Technical Report

UCAM-CL-TR-871
ISSN 1476-2986

Number 871

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Architecture-neutral parallelism via the Join Calculus

Peter R. Calvert

July 2015

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 2014 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Ever since the UNCOL efforts in the 1960s, compilers have sought to use both source-language-neutral and architecture-neutral intermediate representations. The advent of web applets led to the JVM where such a representation was also used for distribution. This trend has continued and now many mainstream applications are distributed using the JVM or .NET formats. These languages can be efficiently run on a target architecture (e.g. using JIT techniques). However, such intermediate languages have been predominantly sequential, supporting only rudimentary concurrency primitives such as threads. This thesis proposes a parallel intermediate representation with analogous goals. The specific contributions made in this work are based around a *join calculus abstract machine* (JCAM). These can be broadly categorised into three sections.

The first contribution is the definition of the abstract machine itself. The standard join calculus is modified to prevent implicit sharing of data as this is undesirable in non-shared memory architectures. It is then condensed into three primitive operations that can be targeted by optimisations and analyses. I claim that these three simple operations capture all the common styles of concurrency and parallelism used in current programming languages.

The work goes on to show how the JCAM intermediate representation can be implemented on shared-memory multi-core machines with acceptable overheads. This process illustrates some key program properties that can be exploited to give significant benefits in certain scenarios.

Finally, conventional control-flow analyses are adapted to the join calculus to allow the properties required for optimising compilation to be inferred. Along with the prototype compiler, this illustrates the JCAM's capabilities as a universal representation for parallelism.

Acknowledgements

This dissertation would never have been completed without the help and support of a number of people. My supervisor, Alan Mycroft, has allowed me the freedom to explore my own ideas, and been invaluable in teaching me how to present them.

I have been fortunate for the discussions that I've had with numerous members of the Computer Laboratory over the last four years, particularly with other members of the programming research group. There are too many to name, but particular thanks to Dominic, Robin, Jukka, Tomas, Raoul, Janina, Stephen and Raphael. I must also thank all those who supervised me during my time as an undergraduate, especially Andy Rice, for their enthusiasm which encouraged me to pursue a PhD.

I am also grateful to the Schiff Foundation for their funding which enabled me to undertake this research, and to Trinity College for its support during my time at Cambridge.

Finally, I would like to thank my parents for their endless encouragement and advice.

Contents

1 Introduction **11**

1.1 Research Context . . . 12

1.1.1 Increased Development Productivity . . . 12

1.1.2 Reduced Barrier to Research . . . 13

1.1.3 Elegance . . . 13

1.2 Outline and Contributions . . . 14

1.3 General Notation . . . 15

1.4 Publications . . . 15

2 Technical Background **17**

2.1 Computer Architecture . . . 17

2.1.1 Moore's Law . . . 17

2.1.2 The Power Wall . . . 18

2.1.3 Multi-Core . . . 18

2.1.4 Vector Processors . . . 18

2.1.5 Heterogeneous Architectures . . . 20

2.1.6 Moving away from Uniform Memory . . . 20

2.1.7 Dataflow Architectures . . . 21

2.2 Virtual Machines and Intermediate Representations . . . 21

2.2.1 Conventional Approaches . . . 22

2.2.2 Dataflow Approaches . . . 23

2.2.3 Alternative Techniques . . . 23

2.3 Programming Language Models of Parallelism . . . 25

2.3.1 Threads and Tasks . . . 25

2.3.2 CUDA and OpenCL . . . 31

2.3.3 Nested Data Parallelism . . . 32

2.3.4 Embedded Domain Specific Languages . . . 33

2.3.5 Dataflow and Streaming Languages . . . 33

2.3.6 Message Passing . . . 34

2.4 The Join Calculus . . . 36

2.4.1 Origins, Semantics and Related Models . . . 37

2.4.2 JoCaml . . . 39

2.4.3 The Joins Library . . . 40

2.5 Control-Flow Analysis . . . 43

2.5.1 Constraint-based . . . 43

2.5.2 Abstract Interpretation . . . 45

2.5.3 Call Strings . . . 48

2.5.4 Escape-based Techniques . . . 49
2.6 Summary . . . 49

3 The Join Calculus Abstract Machine 51
3.1 Arriving at the Join Calculus . . . 51
3.1.1 Supporting Non-Deterministic Choice . . . 51
3.1.2 Exposing Fine-Grained Parallelism . . . 52
3.1.3 Reintroducing Functions and Dynamic Behaviour . . . 54
3.1.4 Granularity . . . 57
3.2 Making Data Dependencies Explicit . . . 58
3.3 The Join Calculus Abstract Machine . . . 61
3.3.1 Semantics . . . 63
3.3.2 Paths and Traces . . . 65
3.4 Relation to Other Models . . . 66
3.4.1 Coordination Zoo . . . 67
3.4.2 Streaming Computations . . . 69
3.4.3 Task-based Parallelism . . . 70
3.4.4 Transactions . . . 74
3.4.5 Objects and Actors . . . 75
3.4.6 Complex Memories . . . 76
3.5 Summary . . . 81

4 Efficient Implementation 83
4.1 A Baseline Implementation . . . 83
4.1.1 Specialised Emission Functions . . . 84
4.1.2 Join Calculus Data Structures . . . 85
4.1.3 Work Stealing for the Join Calculus . . . 87
4.2 Profiling Bottlenecks . . . 88
4.3 Closed Definitions . . . 88
4.4 Bounded Queues . . . 90
4.4.1 Annotations . . . 92
4.4.2 Implementation . . . 92
4.5 Inlining . . . 93
4.5.1 Transition Inlining . . . 94
4.5.2 Definition Inlining . . . 95
4.6 Summary . . . 96

5 Control Flow Analysis 97
5.1 Translating 0-CFA to the Join Calculus . . . 97
5.2 Dealing with Message Interaction: 0-LCFA . . . 99
5.3 Abstracting Call-DAGs: k-LCFA . . . 102
5.4 Correctness of k-LCFA . . . 107
5.4.1 Constraint Generation . . . 108
5.4.2 Closure Algorithm . . . 112
5.5 Queue Bounding . . . 114
5.6 Closedness of Definitions . . . 115
5.7 Worked Examples . . . 115
5.7.1 Foreground call-strings example (‘handshake’) . . . 116

5.7.2 Closedness example (`fib`) . . . 118
5.8 Summary . . . 120

6 Evaluation and Discussion 121
6.1 Test Environments . . . 121
6.2 Benchmarks . . . 122
6.2.1 `fib` . . . 123
6.2.2 `nqueens` . . . 123
6.2.3 `quicksort` . . . 123
6.2.4 `locks` . . . 123
6.2.5 `barrier` . . . 123
6.2.6 `rwlock` . . . 123
6.2.7 `queue` . . . 124
6.2.8 `blackscholes` . . . 124
6.3 Sequential Performance . . . 124
6.4 Scaling to Multiple Cores . . . 126
6.5 Accuracy of Analysis . . . 131
6.6 Better Scheduling . . . 132
6.6.1 The Scheduling Problem . . . 132
6.6.2 Improvements on Basic Work Stealing . . . 133
6.6.3 Transition Inlining . . . 134
6.7 Summary . . . 136

7 Conclusion 137
7.1 Compilation to Other Architectures . . . 138
7.2 Further Analysis . . . 138
7.3 Better Scheduling and Inlining . . . 139
7.4 Final Remarks . . . 139

Bibliography 149

Chapter 1

Introduction

The shift in processor design from ever-increasing clock speeds to multi- and many-core parallelism has been well-documented [109]. Shared-memory multicore systems are now ubiquitous; clusters of these common place; general purpose GPUs mainstream; and the range of esoteric research designs ever-expanding. Indeed, techniques to exploit such architectures have been an especially hot research topic over the last decade. However, for the most part, these have considered a restricted set of problem domains or architectures. It has even been argued that building specialised compilers from each form of parallelism to each target is the only viable approach [27, 86].

Such a conclusion does not sit well in computer science, a discipline almost defined by the view that "all problems ... can be solved by another level of indirection" (David Wheeler). Nowhere is this more true than in the compiler community which has used abstractions so effectively for sequential architectures. The Java Virtual Machine (JVM) [69], .NET's Common Language Runtime (CLR) [36] and the LLVM project [1], currently three of the most popular compiler frameworks, are all based around intermediate representations (IRs) that support both multiple languages and targets. The portability that these provide is arguably the main reason for their success—programmers want their programs to be agnostic of target architecture. However, IRs also allow analyses and optimisations to be shared between compilers. Along with this, new language frontends can make use of existing backends and vice-versa, vastly reducing the human effort required in developing new compilers. This benefit was first noticed during research into UNCOL [107]. Indeed, the total effort becomes proportional to $M + N$ for M languages and N targets, rather than to $M \times N$. However, it is not sufficient to simply take the union of the different primitives as this would imply that the IR needs expanding for each new language. Instead, we need a simplifying abstraction to which these can not only be compiled, but also extracted so that features of different architectures can be exploited.

Such frameworks also remove many of the barriers to research in specific parts of the compilation process—as has been shown by the flurry of papers related to the JVM, and more recently centred around LLVM.

As suggested by the popularity of managed languages, introducing the correct IR abstractions also tends to make life easier for application developers. It can be argued that this is because offering the *right* abstractions allows the crude ones to be removed—for instance the replacement of manual memory management by garbage collection and references.

Current managed VMs provide little more than threading for concurrency, and even this is provided through libraries rather than machine instructions. Synchronisation is then coordinated manually through shared memory or perhaps simple locks. Compiling other primitives to such IRs loses information and therefore restricts optimisation. My research has focussed on replacing these with the *right* primitives in order to bring the aforementioned benefits of a common IR to a parallel setting. This dissertation presents the results of my efforts in support of the following thesis:

> Using the join calculus at the intermediate representation level allows compilation from a variety of parallel source languages to a range of parallel hardware, without the need for specialised compilers, and whilst maintaining a high level of performance.

In particular, I propose a new abstract machine[1], justify its suitability for purpose, and detail how it can be efficiently implemented to give a feasible solution.

Of course, this is not a silver bullet for parallel programming: programmers must still make the parallelism explicit in the IR via a language frontend—there is no magical auto-parallelisation; and while there are significant benefits to a universal IR, there are inevitably trade-offs. David Wheeler famously completed the earlier quotation by stating that abstraction “usually will create another problem”—namely overheads. The prototype implementation presented is competitive but, as one would expect, slower than the direct production compilers. However, just as the performance of Java Virtual Machines has improved over time [68], further research can only close the gap.

1.1 Research Context

The research contained in this thesis sits at the junction between hardware and programming languages. There is also a significant theoretical influence. It is therefore important to set the scene for why this research is relevant. Here I first describe its practical benefits to developers and researchers, before commenting on it from a more pure computer science sense.

1.1.1 Increased Development Productivity

It is becoming clear that modern systems are not only increasingly parallel but also heterogeneous. Common examples include IBM’s Cell Broadband Engine and also CPU-GPU combinations. As well as different processing capabilities, the different cores have access to separate memories, with data transfers needing to be managed explicitly. As these become more popular, how can we continue to offer developers the performance portability that they have come to expect from the JVM on sequential architectures? That is, enable a single program to achieve good (if not optimal) performance on any system. For example, with NUMA multi-core processors and CPU-GPU combinations, placement, scheduling and indeed algorithm choices affect the overall execution time and, for portable programs, must adapt to the target machine at either load-time or run-time.

Being able to achieve this portability is crucial to allowing developers to continue to produce higher performance code. Without a single common representation for programs,

[1] In this work, I treat the terms *abstract machine* and *virtual machine* as more or less interchangeable. In general, I will use the former when talking about the design of the machine itself, and the latter when considering its implementation.

developers will be forced to hand-optimise their software for each platform in order to get the best results.

Despite large amounts of research, auto-parallelisation is still restricted to specific scenarios (e.g. vectorisation). As a result, it is important that any common representation can express any parallelism in a source-language so that it need not be redetected via analysis when generating code for a target.

1.1.2 Reduced Barrier to Research

As well as helping end-users, a common representation serves three research communities. A common problem when evaluating new architectures or languages is that of comparing prototype compilers with established compilers for the status quo. In a sequential setting this problem has been mitigated by tools such as the JVM and LLVM. A standard intermediate representation for parallelism would therefore be a significant aid to the development of parallel languages and architectures.

It also serves as a testing harness for those working on the analysis of parallel programs. By implementing new techniques in terms of the universal IR, it becomes possible to test its effectiveness across a wider variety of scenarios. Indeed, further work on auto-parallelisation could try to convert sequential parts of programs in this IR into equivalents that run in parallel.

In each of these cases, the researcher would reuse the majority of an established toolchain, replacing just the parts corresponding to their work.

1.1.3 Elegance

Whilst the changes in parallel hardware are relatively well understood, it is currently much less clear what the corresponding changes in programming languages and compilers should be. Even aside from the motivations above, computer science is a discipline that prides itself on abstractions, and not enough attention has been given to the abstraction of parallelism. Macrakis' summary of early developments in intermediate representations demonstrates that such elegance has always been a goal—"a universal intermediate language has been a dream for many years" [70]. This has been repeated more recently, with authors feeling that [122]:

> The multicore trend . . . behooves us as a community to study the fundamental requirements in parallel execution models and explore how they can be supported by first-class abstractions at the IL [intermediate language] level.

And others conveying the key requirement that [64]:

> One needs to be able to represent as many of the existing (and, hopefully, future) parallel constructs while minimizing the number of new concepts introduced in the parallel IR.

Another key point made by Macrakis is as follows:

> ANDF [Architecture-Neutral Distribution Format] is not a tool for making non-portable software into portable software, but a tool for *distributing* portable software, which must thus provide *mechanisms* needed by portable software.

That is to say, a two-threaded program should be expressible in our IR, but it is not expected that our IR will magically allow it to run with better performance on a quad-core processor.

An apt comparison is to foreign language translation. Our approach is akin to using two expert translators to translate from, say, Norwegian to Chinese, where both are fluent in English and use this as an intermediary step. Of course, some nuances will be 'lost in translation', and a fluent speaker of both languages will produce a better result. However, in the absence of such a person, the results of the two translators are almost certainly better and easier to obtain, than one would attain from attempting to teach Chinese to the Norwegian or vice-versa.

1.2 Outline and Contributions

This dissertation's contributions are structured as follows:

Chapter 2 provides an overview of the current state of affairs. This includes the trends in hardware, compiler representations and programming models, and attempts to group them into broad categories. I also provide introductions to the *join calculus* upon which this work is based, and to the analysis techniques that will be employed in Chapter 5.

Chapter 3 introduces and justifies the *Join Calculus Abstract Machine* (JCAM) which forms the basis of this entire thesis. It is argued that this representation captures the quintessential features of parallel computation. The introduction develops the model from more familiar starting points in a series of distinct steps—each intended to address a key design criterion. Formal semantics for the final abstract machine are then given in a *small-step* style. I show that it can elegantly and efficiently encode the features of other IRs and language paradigms without simply offering the union of all other primitives.

Chapter 4 demonstrates that it is possible to build high-performance implementations of the JCAM primitives. As well as drawing on existing approaches, it is shown that specialised techniques based on program annotations can offer significant benefits. In particular, these exploit two common cases: (i) where hardware parallelism has been exhausted and we wish to execute the program with minimal overheads sequentially; and (ii) where message queues obey predetermined constraints on their size. The advantages of these optimisations are demonstrated with microbenchmarks and my prototype compiler *Dovetail*.

Chapter 5 presents analysis techniques derived from traditional control flow analyses (CFAs) that allow inference of the properties required by Chapter 4. The combination of message sends and join patterns forces a novel approach to k-CFA, with standard call-string histories not being applicable. The importance of join calculus definition instances in the semantics of the abstract machine also forces other changes to the analysis. Using the results of the control flow analysis, I demonstrate how to infer both some message queue bounds and also definition *closedness* (a key property for achieving fast sequential performance). As well as serving the needs of Chapter 4, this shows the JCAM to be a feasible focus of analysis. This chapter includes a formal proof of k-LCFA correctness.

Chapter 6 finally provides a holistic evaluation of the work using a range of larger benchmarks. This assesses both the final performance, and also the effect of the analyses and optimisations. I also investigate cases that the techniques used do not cover, and suggest areas where this work could be improved in the future.

1.3 General Notation

The topics introduced throughout this thesis frequently make use of certain concepts. The notation for these is therefore introduced here.

Both the Petri-net and join calculus models will be defined by *multisets*. I will use $\mathbf{m}X$ to denote the set of all multisets over the set X. This is equivalent to $X \mapsto \mathbb{N}_0$. The operators $+$ and $-$ on these multisets are simply the lifting of operators onto the function form. To reduce clutter, I abuse the notation slightly by writing $X + x$ instead of $X + \{x\}$ when adding a single element to the multiset.

The set of all possible sequences over X is written X^*, and those bounded by length k as $X^{\leq k}$. A sequence is written $\bar{v}$ and has elements v_i. Fixed-length sequences are also referred to as tuples. Concatenation is denoted by $\cdot$ and ϵ gives the empty sequence.

Diagrams depicting program graphs generally follow the convention that boxes ($\Box$) represent computation while circles ($\bigcirc$) represent data.

When referring to program execution, $\rightarrow$ (or other varieties of arrow) always indicates a single program step, while $\rightarrow^*$ any number of steps. The semantics of Chapter 3 attempt to follow other formalisations of LLVM IR [121].

Especially in Chapter 5, common use is made of abstract values. These are represented by *hatted* symbols—for example, $\hat{X}$ corresponds to the abstract version of the concrete value X. The presentation of constraints takes most inspiration from Faxén's work [38].

1.4 Publications

Some of the work associated with this thesis has already been published. More specifically:

- Some early ideas concerning an intermediate representation based on Petri-nets, and a construction similar to Section 3.4.6 were presented at the *Euro-Par* conference in August 2011 [22]. These ideas also appeared on a poster at the ACACES summer school in 2011.

- The flattened join calculus and an updated version of the EuroPar work to use the join calculus was then published at the *Programming Language Approaches to Concurrency and Communication-cEntric Software* (PLACES) workshop in March 2012 [24].

- The majority of the theoretical work in Chapter 5 on analyses and transformations has been published at the *Static Analysis Symposium* (SAS) in September 2012 [23]. This paper also made use of the JCAM with its simple instruction set for representing the join calculus—although it was presented in a stack-based form similar to the JVM.

The Dovetail compiler developed in Chapter 4 is available from the author's GitHub account [21]. It consists of the compiler itself, written in 2000 lines of ML, along with a runtime library written in a combination of C and LLVM assembly. To build these requires OCaml and the LLVM bindings for OCaml, along with the Boehm garbage collector [18]. The repository also includes all the benchmarks used in Chapter 6, comprising approximately 1500 lines of JCAM assembly, and the corresponding C, Wool and Java versions.

Chapter 2

Technical Background

By its very nature, designing a suitable IR requires a strong understanding of the styles of parallelism present in languages and those supported by hardware. This chapter gives an overview of the most significant varieties in Sections 2.1 and 2.3 respectively. A survey of existing compiler representations that link between these, and their support for parallelism, is provided in Section 2.2. I also introduce the join calculus (Section 2.4) which is used extensively in this dissertation, and the work that has previously been done on its implementation. Finally in Section 2.5, I examine several analysis techniques that provide relevant background for Chapter 5, where I use program analysis to enable optimisations.

2.1 Computer Architecture

Given that the requirement for parallelisation has been driven by limitations in the sequential scaling of hardware, it is important to understand these factors. This section explores the limits of sequential speed as well as looking at the new techniques used to circumvent these limitations. Section 2.1.7 also touches on dataflow architectures. Whilst these were a more active area of research in the 1970s, their ideas are relevant to the approach taken in this thesis.

2.1.1 Moore's Law

Moore's Law is well known: that the number of transistors that can be squeezed into a given area doubles every two years [77]. With signals therefore having to travel smaller distances, clock frequencies initially tended to increase at a similar rate.[1] Through the later decades of the twentieth century, this was the basis of the huge performance improvements. For a software developer, the massive attraction of this period was that everything stayed sequential. Therefore, all software could benefit immediately from improvements in hardware, and get *easy* performance gains.

[1] Due to other improvements in design, clock frequencies in fact doubled every 18 months, and this is often (incorrectly) quoted as Moore's Law.

2.1.2 The Power Wall

Unfortunately, an exponential trend such as doubling of clock frequencies is always going to struggle to last indefinitely. While Moore's Law has continued to the current day,[2] clock frequencies plateaued early in the 2000s. This is due to power usage of a transistor increasing linearly with frequency.[3] Providing this power is not problematic, but it must also be dissipated by the chip. Keeping chips with such high power density cool is simply not feasible in the vast majority of cases.

Sequential processors have circumvented this *power wall* to a certain extent with *superscalar* techniques. These allow *instruction-level parallelism* (ILP) by executing multiple instructions at once using multiple functional units, and allowing instructions to be reordered by examining data dependencies. The result is faster execution for the same clock rate. Implementing instruction reordering is not dissimilar to the dataflow style of processors examined in Section 2.1.7.

For the purposes of this thesis, the benefits of ILP can be regarded simply as part of a sequential core's performance.

2.1.3 Multi-Core

As a consequence of the power wall, and also because it becomes increasingly difficult to find further parallelism through ILP, *multi-core* processors were developed. These have become widespread over the past decade, and the number of cores available on a chip is continuing to rise. While *symmetric multiprocessing* had been popular long before the plateau in clock frequencies, the introduction of multi-core processors has certainly encouraged parallel research. These architectures can be described as *multiple instruction multiple data* (MIMD). Unlike improvements due to clock speed and ILP, taking advantage of multiple cores requires significant alterations to programs.

2.1.4 Vector Processors

Vector processors were originally introduced into supercomputers as a way of achieving higher performance [100]. By performing the same operation on multiple items of data (*single instruction multiple data* or SIMD), the instruction fetch and decode steps only need to be performed once. It is also possible, and common, to add extra functional units that allow the different data to be processed in parallel. For *data-parallel* computations, this is more efficient than a multi- or many-core MIMD approach.

SIMD-style instructions have been commonplace amongst mainstream CPUs for a number of years (e.g. the MMX, SSE and AVX extensions). Graphics processors are another prevalent instance of vector processing. While these were originally intended for graphics rendering, use for general-purpose computation (i.e. GPGPU) is now fairly standard, as described in Section 2.3.2. The architecture of a typical graphics processor is shown in Figure 2.1. Each of the processors within a GPU's multiprocessor executes the same instructions, and in turn each of these supports multiple hardware threads.

As is a common theme in parallelism, it is clear that the development of vector processors was very much driven by the hardware, rather than the appeal of it as a software

[2] Opinion varies on how much longer it is likely to last.

[3] More specifically, the power usage of a single transistor is proportional to CV^2f where C is the capacitance of the load, V the voltage and f the frequency.

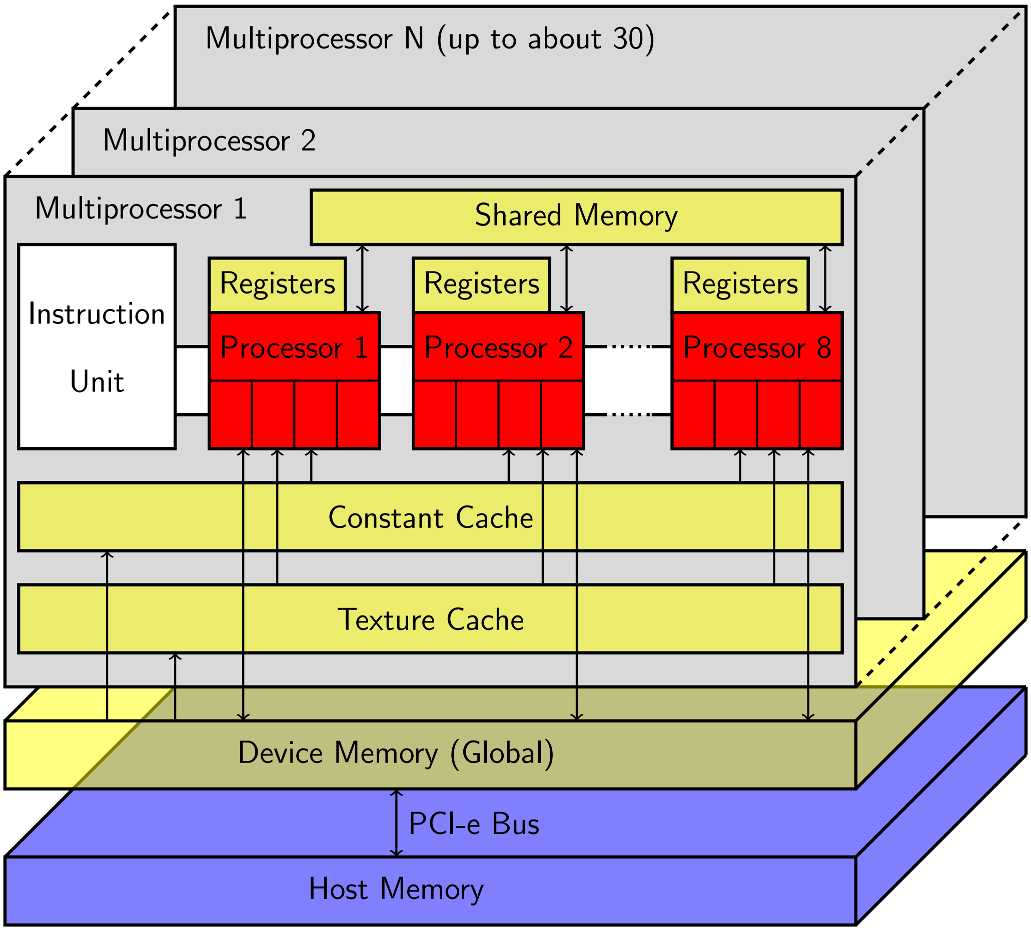


Figure 2.1: NVIDIA Hardware Architecture (taken from [20]).
(Based on a figure used in various NVIDIA presentations.)

model. An example of where they can be difficult to work with is when the operations being performed on data points require the use of `if` statements. If any threads tied to the same instruction by hardware need to take different paths through code, then the different paths must be executed sequentially after each other, with the irrelevant threads being masked off in each case.

2.1.5 Heterogeneous Architectures

Given the variety in processor designs now available, there is considerable interest in producing systems of heterogeneous cores. The most common example of this is the standard pairing of a graphics processor with a multi-core CPU. However, there are many other cases too, such as the IBM Cell Broadband Engine used in the Playstation 3 which has a main *Power Processing Element* (PPE) and eight[4] smaller *Synergistic Processing Elements* (SPEs) [30]. The motivation is that in a program there may be some tasks better suited to one variety and others to another. Even when all cores share the same instruction set, there are benefits to having a powerful core that can execute chunks of sequential code, along with many less powerful cores that are suited to parallel regions.

Although this offers the opportunity for better performance, it also presents a scheduling challenge—adding a spacial dimension to the existing temporal problem. While we will see some work later (Sections 2.3.1.3 and 2.3.5) that addresses this automatically, to date the majority of programs are written specifically for a certain architecture and the placement of computation is managed manually by the developer.

There has also been work on architectures that allow homogeneous *tiles* within a processor to be reconfigured depending on the exact needs of an application [110]. However, this thesis focuses on software adapting to hardware. It is unclear how one would approach a situation where both hardware and software are trying to adapt, as there are potentially too many degrees of freedom for a scheduler to consider.

2.1.6 Moving away from Uniform Memory

The CPU-GPU combination that we have already seen is clearly an example where there is not a single shared memory across the whole architecture. However, even within a homogeneous multi- or many-core scenario it is unlikely that architectures will always be able to offer cache-coherent shared memory. Unfortunately, as the number of cores grows, the overheads involved in maintaining cache-coherence between the grid of cores on a chip become prohibitive. Indeed, Intel's research into scalable many-core chips has abandoned cache-coherence, emulating it in software if required by a program [115].

These *non-uniform memory architectures* (NUMA) require management from software to ensure that data is in the right place for a computation. This data movement is performed using messages, which of course have a cost. It is managing these costs that make these architectures so challenging. A lot of work focuses on the CPU-GPU scenario, but the *partitioned global address space* (PGAS) model [32] has become relatively popular and is used within the X10 language. X10's other features can be categorised as fine-grained task parallelism and are discussed in Section 2.3.1.2. Under a PGAS model, the language exposes the reality that not all memory references are local to the current

[4] Note that in the Playstation 3 only seven of the SPEs are operational. This allowed the manufacturing yield of the processor to be increased.

computation, and therefore cannot be accessed as cheaply. Typically languages will insert the necessary data transfers for non-local dereferences, and also provide constructs that move computation to the local location of data.

2.1.7 Dataflow Architectures

Dataflow machines were an especially hot topic of research during the 1970s, with work being conducted in research departments across the world, including Manchester and MIT. The underlying idea in these is that rather than expressing programs as an ordered list of instructions, the dependencies between instructions are described. This allows the instructions to be executed in parallel. There were two distinct types of dataflow architecture.

Static Dataflow was the earlier approach and allows the use of standard memories. Dependencies are described using the memory addresses of the predecessor instructions. Unfortunately this only allows a single instance of an instruction to be executed at once.

Dynamic Dataflow tries to overcome this by using *content-addressable memory* (CAM). This allows multiple instances of the same instruction to be executed at once. However, it is worth noting that CAMs are very expensive to implement.

Ultimately, it became clear that dataflow architectures were too fine-grained to be effective. The synchronisation and coordination required to execute a single instruction often eclipsed the time taken to perform the instruction itself. Despite this, as already mentioned, many ideas from this research do live on. Modern superscalar processors use dataflow techniques to implement instruction reordering. Typical reordering windows contain between 100 and 200 micro-operation (μop) entries [53, slide 24]. This allows them to be stored in dedicated hardware, overcoming the issues encountered with fully-fledged dataflow architectures.

2.2 Virtual Machines and Intermediate Representations

It could certainly be argued that intermediate representations, and architecture-neutral distribution formats, for sequential architectures are a solved problem. Whilst there are differences between the representations used in the main compilers, the core principles are the same. This can be attributed to some extent to the fact that sequential architectures in themselves have converged on fairly similar, RISC-like instruction sets.

It is unsurprising that the mainstream representations are all varieties of control-flow graphs, since there are no commonly available processors with anything but functions and branching in their instruction sets to describe program flow—even if they use dataflow techniques internally to reorder instructions (Section 2.1.7). Since performance improvements on a sequential architecture result in instructions being executed faster, this translates to a faster passage through the control-flow graph, without any need for the program to be rewritten. Section 2.2.1 reviews several control-flow-based representations and their adaptation to the advent of parallelism. Sections 2.2.2 and 2.2.3 look at less mainstream work that expresses dependencies within the IR.

2.2.1 Conventional Approaches

As mentioned in the introduction (Chapter 1), efforts to produce uniform intermediate representations [107] and architecture-neutral distribution formats [70] date back to the early 1960s. The first of these to receive widespread use was BCPL's OCODE [97]. More recently, the JVM [69], .NET's CLR [36] and LLVM [1] have all become popular. Despite some differences in design decisions between these, I argue in this section that they are all very similar in nature.

The first key difference is that the JVM and .NET are very obviously slanted towards the implementation of object-oriented languages, while LLVM offers a more traditional RISC set of operations. Secondly, the JVM and LLVM are at opposite ends of the spectrum with regards to offering runtime support. The JVM is very much a managed environment with garbage collection and restrictions that prevent uncaught runtime errors. On the other hand, LLVM makes no effort to prevent program crashes, simply offering a universal language that can be compiled to a number of targets. Microsoft's .NET lies somewhere in the middle, supporting both managed and unmanaged operations. I view the choice between stack (JVM and .NET) and register (LLVM) machines as insignificant—converting between the two is relatively trivial. The register machine adopted by LLVM uses a *single static assignment* style where each register is only assigned to once. In order to support loops and other convergent control-flow structures, ϕ-nodes are used, which allow assignment based on the path taken into a basic block by specifying the value to use for each predecessor.

All three intermediate representations are effectively control-flow graphs within an overall structure of functions. One can view the object-oriented nature of the JVM and .NET simply as one way of providing indirect function calls. The choice between manual and automatic memory management is not relevant to this dissertation.

These representations have all been popular as targets for a number of other languages (e.g. Scala). However, this is not necessarily because they are a natural fit. Indeed, compiling functional and dynamic languages to the JVM is relatively difficult to do well, and the subject of a number of research papers [10, 98]. Despite this, frontend developers are drawn to these by the desire to reuse the mature optimisations that they provide and achieve portability. This is confirmed by the presence of other VMs, such as Parrot [89] which is far better suited to dynamic languages, that have received comparatively little attention.

Unfortunately, there is very little support for parallelism in any of these VMs. All three do define memory models to allow reasoning about threaded programs. Beyond that, they expect threads to be provided by libraries.[5] A slight exception to this is that the JVM does provide instructions to enter and leave mutex locks. Compiling other primitives to threads does not truly express the parallel nature of the high-level language, with most information about the original constructs thrown away and difficult to reconstruct. These virtual machines therefore struggle to perform any optimisation of parallel or concurrent constructs within programs.

It is worth noting that there has been some success in compilation of these sequential IRs to graphics processors. This was first done as an extension to the Jikes implementation of the JVM [67], however, subsequent efforts in the context of the JVM both by myself [20] and others [88] have preferred to perform a transformation on bytecode ahead-of-time.

[5] Java and .NET both provide threading in their standard libraries, even though under the hood their implementations are heavily tied into the virtual machine. LLVM is less tied to a single implementation.

There has also been work to produce NVIDIA's PTX assembly code from LLVM bitcode [96], and an (albeit different) PTX backend is now part of the mainline LLVM source tree. However, in all cases these simply produce bodies of GPU kernels, relying on other annotations or code to express the grid across which this should be run.

There has also been more theoretical work to produce parallel *bridging models* from a sequential base. The best-known of these is Valiant's *bulk synchronous parallel* (BSP) model [114]. In BSP, execution is a sequence of *supersteps*. Within a step, computation occurs on multiple threads, with disjoint local memories. These synchronise at the end of each superstep. The threads can also get or put values from remote memory, and these accesses also synchronise at the end of the superstep.

2.2.2 Dataflow Approaches

While control-flow graphs have been more common, there has also been a significant amount of work on data-flow graph representations. These tend to have to find some way of supporting control constructs, such as conditional branching and loops. Perhaps the most elegant approach is that of the *value state dependency graph* (VSDG) [60]. It takes inspiration from single static assignment form and uses a combination of *state edges* and ϕ-nodes to encode all control flow. These state edges can only describe one thread of execution through the graph, so there is no control parallelism (i.e. multiple tasks or threads). It is of course preferable to minimise the number of nodes that are involved in the state edges. Allowing such freedom in the ordering of instructions is beneficial for both the original aim of the VSDG, register allocation and code motion, and also for parallelism.

2.2.3 Alternative Techniques

Other work [122] has acknowledged some of the same issues that this work is trying to address—namely, that compiling parallel constructs to a sequential IR loses information and therefore might restrict optimisation. However, the majority of other work is intended to be specific to a single source-language. Some very recent work aims for generality [64], balancing this with a high-degree of pragmatism in an attempt to ensure it can be integrated with existing compiler technology. This subsection describes these approaches so that my IR (Chapter 3) can be seen in context.

X10. There has been a large amount of research surrounding X10 [102] and its descendent languages [28]—largely led by Vivek Sarkar. They propose using multiple levels of *parallel intermediate representation* (PIR) for the compilation of task parallel languages [122]. The 'highest' of these, the *High Level PIR* (HPIR), is effectively a slightly simplified abstract syntax tree. It is based around a *region structure graph* (RSG). The most significant part of this is the *region structure tree* (RST). This encodes the program as a tree where leaf nodes correspond to IR instructions, and internal nodes (*regions*) to control constructs—for example, `async`, `finish`, and both sequential and parallel loops. Each of these regions is then backed up by a *region control flow graph* (RCFG) that describes how the immediate children of the region relate to each other. The next level down of IR replaces parallel loops with a sequential loop of `async`s, before the lowest reduces parallel constructs to runtime calls.

VCODE. Another approach tied closely to a single programming paradigm is `VCODE` [13]. This develops a conventional stack-based IR to use only vector datatypes, and support nested parallelism. It does this indirectly using *segment descriptors*. These partition another vector into *segments*. Scan and permutation instructions then act within the segments. For example, the `+_scan` instruction computes the (exclusive) prefix sum within each segment:

$$\begin{aligned} \text{Vector} &= \texttt{[1 3 5 7 11 13 15]} \\ \text{Segment Descriptor} &= \texttt{[4 3]} \\ &\boxed{\texttt{+\_scan}} \\ \text{Result} &= \texttt{[0 1 4 9 0 11 24]} \end{aligned}$$

The process of compiling nested parallelism into this representation is described in Section 2.3.3.

SPIRE. The most recent work in this field [64] describes a transformation process (SPIRE) for generating PIRs from an existing sequential language. It does this through three alterations[6] that allow control and data parallelism, but fail to offer dataflow or event-driven support:

Execution Style. Whenever groups of statements are introduced by the IR (e.g. blocks and loops), an annotation, that specifies whether the statements should be executed sequentially or in parallel, is added.

Synchronisation. The authors propose that synchronisation of language constructs is specified in two ways.

1. Each statement is given a synchronisation attribute. This indicates, for example, whether the statement should be executed asynchronously by another thread, whether the statement needs to execute atomically, whether all children spawned by the instruction must complete before the statement concludes, etc.
2. To perform more coarse-grained synchronisation, *events* are used. These operate as a counter with the following operations:
   - `newEvent(int i)` creates a new event with an initial value of `i`.
   - `signal(event e)` increments the value of the event by one.
   - `wait(event e)` waits until the value of the event is strictly greater than 0. It then decrements the event before allowing the thread to continue.

   An event with a positive initial `i` will allow `i` threads to call `wait` and continue, before the next blocks awaiting a `signal`. Events with an initial value that is negative are useful, for example in the pseudo-code of Figure 2.2 which implements a barrier. Without the second `signal` call, only one thread would move past the barrier. With it, the threads resume, as the authors put it, in a "chain-like fashion".

Data Distribution. SPIRE assumes that the existing sequential representation supports shared memory. To this, it adds simple send and receive message passing operations.

[6] The SPIRE work specifically modifies the *PIPS* intermediate language, which they describe as "a comprehensive source-to-source compilation platform", as well as LLVM.

```
ph = newEvent(-(n-1));

for(int j = 1; j <= n; j++) {
  spawn {
    S;
    signal(ph);
    wait(ph);
    signal(ph);
    S';
  }
}
```

Figure 2.2: Implementing a barrier (between `S` and `S'`) with SPIRE's *events*.

2.3 Programming Language Models of Parallelism

This section presents the models, languages and primitives that are currently available for developers to use. I think it would be fair to say that, in the vast majority of cases,[7] performance portability was not a primary concern, with the focus instead being on performance in a specific scenario, or ease of use. The lack of intermediate representation support for parallelism also means that a large amount of the work covered in this section is implemented as non-transparent libraries. To allow analysis and optimisation of these approaches, we need either direct compiler support for the primitives, or a common substrate in which parallel constructs can be expressed transparently. All concepts covered by other recent survey work [63] are included here, offering reassurance that my work has not neglected to consider any key styles of concurrency.

2.3.1 Threads and Tasks

Software threads are very much the de-facto language construct for both parallelism and concurrency. It can be argued that they came about in two distinct ways: as a reflection of how hardware operates; and as an obvious way of achieving concurrency. However, they were never designed as a sensible high-level programming model for parallelism. The key difference between concurrency and parallelism in the context of this thesis is *who chooses the number of threads*. For the purposes of concurrency, a programmer can freely pick a number of threads based purely on the number of things that need to occur without blocking each other. However, to achieve good parallel performance the number of heavyweight software threads needs to be closely matched to the number of threads the architecture can execute at once. Unfortunately, with varying numbers of cores between different processors it becomes very difficult for a program expressed with multiple threads to be truly agnostic of the target architecture.

[7] StarPU and OpenCL spring to mind as exceptions.

2.3.1.1 Coordination

In general, coordination between threads is performed using shared memory along with a combination of synchronisation primitives. By far the most popular primitives are *mutexes* (mutual exclusion locks), *condition variables*, *monitors* and *barriers*—which are all discussed in this section.

Mutexes. Whilst probably being the best known primitive, mutexes are arguably one of the hardest to use correctly. If separate threads attempt to acquire locks in differing orders, then a program will deadlock. Managing a strict order within one's own code is certainly possible, but this becomes much more difficult when composing different libraries. Despite these failings, any universal representation of parallelism will either need locks as a first-class primitive, or a way of encoding them.

Condition Variables. These allow threads to place themselves (via the `wait` operation) on a queue of threads corresponding to some kind of condition or event. Another thread can then indicate that this condition has been met, waking up either one (`notify`) or all (`notifyAll`) of the threads waiting on it. A common pitfall is if a `notify` occurs between another thread checking a condition and calling `wait`. In this scenario (the *lost wakeup* problem), the waiting thread may never be woken up. A second issue is that some implementations may result in *spurious wakeups*, it is therefore typical to call the `wait` operation from within a loop that checks the condition.

Monitors. First proposed by Hoare [57], monitors are simply the pairing of mutexes with condition variables. In addition, a thread that `waits` on a condition variable temporarily releases the mutex. Whilst originally proposed for managing operating system resources, monitors are ubiquitous across concurrent programming.

Barriers. Perhaps the easiest to use of the conventional synchronisation techniques, barriers ensure that a certain number of threads have reached a certain point before allowing any of them to proceed.

These are all provided by POSIX threads (commonly known as *pthreads*) which is the standard library used within the C programming language. OpenMP [34] is a standardised API that provides many of the primitives in this section for the C, C++ and Fortran languages, as well as other patterns—for example, to help with data copying between threads, and performing reductions. The JVM provides monitors, with the mutex operations forming part of the instruction set as noted earlier, and the condition variable operations part of the in-built `Object` class. *Thread Building Blocks* [91] from Intel is also a commonly used toolkit of such coordination primitives, along with implementations of some standard concurrent containers. It also provides support for performing some data parallel operations such as `parallel_for` loops.

Atomic Blocks. A final paradigm to consider is that of *transactions* with `atomic` blocks. Whilst the implementation of these using software transactional memory (STM) is relatively complex, they do offer a far easier programming interface in a multi-threaded environment than those above, and competitive performance. In particular, they compose naturally without introducing the risk of deadlock. Implementations have been produced across a range of languages from Haskell, to Java, to C++.

An interesting development of the STM work is to attempt eliding more conventional locks [99]. Whilst this does not improve the programming interface, it allows developers to use more coarse-grained locking schemes without worrying that they will affect scalability. It also allows the performance of existing code to be improved without rewriting it.

2.3.1.2 Fine Grained

The move towards finer-grained tasks began in the 1990s with the Cilk programming language [16]. Since then, X10 [102], the Wool work-stealing library [39], and others have taken similar approaches. In these languages, function calls can be `spawn`ed, allowing them to be executed by a different *worker*. Primitives are then provided to wait for spawned children to complete, although the exact semantics of these varies slightly between languages. In Cilk, `sync` waits for all children to finish. On the other hand, Wool's `SYNC` simply waits for the most recent spawn in a stack-like manner.

In all these languages, program tasks are separated from hardware threads by *workers*. There is typically one such worker per hardware core, and their sole purpose is to work through a list of (user-space) tasks that have not yet been completed. *Work stealing* [15] has emerged as the de-facto scheduling technique amongst implementations of fine-grained tasks. While implementations of the work queues and stealing operation can be complex, the basic concept is very straightforward. Each worker maintains a separate queue of tasks, and works through this. Once a worker's own queue is exhausted, it attempts to *steal* tasks from another worker.

Both Cilk and Wool focus on delivering high performance in spite of this fine-grained approach, yet do so in very different ways. The research focus of X10 is more on its primitives and approach to non-uniform memory architectures—although we have already discussed its descendants' intermediate representation in Section 2.2.3. X10 also integrates barrier synchronisation with the task spawning model through its *phaser* primitives [104].

In Cilk-5, one of the key techniques is to compile two versions of each function: a *fast clone* and a *slow clone* [49, 90]. The fast clone takes advantage of a key property of Cilk programs—that removing the Cilk keywords results in a sequential C program of identical semantics. The fast clone is executed in the majority of cases, and only converted to the slow clone if it is stolen. This follows the authors' *work-first principle* whereby they try to ensure that any overheads associated with the model are only incurred when parallel coordination occurs. This ensures that sequential execution of a Cilk program, or program-part, is only a little slower than the sequential C equivalent. To understand the mechanics of how this works, we must first describe the order in which Cilk places items on its work queue. Whenever a `spawn` is encountered, it is the continuation of the parent that is placed on the work queue, not the function being spawned. In the fast clone (see Figure 2.3), after the spawned procedure has completed, the generated code simply checks whether the continuation has been stolen—if so, then it can simply return. However, the optimisation is cleverer than just that. Parents are always pushed onto the work stack before children and steals are performed from the opposite end of the work queue. Therefore, we know that a task can only ever be stolen if its parent has already been stolen and switched to the slow clone. Hence in the fast clone, when it returns early due to the continuation having been stolen, it can simply return a dummy value which is always ignored by the slow clone. When the fast code executes to completion, it can make use of the return value as in a normal C program, eliminating any coordination overhead. A consequence of this approach is that the slow clone needs to be able to execute starting

```
int fib(int n) {
  fib_frame *f = alloc(sizeof(*f));  Continuation frame
  f->sig = fib_sig;                  Includes slow clone's code pointer

  if(n < 2) {
    free(f, sizeof(*f));
    return n;
  } else {
    int x, y;
    f->entry = 1;                    Continuation is after first spawn
    f->n = n;                        Save live variables
    push(f);                         Push frame
    x = fib (n-1);                   Call spawned function

    if(pop(x) == FAILURE) {          Try to pop continuation frame
      return 0;                      If stolen, return dummy value
                       (note x was passed to pop so that it is available to the thief later)
    }

    ...                              Second spawn

    ;                               sync is free

    free(f, sizeof(*f));
    return (x+y);
  }
}
```

Figure 2.3: Example of fast clone for `fib` in Cilk (based on Figure 3–2 of [90]).

from any possible continuation in the task (as indicated by `f->entry`). However, since this code is only executed after a steal, its overheads affect the overall performance much less.

On the other hand, as a library approach, Wool cannot compile tasks twice. It therefore seeks to minimise the overheads associated with `SPAWN` and `SYNC` operations. It does this in a number of ways. Whilst some of the optimisations are very specific to the `SPAWN/SYNC` model, there are a number of insights that are relevant to the implementation of any concurrency primitives [40]. Firstly, Wool attempts to reduce indirection by keeping the task descriptors in the work queue itself rather than storing pointers to them. It also attempts to reduce contention on the work queue by dividing it into a *private* area and a *shared* area. A worker can operate on its own private area without fear of contending with other workers. This eliminates the need for expensive memory barriers and compare-and-swap operations. The success achieved through these is encouraging for any implementor of fine-grained concurrency primitives. A later paper [41] discusses strategies for choosing a victim worker during a steal. This could be relevant to future work based on this dissertation but is not discussed here. Actual performance figures for Wool are available in Chapter 6, where it is used as one of several baselines against which my work is evaluated.

Finally, I wish to mention the *Chase-Lev work-queue* [29]. The algorithm itself is shown in Figure 2.4. While Cilk and Wool have focussed on improving the performance of coordination, this approach is still the standard approach for a simple work-queue without coordination (i.e. where tasks do not produce results, and do not wait on the completion of other tasks directly). The goals of the implementation are similar to Cilk's work-first principle, in that they attempt to minimise the cost of operations by the owner worker on the queue. It ensures that push operations by the owner do not require any compare-and-swap (CAS) operations. Similarly, the owner can pop items without a CAS provided that this does not leave the queue empty. The continuation-passing style of my work means that this approach is particularly relevant, and used in my implementation (Chapter 4).

2.3.1.3 StarPU

StarPU [8] is an attempt, by Augonnet and others, to provide a task scheduling library in the context of heterogeneous systems. This extends the authors' earlier work which automates transfers between CPU and GPU memories [7] with a *task execution engine*. They provide standardised interfaces to both program developers and scheduling policy researchers. Each task can be defined with dependencies, and must specify what data are required by the task and how they are accessed (i.e. read-only, write-only or read-write). A number of implementations can then be provided for each task—for example, a GPU version as well as standard CPU code. StarPU manages the dependencies between tasks, automates data transfers, and uses the specified scheduler to choose between versions of a task. All this is done with the aim of optimising performance. The authors argue that this is the only feasible approach for heterogeneous architectures, and that "it is very unlikely that writing portable code which efficiently maps tasks statically is either possible or even productive".

All schedulers in StarPU conform to a very simple interface. They provide two methods, `push` and `pop`, for each computation unit (or *worker*). This accommodates a range of scheduling policies (e.g. work stealing and list scheduling) whilst still being straight-

```
void pushBottom(q, T value) {
  long b = q.bottom;
  long t = q.top;

  if((b - t) >= q.capacity - 1) {
     Grow array
  }

  q.array[b] = value;
  q.bottom = b + 1;
}

T steal(q) {
  long t = q.top;
  long b = q.bottom;

  if(t >= b) return Empty;

  T value = q.array[t];

  if(!cas(&q.top, t, t + 1)) {
    return Abort;
  }

  return value;
}
```

```
T popBottom(q) {
  long b = q.bottom - 1;
  q.bottom = b;
  long t = q.top;

  long size = b - t;

  if(size < 0) {
    bottom = t;
    return Empty;
  }

  T value = q.array[b];

  if(size > 0) {
    return value;
  }

  if(!cas(&q.top, t, t + 1)) {
         value = Empty;
  }

  q.bottom = t + 1;

  return value;
}
```

Figure 2.4: The Chase-Lev Work-Stealing Deque Algorithm [29]

forward enough to implement efficiently. StarPU allows priority and *performance model* annotations to be placed on tasks and implementations respectively. If no performance models are specified, it attempts to infer these automatically with either a pre-calibration run, or dynamically at runtime.

Of the greedy policies that they investigate, *heterogeneous earliest finish time* (HEFT) [112] achieves the best results. It considers a directed-acyclic graph (DAG) of tasks, where each task can be given an expected time on each processor—i.e. exactly the StarPU scenario. An initial *ranking* phase gives each task a priority based on its distance from the last task in the DAG. Tasks are then allocated to the worker on which they will be able to complete first, given that tasks assigned to that worker will execute in priority order.

This work has also been extended to allow for a distributed cluster of heterogeneous machines [6]. This is done by allowing the HEFT scheduling policy to consider an estimate of the time taken to transfer data to the relevant worker. A variant that allows data to be pre-fetched while a task is waiting to execute is also considered.

2.3.2 CUDA and OpenCL

Before CUDA [83] was released in 2007, general purpose use of graphics processing units (GPUs) had to be formulated as graphics operations [54]. CUDA was one of the first frameworks that made programming GPUs more manageable, as in most cases the underlying instruction set is a trade secret. It is therefore inevitable that its features were, and continue to be, very much driven by those available in the underlying hardware. The subsequent OpenCL Specification [78], whilst attempting to be vendor-agnostic and portable, provides an almost identical programming model[8]. Both frameworks are based on C++ with additional keywords for specifying whether functions should be compiled for the host or GPU amongst other things. Whilst the language choice is mostly due to the market share of C/C++, it is a useful indication of the level at which these frameworks operate—i.e. quite close to the hardware.

The OpenCL framework turns out to be a very good example of how basing a *portable* framework around a specific architecture's features can lead to extremely poor performance portability [65]. The difficulty is that although AMD and NVIDIA GPUs appear to offer similar abstractions, their performance characteristics are quite different (e.g. memory access patterns that fit the hardware best). To counteract this, there has been work on parameterised programs that can be *auto-tuned* to a specific GPU by changing loop nesting orders etc.

The programming model for both CUDA and OpenCL is as shown in Figure 2.5. All threads (or *work items*) operate in a data parallel way, however, the level of synchronisation allowed between them depends on whether they are part of the same *block* (or *work group*). Within a block, both frameworks support extra barriers and atomic operations that are not available across the complete *grid*. There is also memory that is shared between threads within a block, but not outside.

[8] For consistency, I use CUDA terminology where there is a discrepancy between the two.

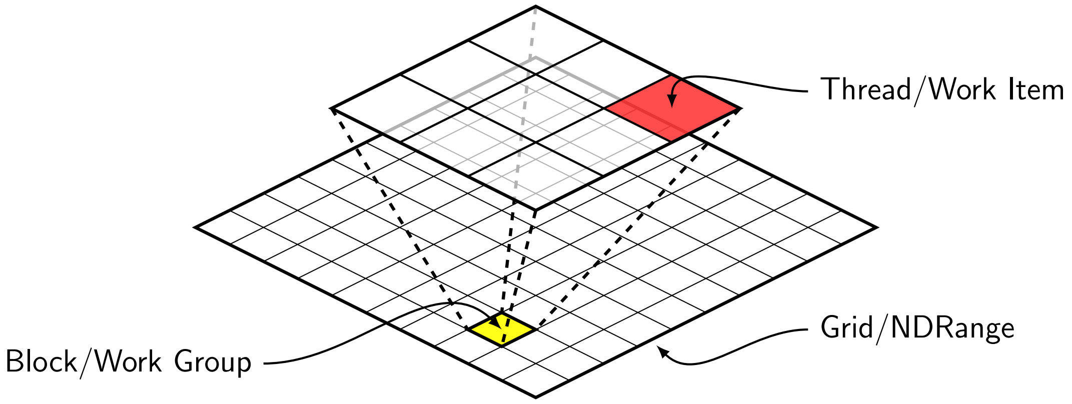


Figure 2.5: Software model of threads under CUDA/OpenCL (adapted from [20]).

```
function qsort(a) =
  if(#a < 2) then a
  else
    let pivot   = [#a/2];
        lesser  = {e in a | e < pivot};
        equal   = {e in a | e == pivot};
        greater = {e in a | e > pivot};
        result  = { qsort(v) : v in [lesser,greater]}
    in result[0] ++ equal ++ result[1];
```

(where # gives the length of a sequence, and ++ represents concatenation)

Figure 2.6: Example NESL quicksort function (taken from [12]).

2.3.3 Nested Data Parallelism

The seminal work on nested data parallelism is Blelloch's *NESL* language [12]. This is a strongly-typed strict first-order functional language, with support for *sequences* whose elements are themselves sequences. Sequences can be nested to an arbitrary depth. Parallelism over these sequences is then offered in two ways:

List Comprehensions. These allow any function in a program to be applied concurrently to each element of a sequence. It is also possible to subselect elements of the sequence. For example (directly from [12]):

`{negate(a) :  a in [3, -4, -9, 5] | a < 4}` ⇒ `[-3, 4, 9]`

In-Built Parallel Operations. e.g. `sum` and `permute`. These can be thought of as building blocks for the rest of an algorithm.

As an example, a parallel version of quicksort is shown in Figure 2.6.

NESL is compiled to the `VCODE` intermediate language introduced in Section 2.2.3. This is done by a process known as *flattening nested parallelism* [14] due to Blelloch and Sabot. All values in NESL are logically encoded as instances of the *pfield* pair-datatype where the first element (*segdes*) is a segment descriptor, and the second (*value*) either another pfield or a sequence of integers. This can of course be compiled trivially into multiple values on the stack and multiple arguments to functions.[9] For a primitive vector, the segment descriptor is a single element sequence containing the vector's length, and the value is the vector itself. For a vector of vectors, the segment descriptor is again the length, but the value is another pfield. This second pfield is the field-wise concatenation of the pfield representations of each sub-vector. For example, the encoding of `[[`a_{00} a_{01}`]` `[`a_{10} a_{11} a_{12}`]` `[`a_{20}`]]` is:

$$\begin{cases}\text{segdes}: & \texttt{[3]}\\ \text{value}: & \begin{cases}\text{segdes}: & \texttt{[2 3 1]}\\ \text{value}: & \texttt{[}a_{00}\ \ a_{01}\ \ a_{10}\ \ a_{11}\ \ a_{12}\ \ a_{20}\texttt{]}\end{cases}\end{cases}$$

Each list comprehension in NESL removes one level from this structure.

2.3.4 Embedded Domain Specific Languages

Another popular approach is to provide a restricted language embedded within another mainstream language—the idea being that the computations expressed are suitable for compilation to a range of platforms, including GPUs and FPGAs. These are typically implemented either with a quoting mechanism (for example within Haskell [71]) or with operator overloading (for example in C# [106], or Python [26]). The *Array Building Blocks* library [80] from Intel also falls into this category, amalgamating earlier work on *Intel Ct* and *Rapidmind*. This code is then compiled for the target architecture in a just-in-time fashion, with the results of this compilation often being cached.

A key advantage of this approach is that it allows new ideas and techniques to be integrated into more mainstream and established languages. However, it is important to remember that this is not a *"silver bullet"*. There is no reason why the embedded language will be any better than the other parallel languages, it might just receive more attention. Indeed, it has been shown that, while they offer *code portability*, it is very difficult, at least in the case of Rapidmind, to achieve *performance portability* "without a deep understanding of the hardware and RapidMind's internal mode of operation" [31].

2.3.5 Dataflow and Streaming Languages

Around the same time that dataflow architectures were receiving attention, there was also research into dataflow programming languages. The first of these was Lucid [117], which introduced the idea of a variable representing a stream of values. A few simple examples of this style of programming are shown in Figure 2.7. More recently, *streaming languages*, some distributed computing frameworks and, to a lesser extent, *co-ordination languages* have continued efforts into data-centric programming.

Streaming languages are particularly applicable to signal processing and media applications. A lot of the work on them has been based around *StreaMIT* [111]. The key construct in StreaMIT programs is a filter. These can be combined into a stream program. A filter takes a source channel, performs operations using `peek` and `pop` operations, and then places results onto an output channel. There are also constructs for supporting

[9] Wherever a pfield is expected, it will be known how deep the nesting of the value field is.

```
fac where
   n   = 1 fby (n + 1);
   fac = 1 fby (fac * n);
  end

fib where
   fib = 0 fby (1 fby (fib + next fib));
  end
```

(where `fby` means ‘followed by’, and `next` advances the stream one element.)

Figure 2.7: Simple Examples of LUCID Programs.

feedback loops and splitting a channel into multiple channels (or joining multiple channels into one). As an example, consider the Fibonacci program in Figure 2.8. Work on streaming languages has included consideration of their mapping to heterogeneous architectures through various forms of scheduling. These are discussed further in Section 6.6.2.

The first large-scale distributed computing model to be based on dataflow ideas was Google’s *map-reduce* [35]. In this framework, there is an initial data-parallel step (map) before these values are reduced to a result. Further research has resulted in more general *dynamic dataflow graphs* such as CIEL [79]. Under such models, a task can spawn a child dataflow graph and delegate its result to this new graph. CIEL forbids cyclic graphs, but it is perfectly valid for a graph to recursively create copies of itself as children. These approaches are in a similar vein to *coordination languages*, where sequential functions are linked together as dependencies.

2.3.6 Message Passing

Message passing has always been a popular approach, both in practice and amongst process calculi. Popular implementations of message passing were driven by the need for a way of communicating between nodes of a cluster. However, the appeal in the theoretical arena was the cleaner behaviour they offer compared to shared memory communication.

The most common standard for message passing is MPI [44]. As well as this library-based approach, there are a number of languages with message passing as a central primitive. A language that I will explore in some detail is *Concurrent ML* (CML) [95], since it provides a good overview of the concepts incorporated in the other languages.

Built on top of standard ML, CML provides threads with blocking and non-blocking operations on channels for synchronous communication. It does this using *events*, which were first explored in the PML research language [94]. A selection of the standard primitives in CML is shown in Figure 2.9. All operations on channels (or I/O, timeouts etc.) produce an event, but return immediately without the operation actually taking place. Composing any of the channel operations with `sync` then results in the blocking version. Conversely combining it with `poll` gives a non-blocking equivalent. The other two operations on events are `wrap` which acts like a typical map primitive, and `choose` which produces a composite event that occurs once one of the sub-events can (and does) occur.

Erlang [5] is heavily used within the telecomms industry, and offers similar constructs

```
class Fibonacci extends FeedbackLoop {
  void init() {
    setDelay(2);
    setJoiner(WEIGHTED_ROUND_ROBIN(0,1));
    setBody(new Filter() {
      Channel input = new IntChannel();
      Channel output = new IntChannel();
      void work() {
        output.push(input.peek(0) + input.peek(1));
        input.pop();
      }
    });
    setSplitter(DUPLICATE);
  }

  int initPath(int index) {
    return index;
  }
}
```

Figure 2.8: StreaMIT program for computing the Fibonacci stream (taken from [111]).

```
val spawn     : (unit -> unit) -> thread_id

val choose    : 'a event list -> 'a event
val wrap      : ('a event * ('a -> 'b)) -> 'b event
val sync      : 'a event -> 'a
val poll      : 'a event -> 'a option

val channel   : unit -> 'a chan
val receive   : 'a chan -> 'a event
val transmit  : ('a chan * 'a) -> unit event

val waitUntil : time -> unit event
val timeout   : time -> unit event

val accept    : 'a chan -> 'a          = sync o receive
val send      : ('a chan * 'a) -> unit = sync o transmit
```

Figure 2.9: A Selection of Primitives in Concurrent ML.

Name	**Based On**	**Distinguishing Features**
JoCaml	OCaml	Compiled—*recent*
Polyphonic C# / Cω	C#	
Joins Library	C#	Lock-free implementation—*recent*
Boost Library	C++	
Join Java	Java	
Hardware Join Java		Matching done in hardware
JErlang	Erlang	
Funnel	Compiled to JVM	

Table 2.1: Selection of join calculus implementations.

to CML except with *asynchronous* channels. XMOS use a channel-based version of C (XC [119]) to program their multicore microcontrollers[10]. The Manticore research language [43] combines these CML primitives with the nested data parallelism from Section 2.3.3.

As stated above, many process calculi are also based around messages. *Communicating Sequential Processes* (CSP) [58] is normally seen as the ancestor of the CML and XMOS work. However, it does not support higher-order programs where channel names themselves are communicated. The π-calculus [76] does support this, along with dynamic creation of new names. Rather than describing the π-calculus, I will describe the *join calculus*, a closely related model, in Section 2.4.

Actors [56] is another formal model that has been adapted for use in programming. It is based around the idea that no state is shared between actors, which instead communicate via messages. Upon receiving a message, an actor may alter local state, create new actors, and send messages. It is not dissimilar to the join calculus introduced in the next section—however, it has no equivalent of join patterns.

2.4 The Join Calculus

The *join calculus* was introduced as a model of concurrent and distributed computation [45]. Its elegant primitives have since formed the basis of many concurrency extensions to existing languages—both functional [33, 84] and imperative [116, 9]—and also of libraries [101], as researchers have looked for paradigms that allow developers to express parallelism naturally, without introducing the intermittent *"data race"* bugs often associated with concurrency. A complete list of implementations known to the author is given in Table 2.1.

The basis of the join calculus is to declare computations as reactions to certain messages being available. By doing so, it offers a strong language for coordination and makes dependencies between computations very explicit.

The variant of the join calculus syntax that I will use is shown in Figure 2.10. This is very similar to the original used by Fournet [45], however, I use & rather than | to represent joined messages as has become more standard. In this core calculus, terms can either be *definitions*, asynchronous *emissions* of *messages* to a *channel*, or a composition $M \,\&\, N$ of other terms. This composition can be thought of as parallel. When a definition

[10] XMOS, their microcontrollers and XC can be seen as descendants of INMOS, the *transputer* and `occam` respectively.

Channels	x, y, z
Terms	$M, N = \textbf{def } D \textbf{ in } M \mid x(\bar{y}) \mid M \,\&\, N$
Definitions	$D, E = J \rhd M \mid D, E \mid \epsilon$
Patterns	$I, J = x(\bar{y}) \mid I \,\&\, J$

Figure 2.10: Syntax of the Core Join Calculus.

term is encountered, it introduces new *transition rules* of the form $J \rhd M$ into the program. Any channels defined within J are freshly instantiated, and distinct from other occurrences of the same definition. Each channel is associated with an unordered message queue. Once messages emitted to channels match the join pattern of a transition rule, the right-hand-side of the rule can be executed. The formal semantics of this are given in Section 2.4.1 below. Other work on the join calculus sometimes refers to channels as *names* and *signals*.

Many presentations and implementations of the calculus also allow for *synchronous* channels. As the name suggests, these allow calls to block awaiting a return value. Whilst very useful as a programming concept, these can easily be encoded with standard, asynchronous channels in a continuation-passing style. The transformation is straightforward and similar to that for the λ-calculus.

Another restriction commonly made is to forbid *non-linear* join-patterns. A linear pattern is one in which each channel can appear at most once. Not dealing with these tends to simplify implementations slightly, although they can be useful in implementing barriers and reductions.

2.4.1 Origins, Semantics and Related Models

Fournet's original paper on the join calculus [45] introduced it as a *reflexive* chemical abstract machine (ChAM), since the instantiation of definitions allows reaction rules to be added *to* the program *by* the program. Its semantics were therefore described in a similar way to the original ChAM, with messages meeting to form *molecules*, and then reactions occurring when those molecules matched a join pattern. These semantics operate on a *solution* $\mathcal{R} \vdash \mathcal{M}$, where $\mathcal{R}$ is a multiset of active transition rules (i.e. $J \rhd M$) in the program and $\mathcal{M}$ is a multiset of current messages and molecules (i.e. terms M). The rules of this approach are shown in Figure 2.11. The first two (*join* and *def*) are structural equivalences, while the third is a reduction step. Whilst the first rule may need to be used in both directions, there is never a need to move transition rules back from $\mathcal{R}$ to $\mathcal{M}$ with *def*. Note that only the members of the multisets involved in the rule are shown. I also do not include Fournet's *"str-and"* rule[11] since I consider the *def* rule to introduce the transition rules separately to the $\mathcal{R}$ multiset.

However, this is not the only presentation that has been offered. The original paper, and Odersky's work [85, 84], give a rewriting style of semantics. Instead of using multisets of terms and transition rules, these instead operate on the syntax of the calculus. In order to do this, a larger number of structural equivalences are defined, along with reduction contexts. These effectively emulate the properties that are gained for free from multisets. I do not make use of this style of semantics, so will not introduce them formally.

[11] With the notation of Figure 2.11, it would read: $(D, E) \vdash M \rightleftharpoons D, E \vdash M$.

$$\begin{array}{rclr} \vdash M \,\&\, N & \rightleftharpoons & \vdash M, N & \text{(join)} \\ \vdash \textbf{def } D \textbf{ in } M & \rightleftharpoons & D\sigma_{\text{dv}} \vdash M\sigma_{\text{dv}} & \text{(def)} \\ J \rhd M \vdash J\sigma_{\text{rv}} & \rightarrow & J \rhd M \vdash M\sigma_{\text{rv}} & \text{(reduce)} \end{array}$$

where σ_{dv} and σ_{rv} are substitutions such that:

σ_{dv} instantiates the defined channels of D to fresh names that do not appear elsewhere in $\mathcal{R}$.

σ_{rv} substitutes the values transmitted on the right-hand side with the named parameters in the join pattern J.

Figure 2.11: Chemical Abstract Machine Semantics for the Join Calculus.

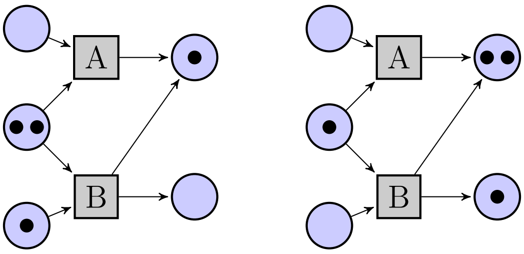


Figure 2.12: Firing Step for a Simple Petri-net.

Odersky's papers also highlight the connection between the join calculus and Petri-nets. Petri-nets are a far older model for formalising concurrent systems. Much like a join calculus transition requires all of the queues in its join pattern to contain a message, a Petri-net transition (drawn □) requires all of its *pre-places* (drawn ◯) to contain a *token* (drawn •). After the transition fires, these tokens are removed, and a token placed at each of the transition's *post-places*. Figure 2.12 shows a firing occurring of transition B in a simple Petri-net. Another example is a simple merge-sort of a 4-element list as shown in Figure 2.13 (the '2's in the diagram show *multiplicities* where multiple tokens are taken from, or given to, a single place in a firing). However, unlike the join calculus, which allows for dynamic creation of message queues, the structure of a Petri-net is entirely static. Note that the *multiplicities* used in Figure 2.13 on pre-places correspond directly to non-linear patterns in the join calculus.

More formally an (uncoloured, place-transition) Petri-net can be defined as a tuple consisting of:

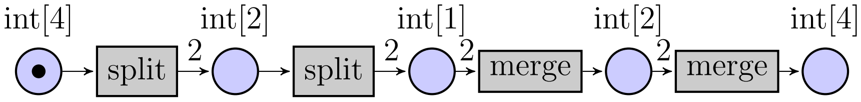


Figure 2.13: Petri-net for *merge-sort* of a 4-element list.

- A set of *places* P.

- A set of *transitions* T.

- A *pre-place* function ${}^{\bullet}\_ : T \to \mathbf{m}P$. These are the places which must contain tokens in order for a transition to fire. Note that ${}^{\bullet}\_$ gives a multiset, allowing multiplicities on arcs to be specified.

- A *post-place* function $\_^{\bullet} : T \to \mathbf{m}P$.

The state of a Petri-net is then given by a *marking* $M \in \mathbf{m}P$. These are sometimes also referred to as *token distributions.* This allows us to define the *firing rule* $M \to M'$ of the Petri-net as follows:

$$M \xrightarrow{t} M' \iff {}^{\bullet}t \leq M \text{ and } M' = M - {}^{\bullet}t + t^{\bullet}$$

Vector addition systems are an equivalent model where each vector has an element per place. Both markings and transitions can then be represented by such vectors (markings are not allowed negative values).

A standard extension to Petri-nets is *coloured* Petri-nets (CP-nets) [59] which allows tokens to take values. These associate a type, or set of colours, with each place. Tokens at a place must then be of one of these colours. Transitions can also be guarded by expressions that check the values of incoming tokens.

2.4.2 JoCaml

JoCaml [33] was the first mainstream implementation of the calculus. Prior to my own work, Le Fessant and Maranget's work [66] on it was also the only attempt to use compiler analysis and transformations on the calculus to improve performance. Their work was based on a compiler for the `join` language as well as JoCaml. This section will summarise the key techniques in both.

In both implementations, a channel can be in one of several states depending on the length of its message queue. Initially, there are just two states, $\{\texttt{0}, \texttt{N}\}$. Their approach uses a finite state machine (FSM) for each definition to determine when matching is possible. Each combination of the definition channel states corresponds to a state in this machine.

For JoCaml, the state is stored as a bit vector. If a new message arrives on a channel of state `N` then no new checks need to be made (assuming only linear patterns). If the status is changed, then a list of bit vectors corresponding to transitions is traversed to see if a match has occurred. The performance is therefore very dependent on the speed of these bit vector comparisons.

In the case of `join`, the state of a definition is maintained by a vector of function pointers for each channel in the definition. On updating the state of an instance, these are also updated. The code referred to is therefore specialised to a specific definition state, and does not need to check any channels explicitly. This makes message sends very cheap since they simply advance the FSM, and check whether the new state allows a match (which is known statically). After performing a match, the implementation must perform a few more checks to calculate the new state. More specifically, consuming a message on a channel marked `N` could either leave it in the `N` state, or move it to `0`.

To optimise the common case where only a single message is waiting in a channel's queue, they add a `1` state to the `join` compiler. The reasoning for the addition of the

`1` status is that `0` and `1` can be implemented efficiently with a single memory location, whereas `N` requires an actual queue for the channel. The `1` state also improves the post-matching overhead, since a `1` channel will always move to `0` after a message is consumed. Unfortunately as the JoCaml implementation requires channel statuses to be stored in a single bit, the `1` state is not applicable.

The `join` compiler goes further and tries to eliminate the possibility of the `N` state using a *"rudimentary name usage analyzer"* that suffices for certain cases (although the analyser itself is not described). This allows the code pointer for such channels (called *state channels*) to remain unchanged, since messages will only be sent when it lies in the `0` state. This approach is not dissimilar to the queue bounding efforts that will be made later in Chapters 4 and 5.

A final observation made is that the state space of a definition can grow very rapidly, and in an implementation where each overall state gives new code pointers for a channel's message send function, this can cause problems. For instance, in a definition that looks like:[12]

```
def create(x_0,k) ▷
  def S(x) & f_1() ▷ P_1(x),
      S(x) & f_2() ▷ P_2(x),
      ...
      S(x) & f_n() ▷ P_n(x)
  in
    S(x_0) & k(f_1, f_2, ..., f_n)
```

Since we have $O(n)$ channels, the state space grows with 2^n even though there are only $O(n)$ transitions. The cost of doing a dynamic search of transitions (as in JoCaml) is therefore more appealing than the state space explosion. They therefore suggest using a dynamic state (written as '`?`') for channels which their name usage analyser cannot track. This allows many of the states of the FSM to be collapsed.

2.4.3 The Joins Library

Claudio Russo's "Joins Library" [101] implementation of the calculus makes use of generic types and delegates in C# to offer join calculus coordination primitives. As an illustration of the library's API, a simple example of a single-item buffer is given below (taken from the second paper [113]). With this library, each definition instance is defined when it is created. `Join.Create` therefore represents creating an instance of an empty definition to which channels and transitions are then added by `Init` and `When`.

```
  class Buffer<T> {
    public readonly Asynchronous.Channel<T> Put;
    public readonly Synchronous<T>.Channel Get;
    public Buffer() {
      Join j = Join.Create();
      j.Init(out Put);
      j.Init(out Get);
```

[12] I have used syntax similar to that introduced in this section, rather than JoCaml/`join`'s actual syntax.

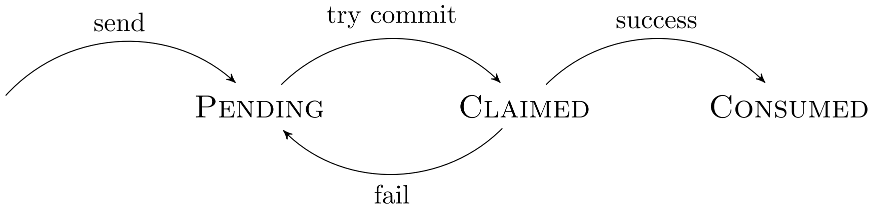


Figure 2.14: State Machine for Joins Library Message Implementation

```
    j.When(Get).And(Put).Do(t => { return t; });
  }
}
```

One side-effect of this library approach is that it allows the dynamic creation of joins. Therefore it is possible to create specific size barriers using non-linear join patterns as shown below (again taken from the original papers). This can only be supported in a compiler approach (such as my own work, or JoCaml) through the introduction of dynamic code generation.

```
class SymmetricBarrier {
  public readonly Synchronous.Channel Arrive;
  public SymmetricBarrier(int n) {
    Join j = Join.Create();
    j.Init(out Arrive);
    var pat = j.When(Arrive);
    for (int i = 1; i < n; i++) pat = pat.And(Arrive);
    pat.Do(() => { });
  }
}
```

Early versions of the library focussed on offering an easy-to-use API for programmers, and paid little attention to performance. Indeed it was implemented with coarse-grained locks on each message queue, restricting the scalability to large numbers of cores. However, more recently Aaron Turon, together with Russo, has developed a second implementation [113], based on lock-free queues, with careful consideration being given to the overheads of matching join patterns. The results that they achieved demonstrated that using the join calculus to implement primitives can match, and even exceed, the performance of more conventional approaches in some cases.

The main difficulty in implementing the join calculus is performing the atomic check and removal of messages from each message queue named in the join pattern. The original implementation attempts to overcome this difficulty with coarse-grained locking and by minimising the length of the critical sections. It is also beneficial to minimise the number of patterns that are checked. For this reason, the Joins Library implementation forms and maintains a list of all transitions affected by a given channel, unlike the JoCaml implementation which simply keeps a single list for the whole definition.

Turon's fine-grained approach uses lock-free queues (based on [72]) where each message is in one of three states: pending, claimed or consumed. The state machine between these is shown in Figure 2.14. The states are used in the following ways:

Pending. Messages are initially placed onto the queue in this state and it indicates that they are available for matching.

Claimed. When a message has been reserved for a match, it is placed in this state. However, if the runtime fails to claim messages for the other channels specified in the join pattern, then the state can be rolled back to *pending*.

Consumed. Once the content of the message has been used, the status is switched to consumed. This indicates that the message has been logically deleted, even though it still exists in the queue. By doing this, it allows a message to be 'deleted' from the queue with an atomic operation. The standard Michael-Scott queue (as is typical) only allows removal of nodes from one end, so this technique is necessary to allow messages mid-way to be deleted.

Matching against join patterns is therefore done in a *two-phase commit* manner. First the matching code examines each channel in the pattern trying to find a *pending* message. If such a message is found for each channel, then it will try to 'commit' the match by switching these messages to *claimed* using a compare-and-swap operation. This will fail for any messages that have since been matched by another worker. In this case, any claimed messages are reverted to pending. Only once the values have been extracted from all these messages are their statuses changed to *consumed*.

Given this implementation, it is clear that the unordered nature of join calculus message queues offers significant performance benefits. Were the queues to be ordered, then any rollback of a *claimed* message would force any matches using subsequent messages to also be rolled back.

To ensure that no matches are missed, it is necessary to retry matching if, when searching for a pending message, a claimed message is seen. This is because the claimed message could later be reverted to pending if the competing match fails to complete. A full proof of correctness of this approach is given in Turon and Russo's paper. However, the general idea is that the addition of a message to a queue has a linearisation point. That is to say that there is a notion of a global ordering of messages in the system. By retrying when a claimed message is seen, we ensure that if a match is possible using the message and messages that arrived before it, then the matching procedure will either succeed or terminate due to the message being consumed by another worker.

Their evaluation of this work demonstrates that despite the effort put into shortening the critical sections of the original implementation, the fine-grained approach almost always performs better. Furthermore, the performance of concurrency primitives implemented via the join calculus and their new implementation is competitive with, and sometimes even beats, the standard implementations found in the .NET libraries. Whilst unexpected, since one would expect the .NET library to be highly optimised and it can utilise exactly the same primitives that underlie the Joins Library, it does offer encouragement to this thesis' overall aim—i.e. that a universal simplifying representation need not introduce prohibitive overheads.

This implementation technique can also support non-linear patterns relatively easily. When traversing the message queue looking for pending messages, one simply keeps going until the correct number have been found. In this case, matches must be retried if the sum of pending and claimed messages seen during this traversal is equal to, or greater than, the number of messages required from the channel for the join pattern.

$$\frac{}{\rho \vdash x \to \rho(x)}\ (\textsc{var}) \qquad \frac{}{\rho \vdash c \to c}\ (\textsc{cons}) \qquad \frac{}{\rho \vdash \lambda x.e \to (\rho, \lambda x.e)}\ (\textsc{lam})$$

$$\frac{\rho(x_1) = (\rho', \lambda x.e) \qquad \rho'[x \mapsto \rho(x_2)] \vdash e \to v}{\rho \vdash x_1 x_2 \to v}\ (\textsc{app})$$

$$\frac{\rho \vdash e_1 \to v_1 \qquad \rho[x \mapsto v_1] \vdash e_2 \to v_2}{\rho \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \to v_2}\ (\textsc{let})$$

Figure 2.15: Semantics for the Flattened λ-Calculus.

2.5 Control-Flow Analysis

Control-flow analysis (CFA) has been a central technique for implementing higher-order languages. Without it, it is impossible to perform optimisations such as inlining except in trivial (i.e. first-order) cases. As is shown in Chapter 4, this is also the case for some optimisations of the join calculus. Whilst ignored by the Joins Library implementation, JoCaml did explore the use of simple analysis and it is this that I build on in this thesis.

In the literature, there are two main styles of control-flow analysis: the constraint-based approach originally developed by Heintze [55] and popularised by the Nielsons [82]; and the abstract interpretation method by Shivers [105] and more recently revisited by Might and others (e.g. [74, 73, 75]). This section will explore these two techniques, before talking about *call-strings* as a way of abstracting program histories and improving accuracy in the λ-calculus. Finally, I look at existing work on using *escape-based* techniques to analyse parts of programs. This happens to have been adapted to Concurrent ML [93].

2.5.1 Constraint-based

The style and notation of constraint-based analysis presented here, and used throughout Chapter 5, is based on that presented by Faxén [38]. Here I introduce his *polymorphic* technique in the context of the *flattened* λ-calculus with the semantics of Figure 2.15. This forbids anonymous sub-expressions in function applications.

$$e ::= x \mid c \mid x_1\, x_2 \mid \lambda x.e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$$

Each program expression and variable is associated with a *flow variable* α. The analysis will then first construct constraints over these flow variables, and *flow values*—i.e. constants and *flow closures* (also referred to as *type schemes* in Faxén's work). It then needs to solve these constraints by assigning a set of flow values to each flow variable. This assignment Φ is called a *model of the constraint set.* The complete analysis is shown in Figure 2.16.

The constraints are generated using inference rules for a judgement $S, \hat{\rho} \vdash e : \alpha$. The *flow environment* $\hat{\rho}$ is an abstract version of the environment ρ in Figure 2.15, mapping program variables to flow variables. In this judgement, S is a constraint set and α a flow variable corresponding to the expression e. Constraint sets and flow environments also appear in flow closures.

Constraint Syntax:

$$\begin{aligned} S \subseteq \text{Constraint} &::= \alpha_1 \supseteq \alpha_2 \mid \alpha \supseteq \{c\} \mid \alpha \supseteq \{w\} \mid \alpha_1 \mapsto \alpha_2 \supseteq \alpha_3 \\ w \in \text{FlowClosure} &::= (S, \rho \mid \lambda x.e : \alpha_1 \mapsto \alpha_2) \end{aligned}$$

Constraint Generation Rules: (with judgement form $S, \hat{\rho} \vdash e : \alpha$)

$$\frac{}{\{\alpha \supseteq \hat{\rho}(x)\}, \hat{\rho} \vdash x : \alpha}\ (\textsc{var}) \qquad \frac{}{\{\alpha \supseteq c\}, \hat{\rho} \vdash c : \alpha}\ (\textsc{cons})$$

$$\frac{S, \hat{\rho}[x \mapsto \alpha_1] \vdash e : \alpha_2}{\{\alpha \supseteq (S, \hat{\rho} \mid \lambda x.e : \alpha_1 \mapsto \alpha_2)\}, \hat{\rho} \vdash \lambda x.e : \alpha}\ (\textsc{lam})$$

$$\frac{}{\{\hat{\rho}(x_2) \mapsto \alpha \supseteq \hat{\rho}(x_1)\}, \hat{\rho} \vdash x_1 x_2 : \alpha}\ (\textsc{app})$$

$$\frac{S_1, \hat{\rho} \vdash e_1 : \alpha_1 \qquad S_2, \hat{\rho}[x \mapsto \alpha_1] \vdash e_2 : \alpha_2}{S_1 \cup S_2, \hat{\rho} \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \alpha_2}\ (\textsc{let})$$

Model of Constraints: ($\Phi \models S$ iff $\Phi \models s$ for all $s \in S$)

$$\begin{aligned} \Phi \models \alpha_1 \supseteq \alpha_2 &\iff \Phi(\alpha_1) \supseteq \Phi(\alpha_2) \\ \Phi \models \alpha \supseteq \{w\} &\iff w \in \Phi(\alpha) \\ \Phi \models \alpha \supseteq \{c\} &\iff c \in \Phi(\alpha) \\ \Phi \models \alpha_1 \mapsto \alpha_2 &\iff w \in \Phi(\alpha) \implies \Phi \models \mathcal{I}(w, \alpha_1 \mapsto \alpha_2) \end{aligned}$$

Closure of Constraint Sets: $S^+ \supseteq S$

$$\begin{aligned} \{\alpha_1 \supseteq \alpha_2, \alpha_2 \supseteq \{w\}\} \subseteq S^+ &\implies \{\alpha_1 \supseteq \{w\}\} \subseteq S^+ \\ \{\alpha_1 \supseteq \alpha_2, \alpha_2 \supseteq \{c\}\} \subseteq S^+ &\implies \{\alpha_1 \supseteq \{c\}\} \subseteq S^+ \\ \{\alpha_1 \mapsto \alpha_2 \supseteq \alpha_3, \alpha_3 \supseteq \{w\}\} \subseteq S^+ &\implies \mathcal{I}(w, \alpha_1 \mapsto \alpha_2) \subseteq S^+ \end{aligned}$$

Instantiation of Flow Closures:

$$\mathcal{I}((S, \hat{\rho} \mid e : \alpha_1' \mapsto \alpha_2'), \alpha_1 \mapsto \alpha_2) = \exists \alpha_1'', \alpha_2''. S[\alpha_1''/\alpha_1', \alpha_2''/\alpha_2'] \cup \{\alpha_1'' \supseteq \alpha_1, \alpha_2 \supseteq \alpha_2''\}$$

Figure 2.16: Faxén's Polymorphic Control-Flow Analysis for the λ-calculus.

There are four forms of constraint. The first two are straightforward. The syntax $\alpha_1 \supseteq \alpha_2$ is used to indicate that all values represented by α_2 could flow to α_1, while $\alpha \supseteq \{c\}$ indicates that c flows to α.

When encountering a λ-abstraction $\lambda x.e$, constraints are generated for the body e. However, to allow analyses more precise than a simple monovariant approach, these are saved in the flow closure $w = (S, \hat{\rho} \mid \lambda x.e : \alpha_1 \mapsto \alpha_2)$. A flow closure states the the constraints S describe the function $\lambda x.e$ when it is passed an argument from α_1 in an environment $\hat{\rho}$, with the result represented by α_2. This allows the constraints to be instantiated using $\mathcal{I}$ for different call-sites by performing substitutions on α_1 and α_2.

Finally, there are *application constraints* $\alpha_1 \mapsto \alpha_2 \supseteq \alpha_3$. These represent a call-site making use of a function from α_3, with argument from α_1, and with the result flowing to α_2. The model of this constraint instantiates the constraints in each flow closure represented by α_3, substituting possibly fresh flow variables for the argument and result. Note that the specification of this analysis is independent of the level of precision—this is discussed further in Section 2.5.3. Not performing the substitution at all results in a monovariant, or zeroth-order (0-CFA), analysis.

Forming a solution Φ is done via a closure operation (again in Figure 2.16) on the constraints. This follows naturally from the specification of a model for each constraint. The final solution can be read off from the $\alpha \supseteq \{w\}$ and $\alpha \supseteq \{c\}$ constraints in the closed set.

2.5.2 Abstract Interpretation

In constraint methods, we first describe the *flow* of data through the program using constraints, and then solve for these. Abstract interpretation techniques tend to merge the two steps, constructing a mathematical formula representing the solution as it walks through the program. This section introduces abstract-interpretation-based CFA for the pure continuation-passing λ-calculus. The style used follows the generic technique presented by Might for abstracting arbitrary semantics mechanically [73]. The syntax and semantics of the calculus he uses are shown in Figure 2.17.

The obvious first step in attempting to produce a CFA for this language is to replace each domain with an abstract version (or as Might puts it "throw hats on everything"). We would also expect a CFA to deal with sets of values rather than single ones, so we try an abstract domain of $\widehat{\mathrm{Env}} = \mathrm{Var} \rightharpoonup \mathcal{P}(\widehat{\mathrm{Clo}})$. However, this results in an infinite abstract domain since a closure could contain an environment which in turn contains the closure in an infinite cycle—i.e.

$$\hat{c}_\infty = (\lambda x.\ \ldots, [x \mapsto \{\hat{c}_\infty\}])$$

To ensure the abstraction remains finite, we need a way to ensure that at some point closures are reused.[13] This can be done by "snipping" (sic), the link from Env to Clo and introducing a store. This change to the semantics of Figure 2.17 is shown in Figure 2.18 (figure taken from [73]). Now the environment associates variables with *abstract addresses* $\widehat{\mathrm{Addr}}$, which in turn are associated with a set of closures by the store that is part of the abstract state $\hat{\Sigma}$. This results in the abstract analysis shown in Figure 2.19.

In this analysis, note that $\leadsto$ offers multiple successor states. It is therefore necessary to consider all reachable states and then take results from all resultant stores. Since each

[13] Might points out an alternative would be to reuse environments, but this is not an approach which has been used for the λ-calculus.

Syntax:

$$\begin{aligned} f, e \in \text{Exp} &= \text{Var} \cup \text{Lam} \\ \mathit{lam} \in \text{Lam} &::= \texttt{(}\lambda \ \texttt{(}x_1 \ldots x_n\texttt{)} \ \mathit{call}\texttt{)} \\ x \in \text{Var} &= \text{a set of identifiers} \\ \mathit{call} \in \text{Call} &::= \texttt{(}f \ \ e_1 \ldots e_n\texttt{)} \end{aligned}$$

State-Space:

$$\begin{aligned} \Sigma &= \text{Call} \times \text{Env} \\ \rho \in \text{Env} &= \text{Var} \rightharpoonup \text{Clo} \\ \text{Clo} &= \text{Lam} \times \text{Env} \end{aligned}$$

Semantics:

$$\llbracket \texttt{(}f \ \ e_1 \ldots e_n\texttt{)} \rrbracket, \rho \longrightarrow \mathit{call}, \rho'[x_i \mapsto \varepsilon(e_i, \rho)]$$

where:

$$(\llbracket \texttt{(}\lambda \ \texttt{(}x_1 \ldots x_n\texttt{)} \ \mathit{call}\texttt{)} \rrbracket, \rho') = \varepsilon(f, \rho)$$

and the argument evaluator $\varepsilon : \text{Exp} \times \text{Env} \rightharpoonup \text{Clo}$ is defined as:

$$\varepsilon(e, \rho) = \begin{cases} \rho(e) & \text{if } e \in \text{Var} \\ (e, \rho) & \text{if } e \in \text{Lam} \end{cases}$$

Figure 2.17: Pure Continuation-Passing λ-Calculus.

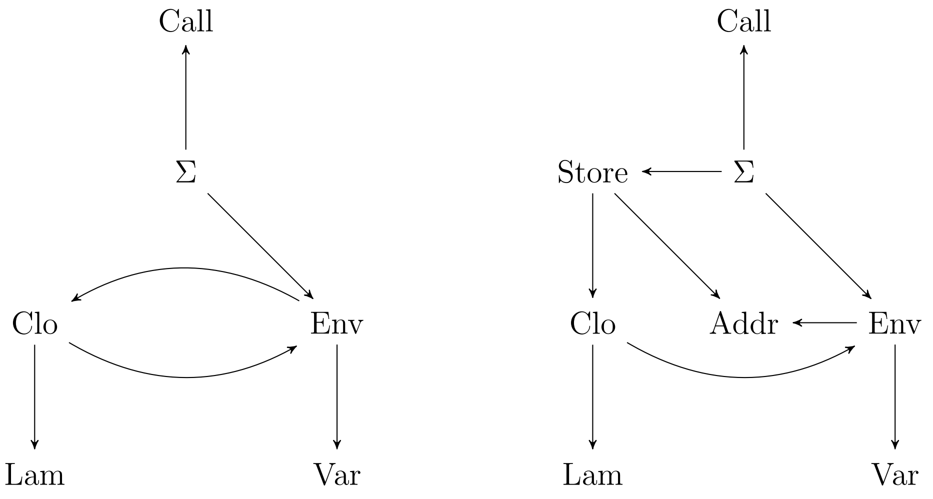


Figure 2.18: Removing cycles in the CPS λ-calculus state-space (taken from [73]).

Abstract State-Space:

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathrm{Call} \times \widehat{\mathrm{Env}} \times \widehat{\mathrm{Store}} \\
\hat{\rho} \in \widehat{\mathrm{Env}} &= \mathrm{Var} \rightharpoonup \widehat{\mathrm{Addr}} \\
\hat{\sigma} \in \widehat{\mathrm{Store}} &= \widehat{\mathrm{Addr}} \rightharpoonup \mathcal{P}(\widehat{\mathrm{Clo}}) \\
\mathrm{Clo} &= \mathrm{Lam} \times \widehat{\mathrm{Env}} \\
\hat{a} \in \widehat{\mathrm{Addr}} &= \text{a finite set of addresses}
\end{aligned}$$

Abstract Semantics:

$$\overbrace{[\![\texttt{(}f\ \ e_1 \ldots e_n\texttt{)}]\!], \hat{\rho}, \hat{\sigma}}^{\varsigma} \rightsquigarrow \mathit{call}, \hat{\rho}'[v_i \mapsto \hat{a}_i], \hat{\sigma} \sqcup [\hat{\sigma}_i \mapsto \hat{\varepsilon}(e_i, \hat{\rho}, \hat{\sigma})]$$

where:

$$\begin{aligned}
([\![\texttt{(}\lambda\ \ \texttt{(}v_1 \ldots v_n\texttt{)}\ \ \mathit{call}\texttt{)}]\!], \hat{\rho}') &= \hat{\varepsilon}(f, \hat{\rho}, \hat{\sigma}) \\
\hat{a}_i &= \widehat{\mathrm{alloc}}(v_i, \varsigma)
\end{aligned}$$

and the abstract argument evaluator $\hat{\varepsilon} : \mathrm{Exp} \times \widehat{\mathrm{Env}} \times \widehat{\mathrm{Store}} \rightharpoonup \mathcal{P}(\widehat{\mathrm{Clo}})$ is:

$$\hat{\varepsilon}(e, \hat{\rho}, \hat{\sigma}) = \begin{cases} \hat{\sigma}(\hat{\rho}(e)) & \text{if } e \in \mathrm{Var} \\ \{(e, \hat{\rho})\} & \text{if } e \in \mathrm{Lam} \end{cases}$$

Figure 2.19: CFA for CPS λ-Calculus using Abstract Interpretation.

step is monotonic with respect to $\hat{\sigma}$, it is possible to use a single store for this exploration [105].

As yet the abstract allocation function $\widehat{\text{alloc}} : \text{Var} \times \hat{\Sigma} \to \widehat{\text{Addr}}$ has not been defined. The exact definition of this will alter the accuracy of the analysis. For instance, we can recover a zeroth-order analysis (0-CFA) as follows:

$$\begin{aligned} \widehat{\text{Addr}} &= \text{Var} \\ \widehat{\text{alloc}}(v, \varsigma) &= v \end{aligned}$$

I will discuss other allocation functions in Section 2.5.3.

Comparing this to the constraint-based approach introduced above, it is clear that the abstract addresses ($\widehat{\text{Addr}}$) serve a similar purpose to flow variables, in that it is the extent to which the size of this domain is restricted that affects the precision. The store that is added for abstract interpretation also provides a level of indirection that is similar in spirit to that provided by the constraints themselves.

2.5.3 Call Strings

The analyses in Figures 2.16 and 2.19 have been presented in a manner independent of precision. In the constraint-based approach, the precision depends on the exact implementation when choosing substitutions for the argument and return flow variables of a function prior to flow closure instantiation. For abstract interpretation, it depends on the $\widehat{\text{alloc}}$ function definition. Whilst there are other approaches, the most common approach is to implement these choices using *call-strings*.

In the concrete semantics, the current point in the program is accurately described by the *call-stack*. This is especially true in continuation-passing style since call-sites are never removed from the call-stack.[14] It is therefore reasonable to attempt to improve accuracy of 0-CFA by making use of an abstract version of the call-stack. As a sequence, the call-stack is also particularly easy to abstract. We form a kth order CFA (traditionally called k-CFA) simply by considering the most recent k items on the call-stack as our call-string.

To support call-strings in Figure 2.19, we can add a *time* component to the abstract state $\hat{\Sigma}$. Each transition then maintains this time as the k most recent call-sites. More specifically:

$$\begin{aligned} \widehat{\text{Time}} &= \text{Call}^k \\ \widehat{\text{Addr}} &= \text{Var} \times \widehat{\text{Time}} \\ \hat{\Sigma} &= \text{Call} \times \widehat{\text{Env}} \times \widehat{\text{Store}} \times \widehat{\text{Time}} \\ &\text{etc.} \end{aligned}$$

For the constraint based approach, the $\exists$ operator can be viewed as mimicking a call to $\widehat{\text{alloc}}$. It will therefore pick the flow variables based on the k most recent call sites. Note that since the constraint-based analysis above works on a λ-calculus with only a single argument per function (unlike the CPS version used for the abstract interpretation approach), the Var argument to $\widehat{\text{alloc}}$ is not applicable.

[14] Of course, CPS languages are not actually implemented with a call-stack unless specific techniques are employed to avoid stack overflow.

The join calculus does not offer a ready alternative to call-stacks. An alternative approach is discussed in Section 5.3.

2.5.4 Escape-based Techniques

One attempt to analyse the communication patterns of Concurrent ML [93] is based on Reppy's *type-sensitive* improvements [92] of Serrano's 0-CFA [103]. The key difference between this approach and those already described is that it is modular—i.e. can analyse *part* of a program (for example, an ML module). It does this by introducing an *unknown* abstract value $\top$, alongside finite sets of (flow) closures. A call to an unknown value could correspond to a call of any function that *escapes*. Reppy describes the notion of escaping particularly well:

> A function is said to *escape* if it can be called at unknown [call-]sites. There are several ways that a function might escape: it may be exported by the module, it may be passed as an argument to an unknown function, or it may be returned as the result of an escaping function. If a function f escapes, then [the solution] $\mathcal{A}$ [of the analysis] must be defined to map f's parameters to $\top$, since we have no way of approximating them.

In using this technique for CML, Reppy uses type information to partition the set of escaping values based on their type. This is particularly useful for abstract types where we know that only values which escape could be passed back as arguments (since code external to the module has no way of generating new values of such types).

2.6 Summary

This chapter has introduced the wide range of previous work that is required to understand the context of this thesis.

Section 2.1 explained why modern hardware has been forced to become parallel rather than offering the more conventional speed ups. It also served to describe the types of hardware that an architecture-neutral representation of parallel needs to target.

Section 2.2 described the current state of compiler intermediate representations and architecture-neutral formats. This makes it clear that none of the mainstream VMs provide adequate support for parallelism, and that existing research tends to be focussed on a single source-language.

Section 2.3 went through the paradigms currently on offer to software developers. My work has deliberately steered clear of the crowded field of parallel programming language design, however, this gives an overview of the primitives that should be encodable in a parallel intermediate representation.

The final two sections are less general and provide the prerequisite background for the specific approach taken in this thesis.

Section 2.4 introduced the **Join Calculus**, along with its concepts, semantics, origins and current implementations. This model forms the basis of the intermediate representation introduced in this thesis.

Section 2.5 gave two styles of approach to doing control-flow analysis on the λ-calculus as a way of introduction. CFA will be formulated for the join calculus in Chapter 5.

The remainder of this dissertation discusses how to use these to address the shortcomings of existing IRs, and provide a common substrate for parallelism and concurrency.

Being abstract is something profoundly different from being vague... The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

— *E. W. Dijkstra*

Chapter 3

The Join Calculus Abstract Machine

Chapter 2 observed that mainstream compiler representations are all based around the general principle of control-flow graphs—fundamentally a sequential concept. They almost universally then retrofit parallelism through some form of threading (be they raw kernel ones, lightweight user threads, or some form of data-parallel construct as in CUDA and OpenCL) and rely on shared memory for coordination. However, threads are very much a hardware implementation of parallelism rather than a core principle. Whilst other primitives can of course be built on top of this foundation, doing so hides the true nature of the computation and hinders any attempts made by the compiler to adapt to parallel architectures. In much the same way, the operand stack in .NET and the JVM, or virtual registers in LLVM, provide a simple model that can be implemented by either memory or hardware registers. None of these expose hardware registers and, in the case of .NET and the JVM, even the view of memory is very restricted.

From the survey of existing languages and architectures in Chapter 2, three features stand out as being essential to a modern parallel intermediate representation: explicit fine-grained parallelism with flexible coordination, choice, and support for non-uniform memory models. This chapter starts (Sections 3.1 and 3.2) by demonstrating how key features of existing models that fare well in each of these areas can be merged, resulting first in Petri-nets and then in the join calculus. In Section 3.3, I condense the key operations of the calculus into three main primitives that allow an abstract machine to be defined. The remainder of the chapter then offers examples of this abstract machine's usage to confirm its universality, both allowing compilation to a variety of architectures and also being language-neutral.

3.1 Arriving at the Join Calculus

With the exception of the dataflow architectures explored in the 1970s (Section 2.1.7), all parallel hardware is built as an extension to a sequential base. It is therefore reasonable to use control-flow graphs (CFGs) as the starting point for the development of my model.

3.1.1 Supporting Non-Deterministic Choice

Whilst explicit non-deterministic choice is relatively rare in real-world programming languages, it is in fact a common interpretation of certain features in CFGs. This is most commonly encountered in static analysis, where conditional branching is often treated as

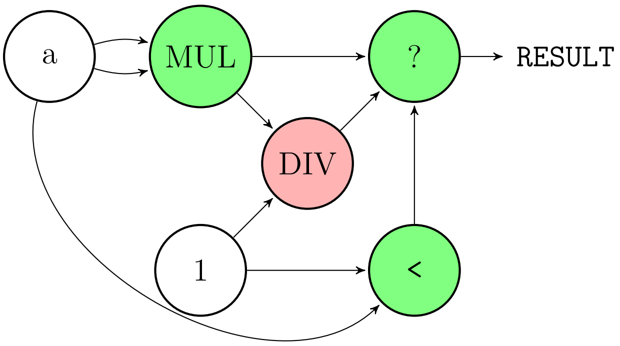


Figure 3.1: Dataflow Graph for `(a < 1) ? (1 / (a * a)) : (a * a)`. Green highlighting indicates nodes executed for `a` = 3.

such. Indeed, any fork[1] in a standard CFG represents a choice—even if it is deterministic under the usual language semantics. I start by allowing forks that are actually non-deterministic. This allows implementation choices that may depend on the final target to be expressed in the IR and left for a load-time or runtime scheduler to decide.

At this point it is worth clarifying the type of choice that I wish to support. Systems such as StarPU [8] (Section 2.3.1.3) allow multiple implementations of program parts, allowing the runtime to pick whichever it expects to execute most quickly. For programs to have *portable* performance, choices such as this must be made automatically and separately from compilation. This is an example of *performance non-determinism*, where run-time or load-time decisions affect the overall execution time. This can be distinguished from *I/O non-determinism*, where run-time decisions might alter the *result* of the program.

At the IR level, I choose not to explicitly forbid I/O non-determinism, with its behaviour simply treated as undefined. This is in much the same spirit that it is possible to attempt dereferencing null references or invalid pointers in other representations. Compiler transformations are therefore able to assume that all choices lead to the same result being calculated—just as a C compiler assumes non-null pointers for the purposes of optimisation (since the result of a null-dereference is undefined anyway).

Until parallelism is added, the expressive power of arbitrary non-deterministic forks is very similar to the StarPU approach of allowing multiple implementations of each CFG node. However, StarPU is unable to encode scenarios such as *either perform operation A once or perform operation B ten times (possibly in parallel) each*, since every task is a single unit that must be completed. I wish, and will be able, to encode these possibilities which are important when the amount of parallelism available is unknown.

3.1.2 Exposing Fine-Grained Parallelism

In direct contrast to control-flow graphs, dataflow graphs allow very explicit parallelism, but no branching or non-determinism. In a naïve implementation, every computation node in the graph will be performed at some point, and the edges give the dependencies between them. In a pure side-effect-free language, conditionals can still be expressed (for example, Figure 3.1), and demand-driven execution used to minimise excess computation.

[1] This ignores extensions to CFGs that allow parallel forks or spawns.

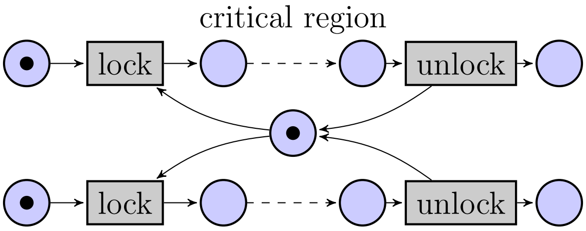


Figure 3.2: Petri-net for a critical region using a *mutual exclusion lock*.

However, this dataflow model does not meet the requirements of a general-purpose IR, which needs to support state (or at least a natural encoding of it). Furthermore, I have already stated non-determinism as one of my key requirements.

In many ways, control-flow and dataflow graphs can be seen as duals of each other—in a CFG, the data implicitly travels between the computations as controlled by the program counter, whereas in a DFG it is the control-flow that is absent. One can make the data in a control-flow graph more explicit by imagining that each edge of the graph consists of a single-item buffer, even though in reality data is passed using registers or memory. Similarly, in a dataflow graph one can imagine that each node consists first of a computation step, that waits for data to be available at all predecessors before computing the result, followed by a *persistent* single-item buffer into which the result is placed. The need for this buffer to be persistent stems from the fact a result can be used multiple times in a dataflow graph. It is possible to remove the need for this by incorporating some form of duplication primitive—these would come 'for-free' if nodes producing multiple results were allowed.

Combining these two models results in a form of (coloured) Petri-net [59] (as introduced in Section 2.4.1) supporting both parallelism and non-determinism. Recall that execution proceeds by the firing of *transitions*. When a transition *fires*, it removes tokens from its *pre-places*, applies a function to them, and adds the results to its *post-places*.

The atomic nature of token consumption from pre-places allows Petri-nets to express concurrency primitives well—for example, mutual exclusion (Figure 3.2). This example makes use of a cycle within the net which is a common pattern in Petri-nets for encoding state. The beauty of this pattern is that it allows the model to express imperative, mutable-state programming without the need for any extra primitives, such as references. It is not dissimilar to threading state through a program as might be done with monads, however, Petri-nets allow different components of state to be brought in as and when they are needed rather than being threaded through every step.

Attempting to merge control-flow and data-flow graph features is not a new idea. The VSDG [60] (mentioned in Section 2.2.2) takes a similar approach but maintains a distinction between the two types of edges. Petri-nets can therefore be viewed as a generalisation of the VSDG.

The computation that a transition performs could be defined using most existing representations for sequential programs (e.g. functions in LLVM's IR [1]). However, as I will only deal with pure transitions,[2] transition bodies may not access or modify any global state. Since Petri-nets are not my final model I will not polish this definition too much. However, a flavour for Petri-net semantics, including the *firing rule*, was given in Section 2.4.1.

[2] As previously noted, in this work all state is encoded through cycles in the net.

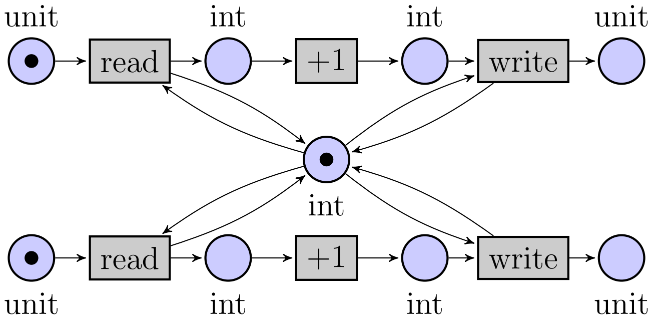


Figure 3.3: Petri-net without confluence ($x := x + 1 \parallel x := x + 1$).

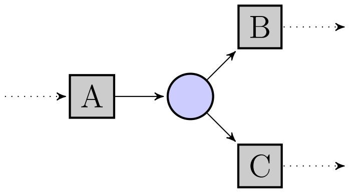


Figure 3.4: Non-deterministic Choice between Alternatives in Petri-nets

The firing of Petri-nets is non-deterministic, as intended. However, this allows all forms of non-determinism to be expressed, not just performance non-determinism (e.g. Figure 3.3). The assumption that I stated earlier (Section 3.1.1) of I/O determinism can be expressed by assuming *confluent* nets—i.e. for all markings $M_1, M_2, M_3 \in \mathbb{M}$:

$$(M_1 \rightarrow^* M_2) \wedge (M_1 \rightarrow^* M_3) \implies \exists M_4 \in \mathbb{M}.((M_2 \rightarrow^* M_4) \wedge (M_3 \rightarrow^* M_4))$$

However, while supporting non-deterministic choice between alternatives (Figure 3.4), the simple definition of Petri-nets does not support deterministic conditional branching. Jensen's work on coloured Petri-nets [59] addresses this by restricting the domain of values accepted by a transition with *guards*. Although this was the approach that I followed in my earlier work [22], this issue can be addressed more elegantly as shown in the following section.

3.1.3 Reintroducing Functions and Dynamic Behaviour

Moving to Petri-nets enables support for parallelism and non-determinism in a natural way, but it does force some more traditional features of programming to be reconsidered. I have already mentioned that conditional branching becomes more difficult to support. However, this alone would be easy to work around. Of more concern is the difficulty in allowing abstraction and dynamic allocation.

In both dataflow and control-flow graphs, functions are easy to support and allow reuse of program parts. With control-flow graphs, functions are defined in terms of control-flow. In particular, execution transfers from the call-site to the function, before returning when the function completes. This can be viewed as allowing a node of the CFG to be defined as referencing another graph (Figure 3.5). Inlining therefore simply replaces the node with the referenced graph, and trivially offers identical behaviour in a sequential context. We can apply a similar idea to dataflow graphs, allowing a node of the graph to be defined

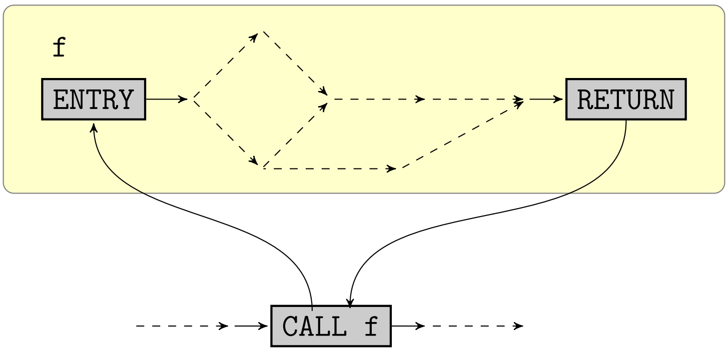


Figure 3.5: Functions as sub-graphs in a control-flow graph.

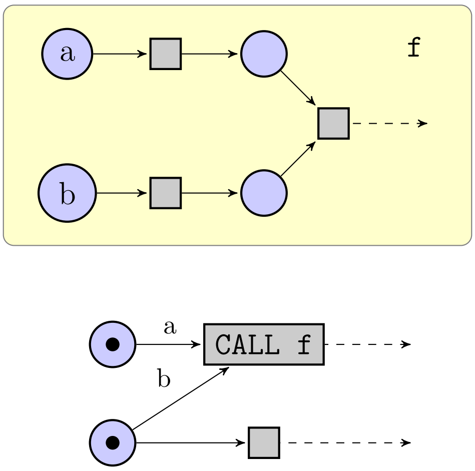


Figure 3.6: A Petri-net sub-graph.

```
def @memcell(i,k) ▷
  def get(k)   & val(x) ▷ val(x) & k(x),
      set(x,k) & val(y) ▷ val(x) & k()
  in val(i) & k(get,set)
```

Figure 3.7: Memory cell encoding in the join calculus.

by a separate dataflow graph. Even though dataflow graphs allow parallelism, inlining still preserves behaviour since there is no mutable state.

However, if we apply a similar approach to Petri-nets, then we run into problems when attempting to represent functions. Consider the example in Figure 3.6: before inlining, the subgraph would be expected to execute atomically (i.e. it always consumes both tokens or neither). If it did not, then any use of a subgraph would need to understand at what point within that graph tokens would be taken from places. Once inlined this restriction does not apply (i.e. it might consume just one token, possibly resulting in deadlock if there are two such functions). As a programming model, this therefore fails to provide satisfactory abstraction. Furthermore, programmers expect to have access to some variety of indirect function call. This is easy to support in control-flow graphs where each function has a single predecessor, but it is unclear how this adapts to the dataflow and Petri-net cases where there may be multiple predecessors to a function use. In the case of the VSDG mentioned earlier, this is not a problem as the model has only a single *control-flow token*.

It is worth pointing out that such a notion of substitution in Petri-nets is not without use. When nets are instead being used to model hardware, or interacting systems, then hierarchical nets [59] provide an effective way of composing separate components, and instantiating the same sub-net (or *page*) multiple times.

The second problem is that by encoding mutable state in the Petri-net structure itself, one needs to be able to dynamically enlarge the net at runtime in order to support dynamic allocation of state. Again, the straightforward subgraph notion of functions does not allow this.

Whilst the join calculus was not originally introduced as an extension of the Petri-net model, the similarities between the two models have been discussed previously by Odersky [84]. Just as a Petri-net transition has a fixed multi-set of *pre-places*, each transition in the join calculus has a fixed *join pattern* defining its input channels. The key difference is that the join calculus is higher-order, allowing channels to be passed as values, and for the output channels to depend on its inputs—unlike Petri-nets where the *post-places* of a transition are fixed. This simple modification allows use of continuations to support functions. Moreover, while nets are static at runtime, a join calculus program can create new instances of definitions (containing channels and transition rules) at runtime. Although these cannot match on existing channels, existing transitions can send messages to the new channels. This allows dynamic state through definitions such as that in Figure 3.7. Note however that the join calculus does not introduce dynamically changing pre-places, and this key restriction allows compilation without unmanageable overheads. Referring back to the dataflow architectures mentioned in Section 2.1.7, it is worth observing that the join calculus is not dissimilar to the concept of *dynamic dataflow machines*. In such a comparison, Petri-nets are rather like *static* dataflow.

As mentioned in Section 2.4, much as the λ-calculus can be defined in both continuation-passing (CPS) and direct styles, so can the join calculus. CPS is advantageous for my use case—indeed it is commonly used in compiler representations for functional languages [4]. This is because it makes all control-flow explicit. It also reduces the number of primitives that need to be handled by a compiler. During his work on control-flow analysis of the λ-calculus [74], Might notes that in functional languages, λ-abstraction "represents, in one construct, the fundamental data, control, and environment structure of these languages", and that "by using CPS, our *principal* concern—function call—becomes our *only* concern". In the CPS form of the join calculus, channels become the only concern, with the benefit that they also represent coordination in a parallel setting.

3.1.4 Granularity

As with most compiler problems, parallelism can be considered at a number of different levels, and the join calculus could arguably be applied at each of these:

Instruction Level Mostly dealt with by hardware, although compiler instruction scheduling does have an effect. I could arguably use the join calculus as the basis of new hardware architectures in a similar vein to the dataflow machines of the 1970s. However, it is doubtful that the high-level of overheads would ever be overcome, and the large amount of research and development that has led to modern superscalar processors means that pragmatically any other designs will struggle to catch up.

Blocks[3] As fine-grained implementations have improved, it is not uncommon for languages to allow code blocks to be spawned. Indeed, this is quite common in the case of parallel 'for' loops (e.g. in OpenMP). This is the level at which auto-parallelisation techniques have attempted to operate. The join calculus, especially in a continuation-passing style, is particularly well suited to expressing this—assuming an efficient implementation.

Functions This is the level that Wool and Cilk (Section 2.3.1.2) have demonstrated as being feasible. From a compiler perspective this is not dissimilar to basic blocks, although Wool in particular shows that this level can be implemented without the compiler removing, or 'inlining', excess parallelism (i.e. without *coarsening* the granularity).

Large Scale Distributed computing was one of the original stated goals of the join calculus [45], although it is perhaps better introduced in a later tutorial [46]. This space has also seen many more models tried (e.g. map-reduce and CIEL—see Section 2.3.5) than for compiler IRs. I believe this is because less concern needs to be paid to overheads. Whilst distributed scenarios introduce extra problems that are beyond the scope of this thesis (such as fault tolerance), any model that can expose finer parallelism is arguably a good basis for future work in large scale situations. Indeed, there are many common features between the dynamic data-flow graphs of CIEL, and the join calculus.

[3] Code blocks and basic blocks do not correspond directly (especially in CPS), but are similar enough to not be distinguished in this discussion.

Years of research indicate that discovering parallelism at a finer granularity than originally expressed in a program is incredibly difficult. Any representation should therefore operate at the finest level at which one might wish to express parallelism—i.e. basic blocks. As stated above, other recent work (e.g. Wool [39, 40]) has shown that even without compiler optimisations, careful implementation of function granularity primitives can give good performance. Whilst not within the scope of this thesis, I also believe that the results of the analysis in Chapter 5 could be used to coarsen the level of parallelism when appropriate, and this is discussed in Section 6.6.3.

It is true that the shortcoming of dataflow machines was that the instruction-level dataflow was too fine-grained. However, even a basic block is significantly more coarse than an instruction. Using the join calculus at this level is similar in spirit to coordination languages. Techniques described in Chapter 4 will ensure that coordination overheads are minimised in common cases. The approach of my abstract machine is therefore to compile everything to join calculus definitions, rather than maintaining support for standard functions in the representation. Chapters 4 and 5 show how this can be implemented without incurring prohibitive overheads.

3.2 Making Data Dependencies Explicit

One of the stated requirements for my representation is support for non-uniform memory architectures (NUMA). The most established programming model for such systems is message passing (e.g. MPI and Erlang). The basic premise of this model is that all data must be explicitly received, with no data being implicitly shared. Although the join calculus is already a message-passing approach of similar granularity to Erlang,[4] its standard formulation does not satisfy this requirement, since the nested nature of definitions allows values to be captured in free variables. For example, observe that `N` and `k` are both implicitly used by the program[5] in Figure 3.8. Allowing this in the final IR would mean that some data transfers have to be inserted automatically. Whilst this may be possible, it introduces costs that are hidden from any scheduling system.

My final formulation of the join calculus therefore forbids the nesting of definitions. Along with the lack of mutable state,[6] this ensures all data dependencies are explicit. In my *flattened* version of the join calculus, programs consist of a list of definitions. This necessitates a special type of channel, *constructors*, that are used to create and instantiate a join definition.

Standard channels have definition-local scope and instances of them are first-class values. By contrast *constructor channel names*, marked by $@f$, are exported from definitions, but do not give first-class values. Invoking a constructor creates an *instance* of the definition which consists of a *channel instance* (conceptually a message queue) for each of the definition's channel names. Messages are placed on the queues by emission calls as before, and when sufficient messages are available on these queues, one of the transitions can be fired. Channel values are first-class values and resemble closures in the λ-calculus—semantically being pairs consisting of the channel name and instance identifier. A constructor would normally pass out some of its definition's channel instances as

[4] MPI tends to be much coarser with messages only used where communication is intended, rather than as a general programming concept.

[5] The syntax of this program adds some basic pseudo-code on top of the core join calculus.

[6] Recall that state is encoded with recursive transition rules (equivalent to cyclic Petri-nets).

```
def sort(numbers, k) ▷
  let N = numbers.length in
  def split(a) ▷
        let n = a.length in
        if n = 1 then merge(a)
                 else split(a[0→n/2-1]) & split(a[n/2→n-1]),
      merge(a) & merge(b) ▷
        if a.length + b.length = N then do_merge(a, b, k)
                                   else do_merge(a, b, merge)
  in split(numbers)
```

`do_merge` is a functional-style procedure that merges the sorted arrays `a` and `b` into a new sorted array that is passed to its continuation (`k` or `merge`).

Figure 3.8: Implicit use of data in the join calculus.

Channels	x, y, z
Program	$P = \textbf{def } D; P \mid \epsilon$
Definitions	$D, E = J \triangleright M \mid @x(\bar{y}) \triangleright M \mid D, E \mid \epsilon$
Terms	$M, N = @x(\bar{y}) \mid x(\bar{y}) \mid M \mathbin{\&} N$
Patterns	$I, J = x(\bar{y}) \mid I \mathbin{\&} J$

Figure 3.9: Syntax of the Flattened Join Calculus.

these are not otherwise exported. Note that if rules are restricted to have a single channel in the left-hand-side pattern, the calculus becomes a traditional functional language and definition instances play no interesting role.

The syntax of my flattened join calculus is given in Figure 3.9. Its semantics can be expressed in a similar manner to the original reflexive ChAM style (Section 2.4.1). However, since the top-level definitions are no longer part of terms in the calculus I lift them out. I assume that channel names are globally unique, and this can trivially be achieved with renaming. Rather than replicating definitions on every instantiation (as was done in Figure 2.11 with the σ_{dv} substitution), these semantics use identifiers on channel names, written with a superscript—for example, $x^{\mathrm{id}}(\bar{y})$. Rules now take the form $P \vdash \mathcal{M} \rightleftharpoons \mathcal{M}'$ and $P \vdash \mathcal{M} \rightarrow \mathcal{M}'$, and are given in Figure 3.10.

It is worth noting that the grouping of transition rules into definitions, while useful for legibility, is actually semantically redundant. A program with a single definition containing all the transition rules would behave identically.

However, despite forbidding nested definitions, they can easily be encoded by a process similar to both *lambda-lifting* and Java's *inner-classes*. In particular, any program similar to:

```
def ...,
    a(x,k) ▷
```

$$\begin{array}{rcr} P \vdash & M \,\&\, N \rightleftharpoons M, N & \text{(join)} \\ P \vdash & @x(\bar{y}) \rightleftharpoons @x^{\texttt{new\_id}}(\bar{y}) & \text{(construct)} \\ \textbf{def } J \rhd M, D; P \vdash & J\sigma \rightarrow M\sigma & \text{(reduce)} \end{array}$$

where `new_id` is an unused identifier and σ is a substitution that operates on:

Parameters. Replacing the formal parameters in the join pattern J with the actual parameters transmitted.

Channels. Replacing unidentified channel names in J with identified equivalents (i.e. $x \mapsto x^{\text{id}}$). The same identifier is used for all channels.

Figure 3.10: Chemical Abstract Machine Semantics for the Flattened Join Calculus.

```
def f(m) ▷ m(x * 2)
in  k(f)
```

can be rewritten using extra channels for values that would have been implicitly shared. In this case:

```
def @unnested(x,k) ▷ temp(x) & k(f),
    temp(x)        ▷ temp(x) & temp(x),
    f(m) & temp(x) ▷ temp(x) & m(x * 2);

def ...,
    a(x,k) ▷ @unnested(x,k);
```

It is worth noting that, unlike λ-lifting which cannot support first-class function values without adding explicit environments, this encoding does not impose any restrictions on the use of values. This is because I store the extra values in fresh channels so that existing channels do not change type. Conventional lifting would alter `f` to take `x` as an extra argument and pass it at all call-sites—hence preventing any use of `f` outside of `a` (e.g. by the continuation `k`).

Unfortunately, the extra channel could cause serialisation of many transition firings within the definition if each transition were simply to match on the new channel and then re-emit a message with a same value. There are two possible solutions to this. The first is with *duplication* transitions (as shown above) that allow as many copies of the `temp` message to be created as required. This relies on the scheduler not to perform excessive duplications. The second is to rely on the compiler to elide the serialisation in a similar way to software transactional memory, or previous work on optimistic lock removal [99]. A flattened version of the mergesort example is shown in Figure 3.11. As I will show in Section 3.4.6, my approach will be to model all transfers explicitly as transitions in the IR at runtime.

```
def @sort(numbers, k) ▷ split(numbers) & info(numbers.length, k),
    info(N, k)        ▷ info(N, k) & info(N, k),

    split(a) ▷
      let n = a.length in
      if n = 1 then merge(a)
               else split(a[0→$\frac{n}{2}$-1]) & split(a[$\frac{n}{2}$→n-1]),

    merge(a) & merge(b) & info(N, k) ▷
      info(N, k) &
      if a.length + b.length = N then do_merge(a, b, k)
                                 else do_merge(a, b, merge)
```

Figure 3.11: Flattened version of merge sort.

3.3 The Join Calculus Abstract Machine

For the purposes of an intermediate representation, my flattened version of the join calculus can be supported by just three key operations: instance construction, message emission, and local channel introduction. These, along with the pattern matching, or *firing rule* semantics, take the place of method or function calls in a traditional IR. The other operations of the IR can be left unchanged, except function return values and instructions affecting global state which must be encoded using CPS and cyclic messages as previously mentioned. Indeed, the flattening alteration itself already went a long way towards making the join calculus look more like conventional representations (e.g. the JVM or LLVM).

Instance Construction (**construct**): Creates a new instance of the definition containing the named constructor, and places the arguments in a message on the queue for that constructor's channel within the newly created definition.

Message Emission (**emit**): Places the arguments in a message on the given channel's message queue.

Local Channel Introduction (**load.channel**): Gives the channel value for the named channel in the current instance.

In the interests of generality, this section will aim to describe the semantics of the key JCAM operations in a way that allows them to be combined with a variety of other instructions. To do this, I will assume that the complete IR will take a single-static assignment form in a similar style to LLVM. The non-JCAM instructions within this IR will be referred to as *local instructions* (e.g. **add**). As interaction with global state is forbidden, these will act on state that is restricted to being within a transition firing. If an IR wanted to support in-place mutable structures or arrays, this would be possible, however, they would have to respect a memory-isolation property by only allowing writes when a transition firing held the sole reference.[7]

[7] Wool and Cilk make similar assumptions on use of shared memory by spawned functions.

$$
\begin{array}{lrl}
\text{Programs} & p ::= & \overline{\mathit{def}} \\
\text{Definitions} & \mathit{def} ::= & \textbf{definition}\ \{\overline{\mathit{prod}}\} \\
 & \mathit{prod} ::= & \textbf{channel}\ \mathit{id}(\overline{\mathit{typ}}) \mid \textbf{transition}\ \overline{\mathit{note}}\{\bar{b}\} \\
\text{Pattern Notes} & \mathit{note} ::= & \mathit{id}(\overline{\mathit{arg}}) \\
\text{Types} & \mathit{typ} ::= & \mathbf{i}\mathit{sz} \mid (\overline{\mathit{typ}}) \mid \ldots \\
\text{Arguments} & \mathit{arg} ::= & \mathit{typ}\ \mathit{id} \\
\text{Values} & v ::= & \mathit{id} \mid \mathit{const} \\
\text{Blocks} & b ::= & l\ \bar{\phi}\ \bar{c} \\
\phi\ \text{nodes} & \phi ::= & \mathit{id} = \textbf{phi}\ \mathit{typ}\overline{[\mathit{val}, l]} \\
\text{Commands} & c ::= & \textbf{emit}\ \mathit{val}(\overline{\mathit{param}}) \\
 & \mid & \textbf{construct}\ @\mathit{id}(\overline{\mathit{param}}) \\
 & \mid & \mathit{id} = \textbf{load.channel}\ \mathit{id} \\
 & \mid & \mathit{term} \\
 & \mid & \ldots\quad (\mathit{local}\ \text{instructions—e.g.}\ \mathit{id} = \textbf{add}\ \mathit{typ}\ \mathit{id}, \mathit{id}) \\
\text{Terminators} & \mathit{term} ::= & \textbf{br}\ \mathit{val}\ l_1\ l_2 \mid \textbf{br}\ l \mid \textbf{finish}
\end{array}
$$

Figure 3.12: Syntax of a JCAM IR.

The basis of the syntax for such an IR is shown in Figure 3.12.[8] As in the flattened calculus, each program consists of a list of definitions. In turn, each definition consists of channel declarations and transitions. Indeed, compiling from the flattened calculus into the abstract machine is a straightforward conversion. The only command that is not explicit in the calculus is **load.channel**, which forces programs to be explicit when an identifier is being used to refer to a channel in the current instance rather than a local variable.

Like in LLVM, each identifier *id* begins with either `@` which is used for constructors, or % which represents local variables. It is convenient in examples to elide **load.channel** commands, with %*id* implicitly performing %*id* = **load.channel** *id* if the local variable *id* is not already defined. The Dovetail implementation [21] presented in Chapter 4 supports this abbreviation. However, for the purposes of semantics and the analysis in Chapter 5, it is convenient to keep it explicit.

The types shown correspond to arbitrary bit-width integers $\mathbf{i}\mathit{sz}$ (e.g. **i**32 or **i**64) and channel types $(\overline{\mathit{typ}})$. Whilst it is possible to perform type inference on the join calculus [48], this is not a focus of my work, so the IR is explicitly typed in a very similar vein to LLVM.

Each transition definition is made up of a number of labelled *blocks* $b = (l, \bar{\phi}, \bar{c})$. The ϕ-nodes allow use of previous values at merge-points in the control-flow graph, as is standard in an SSA representation. Commands in the IR make use of *values* which are either local variables or constants. The basic syntax shown only includes the three join calculus primitives discussed above. The list of commands $\bar{c}$ must conclude with a single

[8] This is the only place where I need a name for the occurrence of a channel in a pattern. The term *note* is taken from Polyphonic C#, where patterns are referred to as *chords*.

terminator instruction that either **br**anches to another block, or **finish**es the transition. Terminators may not appear any earlier in this list.

This gives transition bodies the same expressiveness as a conventional IR function body, with the exception of access to mutable state. I show in Chapters 4 and 5 that the requirement of encoding state need not be particularly costly in terms of performance. It is true, however, that the lack of mutable state is a disruptive change from the point-of-view of frontend compilers. The join calculus primitives themselves are backwards-compatible with functions, so do not cause more disruption than any other continuation-passing style representation.

3.3.1 Semantics

Rather than the Chemical Abstract Machine [45] or the rewriting-style [84] semantics previously used for the join calculus, the operation of the key primitives will be described with more traditional small-step semantics that can also be used for the local instructions. This style is very similar to Petri-net semantics, in particular markings and the firing rule, and corresponds more closely to actual implementations of the calculus (although these would typically use per-channel queues instead of a single multiset). It also provides a more natural fit for typical analysis techniques, and the control-flow analysis of Chapter 5. As far as possible, the notation that I will use follows the recent *Vellvm* work on formalising the LLVM intermediate language [121].

The execution of a JCAM program p is specified by rules as shown in Figure 3.13, with the form $p \vdash S \twoheadrightarrow S'$. These are transitions between *machine states* S of the form Γ, Σ, where Γ provides a *global environment* and Σ is the state of the transition currently firing. This treatment therefore runs the right-hand-side of a rule to completion (**finish**) before firing another rule. Although this results in transition-level interleaving, it describes the same observable behaviour as more fine-grained interleavings, or even truly parallel semantics.

The global environment $\Gamma = M, \theta$, consists of a Petri-net style marking M and a global *timestamp* θ. Within the semantics, every definition instance will be distinguished by a timestamp, and θ simply acts as a global counter used to allocate these. Note that θ could easily be split into worker-local counters, so an implementation does not necessarily need a global lock on such a counter.

The current firing's state $\Sigma = (t, l, \bar{c}, \rho, \bar{b})$ is more complex. It starts with the identifier t of the current executing instance, which is used for producing channel values whenever **load**.**channel** is executed. This is followed by the label l of the current block, which is necessary for evaluating the successor block's ϕ-nodes after the end of the block. The list of commands $\bar{c}$ is referred to as the *continuation sequence* by Vellvm. This is the list of commands that are yet to be executed in the block. Local variables are assigned values in the local environment ρ. Finally, $\bar{b}$ is the list of blocks in the currently firing transition and used after branch instructions (since the branch instruction only specifies a label and not other details of the block).

The rules also rely on the semantics of the local instructions. These are defined by the $c \vdash \rho \rightarrow \rho'$ relation. I assume this is already defined, and will not use any instructions in examples that do not have intuitive semantics. Chapter 5 will assume that these local

[9] In the semantics, p corresponds to the complete program being executed. The usage of it, $(\overline{f_i(\overline{x_i})}\{\bar{b}\}) \in p$, is simply requiring that we select a transition rule that appears in the program.

Domains:

$$\begin{aligned}
\alpha, \theta, t \in \text{Time} &= \mathbb{N}_0 \\
(f, t), (f, \theta) \in \text{ChannelValue} &= \text{Channel} \times \text{Time} \\
v \in \text{Value} &\supseteq \text{ChannelValue} \\
M, \Delta \in \text{Marking} &= \mathbf{m}(\text{ChannelValue} \times \text{Value}^*) \\
\Gamma \in \text{GlobalEnv} &= \text{Marking} \times \text{Time} \\
\rho \in \text{LocalEnv} &= \text{Identifier} \to \text{Value} \\
b \in \text{Block} &= \text{Label} \times \text{Phi}^* \times \text{Command}^* \\
\Sigma = (t, l, \bar{c}, \mathit{term}, \rho, \bar{b}) \in \text{LocalState} &= \text{Time} \times \text{Label} \times \text{Command}^* \\
&\quad \times \text{LocalEnv} \times \text{Block}^* \\
(\Gamma, \Sigma) \in \text{State} &= \text{GlobalEnv} \times \text{LocalState}
\end{aligned}$$

Global Semantics[9]: $p \vdash \Gamma, \Sigma \twoheadrightarrow \Gamma', \Sigma'$

$$\frac{\begin{array}{c}\Delta = \{((f_i, \alpha), \bar{v_i}) \mid 1 \leq i \leq n\} \\ \overline{(f_i(\overline{x_i})}\{\bar{b}\}) \in p \qquad b_0 = (l_0, [], \bar{c}) \qquad \rho = [\bar{x}_i \mapsto \bar{v_i} \mid 1 \leq i \leq n]\end{array}}{p \vdash (M + \Delta, \theta), (\_, \_, [\textbf{finish}], \_, \_) \twoheadrightarrow (M, \theta), (\alpha, l_0, \bar{c}, \rho, \bar{b})} \ (\text{FIRE})$$

$$\frac{c_0 = (\textbf{emit}\ x(\bar{y})) \qquad \bar{v} = \rho(\bar{y}) \qquad (f, \alpha) = \rho(x)}{p \vdash (M, \theta), (t, l, c_0 \cdot \bar{c}, \rho, \bar{b}) \twoheadrightarrow (M + ((f, \alpha), \bar{v}), \theta), (t, l, \bar{c}, \rho, \bar{b})} \ (\text{EMIT})$$

$$\frac{c_0 = (\textbf{construct}\ f(\bar{x})) \qquad \bar{v} = \rho(\bar{x})}{p \vdash (M, \theta), (t, l, c_0 \cdot \bar{c}, \rho, \bar{b}) \twoheadrightarrow (M + ((f, \theta), \bar{v}), \theta + 1), (t, l, \bar{c}, \rho, \bar{b})} \ (\text{CONSTRUCT})$$

$$\frac{c_0 = (x = \textbf{load.channel}\ f)}{p \vdash \Gamma, (t, l, c_0 \cdot \bar{c}, \rho, \bar{b}) \twoheadrightarrow \Gamma, (t, l, \bar{c}, \rho[x \mapsto (f, t)], \bar{b})} \ (\text{LOAD.CHANNEL})$$

$$\frac{c_0 \vdash \rho \to \rho'}{p \vdash \Gamma, (t, l, c_0 \cdot \bar{c}, \rho, \bar{b}) \twoheadrightarrow \Gamma, (t, l, \bar{c}, \rho', \bar{b})} \ (\text{LOCAL})$$

$$\frac{(l_1, \bar{\phi}, \bar{c}) \in \bar{b} \qquad \rho' = \rho[\mathrm{x} \mapsto \rho(y) \mid (y, l_0) \in \bar{v} \wedge (x = \textbf{phi}\ \bar{v}) \in \bar{\phi}]}{p \vdash \Gamma, (t, l_0, [\textbf{br}\ l_1], \rho, \bar{b}) \twoheadrightarrow \Gamma, (t, l_1, \bar{c}, \rho', \bar{b})} \ (\text{BR})$$

Figure 3.13: Generic operational semantics of a Join Calculus Abstract Machine (JCAM)

instructions all operate on primitive values (i.e. no structures or arrays), as this simplifies the presentation of my control-flow analysis.

The firing rule of the semantics defines the characteristic join calculus behaviour in a similar way to how the Petri-net semantics were described in Section 2.4.1. It requires that the current marking M contains messages Δ that match the join pattern of a rule $(\overline{f_i(\overline{x_i})\{\bar{b}\}})$ These messages must also all be associated with the same instance α. I will refer to the set of all transition rules as $\mathbb{T}$, and all channels as $\mathbb{C}$. The complete program is p. The JCAM starts with a marking placing a single message at a `main` constructor as follows:

$$\Gamma = (\{((@\mathtt{main}, 0), \bar{v})\}, 1)$$
$$\Sigma = (\_, \_, [\mathbf{finish}], \_, \_)$$

I require that the program provides a *main* constructor typed in accordance with the input $\bar{v}$. This input must only contain primitive values, or channel values (e.g. giving access to system calls) that cannot conflict with any program definition instances—present or future. This can be achieved by drawing their instance identifiers from a disjoint set—e.g. the negative integers.

3.3.2 Paths and Traces

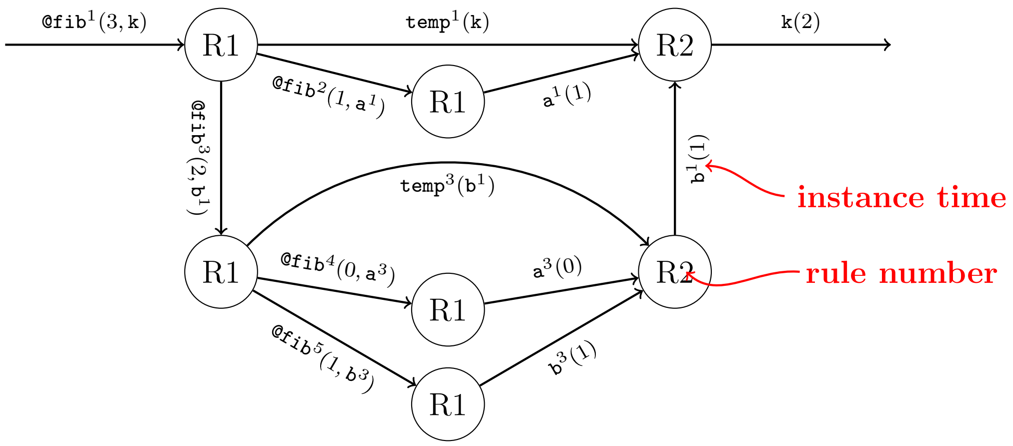
With any representation, it is useful to be able to describe possible *paths* through the program, and actual execution *traces*. In standard functional languages and imperative languages, this is done through use of call-strings as discussed in Section 2.5.3. The obvious equivalent for the join calculus and JCAM would be to only consider firing transitions in the semantics, producing a record of which transitions have fired. However, unlike in the λ-calculus where the relationship between function calls can be described accurately by a list, the relationship between join calculus transition firings is much more complex.

Each firing of a transition is referred to as an *occurrence* v. I will also make use of a *big-step* semantics, with $\Downarrow_r$ representing the sequence of steps from a firing of transition rule r until the next firing. These $\Downarrow$ relations act between markings M. When the transition is not given, I am considering all possible firings—i.e. $\Downarrow$ is shorthand for $\bigcup_r \Downarrow_r$. From this, I can make use of work on paths in Petri-nets. There are two main approaches in the literature: *causal nets* [81] and *pomsets* [87]. Pomsets correspond to directed-acyclic graphs and are therefore a more useful representation when discussing execution costs and scheduling later (Sections 3.4.6 and 6.6.2). An example of a JCAM program path is given in Figure 3.14 where each node represents a transition rule firing.

Definition 1 *A* **(pomset) path** *is a triple* $(V, \leq, \mu)$ *where* V *is a set of occurrences labelled with a transition and instance identifier by* $\mu : V \rightarrow \mathbb{T} \times \mathrm{Time}$. $\leq$ *defines a partial order on* V. *Two pomset paths are considered equivalent if there is an isomorphism between them.*

I will use $\mathbb{P}$ to give the set of all such paths. Note that for two transition occurrences $v_1, v_2 \in V$, if neither $v_1 \leq v_2$ nor $v_2 \leq v_1$ then the occurrences can occur in parallel, whereas otherwise they must fire sequentially.

Of course, not all these paths are actually *feasible*, and this is determined by the $\Downarrow$ relation. In the same way that not every destination of function calls in the λ-calculus can be determined statically, $\Downarrow$ also depends on runtime values.



```
R1. def @fib(x, k) ▷
          if x < 2 then k(x)
                   else temp(k) & @fib(x-2,a) & @fib(x-1,b),
R2.     temp(k) & a(x) & b(y) ▷ k(x + y)
```

Figure 3.14: Call DAG for `@fib(3,k)`

A final observation is that the $\Downarrow_r$ relation is ripe for annotation with expected time costs. This is of particular importance to the encoding on NUMA and heterogeneous architectures explored in Section 3.4.6. Where the time cost of a transition is referred to, I use a *duration function* $\langle\_\rangle : \mathbb{T} \to \mathbb{R}_+$ to give an estimate of how long it takes to fire. Assuming such durations are associated with each transition, it is reasonable to ask how long a path will take to execute. Given a pomset path $\mathfrak{p} = (V, \leq, \mu) \in \mathbb{P}$, this is given by $\langle\mathfrak{p}\rangle = \max_{v \in V}(f(v))$ where the finish time $f(v)$ of an occurrence is given by:

$$f(v) = \begin{cases} \max_{\{w \in V \mid w \leq v\}}(f(w)) + \langle\mu(v)\rangle & \text{if } \exists w \in V : w \leq v \\ \langle\mu(v)\rangle & \text{otherwise} \end{cases}$$

In executing a JCAM program, the aim is to choose a trace with *minimum duration*.

There has been some other work on incorporating the notion of time with the join calculus [19]. However, this was used to specify timings rather than measure them. For example, the program in Figure 3.15 executes the process P every t time steps, until the `stop` message which occurs after $3t$ steps. It does this both by allowing transitions to take a certain amount of time to fire (e.g. $\overset{t}{\triangleright}$) and by delayed emission of messages (e.g. $3t$: `stop`()). It is therefore non-deterministic whether P is executed a 4th time.

3.4 Relation to Other Models

This section gives a more detailed explanation of the compilation from a few styles of language, and of an assortment of synchronisation primitives. In doing so, I will demonstrate the suitability of the JCAM for both imperative and functional languages.

```
                          0
def start()               ▷ P & trigger(),
                          0
    stop() & trigger()    ▷  null process,
                          t
    trigger()             ▷ P & trigger()
in  start() & 3t:stop()
```

Figure 3.15: A *Timed Join Calculus* Program (taken from [19])

3.4.1 Coordination Zoo

Section 2.3.1.1 introduced a range of common primitives used for coordination in the context of multi-threaded programs. Here I show that all of these can easily be encoded in the JCAM. I also give an encoding of *futures*, *synchronous channels*, and a *reader-writer* lock.

Mutexes

A very basic mutex can be encoded as follows.

```
def @mutex(k)        ▷ free() & k({lock,unlock}),
    lock(k) & free() ▷ k(),
    unlock(k)        ▷ free() & k()
```

This is not reentrant, but this is because the join calculus does not have any notion of a thread, so cannot determine whether it is already held. A more advanced lock could be produced by requiring programming language threads to pass an identifier, `t`, to the `lock` and `unlock` channels (and assuming that there will not be two concurrent operations on the lock with the same thread identifier):

```
def @reentrant_mutex(k)       ▷ free() & k({lock,unlock}),
    lock(t,k)    & free()     ▷ used(t,1) & k(),
    lock(t1,k)   & used(t2,i) ▷ if t1 = t2 then used(t1,i+1) & k()
                                             else wait(t1,k),
    wait(t,k)    & free()     ▷ used(t,1) & k(),
    unlock(t1,k) & used(t2,i) ▷ (assert that t1 = t2)
                                if i = 1 then free()
                                          else used(t1,i-1);
                                k()
```

Condition Variables

Now I present an encoding of condition variables in the JCAM. This construct was the most difficult to encode of those I have considered. Despite this, the final result is elegant if a little difficult to read at first. A simpler version could be achieved if recursive structure types were supported, however, there is merit in offering this primitive in the bare flattened join calculus.

The fork definition provides a linked list of continuations that should be woken by a notification. The head node of this list is stored in the condvar instance in the next channel. Each wait operation adds to the head of this list, while notify operations remove the head. A full traversal of the list is done when notifyAll is encountered.

```
def @fork(f,n,nA,next) ▷ data(f,n,nA) & next(notify,notifyAll),
    notify(next,k) & data(f,n,nA) ▷ f() & next(n,nA) & k(),
    notifyAll(k)   & data(f,n,nA) ▷ f() & nA(k)

def @condvar(k)                 ▷ next(n_finish,nA_finish)
                                   & k({wait,notify,notifyAll}),
    n_finish(next,k)            ▷ next(n_finish,nA_finish) & k(),
    nA_finish(k)                ▷ k(),
    wait(f)      & next(n,nA) ▷ @fork(f,n,nA,next),
    notify(k)    & next(n,nA) ▷ n(next,k),
    notifyAll(k) & next(n,nA) ▷ nA(k) & next(n_finish,nA_finish)
```

Barriers

The encodings of barriers in the latest Joins Library paper [113] effectively make use of dynamic code generation. On creation of an n-way barrier they produce: n channels; and either a single rule that matches on all of these, or a tree of rules with intermediate channels. If n is not known at compile time, this is not directly possible in the JCAM.

However, it is of course possible to implement a barrier more simply using a count, along with a queue of threads that need to be released. The code for this is given below. The second state definition is required to allow the barrier to be reused. It prevents continuations yet to be released from one occurrence of the barrier being confused with early arrivals for the next iteration.

```
def @barrier(n,k)                 ▷ @new_state(n, state) & k(done),
    done(k) & state(r,n,enq,rel) ▷ enq(k) &
                                    if r>1 then state(r-1,n,enq,rel)
                                           else n×rel()
                                                & @new_state(n,state)

def @new_state(n,k)               ▷ k(n,n,enqueue,release),
    enqueue(k) & release()        ▷ k()
```

A tree-based approach is also possible, but unfortunately requires each thread entering the barrier to know how many other threads it will synchronise with. This method is demonstrated with the reduction below.[10] Each computation produces a result that is then combined using the ⊕ operator.

```
def @reduction(k)                   ▷ k(done, get),
    done(a, i, M) & done(b, j, N) ▷ (assert that M = N)
                                      let x = a ⊕ b in
                                      let count = i + j in
```

[10] Note that this requires non-linear join patterns—my Dovetail implementation does not currently support these.

```
                                    if count=M then result(x)
                                                else done(x,count,M),
    result(x) & get(k)             ▷ k(x)
```

Each message on the `done` queue contains:

- A value.
- The number of computations it represents—initially 1 for each computation's immediate result.
- The total number of computations expected—which should be constant across all messages.

Futures

This encoding creates a future that evaluates the function f with argument x, and provides a channel `get` that can be used to access the result. Making a call to `get` blocks until the result is ready.

```
def @future(f,x,k)     ▷ f(x,result) & k(get),
    result(y) & get(k) ▷ k(y)
```

Reader-Writer Lock

As with the other encodings in this section, a reader-writer lock encodes elegantly into the JCAM. This example is translated from that given in the initial Joins Library work [101].

```
def @readerwriter(k)             ▷ idle() & k(
                                      getRead, releaseRead,
                                      getWrite,releaseWrite
                                    ),
    getWrite(k)    & idle()      ▷ k(),
    releaseWrite(k)              ▷ idle() & k(),
    getRead(k)     & idle()      ▷ sharing(1)     & k(),
    getRead(k)     & sharing(n)  ▷ sharing(n + 1) & k(),
    releaseRead(k) & sharing(n)  ▷ (if n = 1 then idle()
                                              else sharing(n - 1)
                                    ) & k()
```

3.4.2 Streaming Computations

Since the join calculus is a close relative of dataflow graphs, one would hope that it offers a straightforward encoding of streaming and pipelined languages. The key primitive that these languages offer which the join calculus does not immediately provide is that of ordered queues. Channel message queues in the join calculus do not have any order guarantees. Here I give two approaches to encoding ordering.

One-Place Buffer. The easiest way to keep ordering is to prevent the producer from getting ahead of the consumer. This can be done by a one-place buffer as follows:

```
def @queue(k)           ▷ k(take, put),
    take(m) & put(v, n) ▷ m(v) & n()
```

This of course constrains the computation significantly. A common idea in streaming runtimes is that a node in the graph can be duplicated to keep up with demand, with results of the node still appearing in order. Whilst it would be easy to write a queue that alternates a producer between one of N consumers (with a corresponding queue that merges N producers back into a single consumer), this would be for a fixed N and not allow dynamic adaptation. In order to allow this adaptation, the values being communicated between nodes should be futures. This way a varying number of computations at each node can be performed at once. There is no need to allow more than one future to be in-flight between each node as no real computation is performed until the future is evaluated.

Arbitrary-sized Queue. Of course, arbitrary-sized ordered queues are still a useful primitive. These are relatively easy to provide with coarse-grained locking, just as Java's simple `Collections.synchronizedList(...)` wrapper. Here I describe an approach that instead performs locking in a much finer manner on each node of the list.

```
def @queue(k)                 ▷ tail(head) & k(take, put),
    take(k) & head(v, next)   ▷ k(v) & next(head),
    put(v, k) & tail(set)     ▷ k() & @node(v, set, tail)

def @node(v, m, n)            ▷ m(v, get) & n(set)
    get(k) & set(v, next)     ▷ k(v, next)
```

This describes a complex data structure that is similar to Java's `LinkedBlockingQueue`, whilst capturing its communication characteristics more accurately than a Java implementation ever could.

3.4.3 Task-based Parallelism

Fine-grained task-based parallelism is probably the most popular of parallel programming languages. It is therefore important that it encodes neatly in the JCAM. Fernandez studied this encoding in his master's thesis [42]. This section presents his solutions, specifically considering the Cilk language.

The key primitives to support are, of course, `spawn` and `sync`. There are, however, two other[11] more obscure keywords in Cilk that I also consider as an exercise in compilation of a complete language—`abort` and `inlet`. They were specifically mentioned by Fernandez as problematic. I seek to counter that argument.

[11] Cilk does also have a `cilk` keyword that specifies a function as being spawnable. I will only consider such functions so do not need to consider the keyword any further.

Basic Case

Fernandez starts by considering the simple case where all `spawn` and `sync` constructs are outside control-flow structures such as `if` and loops. For example, a simple Fibonacci program as in Figure 3.16. This figure shows not only the Cilk program but also its translation into the JCAM.

As with compilation of any function to the JCAM, each Cilk function must form its own definition, with each call creating a new instance. This is required to prevent messages from different call frames interfering in the case of recursive (i.e. where two instances of a function would be on the conventional call stack at once) or parallel calls. After a normal function call, all live variables are passed to one continuation channel, and the function is given a second continuation channel. The code of the continuation is then defined in a transition rule that matches on both of these. In the event of `spawn` operations, further continuation channels are simply added for the function call. However, rather than matching on these in the transition rule of the code immediately following the spawn, it is necessary to wait until the `sync` operation to require a message.

Within Control Structures

Encoding `spawn` and `sync` operations that occur within conditionals and loops is more difficult as the exact number of child tasks is not known at compile-time. In Cilk specifically, each `sync` acts as a barrier on all outstanding children (unlike Wool which synchronises on the next child). It is therefore only necessary to maintain a count of outstanding children (`%c` below). For each `spawn`-site, we then introduce a continuation channel `kspawn`$_f$. We then require multiple rules as follows for a `sync`-site i:

```
transition %sync_i(i32 %c, ... local vars ...) %kspawn_f(T %x) {
  Update local variables with result %x
  %done = cmp eq i32 %c, 1
  br %done, label %end, label %wait

done:
  emit %postsync_i(... local vars ...)
  finish

wait:
  %c_m1 = sub i32 %c, 1
  emit %sync_i(i32 %c_m1, ... local vars ...)
  finish
}
```

A further complication occurs when the result of a spawn is assigned to a location dependent on other variables—i.e. an array element (or multiple levels of array elements). In this case it is necessary to use a wrapper that can pass these extra parameters to the result—for example `a[i] = spawn f(x)` might be compiled as in Figure 3.17.

Speculative Computation

Cilk's `inlet` keyword allows functions to be defined that are used to handle the results of spawned children. For example, if we were looking for a solution with the minimum

```
int fib(int n) {
  if(n <= 1)
    return n;

  int x = spawn fib(n - 1);
  int y = spawn fib(n - 2);

  sync;

  return x + y;
}
```

$\Downarrow$

```
definition {
  channel @fib(i32, (i32))
  channel %a(i32)
  channel %b(i32)
  channel %temp((i32))

  transition @fib(i32 %x, (i32) %k) {
    %base = cmp ult i32 %x, 2
    br %base, label %base_case, label %recurse

  base_case:
    emit %k(i32 %x)
    finish

  recurse:
    emit %temp((i32) %k)

    %x1 = add i32 %x, -1
    construct @fib(i32 %x1, (i32) %a)

    %x2 = add i32 %x, -2
    construct @fib(i32 %x2, (i32) %b)
    finish
  }

  transition %a(i32 %x) %b(i32 %y) %temp((i32) %k) {
    %result = add i32 %x, %y
    emit %k(i32 %result)
    finish
  }
}
```

Figure 3.16: Compilation of `fib` in Cilk to the JCAM.

```
definition {
    ...
    construct @fwrap(i32 %i, S %x, (S, (T)) %f, (i32, T) %kspawn)

  transition %sync_i(i32 %c, ..., [T] %a) %kspawn_f(i32 %i, T %res) {
    store [T] %a, i32 %i, T %res
    As before
  }
}

definition {
  Channel declarations omitted

  transition @fwrap(i32 %i, S %x, (S, (T)) %f, (i32, T) %k) {
    emit %f(S %x, (T) %result)
    emit %temp(i32 i, (i32, T) %k)
    finish
  }

  transition %result(T %r) %temp(i32 %i, (i32, T) %k) {
    emit %k(i32 %i, T %r)
    finish
  }
}
```

Figure 3.17: Compilation of `spawn/sync` within more complex control-flow.

```
cilk solution_t evaluate(param_t p);

cilk solution_t search() {
  solution_t best = NULL;

  inlet void check(solution_t s) {
    if((best == NULL) || (s.cost < best.cost)) {
      best = s;
    }

    if(best.cost == 0) {
      abort;
    }
  }

  for(int i = 0; i < N; i++) {
    check(spawn evaluate(PARAMS[i]));
  }

  sync;
  return best;
}
```

Figure 3.18: Example of Cilk's `inlet` and `abort` keywords.

cost over a search space, this could be written as in Figure 3.18. The `abort` keyword is used to cancel remaining evaluations once we find a zero-cost solution (assuming negative costs are impossible). In the JCAM this would be written as shown in Figure 3.19. Note that the encoding does not necessarily cancel the remaining evaluations, but it will allow the search itself to terminate. As with other scenarios, it is hoped that a scheduler will be able to see that further firings of `evaluate` are wasteful. This is not dissimilar to garbage collecting unused objects in a language such as Java.

3.4.4 Transactions

A more difficult paradigm to consider is that of transactions. This typically provides the programmer with `atomic` blocks and possibly also the opportunity to `abort` or `retry` a transaction. A lot of the issues involved here have been highlighted by Boehm in a discussion paper [17].

He points out that the `abort` primitive, which rolls back the program state to the start of the block, is a feature independent of the concurrency or parallel aspects. Furthermore, he argues that the `retry` statement can only be of use in a scenario where some aspects of coordination are being performed outside of the transactional framework. I will therefore focus solely on the atomic blocks themselves.

Boehm concludes that the most intuitive semantics for such blocks is that of a global mutual exclusion lock. This is of course easy to encode in the join calculus. However,

```
def @evaluate(p, k) ▷ k(...)

def @search(k) ▷ state(None, N, k)
                 & for(p in PARAMS) @evaluate(p, check),
    state(best, n, k) & check(s) ▷
      let next = match best with
        | None   -> s
        | Some b -> if s.cost < b.cost then s else b
      in
      if (next.score == 0) or (n == 1) then k(next)
                                       else state(Some(next), n-1, k)
```

Figure 3.19: Encoding of Figure 3.18 in the JCAM.

```
class Foo {                      def @Foo(k) ▷ S(...) & k({f_1, ...}),
  public f_1(...) {                  S(this) & f_1(..., k) ▷
    ...                   ⇒            let new_this = ... in
  }                                    S(new_this) & ... & k(...),
  ...                                ...
}
```

Figure 3.20: Compilation of Objects to the JCAM.

the performance of such a naïve implementation is poor. Transactional memory (TM) is therefore the de-facto way of implementing these blocks. The key point though is that, as Boehm's title states, "transactional memory should be an implementation technique, not a programming interface".[12] Applying this to my situation, TM should be seen as a way of improving JCAM performance, much as others [99] have used it to improve conventional lock-based programs. This is not something explored in this thesis, but is a promising area for future work.

3.4.5 Objects and Actors

Immutable structures could easily be added to a JCAM without any adverse interaction with the join calculus.[13] From these it would be possible to implement objects using a single *this* channel S that maintains the state between calls to methods, as shown in Figure 3.20. The behaviour of such objects in a parallel context would be exactly as per the *actor model* [56], with only one method of the object executing at any one time.

Supporting inheritance is a little more complex. I do not believe that supporting inheritance directly within the abstract machine is the right approach. This is because

[12] This is further backed up by recent work to implement atomic sections with *lock-inference* techniques, which "infers a set of locks for each atomic section, while attempting to balance the requirements of maximal concurrency, minimal locking overhead and freedom from deadlock" [52].

[13] Although, of course, the control flow-analysis presented in Chapter 5 would need to be altered to incorporate compound values.

there are a number of ways that the join calculus and the notion of inheritance can interact [47]. Tying the abstract machine to a specific behaviour would be unnecessarily restrictive. Instead, I would suggest supporting subtyping of structures within the JCAM. All inheritance could then be resolved in the frontend, with constructor transitions in the JCAM simply returning structures that contain public methods and fields.[14] Supporting interfaces and multiple inheritance of course requires that the subtyping relation between structure types ignores the order of fields (i.e. does not check whether one structure is a prefix of another), but there has been plenty of research looking at how this can be effectively implemented in the context of the JVM [3].

3.4.6 Complex Memories

One of the original motivations for this work was the difficulty in dealing with placement and scheduling on heterogeneous architectures. These two problems are tightly linked and, as argued in Section 1.1, allowing them to be considered as a single problem would be a big advantage. The original join calculus paper [45] did intend the model to be used for distributed as well as concurrent computation, and JoCaml supported this through a registry type setup [33]. However, it is fair to say that this aspect of the language has not received the same attention as the join patterns, and most of the later work has considered the model in the context of shared-memory multi-core systems. I would argue that the join calculus is in fact better suited to an automatic approach to this mapping than the registry. The placement choices in a heterogeneous system can be accurately modelled with the non-determinism of the calculus, allowing them to be seen as standard scheduling choices. Data movement between cores, as well as local computation, is expressed as transition rules. Thus I argue that the JCAM can form a universal intermediate representation, that not only models existing concurrency primitives, but could also adapt to different architectures at load-time or run-time.

The remainder of this section demonstrates how to perform this construction. The work for this section has been presented in two publications: one using Petri-nets [22], and a later one on the join calculus [24].

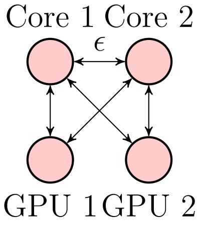
Hardware Model

I restrict myself to a very simple model of heterogeneous architectures which ignores fine details of the memory system such as caches. I consider a system to consist of *processors*, each with a *local memory*, and *interconnects* between them. The cost of accessing this local memory is low and considered part of the computation cost of a transition. Non-local data must be transferred via interconnects before use, at a cost modelled by latency and bandwidth. This is not dissimilar to the partitioned global address space model (PGAS) that is used elsewhere (e.g. X10). I ignore capacity constraints of memories.[15] Formally, a hardware architecture is defined as follows:

Definition 2 *A* **simple heterogeneous hardware model** *H is a 3-tuple (P, i, c) consisting of:*

[14] Public fields will of course correspond to structures containing getter and setter channels for a memory cell encoding.

[15] In situations where not all program data can fit in a single memory, one could imagine data transfer costs varying to prevent a scheduler choosing to transfer more data to a near-full memory.



Here ϵ is typically small, and the other costs are based on actual measurements.

Figure 3.21: Example model for dual-core CPU with 2 general-purpose GPUs.

- *A finite set of* processors P.
- *An* interconnect descriptor *function* $i : (P \times P) \to (\mathbb{R}_+ \times \mathbb{R}_+)^\infty$. *For a pair* (p_1, p_2) *of distinct processors,* $i(p_1, p_2) = (l, b)$ *gives the latency* l *and per-byte cost* b *(=* $\frac{1}{bandwidth}$*) of the interconnect from* p_1 *to* p_2. *I will refer to the cost of transferring* n *bytes of data with the notation* $\langle p_1 \overset{n}{\to} p_2 \rangle = l + n \cdot b$. *When there is no interconnect from* p_1 *to* p_2, $i(p_1, p_2) = \infty$.
- *A* computation cost *function* $c : (\mathbb{T} \times P) \to \mathbb{R}_+^\infty$, *where* ∞ *indicates that the processor cannot perform the transition (e.g. no floating-point support). The set* $\mathbb{T}$ *represents the set of all possible transition rules.*

In practice, the interconnect descriptions and computation costs, which will only ever be approximate, would be given by profiling information. An example model of a multi-core plus GPU architecture is given in Figure 3.21. The inclusion of small costs, such as ϵ in the example, approximates the effect of cache invalidations, when cores share a memory but have separate caches. Memories not associated with a processor can be modelled as a 'null' processor $p_\perp$ with $c(t, p_\perp) = \infty$ for all $t \in \mathbb{T}$.

I assume two sanity constraints: that the memory interconnect is strongly connected,[16] and also that all transitions can be executed somewhere (i.e. $\forall t \in \mathbb{T} : \exists p \in P : c(t, p) \neq \infty$). These properties ensure that the mapping of software onto hardware is also confluent, given a confluent program.

Mapping a Program to a Target Architecture

Given the hardware model above and the JCAM defined earlier, the goal is to model all possible executions of a program on an architecture with a new JCAM program. Each feasible path through this will give a possible execution trace.

I introduce this construction by considering the flattened join calculus program presented earlier that sorts an array of integers using a *merge-sort*-like algorithm (Figure 3.11). There is clearly scope for parallelising both the `split` and `merge` steps—although this may require moving data to another memory. For this example, I take $P = \{\mathtt{x}, \mathtt{y}\}$, and assume both a fully-connected memory interconnect and that all rules can execute on either `x` or `y`. However, the encoding would also work for more complex scenarios.

[16] A graph is strongly connected if for every pair of vertices a and b, there is a path both from a to b, and b to a.

```
def
@sort_x(numbers, k) ▷
  split_x(numbers)
   & info_x(numbers.length, k),

info_x(N, k) ▷ info_x(N, k)
                 & info_x(N, k),

split_x(a) ▷
  let n = length(a) in

  if n = 1 then
    merge_x(a)
  else
    split_x(a[0..(n/2)-1])
     & split_x(a[(n/2)..(n-1)]),

merge_x(a) & merge_x(b)
                  & info_x(N, k) ▷
  info_x(N, k) &
  if a.length + b.length = N then
    do_merge(a, b, k)
  else
    do_merge(a, b, merge_x),
```

A

```
@sort_y(numbers, k) ▷
  split_y(numbers)
   & info_y(numbers.length, k),

info_y(N, k) ▷ info_y(N, k)
                 & info_y(N, k),

split_y(a) ▷
  let n = length(a) in

  if n = 1 then
    merge_y(a)
  else
    split_y(a[0..(n/2)-1])
     & split_y(a[(n/2)..(n-1)]),

merge_y(a) & merge_y(b)
                  & info_y(N, k) ▷
  info_y(N, k) &
  if a.length + b.length = N then
    do_merge(a, b, k)
  else
    do_merge(a, b, merge_y),
```

B

```
info_x(N, k) ▷ info_y(N, "k on y"), info_y(N, k) ▷ info_x(N, "k on x")
split_x(a)   ▷ split_y(a),          split_y(a)   ▷ split_x(a),
merge_x(a)   ▷ merge_y(a),          merge_y(a)   ▷ merge_x(a),
```

C

Figure 3.22: Mapped version of *merge-sort* for a dual-processor system

The intuition behind my construction comes from considering a message value x. In a program marking $M \in \mathbf{m}(\text{ChannelValue} \times \text{Value}^*)$, x must be associated with some channel $f \in \mathbb{C}$. However, the architecture $H = (P, m, c)$, on which the software is run, must store x in some memory $p \in P$. Therefore, the location of a *data* token in a running program is described by a pair from the set $\mathbb{C} \times P$.

If one now considers what might happen to the message containing x, there are two options. *Either:*

- (r, p): The message is used by transition rule $r \in \mathbb{T}$ executing on processor p (where possible), *or*

- (f, p, p', n): The message is transferred to another processor $p' \in P$ via an interconnect (p, p') as part of an n-token transfer.

The first option corresponds to producing a copy of the program for each $p \in P$, omitting any transition rules r for which $c(r, p) = \infty$, giving box "B" of Figure 3.22.

Secondly, I add transitions that correspond to possible data transfers (box "C"). This requires one rule per channel and interconnect pair. However, the higher-order nature of the join calculus means these need to be carefully defined to preserve locality. Specifically, when a channel value such as `k` is transferred it needs to be modified so that it becomes local to the destination processor. This maintains the invariant that the 'computation transitions' introduced by the first part of the construction can only call channels on the same processor.

This results in a new program that is a subset of the following (depending on the interconnect and computability functions). The construction is equivalent to the Cartesian product of hypergraphs, where each transition rule corresponds to a hyperedge.

$$\text{Channels} = \overset{\text{Data Places}}{\boxed{(\mathbb{C} \times P)}} \quad \text{Transitions} = \underset{\text{Computation Transitions}}{\boxed{(\mathbb{T} \times P)}} \cup \overset{\text{Memory Transfers}}{\boxed{(\mathbb{C} \times P^2 \times \mathbb{N})}}$$

Additionally, when the complete system is considered, *resource constraints* are required so that a processor executes only a single transition at any moment, and similarly so that each interconnect is only used for one transfer at a time. Considering the problem in the context of Petri-nets (e.g. Figure 2.13), it would be possible to use mutual exclusion locks similar to Figure 3.2. This would result in a net such as Figure 3.23 for the merge-sort example on a dual-core CPU, where red places correspond to processor constraints and green places to interconnect constraints. However, using such *resource constraint places* is not possible in the join calculus, as they need to be matched on by transition rules in different definition instances (since processor time is shared between these). Changing the calculus to allow this would make it harder to generate an efficient implementation. Instead, I introduce the notion of *workers* to the semantics.

Rather than allowing any number of transition firings to be mid-execution at a given time, each worker can only perform zero or one firing at a time. I also tag each transition

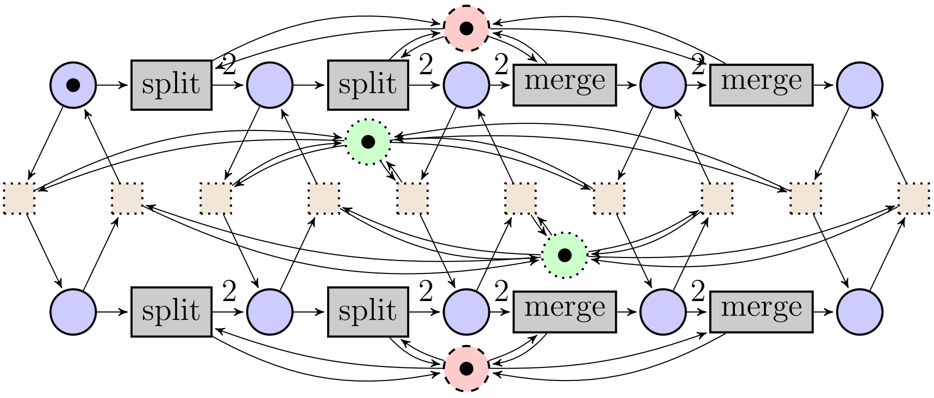


Figure 3.23: *Merge-sort* mapped onto a dual core CPU (types are omitted for clarity, and memory transfers are only shown for $n = 1$).

with the worker that may fire it. In the present example, there would be four workers: `x`, `y`, `xy` and `yx`. The `_x` and `_y` copies of the original program (in boxes "A" and "B" of Figure 3.22) are tagged with the `x` and `y` CPU workers respectively, while the data transfer transitions are tagged with the relevant interconnect worker, `xy` or `yx`.

Modifying the formal semantics of the JCAM to accommodate these changes is relatively straightforward. Previously the overall machine state was described by a pair:

$$(\Gamma, \Sigma) \in \text{State} = (\text{GlobalEnv} \times \text{LocalState})$$

with Γ giving the state of each channel's message queue, and Σ the state of the currently firing transition. To introduce workers, I replace the single firing state with a map of each worker's state as follows:

$$(\Gamma, \Omega) \in \text{WState} = (\text{GlobalEnv} \times (\text{Worker} \to \text{LocalState} \cup \{\texttt{IDLE}\}))$$

The behaviour of each non-idle worker is described by projecting from WState into State (i.e. $(\Gamma, \Omega) \mapsto (\Gamma, \Omega(w))$ for $w \in$ Worker). The firing rule must be defined on the overall machine, and requires an idle worker.

$$\frac{\begin{array}{c}\Delta = \{((f_i, \alpha), \bar{v}_i) \mid 1 \leq i \leq n\} \\ (\overline{f_i(\overline{x_i})}\{\bar{b}\}) \in p \qquad b_0 = (l_0, [], \bar{c}) \qquad \rho = [\bar{x}_i \mapsto \bar{v}_i \mid 1 \leq i \leq n]\end{array}}{p \vdash (M + \Delta, \theta), \Omega[w \mapsto \texttt{IDLE}] \twoheadrightarrow (M, \theta), \Omega[w \mapsto (\alpha, l_0, \bar{c}, \rho, \bar{b})]} \text{ (WFIRE)}$$

$$\frac{}{p \vdash \Gamma, \Omega[w \mapsto (\_, \_, [\textbf{finish}], \_, \_)] \twoheadrightarrow \Gamma, \Omega[w \mapsto \texttt{IDLE}]} \text{ (FINISH)}$$

It is also necessary to make the timestamps worker-local. I do this by drawing them from $(\text{Worker}, \mathbb{N}_0)$ rather than just $\mathbb{N}_0$. The initial state is for all workers to be `IDLE`, and some messages corresponding to program arguments to be available in Γ as before. Unmapped programs can be considered to have just a single worker, as this collapses down to the previous semantics.

When defining the hardware model, I required that the memory be strongly connected. This ensures that data transfers can always be 'undone'. Similarly, since each transition in $\mathbb{T}$ can be performed on some $p \in P$, the mapped version of a confluent program will also be confluent. Therefore, the choice of which transition to fire can only affect performance, not correctness.

Since the hardware model gives costs, the converted program can be supplemented with durations, using the $\langle \_ \rangle$ notation introduced in Section 3.3.2. It can be defined as follows:

$$\begin{aligned} \langle (t,p) \rangle &= c(t,p) \\ \langle (f,p_1,p_2,n) \rangle &= \langle p_1 \overset{n \cdot \text{sizeoftype}(f)}{\rightarrow} p_2 \rangle \end{aligned}$$

To accommodate vector processors such as GPUs, it is necessary to consider the fusing of multiple copies of a single transition. The merged transition will take significantly less time than performing the n transitions individually. Obviously, a real implementation will not enumerate these fusings, but one can view it this way in the abstract. A similar argument also applies to data transfers which can benefit from bulk operations. Further discussion of fusing and inlining is given in Section 6.6.3.

Dealing with Explicit Placement in Programs

The above encoding enlarges a JCAM program to consider placement choices not specified in the original program. However, it will often still be necessary in languages that have some notion of data transfers, or different memories. Typically these will only constrain some choices, and these constraints could be presented as annotations on the IR. Ignoring such annotations would not affect program correctness, so they should be seen as scheduler hints.

For example, in X10 [102] the language simply distinguishes between the current memory and other *places*. The programmer is therefore aware of points at which computation or data is introduced at a previously unspecified place (i.e. the points at which data transfers may occur). Expressing this equates to annotating channels for which data transfer transitions should not be introduced.

Similarly languages that expose heterogeneity between processors can be expressed by hinting that JCAM transitions should only execute in certain locations.

3.5 Summary

This chapter has detailed the rationale for the join calculus abstract machine (JCAM), and also introduced its behaviour more formally. Section 3.4 provided high-level descriptions of how certain paradigms map to the JCAM. In doing so, I have demonstrated that the JCAM is a suitable representation for encoding concurrent programs without throwing away information about their communication patterns. The following chapters take a more concrete view, addressing the challenges of implementing the JCAM with competitive performance.

***dovetail**, n. (woodworking) A type of joint where adjoining boards are fastened by interlocking fan-shaped cutouts.*

Chapter 4

Efficient Implementation

This chapter introduces an implementation of the JCAM (named *Dovetail*) based on existing state-of-the-art techniques and LLVM. Since the implementation is a compiler rather than a library, there are also opportunities to introduce (user- or analysis-generated) annotations that allow for specialised and higher performance implementations of channels and instance instantiation. Making use of these opportunities is particularly important since, unlike previous join calculus implementations, the JCAM does not offer functions, methods, or even mutable state.[1] This is accentuated by CPS rather than direct style. The primitives of the JCAM are therefore very heavily used and must not incur significant overheads. However, whilst the traditional primitives are not available within the JCAM, any implementation can of course make use of any operations offered by the target-architecture—for example shared-global-memory operations to represent optimised channel queues. This work was also encouraged by the performance achieved by Turon and Russo [113] which demonstrated that other concurrency primitives can be implemented competitively in a relatively fine-grained manner with the join calculus.

Section 4.1 starts by describing a compiler based on the techniques used in the latest library implementation of the calculus [113], along with optimisations that a compiler approach enables. I then provide a breakdown of the overheads associated with this implementation (Section 4.2). It is this investigation that guides the remainder of the chapter, with the largest area for improvement shown to be memory allocation. Much of this is associated with heap allocation of definition instances, and Section 4.3 develops a novel technique to improve this. The second improvement is in bounding the size of the channel message queues. This is investigated in Section 4.4. Throughout, a simple Fibonacci micro-benchmark is used. A wider range of programs is considered in Chapter 6, but for the purposes of optimising the raw cost of JCAM operations, this simple case is sufficient.

4.1 A Baseline Implementation

In Section 2.4, I reviewed the two predominant join calculus implementations: JoCaml [33] and the Joins Library [101, 113]. It is clear that Turon and Russo's recent work is the more sophisticated approach, with the better performance, and it therefore forms the basis of my prototype with the exact data structures described in Section 4.1.2. I make

[1] Recall that this is due to a desire to support non-shared memory and also test whether the purity of the join calculus can be offered with acceptable overheads.

some obvious improvements on their approach that I can perform due to taking a compiler rather than library approach (Section 4.1.1). Another influence is Wool's implementation of fine-grained task parallelism [39]. The simple scheduling used in this implementation is also based on work-stealing (Section 4.1.3), although more complex approaches are discussed later in Section 6.6.2.

In order to achieve results that can be compared to more mature compilers, I make use of LLVM [1] and the wide range of optimisations that it can provide. The implementation performs the compilation *ahead-of-time* rather than *just-in-time*. This was done to reduce the effort required to produce a prototype, but in the long run a JIT may be better suited if complex scheduling choices are to be made.

Within transition rule bodies (i.e. *intra-procedural*), classical techniques, such as various dataflow analyses, are applicable without modification. In fact, by using LLVM as the basis of my implementation, most common optimisations are already performed. However, switching from conventional IRs to the JCAM significantly alters the scenario for *inter-procedural* analyses and transformations. JoCaml scratched the surface of what could be done, and I take this further later in this chapter and in Chapter 5.

4.1.1 Specialised Emission Functions

For each channel, the Joins Library maintains a list of transitions that match on it. In turn, each transition is associated with a list of channels that make up its join pattern. These are required in order that a single *emission function*[2] can be used to check for new matches after a message send. The emission function is what is called by code that wishes to send a message to a channel. However, this approach incurs several costs:

- Each message send must traverse these lists. These memory reads go through multiple levels of indirection and will therefore be relatively expensive, even with the aid of caching.

- A single emission function, such as in the Joins Library, will adversely affect branch prediction. It is common in join calculus programs that certain channels in a pattern always receive their messages before others. Ideally a branch predictor would therefore be able to predict the route through the emission function fairly accurately, even if the behaviour is not obvious statically. A single function conflates all conditional branches, causing the behaviours of different channels to be confused.

Since the data contained in these lists is entirely constant, the lists can be completely avoided by compiling specialised emission functions for each channel that unroll the traversal of the list. This is the approach taken by Dovetail. A channel value is therefore not just a pointer to instance specific data, but also a code pointer to this specialised emission function which is shared with other instances of the same channel.

Furthermore, I can optimise these functions for special scenarios where simpler checks can be made. As a simple example, consider channels that always appear as the sole channel in a join pattern—i.e. *functional* channels. Since no coordination is taking place and a match will always occur, I simplify the emission function to immediately enqueue a match. This effectively reduces the overhead of the message send-and-firing to a spawn operation. Wool showed that this need take little more than a standard function call. This very simple optimisation gives the performance gains shown in Figure 4.1.

[2] `AsyncSend<A>(Chan<A> chan, A a)` in the Joins Library[113].

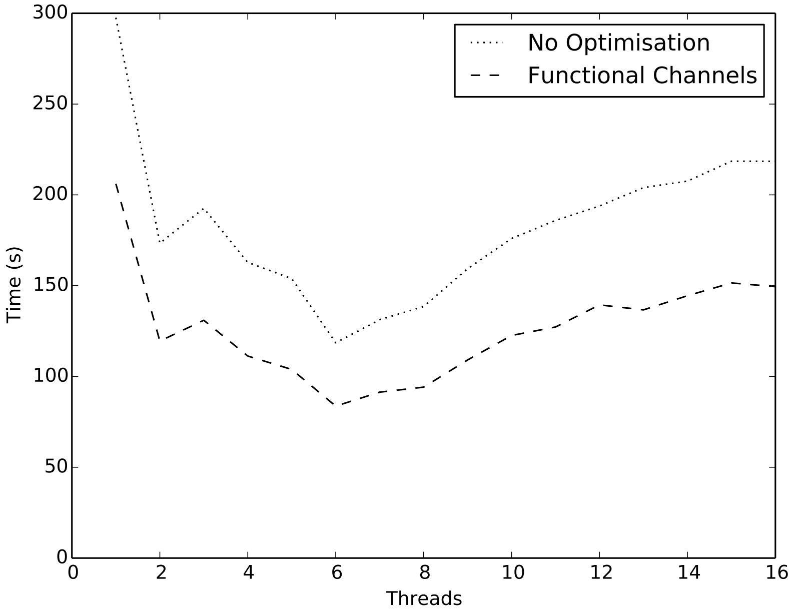


Figure 4.1: Benefit of specialising *functional* channels.

The remainder of the optimisations made in this chapter all rely on this emission function specialisation. While they offer significant benefits, I have no doubt future work could identify other specialisation scenarios ripe for optimisation.

4.1.2 Join Calculus Data Structures

Each join calculus instance is represented by a structure containing a queue for each channel. Later these queue structures may differ depending on the characteristics of the channel, but initially each represents a Michael-Scott queue—probably the best known lock-free queue algorithm and as used by Turon (see Section 2.4.3 and [113]). The nodes in these queues consist of the message value and its status[3]—again as in Turon's Joins Library work. Channel values are represented very simply as a pair consisting of the specialised emission function and a pointer to the instance. An example of the data structures for `fib` is given in Figure 4.2. This corresponds to the state of the program in Figure 3.14 before @`fib`^{4} and @`fib`^{5} are created, and once @`fib`^{2} has already passed its result to `a`^{1}.

Since a traditional function call is replaced by a message send, possibly a matching operation and finally the enqueuing of a match in the JCAM, it is crucial to minimise the overheads in this sequence as much as possible. All calls to the Dovetail runtime (for instance, to enqueue a match) are therefore inlined.[4] The only calls made by an emission function after inlining has been performed are to allocate memory.

The higher-order nature of channel values makes it impossible to inline calls to the

[3] Recall that the two-phase commit of the matching algorithm requires *pending*, *claimed* and *consumed* states.

[4] Dovetail also performs an extra optimisation beyond those performed by LLVM to remove a commonly occurring pattern, whereby switch statements followed by a conditional branch on the same (or negated) condition are pushed down into the destination blocks of the branch.

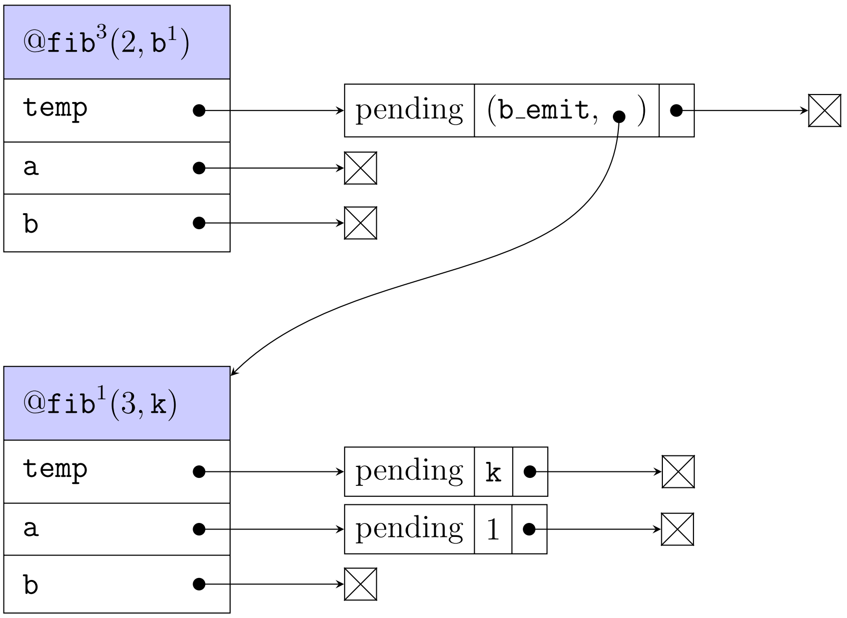


Figure 4.2: Data structure snapshot for `fib`.

emission functions themselves in many cases—although they are marked with LLVM's `inlinehint` attribute. Since these emission functions are only ever called by code that Dovetail generates, the calling convention can be modified to maximise use of registers for arguments, and minimise the cost associated with the call. LLVM provides a number of calling conventions that can be considered for this, including `fastcc`, and several `x86` conventions. Unfortunately, the conventions used by the LLVM-based Haskell and Erlang compilers were not applicable. In the case of Haskell (referred to in LLVM as `cc10`), there seemed to be compatibility issues with the Boehm garbage collector I use. Erlang's calling convention makes calls into the Erlang runtime, so is more difficult to use with other languages. Table 4.1 illustrates the performance of the possible alternatives, relative to the default C convention. These measurements are taken with all other optimisations developed in this chapter turned on, in a multi-threaded execution. This is intended to be most representative of a normal scenario—but as with other measurements in this chapter, the micro-benchmark nature means that the effect of the calling convention is not masked by computation unrelated to the JCAM coordination. Although no significant difference was observed between the standard C convention and `fastcc`, Dovetail uses `fastcc` as it passes more arguments via registers. This calling convention is also used when a worker calls a match (i.e. a transition rule's body).

The calls to the emission functions also often don't make use of data allocated on the stack of the compiled transition making the call. This allows them all to be annotated as candidates for tail call optimisation. Since transitions can only have any effect through messages they send, every transition will have at least one `EMIT` instruction that is a genuine *tail emit*.[5] Adding these annotations therefore has a worthwhile effect on performance, as shown in the second column of Table 4.1.

A final noteworthy implementation detail is the treatment of *dead instances*. These are instances in which no more transitions will ever fire (i.e. no matches are possible and

[5] Note that unlike in the CPS λ-calculus, where each function will make a single (tail) call, join transitions can perform multiple message emits even in CPS form.

Calling Convention	Without Tail Calls	With Tail Calls
LLVM `fastcc`	0.99	0.90
Standard C	1.00	0.90
Fast x86	1.22	1.22
Standard x86	1.22	1.22

Table 4.1: Execution time of calling conventions relative to standard C without tail call optimisation.

no more messages will ever be sent to the instance's channels). One might expect that collecting such instances would be a significant part of the implementation. However, since Dovetail only checks an instance for possible firings after a message is sent to one of its channels, it only need maintain references to an instance in channel values. As a result, only the instances which are reachable via a channel value can ever fire again. Dovetail therefore relies on standard garbage collection to ensure that dead instances are removed from memory.

The garbage collector used in Dovetail is the *Boehm-Demers-Weiser Garbarge Collector* [18]. Just as Section 3.3.1 commented that θ need not be a global counter but worker-specific, the *thread-local* feature of the garbage collector is enabled to reduce contention between workers when allocating instances.

4.1.3 Work Stealing for the Join Calculus

The work queues in Dovetail are standard Chase-Lev deques [29]. The tasks described in these queues correspond to matches of join calculus transitions. As in Wool, the complete task descriptors are stored in the work queue rather than having extra indirection. The individual messages of the match are also copied into the descriptor. This is motivated by considering the cachelines present in the cache at various points.

When a match is first created, all message queues in its join pattern have just been checked for messages. Assuming no false-sharing, all these messages will therefore be in the cache, and accessing the message values should be relatively inexpensive. In many cases, the size of the message value will not be dissimilar to the size of a pointer. The overheads associated with writing the value, rather than a pointer, into the match should therefore be minimal. When the match is executed, the cacheline of the match itself will be present in the cache, so accessing these values should again be inexpensive.

If we instead were to store indirect pointers to the messages, the cost of creating the match would be much the same for the above reasons. However, when executing the match we would need to follow these pointers to the locations where the messages were originally stored. Especially in the case that the match was stolen, this will often result in cache misses.

Of course, using work-stealing as the scheduler for the JCAM is a very simple approach. It is only applicable within a single shared memory (e.g. x86), since any implementation involving data transfers needs to consider the cost of these, as described in Section 3.4.6. There is also no real consideration of non-determinism—although I could feasibly control this by altering the order that transitions are checked in the matching sequence.

	Total[6]	MM	Other
Work Queues (i.e. popping/stealing matches)	2.3%	-	2.3%
Instantiating Definitions	40.6%	35.7%	4.9%
Emission of Messages (including matching)	54.3%	31.5%	22.8%
Actual Work	2.9%	-	2.9%

Table 4.2: Breakdown of `fib` execution (by instruction count).

4.2 Profiling Bottlenecks

If we examine the performance of our `fib` benchmark on a single core using the basic implementation described so far, it becomes clear that memory management is a key bottleneck. A breakdown that separates out this aspect of the execution is shown in Table 4.2. Furthermore it shows the split between the different primitives used. These results were obtained using the `callgrind` tool that forms part of *Valgrind* [120]. "MM" corresponds to any time spent within the Boehm GC `GC_malloc` function.

Unfortunately, the large amount of inlining performed by LLVM on Dovetail's output makes it very difficult to separate out the cost of enqueuing matches on the work queue from the instantiation and emission costs. However, these are not believed to be significant and the low percentage for "Other" within instantiation backs up this hypothesis. Aside from the memory management costs, join calculus matching also stands out. The cost here is due to the way that channel message queues are implemented. Each message is a distinct node in a linked list, and even an empty list includes a sentinel node. Therefore, checking whether a message is available always requires one or more pointer dereferences. This is a cost that should be avoidable, as Section 4.4 shows.

The figure given for "Actual Work" still represents more time than our C baseline takes to execute. This can be explained by the indirect function calls that are required for each `fib` definition to return its result to the continuation, which can be either `a` or `b` (see Figure 3.16). Removing these indirections requires some form of inlining. Dovetail does not do inlining itself, but I give hand-optimised values for the benefit of this in Section 4.5. Even without this inlining, Table 4.2 suggests that removing the memory and queue overheads could result in performance improving by a factor of about 40 on a single core. This is almost exactly the benefit that Dovetail achieves for `fib` (Figure 4.3).

4.3 Closed Definitions

The profiling of the previous section shows that memory management is the main overhead that prevents the baseline implementation from being competitive. Clearly improving the performance of the memory allocator and garbage collector may be possible through a number of techniques. However, a better approach would be to decrease the number of allocations that must be done on the heap. Producing a garbage collector with better performance than the Boehm-Demers-Weiser collector is also beyond the remit of this work. This section presents an entirely new approach to optimisation of the join calculus that aims to do this based on knowledge about characteristics of a definition.

Since the JCAM does not offer mutable state or function calls, instances must regularly be created in cases where a traditional representation would have used space in the current

[6] As a result of rounding for presentation, the percentages given do not sum to exactly 100%.

function's stack frame (or activation record). One key feature of a stack frame is that it can be allocated and deallocated simply by incrementing and decrementing a pointer. Unfortunately, once compiled to join calculus definitions, this can no longer be done in the general case, since the lifetime of an instance is far more loosely defined than for a function, which can never outlive its parent. As mentioned already, memory for instances is therefore allocated on the heap and garbage collected once the instance is dead—far more costly than stack allocation. This is confirmed by the profiling data in Section 4.2. Given that such examples occur often, it is important to optimise these back to stack-allocated instances. Here I show how stack allocation can be used in any case where an instance does not require subsequent messages after its construction. Such instances will be called *closed*.

These are extremely common and result from the compilation of conventional functions, and any other program parts that are self-contained (i.e. do not synchronise or coordinate with other parts). Indeed the encoding of fork-join parallelism (e.g. Cilk) shown in Section 3.4.3 exhibits this property. Closed definitions may even instantiate other definitions, which need not be closed themselves, so long as all these instances are encapsulated by the closed instance.

A precise interpretation of the semantics in Figure 3.13 never executes a transition enabled by a **construct** or **emit** until after the completion of the current transition. This would never allow stack allocation of the instance since the instantiating transition's own stack frame would be deallocated before the new instance ever executed. Even in my implementation, matched transitions will only run before the end of the current transition if they are stolen by another worker. However, if the new instance is instead executed *to completion* immediately following the **construct**, then it can be allocated on the stack.[7] Note that this is only possible for closed definitions where performing as many firings as possible on the new instance and then deallocating it does not risk missing future firings within the instance.

This is simple to implement in a single-threaded scenario—I simply modify the emission functions within the closed definition to immediately execute matches rather than enqueuing them. It is also possible in the presence of concurrency, but this is more difficult since the worker on which the instance is created must wait for any transitions on other cores to complete. This is not straightforward due to the complex synchronisation and coordination patterns that one can specify in the JCAM. Dovetail therefore only optimises closed definitions on a single-core.

This approach is quite similar to Cilk's *fast* and *slow* clones (Section 2.3.1.2). The more common single-core case is optimised in line with the work-first principle. A key difference is that in Cilk it is possible to switch from fast to slow. Unfortunately, once Dovetail decides to execute the 'fast' version of a JCAM instance on a single-core, it is not possible to switch back to 'slow' instances that can be shared between workers until the fast instance terminates.

However, there are advantages to this. It means that queue implementations within fast instances can assume non-concurrent execution and avoid expensive memory barriers. These are required in slow mode to ensure that at least one of two concurrent message-sends, which can match together, sees the other in the matching algorithm. It also fits with the idea that the program is first split into enough separate parts to occupy all

[7] This stack allocation does force tail-call annotations to be removed, but this cost is insignificant compared to the benefit of stack allocation.

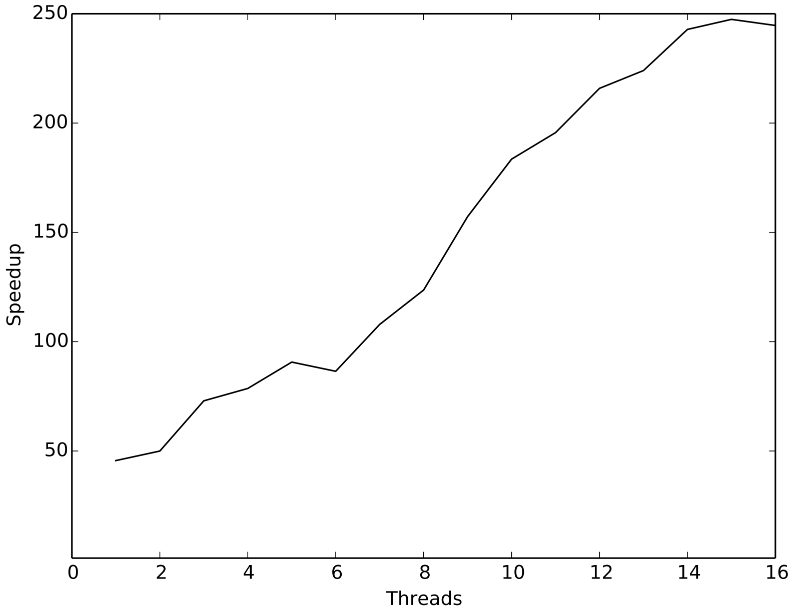


Figure 4.3: Effect of using *closedness* on performance.

available cores, but that after that each core's part should execute at as close to the speed of a sequential version as possible. This prevents the fine-grained nature of the JCAM hindering its performance too much. The decision to switch to fast mode is based on the number of matches present on the worker's queue.

We saw in the previous section the high proportion of runtime spent performing memory allocation. It is therefore unsurprising that this optimisation has a large effect on performance. As shown in Figure 4.3, for cases where closedness is applicable, execution can be sped up by several orders of magnitude.

4.4 Bounded Queues

Another particular case where specialisation of emission functions can offer significant benefits is when the size of a message queue can be bounded statically. As mentioned in Section 2.4.2, this is a scenario that was also considered in earlier work [33]. Given that mutable state is not a first-class primitive in my abstract machine, this is a particularly common pattern. Consider the memory cell encoding:

```
def @memcell(i,k)       ▷ val(i) & k(get, set),
    get(m)   & val(x)   ▷ val(x) & m(x),
    set(x,m) & val(y)   ▷ val(x) & m()
```

It is clear that `val` never escapes from this definition, and also that it will always have exactly one message available. Therefore in a shared-memory scenario, one might hope to optimise this to:

```
def @memcell(i,k)       ▷ (loc := i); k(get, set),
    get(m)              ▷ m(!x),
    set(x,m)            ▷ (loc := x); m()
```

where `loc` now refers to an (instance-local) ML-style memory location corresponding to the single `val` message. This offers two direct benefits:

- When matching, the value is simply read and there is no need to check for an available message.
- When sending a message, a write is performed which is far simpler than enqueuing a value on a lock-free queue.

Unfortunately, this optimisation is not as straightforward as it first seems. There are two problems:

1. The semantics imply that transitions appear atomic to other transitions. If a compare-and-swap operation is added to the memory cell the problem becomes evident.

   ```
   def @memcell(i,k)       ▷ val(i) & k(get, set),
       get(m)     & val(x) ▷ val(x) & m(x),
       set(x,m)   & val(y) ▷ val(x) & m(),
       cas(x,y,m) & val(z) ▷ (if x = z then val(y)
                                         else val(x))
                            & m(z)
   ```

   The above approach would convert this to:

   ```
   def @memcell(i,k)       ▷ (loc := i); k(get, set),
       get(m)              ▷ m(!x),
       set(x,m)            ▷ (loc := x); m()
       cas(x,y,m)          ▷ (if x = !loc then (loc := y)
                                             else ());
                              m(!loc)
   ```

   It is clear that a worker executing the `set` rule is now in a data-race with another worker performing `cas`. This optimisation is therefore only applicable in *fast mode* instance implementations, where concurrency is not an issue.

2. According to the original calculus semantics, it would be expected that the optimised version is equivalent to (note the altered order of message emits):

   ```
   def @memcell(i,k)       ▷ k(get, set); (loc := i),
       get(m)              ▷ m(!x),
       set(x,m)            ▷ m(); (loc := x)
   ```

   Unfortunately, this is not the case. With the reordering, there is no guarantee that the continuations will see the updates to `loc`—this is even true in fast mode. Therefore in addition to knowing that the queue is bound as exactly one item long, it is also necessary to know that all emits to it are in a *head-position*. An emit instruction is said to be in a head-position if it occurs before any new matches could have been produced.

Fortunately, it is often possible to hoist emits to potential 'memory location' style channels so that they do occur in the head position, although this transformation is not provided by the Dovetail prototype.

Theoretically, it is always possible for a compiler to ensure that this property holds. This is a consequence of transition bodies not having any effect on global state. The code for the body of a transition can therefore be generated twice. The first version would only contain emits to these memory channels, and the second to everything else. In cases where it cannot be statically determined whether the channel value will be a memory channel or not, extra runtime checks would be necessary. One would expect the duplication to produce large amounts of dead code that can be deleted. If this were not the case and actual computation (for example, a loop) remained in duplicate, or if many runtime checks were necessary, then the costs of this transformation would quickly outweigh the benefit of replacing a cell channel with a memory channel.

4.4.1 Annotations

To support this scenario and ones like it, Dovetail supports five annotations on channels:

- `lower_bound(i)` specifying that the queue always contains at least i messages.
- `upper_bound(i)` specifying that the queue never contains more than i messages.
- `head` indicates that all emits to the channel are in a head-position.
- `cell` is equivalent to `lower_bound(0) upper_bound(1)`.
- `mem` is equivalent to `lower_bound(1) upper_bound(1) head`.

4.4.2 Implementation

Based on these annotations, different queue implementations are picked for each of the slow and fast modes of a channel. These are in turn called by the specialised emission functions. I force LLVM to inline these calls, so after optimisation the channel emission functions will simply contain the raw code intended with no calls. This often corresponds to the code that would have been written by hand. The actual interface that these queue implementations conform to (Figure 4.4) is much like that of a standard queue, however in the case of 'slow' queues there are extra methods to allow for the two-phase commit style of matching.

Cells

These can be implemented as a memory location along with a status field. To support the two-phase matching algorithm (used in 'slow' mode), this can take the same three statuses as messages in Turon and Russo's work [113]—i.e. pending, claimed or consumed. Furthermore, any update of the status field from pending to claimed must be performed with an atomic compare-and-swap. In fast mode, this is simplified to a simple flag indicating whether the cell is full or empty, and operations on this no longer need to be atomic.

Slow Mode

```
void init      (chan*, sizeof(T))
msg* allocate  (chan*, sizeof(T))
T*   data      (chan*, msg*)
void enqueue   (chan*, msg*)
msg* find      (chan*, bool* retry)
bool try_claim (chan*, msg*)
void revert    (chan*, msg*)
void consume   (chan*, msg*)
void is_consumed(chan*, msg*)
```

Fast Mode

```
void init   (chan*, sizeof(T))
msg* enqueue(chan*, sizeof(T))
T*   data   (chan*, msg*)
msg* find   (chan*, sizeof(T))
void consume(chan*, msg*, sizeof(T))
```

Figure 4.4: Queue implementation interfaces.

Memory Locations in Fast Mode

Rather than actually removing the channel, I provide an implementation of the standard queue interface that often does not need to do anything (e.g. it never has to check whether the queue is empty). Visual inspection has confirmed that, after running LLVM's optimisations, this approach does collapse down to the raw reads and writes without extra instructions.

Results

Figure 4.5 shows the effect of these bounded queue implementations on performance. It is clear that both of these optimisations are worth making. The main benefit in both cases is in preventing memory allocation for each message send. Memory locations offer a smaller further benefit over cells since they apply to a limited selection of channels, and also only make any difference in fast mode.

4.5 Inlining

Something that has not been discussed at all thus far is inlining at the JCAM level. This is a key technique in the compilation of most programming languages, especially those making use of CPS or which are particularly fine-grained.

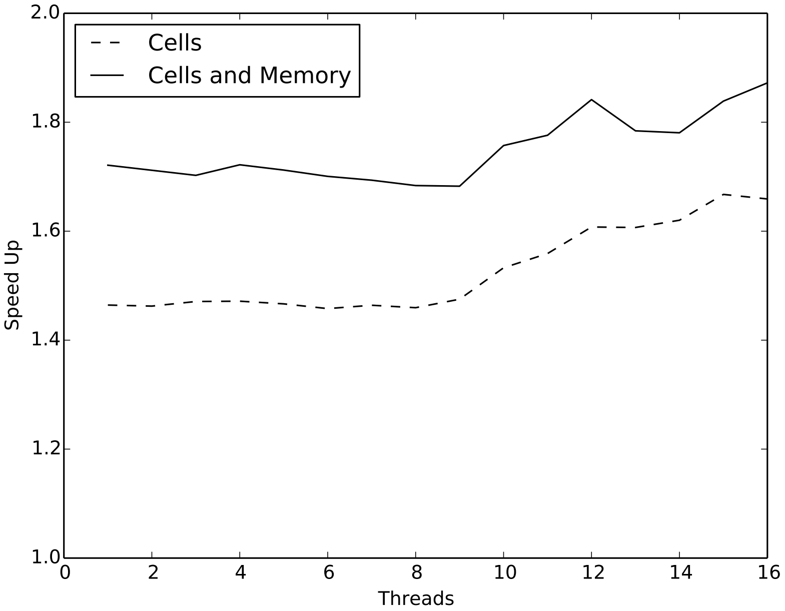


Figure 4.5: Performance of bounded queue implementations.

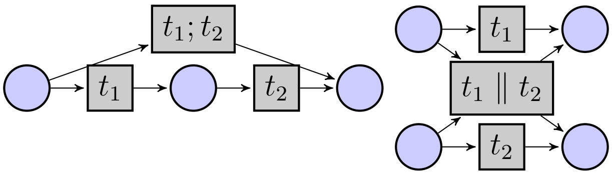


Figure 4.6: Fusing of Transition Rules.

4.5.1 Transition Inlining

The most direct equivalent in the JCAM to conventional function inlining is *transition inlining*. This involves fusing transitions together[8] (e.g. Figure 4.6), and can give multiple results for the same set of transitions (e.g. Figure 4.7). Performing such fusing may cause some of the explicit parallelism to be lost, so this optimisation would ideally be done at load-time or runtime when the target architecture is known.

It is also necessary to retain the original transitions in case the fused transition does not support all the behaviours—for example in Figure 4.6, if only t_1 ever fires, then replacing t_1 and t_2 with just $t_1 \parallel t_2$ alters program behaviour. With the originals retained,

[8] $t_1; t_2$ and $t_1 \parallel t_2$ give sequential and parallel compositions of t_1 and t_2 respectively.

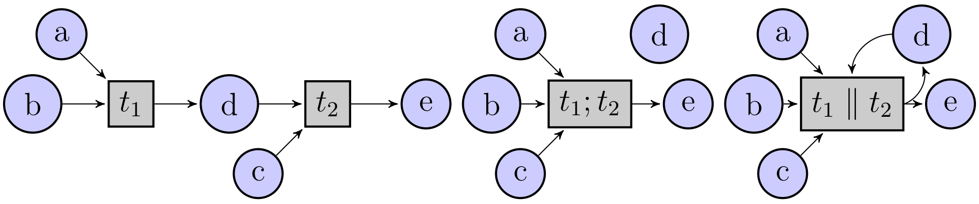


Figure 4.7: Multiple Fusings of Transitions

one must rely on a scheduler to pick the more efficient fused version in the hope that $\langle t_1; t_2\rangle < \langle t_1\rangle + \langle t_2\rangle$ or $\langle t_1 \parallel t_2\rangle < \langle t_1\rangle + \langle t_2\rangle$. For example, with the use of SIMD instructions it might be that $\langle t \parallel t\rangle = \langle t\rangle$. The simple work stealing scheduler in Dovetail is unable to make such choices, and therefore forbids the application of transition inlining. This concept is discussed further in Section 6.6.3.

4.5.2 Definition Inlining

A simpler technique that can be employed is *definition inlining*. This moves whole 'child' instances within a 'parent'. Assuming that channels are α-renamed to avoid conflicts, this clearly preserves semantics—just as two Petri-nets placed next to each other do not interact, the channels of the child do not affect the existing transition rules in the parent, or vice-versa. This transformation is another way to reduce the number of memory allocations that are required, and does not reduce the parallelism available. The transformation also allows transition inlining, which can only occur within a definition, to have a more global effect.

However, since inlining is a static optimisation, it is necessary to know statically that a given **construct** instruction is called at most once. Hence, one approach would be to only inline definitions instantiated within a constructor and outside a loop.[9]

The exact process is to copy non-constructor transitions into the parent definition (α-renaming to preserve globally unique channel names), and then to replace the **construct** instruction with the body of the child's constructor transition rule. Any **finish**es are replaced by branches to the successor of the original **construct**. For example, in the following code:

```
def @main()           ▷ @mutex(s)
    s(p,v)            ▷ ...

def @mutex(k)         ▷ free() & k(lock,unlock)
    lock(k) & free()  ▷ k()
    unlock(k)         ▷ k() & free()
```

inlining `@mutex` gives:

```
def @main()           ▷ free() & s(lock, unlock)
    s(p,v)            ▷ ...
    lock(k) & free()  ▷ k()
    unlock(k)         ▷ k() & free()
```

Applying this approach once to the `fib` benchmark by hand offers a small benefit (Figure 4.8). It also enables transition inlining which increases the performance improvement further.[10]

Whilst not relevant to this implementation, one issue with this approach is that it does enlarge the possible state space of the parent definition. With some implementation techniques (e.g. the `join` language in Section 2.4.2), the states considered grow exponentially with the number of channels. Ideally, the total state space would remain constant,

[9] It may therefore be beneficial to unroll loops before this optimisation.

[10] In the case of `fib`, some transition inlining can be done without risk of deadlock or reduced parallelism.

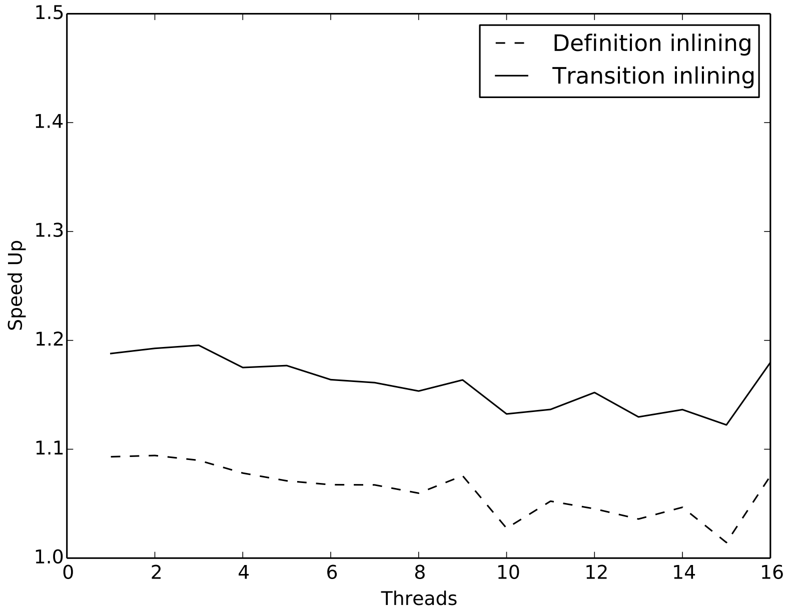


Figure 4.8: Performance benefit of inlining for `fib`.

with part of it simply being transferred from the child definition instance to the parent. This could be achieved by considering *disjoint sets* of channels. Initially, all channels are considered disjoint, but whenever two channels appear in a join pattern together, their respective sets must be unioned. The state spaces of the disjoint sets can then be considered separately avoiding the state explosion (since the inlined channels will clearly be disjoint from all others).

4.6 Summary

In this chapter, I have shown how the performance of the join calculus primitives can be improved. This is done using a mix of existing techniques, and also by taking advantage of opportunities given by the compilation approach. I have highlighted two types of annotation that enable significantly better performance in some common cases. Specifically, the annotations are *closedness* (Section 4.3) and *queue bounds* (Section 4.4). For some of the benchmarks in Chapter 6, I even found that these were required to prevent uncontrollable runtimes and memory usage (in a similar way to functional languages that rely heavily on tail-call optimisation). The next chapter shows how these can be inferred automatically. Chapter 6 will provide a more in-depth evaluation of the implementation's performance over a range of larger benchmarks, rather than the micro-benchmarking used in this chapter. This will show that the overheads of using the join calculus to represent all parts of a program, rather than just coordination as in previous work, are not prohibitive.

Chapter 5

Control Flow Analysis

The previous chapter demonstrated how, with suitable annotations, JCAM programs can be executed with competitive performance. However, in reality these annotations are unlikely to be present, and in any case it would be preferable if programmers and language-frontends need not provide them. Inferring these properties requires *inter-procedural* analysis to understand the message interaction characteristics of the programs. This chapter will show how classical *control flow analysis* (CFA) can be adapted to the join calculus, and how its results can detect both *closed definitions* and bounds on queue lengths.

CFA is traditionally used for inlining of indirect function calls. However, inlining becomes more complex in the join calculus and is beyond the scope of this work. An overview of the issues involved in *transition inlining* is given in Section 6.6.3.

I start with a straightforward translation (Section 5.1) of constraint-based 0-CFA for functional languages, and discuss its shortcomings, before moving onto two more sophisticated analyses. The first of these, 0-LCFA (Section 5.2), introduces the notion of an *instance-local* analysis which can deal with the richer way channels interact in the join calculus compared to call-return in imperative languages. The second, k-LCFA (Section 5.3), tries to improve accuracy in a similar way to higher-order approaches for functional languages by considering the program history. As discussed in Section 3.3.2, join calculus histories are more complex than the linear call-strings applicable to sequential languages, and best thought of as DAGs. My approach therefore requires a novel abstraction of these. Its correctness is shown in Section 5.4.

Finally, Sections 5.5 and 5.6 show how the queue bound and closedness annotations required by Chapter 4 can be inferred from the results of control-flow analysis.

5.1 Translating 0-CFA to the Join Calculus

Abstract interpretation (see Section 2.5) has been successfully used for concurrent versions of the λ-calculus [75]. However, in the presence of non-deterministic join-pattern matching, ensuring that a direct abstraction of the concrete semantics considers all cases is rather more difficult. In particular, the idea of an abstract machine state does not fit easily with the fact that both future and past message sends can interact with the one being analysed.

Constraints, on the other hand, seem a natural choice, since they can describe all possible executions, before being solved to give an actual solution. As far as possible, I

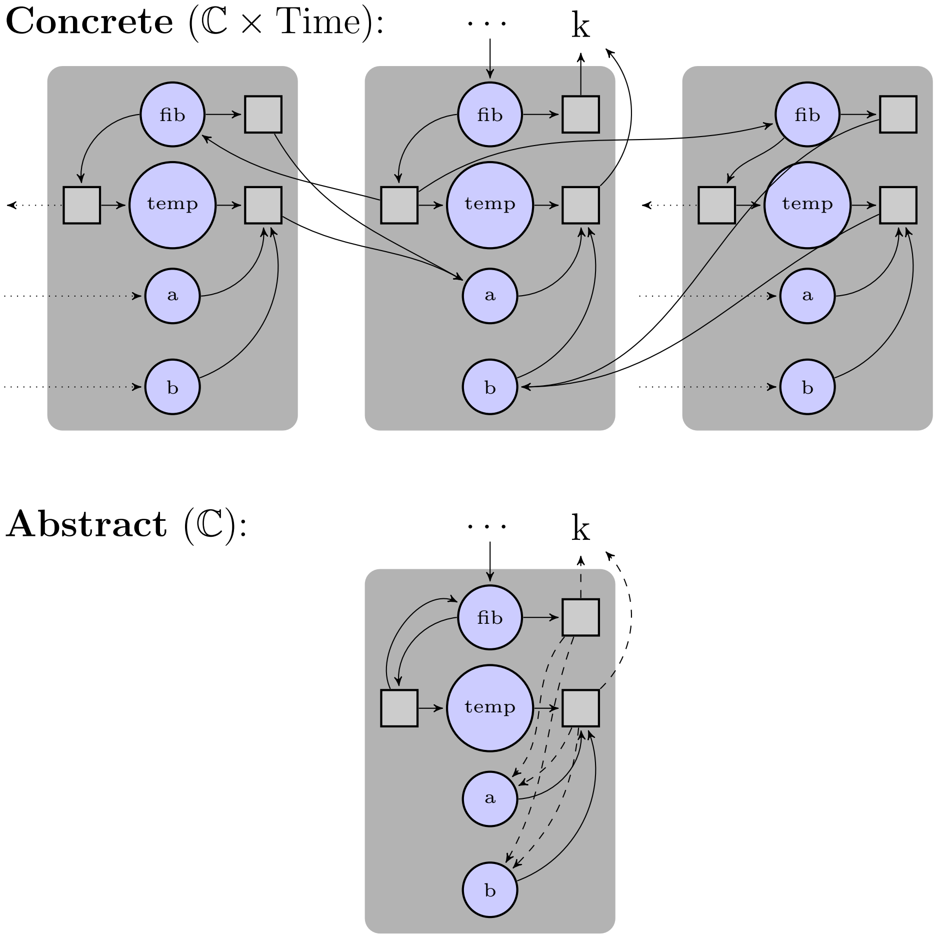


Figure 5.1: Illustration of 0-CFA abstraction for `fib`.

adopt the notation used by Faxén's *polymorphic analysis* [38], combined with the lexical convention that *hatted* names represent abstract domains and values. By starting with a zeroth-order analysis of the JCAM, I hope to provide a gentle introduction to analysis of the join calculus in the context of a well-known approach.

Recall that each channel value in the concrete semantics (Figure 3.13) is taken from the set $\mathbb{C} \times \text{Time}$, where the time component is used as an identifier. For this straightforward abstraction, I discard the instance identifier. This effectively conflates all join calculus definition instances, as is done for the different environments that a closure might receive in a λ-calculus 0-CFA—i.e. when the same flow variables are used for a function at every call-site. For a simple Fibonacci example such as in Figure 3.16, this simplification can be viewed as shown in Figure 5.1. As normal, values for variables in the local environment of the concrete semantics are abstracted to sets of 0-CFA flow values, although I flip between this and the isomorphic function-form that I can refine later. My usages of CFA are only interested in channel values, so all primitives[1] are abstracted to PRIM.

$$\begin{aligned} \hat{f} &\in \widehat{\text{ChannelValue}} = \mathbb{C} \\ \hat{v} &\in \widehat{\text{Value}} = \mathcal{P}(\widehat{\text{ChannelValue}} \cup \{\texttt{PRIM}\}) \\ &\cong (\widehat{\text{ChannelValue}} \cup \{\texttt{PRIM}\}) \to \{\bot, \top\} \end{aligned}$$

I use c to range over $\widehat{\text{ChannelValue}} \cup \{\texttt{PRIM}\}$, while f only ranges over channel values. $\widehat{\text{Value}}$ inherits the order $\bot \sqsubseteq \top$, which in the set view corresponds to $\subseteq$. However, rather than using these values directly, I use flow variables as is typical in a constraint-based approach, along with a constraint set S over these. The local environment that represents intermediate values within a firing is abstracted to $\hat{\rho}$. A constant mapping $\hat{\Gamma} : \mathbb{C} \to \text{FlowVar}^*$ associates a tuple of flow variables with each channel (i.e. one per argument to the channel), and is equivalent to the global environment Γ. These represent all possible values that might be sent in messages to the channel.

The possible constraints are the same as in Faxén's CFA (Section 2.5), except the application constraint $\alpha_1 \mapsto \alpha_2 \subseteq \alpha_3$ is replaced by an *emission constraint* $\bar{\alpha} \mapsto \beta$.

$$\alpha_1 \succeq \alpha_2 \quad | \quad \alpha \succeq \{c\} \quad | \quad \bar{\alpha} \mapsto \beta$$

Constraints are built using the rules in Figure 5.2. The emission constraints generated for **emit** and **construct** instructions can be read as saying that any tuple of values represented by the flow variables $\bar{\alpha}$ could be used to send a message to any of the channel values in β. New flow variables are allocated by $\exists\alpha$, and since there may be cycles in the control-flow graph of rule bodies, an implementation will need to reuse flow variables to ensure termination. Typically, 0-CFA allocates one per program point (i.e. $\exists\alpha$ is treated as α_l at program point l, along with splitting for different variables). However, within a transition body it is trivial to use call strings to make use of different flow variables depending on the path through loops and conditionals. The BR rule is shown as an example of considering intra-transition control flow, but is omitted from the later analyses to save space as it does not change. The constraint set S is then defined as the least set that satisfies the following for each rule $\overline{f_i(\overline{x_i})}\{\bar{b}\}$ in the program.

$$S, \hat{\Gamma} \vdash l_0, \bar{c}, [\overline{x_i} \mapsto \hat{\Gamma}(f_i) \mid 1 \leq i \leq n], \bar{b} \text{ where } b_0 = (l_0, [], \bar{c})$$

Solutions to the analysis are of the form $\Phi : \text{FlowVar} \to \widehat{\text{Value}}$. Figure 5.2 also defines what it means for such a Φ to be a valid model of the constraints, and gives a dynamic transitive closure algorithm for computing S^+. Given S^+, the (least) solution can be read off as:

$$\Phi(\alpha) = \{c \mid (\alpha \succeq \{c\}) \in S^+\}$$

5.2 Dealing with Message Interaction: 0-LCFA

Whilst 0-CFA is useful for functional languages, it is often insufficient for the join calculus as it cannot differentiate between different channel instances. In particular, the firing semantics only allows two messages to interact when they belong to the same instance—for example, P never fires in the following code but might be thought to by 0-CFA:

[1] This analysis is based on a JCAM that does not support structures or arrays. Adding these would not be difficult but would cloud the presentation.

Constraint Syntax:

$$S \subseteq \text{Constraint} ::= \alpha_1 \succeq \alpha_2 \mid \alpha \succeq \{c\} \mid \bar{\alpha} \mapsto \beta \qquad \text{where } c \in \widehat{\text{ChannelValue}} \cup \{\texttt{PRIM}\}$$

Constraint Generation Rules: (with judgement form $S, \hat{\Gamma} \vdash l, \hat{\sigma}$)

$$\frac{}{\{\}, \hat{\Gamma} \vdash l, [\textbf{finish}], \_, \_}\ (\textsc{finish})$$

$$\frac{S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}, \bar{b}}{\{\hat{\rho}(\bar{y}) \mapsto \hat{\rho}(x)\} \cup S, \hat{\Gamma} \vdash l, (\textbf{emit}\ x(\bar{y})) \cdot \bar{c}, \hat{\rho}, \bar{b}}\ (\textsc{emit})$$

$$\frac{\exists \alpha \qquad S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}, \bar{b}}{\{\hat{\rho}(\bar{y}) \mapsto \alpha, \alpha \succeq \{f\}\} \cup S, \hat{\Gamma} \vdash l, (\textbf{construct}\ f(\bar{y})) \cdot \bar{c}, \hat{\rho}, \bar{b}}\ (\textsc{construct})$$

$$\frac{\exists \alpha \qquad S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}[x \mapsto \alpha], \bar{b}}{\{\alpha \succeq \{f\}\} \cup S, \hat{\Gamma} \vdash l, (x = \textbf{load.channel}\ f) \cdot \bar{c}, \hat{\rho}, \bar{b}}\ (\textsc{load.channel})$$

$$\frac{(l_1, \bar{\phi}, \bar{c}) \in \bar{b} \qquad \hat{\rho}' = \hat{\rho}[x \mapsto \hat{\rho}(y) \mid (x = \textbf{phi}\ \bar{v}) \in \bar{\phi} \wedge (y, l_0) \in \bar{v}] \qquad S, \hat{\Gamma} \vdash l_1, \bar{c}, \hat{\rho}', \bar{b}}{S, \hat{\Gamma} \vdash l_0, [\textbf{br}\ l_1], \hat{\rho}, \bar{b}}\ (\textsc{br})$$

Model of Constraints: ($\Phi, \hat{\Gamma} \models S$ iff $\Phi, \hat{\Gamma} \models s$ for all $s \in S$)

$$\begin{aligned}
\Phi, \hat{\Gamma} \models \alpha_1 \succeq \alpha_2 &\iff \Phi(\alpha_1) \sqsupseteq \Phi(\alpha_2) \\
\Phi, \hat{\Gamma} \models \alpha \succeq \{c\} &\iff c \in \Phi(\alpha) \\
\Phi, \hat{\Gamma} \models \bar{\alpha} \mapsto \beta &\iff f \in \Phi(\beta) \implies \Phi, \hat{\Gamma} \models \mathcal{I}(\hat{\Gamma} \mid f, \bar{\alpha})
\end{aligned}$$

Closure of Constraint Sets: $S^+ \supseteq S$

$$\begin{aligned}
\{\alpha_1 \succeq \alpha_2, \alpha_2 \succeq \{c\}\} \subseteq S^+ &\implies \{\alpha_1 \succeq \{c\}\} \subseteq S^+ \\
\{\bar{\alpha} \mapsto \beta, \beta \succeq \{f\}\} \subseteq S^+ &\implies \mathcal{I}(\hat{\Gamma} \mid f, \bar{\alpha}) \subseteq S^+
\end{aligned}$$

Instantiation of Abstract Channel Values:

$$\mathcal{I}(\hat{\Gamma} \mid f, \bar{\alpha}) = \forall i. \{\hat{\Gamma}(f)_i \succeq \alpha_i\}$$

Figure 5.2: Definition of 0-CFA

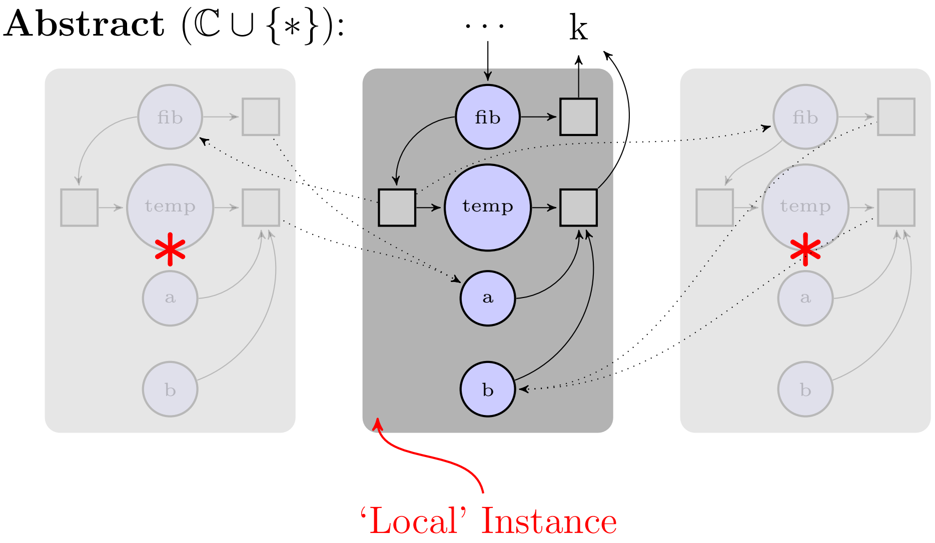


Figure 5.3: Illustration of 0-LCFA for `fib`.

```
def @test(k)             ▷ k(a,b),
    a() & b()            ▷ P

def @main()              ▷ @test(m) & @test(n),
    m(a1,b1) & n(a2,b2)  ▷ a1() & b2()
```

There is no need for such discrimination in functional languages (i.e. predicting whether two closures share the *same* environment, rather than two environments that bind the same values to variables). This is because a functional language's environment is never mutated, unlike the join calculus where messages will be added to and removed from channels.

To do so, it is necessary to abstract the timestamps allocated by **construct**. In past techniques, it is typical to use a call-site history of depth k, in place of the unbounded concrete call string to give instance identifiers.[2] However, in the join calculus, call strings are replaced by more complex traces based on pomsets, or DAGs, as introduced in Section 3.3.2. Forming an abstract version of these *call-DAGs* is further complicated by the non-deterministic choice of messages that is made when a transition fires.

In Section 5.3, I show how to abstract call-DAGs for the purposes of accuracy. However, that technique is not suitable for comparing abstract instances (i.e. it cannot imply either equality or inequality of concrete instances). Instead, I use a naïve refinement that considers two *abstract times*: definitely 'this' instance (i.e. *local*); and possibly another instance. I call the resultant analysis *zeroth-order local CFA* (0-LCFA), which is similar in many ways to the technique used by Reppy and Xiao [93] for Concurrent ML, although it does not make use of type-sensitivity. It can be seen pictorially in Figure 5.3.

[2] Call strings can also improve accuracy (k-CFA, see Section 5.3). However, this and identifying instances are two distinct problems, and for the join calculus I solve them separately.

Channel values are abstracted as follows: a *local* channel value is abstracted as its channel name,[3] ranged over by f as before (discarding the time component); other channel values (either from other definitions or another instance of this definition) are abstracted to a *wildcard* $*$. This wildcard also represents local channels that have escaped the instance and might then be passed back in. It is therefore very similar to the *unknown* abstract value $\top$ used in Serrano's technique [103], and also the concept of a 'most general attacker' in security. However, it does not represent local channels that do not escape. Clearly determining whether a channel escapes is undecidable, but simply requiring that the solution to the analysis is self-consistent results in a safe approximation. Primitive values are not of interest, so for simplicity I represent these by $*$ too—this conveniently captures the fact that other definition instances are able to fabricate any primitive value they wish. The abstract value set therefore changes to:

$$\begin{aligned}\hat{s} \in \widehat{\text{ChannelValue}} &= \mathbb{C} \\ \hat{v} \in \widehat{\text{Value}} &= \mathcal{P}(\widehat{\text{ChannelValue}} \cup \{*\}) \\ &\cong (\widehat{\text{ChannelValue}} \cup \{*\}) \to \{\bot, \top\}\end{aligned}$$

The updated analysis (Figure 5.4) requires a new constraint generation rule for **construct** as well as changes to the model and closure algorithm for emission constraints. These simply ensure that whenever a local channel value may escape to another instance, its $\hat{\Gamma}$ flow variables are updated to include $*$ for each of its arguments. The initial conditions of the analysis must also include the following constraints, which specify that constructors may receive any external value:

$$\forall f \in \text{Constructor}.\{\hat{\Gamma}(f)_i \succeq \{*\} \mid 1 \leq i \leq \text{arity}(f)\} \subseteq S$$

Unlike 0-CFA, this new analysis can be used for queue bounding (Section 5.5), will assist in detecting closed definitions (Section 5.6), and would also be applicable to inlining.

5.3 Abstracting Call-DAGs: k-LCFA

The limitations of my approach so far are the same as those of other zeroth-order and monovariant approaches for the λ-calculus. For example, in the following λ-calculus program, 0-CFA cannot determine that `x` equals `f` and `y` equals `g`—instead thinking that both calls to `id` could return either `f` or `g`.

```
let id = λ k . k in
let x = id f in
let y = id g in ...
```

In the join calculus, this is illustrated by two small examples ('handshake' and 'handshake-with-swap' respectively):

```
A:  a(x,m) & b(y,n) ▷ m(x) & n(y)
B:  a(x,m) & b(y,n) ▷ m(y) & n(x)
```

[3] The abstract value f in 0-LCFA corresponds to a single concrete value (f, this) whereas it previously gave (f, θ) for all θ in 0-CFA.

Constraint Syntax:

$$S \subseteq \text{Constraint} ::= \alpha_1 \succeq \alpha_2 \mid \alpha \succeq \{c\} \mid \bar{\alpha} \mapsto \beta \qquad \text{where } c \in \widehat{\text{ChannelValue}} \cup \{*\}$$

Constraint Generation Rules: (with judgement form $S, \hat{\Gamma} \vdash l, \hat{\sigma}$)

$$\frac{}{\{\}, \hat{\Gamma} \vdash l, [\textbf{finish}], \_, \_}\ (\textsc{finish})$$

$$\frac{S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}, \bar{b}}{\{\hat{\rho}(\bar{y}) \mapsto \hat{\rho}(x)\} \cup S, \hat{\Gamma} \vdash l, (\textbf{emit}\ \ x(\bar{y})) \cdot \bar{c}, \hat{\rho}, \bar{b}}\ (\textsc{emit})$$

$$\frac{\exists \alpha \qquad S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}, \bar{b}}{\{\hat{\rho}(\bar{y}) \mapsto \alpha, \alpha \succeq \{*\}\} \cup S, \hat{\Gamma} \vdash l, (\textbf{construct}\ \ f(\bar{y})) \cdot \bar{c}, \hat{\rho}, \bar{b}}\ (\textsc{construct})$$

$$\frac{\exists \alpha \qquad S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}[x \mapsto \alpha], \bar{b}}{\{\alpha \succeq \{f\}\} \cup S, \hat{\Gamma} \vdash l, (x = \textbf{load.channel}\ \ f) \cdot \bar{c}, \hat{\rho}, \bar{b}}\ (\textsc{load.channel})$$

Model of Constraints: $(\Phi, \hat{\Gamma} \models S$ iff $\Phi, \hat{\Gamma} \models s$ for all $s \in S)$

$$\begin{aligned}
\Phi, \hat{\Gamma} \models \alpha_1 \succeq \alpha_2 \quad &\iff \quad \Phi(\alpha_1) \sqsupseteq \Phi(\alpha_2) \\
\Phi, \hat{\Gamma} \models \alpha \succeq \{c\} \quad &\iff \quad c \in \Phi(\alpha) \\
\Phi, \hat{\Gamma} \models \bar{\alpha} \mapsto \beta \quad &\iff \quad \Big(f \in \Phi(\beta) \implies \Phi, \hat{\Gamma} \models \mathcal{I}(\hat{\Gamma} \mid f, \bar{\alpha})\Big) \\
&\quad\ \wedge \quad \Big(* \in \Phi(\beta) \implies \forall f \in \textstyle\bigcup_i \Phi(\alpha_i). \Phi, \hat{\Gamma} \models \mathcal{E}(\hat{\Gamma} \mid f)\Big)
\end{aligned}$$

Closure of Constraint Sets: $S^+ \supseteq S$

$$\begin{aligned}
\{\alpha_1 \succeq \alpha_2, \alpha_2 \succeq \{c\}\} \subseteq S^+ &\implies \{\alpha_1 \succeq \{c\}\} \subseteq S^+ \\
\{\bar{\alpha} \mapsto \beta, \beta \succeq \{f\}\} \subseteq S^+ &\implies \mathcal{I}(\hat{\Gamma} \mid f, \bar{\alpha}) \subseteq S^+ \\
\{\alpha_i \succeq \{f\}, \bar{\alpha} \mapsto \beta, \beta \succeq \{*\}\} \subseteq S^+ &\implies \mathcal{E}(\hat{\Gamma} \mid f) \subseteq S^+
\end{aligned}$$

Instantiation of Abstract Channel Values:

$$\begin{aligned}
\mathcal{I}(\hat{\Gamma} \mid f, \bar{\alpha}) &= \forall i. \{\hat{\Gamma}(f)_i \succeq \alpha_i\} \\
\mathcal{E}(\hat{\Gamma} \mid f) &= \forall i. \{\hat{\Gamma}(f)_i \succeq \{*\}\}
\end{aligned}$$

Figure 5.4: Definition of 0-LCFA (changes highlighted)

Consider the calls `a(i,p)`, `a(j,q)`, `b(k,r)` and `b(l,s)`. The table below indicates the results of 0-LCFA, compared to the optimum—i.e. what could actually occur on a real execution:

	Example A		Example B	
	0-LCFA	Optimum	0-LCFA	Optimum
`p`	$\{i,j\}$	$\{i\}$	$\{k,l\}$	$\{k,l\}$
`q`	$\{i,j\}$	$\{j\}$	$\{k,l\}$	$\{k,l\}$
`r`	$\{k,l\}$	$\{k\}$	$\{i,j\}$	$\{i,j\}$
`s`	$\{k,l\}$	$\{l\}$	$\{i,j\}$	$\{i,j\}$

The case where the optimum solution is not attained is A. However, in some non-trivial situations the simple approach does as well as possible. As expected, the inaccuracy is due to arguments passed from different call-sites being conflated. It is this issue that I now address, while still allowing for the non-deterministic combination of call-sites (as exemplified by case B above).

In zeroth-order approaches, the problem is our simple approximation of a single flow variable per channel argument, as given by $\hat{\Gamma}$. More accurate k-CFA approaches for the λ-calculus refine this 'global' variable into a set of variables, indexed by the last k call-sites as introduced in Section 2.5.3. However, as already discussed, the join calculus, and therefore the JCAM, gives call-DAGs rather than call strings. Furthermore, they include non-deterministic choices wherever different messages could have been combined.

My approach is to continue to calculate zeroth-order results for the flow variables in $\hat{\Gamma}$, and then use these as *background* information while following each possible (*foreground*) path of the DAG. Along these foreground paths, it is possible to use the standard call-string abstraction to improve accuracy. The trick is to ensure that the inaccurate background information is overridden by the more accurate constraints generated by the *foreground path.* I arrange that the union of the analyses for each path gives a suitable result for the whole DAG. In order to do this, the abstract value domain is further refined to tag each value:

$$\hat{v} \in \widehat{\text{Value}} = (\widehat{\text{ChannelValue}} \cup \{*\}) \to \{\bot, \mathcal{B}, \mathcal{F}\}$$

The ordering $\bot \sqsubseteq \mathcal{B} \sqsubseteq \mathcal{F}$ ensures the $\mathcal{F}$ tag takes priority, since I want the analysis to take more notice of foreground values. Therefore, at a merge point with both foreground and background versions of a value, only the foreground tag will propagate. For convenience, I continue to use set-style notation, with the tag given by annotations—for example:

$$\{\mathcal{F}(c)\} \equiv \lambda x. \begin{cases} \mathcal{F} & \text{if } x = c \\ \bot & \text{otherwise} \end{cases}$$

Note $\{\mathcal{B}(c)\} \sqsubseteq \{\mathcal{F}(c)\}$ and, less obviously, that x ranges over both a and b in:

$$\forall \mathcal{B}(x) \in \{\mathcal{F}(a), \mathcal{B}(b)\}.\ \ldots$$

Figure 5.6 presents the new analysis. Values always start off as being tagged $\mathcal{F}$, and it is only the emission constraint, where values are added to $\hat{\Gamma}$, that later lowers them to $\mathcal{B}$.

Much as Faxén's *polymorphic* analysis of λ-abstractions constructed constraints for the function's body, the **load.channel** instruction now saves constraints for rules matching on

$$
\begin{aligned}
(S, \hat{\Gamma} \mid f_\bullet : \bar{\gamma} \mapsto \diamond) \in \mathcal{W} &\iff \forall(\overline{f_i(\overline{x_i})\{\bar{b}\}}) \in \mathbb{T}.\ \forall i.\ f_i = f_\bullet \implies \\
&\qquad S, \hat{\Gamma} \vdash l_0, \bar{c}, [\overline{x_j} \mapsto \hat{\Gamma}(f_j) \mid j \neq i] + [\overline{x_i} \mapsto \bar{\gamma}], \bar{b} \\
&\qquad\qquad \text{where } b_0 = (l_0, [], \bar{c}) \\
(S, \hat{\Gamma} \mid f_\bullet : \bar{\gamma} \mapsto \diamond) \in \mathcal{W}_f &\iff (S, \hat{\Gamma} \mid f_\bullet : \bar{\gamma} \mapsto \diamond) \in \mathcal{W} \wedge f_\bullet = f
\end{aligned}
$$

Figure 5.5: Sets of valid k-LCFA flow closures.

the channel. This corresponds with a change to the abstract channel values $\widehat{\text{ChannelValue}}$ to make them more like Faxén's *flow closures*:[4]

$$w \in \widehat{\text{ChannelValue}} ::= (S, \hat{\Gamma} \mid f : \bar{\gamma} \mapsto \diamond)$$

Of course, not all such values are valid in the context of a specific program as the constraints S must correspond to the rules that match on f. The set of valid channel values is denoted $\mathcal{W}$, and the valid values for a specific channel f as $\mathcal{W}_f$. Membership of these sets is defined in Figure 5.5.

Examining the model of the emission constraints, note first that it only has any effect for destination values tagged with $\mathcal{F}$. This prevents the background $\mathcal{B}$ values causing inaccuracy. The background part of the instantiation of abstract channels $\mathcal{I}$ and treatment of escaping $\mathcal{E}$ state similar requirements to my 0-LCFA. The $\hat{\Gamma}$ flow variables are predominantly made up of $\mathcal{B}$ values, since these are used to give values to channel arguments not in the current foreground path. The exception is when a channel f escapes the instance, then $\mathcal{F}(*)$ is added to each $\hat{\Gamma}(f)_j$ since the $*$ values are not attributable to any particular call-site, so will not be considered on a foreground path. For this reason, it is still required that the following holds for each rule $\overline{f_i(\overline{x_i})}\{\bar{b}\}$, even though this typically generates very few constraints directly:

$$S, \hat{\Gamma} \vdash l_0, \bar{c}, [\overline{x_i} \mapsto \hat{\Gamma}(f_i) \mid 1 \leq i \leq n], \bar{b} \text{ where } b_0 = (l_0, [], \bar{c})$$

As before, I pass $*$ (actually $\mathcal{F}(*)$) to the entry points of each definition:

$$\forall f \in \text{Constructor}.\ \forall i.\ \{\hat{\Gamma}(f)_i \succeq \{\mathcal{F}(*)\}\} \subseteq S$$

The foreground instantiation half of $\mathcal{I}$ for the emission constraints is new, and performs the analysis along the foreground path. The $\exists\sigma$, which is used in both the model and closure algorithm, is responsible for choosing a substitution with new flow variables, and it is here that the choice of k affects an implementation, as it reuses flow variables for emissions with common foreground history $h \in \text{Label}^{\leq k}$. Although not shown, each emission constraint is implicitly associated with such a foreground call-string. Note that σ may not perform substitutions on flow variables returned by $\hat{\Gamma}$.

The dynamic transitive closure algorithm also changes to accommodate the alterations. In particular, it may introduce a new form of constraint that corresponds to raising tags from $\mathcal{B}$ to $\mathcal{F}$, and lowering them the other way.

As already seen, my 0-LCFA and this k-LCFA essentially perform a form of *escape-analysis*. However, if the results of the k-LCFA for the `memcell` example are calculated,

[4] The $\bar{\gamma} \mapsto \diamond$ syntax is used to make it clear that $\hat{\gamma}$ corresponds to arguments to the channel.

Constraint Syntax:

$$S \subseteq \text{Constraint} ::= \alpha_1 \succeq \mathcal{F}(\alpha_2) \mid \alpha_1 \succeq \mathcal{B}(\alpha_2) \mid \alpha \succeq \{\mathcal{F}(c)\} \mid \alpha \succeq \{\mathcal{B}(c)\} \mid \bar{\alpha} \mapsto \beta$$

Constraint Generation Rules: (with judgement form $S, \hat{\Gamma} \vdash l, \hat{\sigma}$)

$$\frac{}{\{\}, \hat{\Gamma} \vdash l, [\textbf{finish}], \_, \_}\ (\textsc{finish})$$

$$\frac{S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}, \bar{b}}{\{\hat{\rho}(\bar{y}) \mapsto \hat{\rho}(x)\} \cup S, \hat{\Gamma} \vdash l, (\textbf{emit}\ x(\bar{y})) \cdot \bar{c}, \hat{\rho}, \bar{b}}\ (\textsc{emit})$$

$$\frac{\exists \alpha \qquad S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}, \bar{b}}{\{\hat{\rho}(\bar{y}) \mapsto \alpha, \alpha \succeq \{\mathcal{F}(*)\}\} \cup S, \hat{\Gamma} \vdash l, (\textbf{construct}\ f(\bar{y})) \cdot \bar{c}, \hat{\rho}, \bar{b}}\ (\textsc{construct})$$

$$\frac{\exists \alpha, w \qquad S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}[x \mapsto \alpha], \bar{b} \qquad w \in \mathcal{W}_f}{\{\alpha \succeq \{\mathcal{F}(w)\}\} \cup S, \hat{\Gamma} \vdash l, (x = \textbf{load.channel}\ f) \cdot \bar{c}, \hat{\rho}, \bar{b}}\ (\textsc{load.channel})$$

Model of Constraints: ($\Phi, \hat{\Gamma} \models S$ iff $\Phi, \hat{\Gamma} \models s$ for all $s \in S$)

$$\begin{array}{rcl}
\Phi, \hat{\Gamma} \models \alpha_1 \succeq \mathcal{F}(\alpha_2) & \iff & \Phi(\alpha_1) \sqsupseteq \{\mathcal{F}(x) \mid \mathcal{B}(x) \in \Phi(\alpha_2)\} \\
\Phi, \hat{\Gamma} \models \alpha_1 \succeq \mathcal{B}(\alpha_2) & \iff & \Phi(\alpha_1) \sqsupseteq \{\mathcal{B}(x) \mid \mathcal{B}(x) \in \Phi(\alpha_2)\} \\
\Phi, \hat{\Gamma} \models \alpha \succeq \{\mathcal{F}(c)\} & \iff & \mathcal{F}(c) \in \Phi(\alpha) \\
\Phi, \hat{\Gamma} \models \alpha \succeq \{\mathcal{B}(c)\} & \iff & \mathcal{B}(c) \in \Phi(\alpha) \\
\Phi, \hat{\Gamma} \models \bar{\alpha} \mapsto \beta & \iff & \big(\mathcal{F}(w) \in \Phi(\beta) \implies \Phi, \hat{\Gamma} \models \mathcal{I}(w, \bar{\alpha})\big) \\
 & \wedge & \big(\mathcal{F}(*) \in \Phi(\beta) \implies \forall \mathcal{B}(w) \in \bigcup_i \Phi(\alpha_i) . \Phi, \hat{\Gamma} \models \mathcal{E}(w)\big)
\end{array}$$

Closure of Constraint Sets: $S^+ \supseteq S$

$$\begin{array}{rcl}
\{\alpha_1 \succeq \mathcal{F}(\alpha_2), \alpha_2 \succeq \{\mathcal{F}(c)\}\} \subseteq S^+ & \implies & \{\alpha_1 \succeq \{\mathcal{F}(c)\}\} \subseteq S^+ \\
\{\alpha_1 \succeq \mathcal{F}(\alpha_2), \alpha_2 \succeq \{\mathcal{B}(c)\}\} \subseteq S^+ & \implies & \{\alpha_1 \succeq \{\mathcal{F}(c)\}\} \subseteq S^+ \\
\{\alpha_1 \succeq \mathcal{B}(\alpha_2), \alpha_2 \succeq \{\mathcal{F}(c)\}\} \subseteq S^+ & \implies & \{\alpha_1 \succeq \{\mathcal{B}(c)\}\} \subseteq S^+ \\
\{\alpha_1 \succeq \mathcal{B}(\alpha_2), \alpha_2 \succeq \{\mathcal{B}(c)\}\} \subseteq S^+ & \implies & \{\alpha_1 \succeq \{\mathcal{B}(c)\}\} \subseteq S^+ \\
\{\bar{\alpha} \mapsto \beta, \beta \succeq \{\mathcal{F}(w)\}\} \subseteq S^+ & \implies & \mathcal{I}(w, \bar{\alpha}) \subseteq S^+ \\
\{\alpha_i \succeq \{\mathcal{B}(w)\}, \bar{\alpha} \mapsto \beta, \beta \succeq \{\mathcal{F}(*)\}\} \subseteq S^+ & \implies & \mathcal{E}(w) \subseteq S^+ \\
\{\alpha_i \succeq \{\mathcal{F}(w)\}, \bar{\alpha} \mapsto \beta, \beta \succeq \{\mathcal{F}(*)\}\} \subseteq S^+ & \implies & \mathcal{E}(w) \subseteq S^+
\end{array}$$

Instantiation of Abstract Channel Values:

$$\mathcal{I}((S, \hat{\Gamma} \mid f : \bar{\gamma} \mapsto \diamond), \bar{\alpha}) = \exists \sigma . S\sigma \cup \forall i . \{\sigma(\gamma_i) \succeq \mathcal{F}(\alpha_i), \hat{\Gamma}(f)_i \succeq \mathcal{B}(\alpha_i)\}$$

$$\mathcal{E}((S, \hat{\Gamma} \mid f : \bar{\gamma} \mapsto \diamond)) = \forall i . \{\hat{\Gamma}(f)_i \succeq \{\mathcal{F}(*)\}\}$$

Figure 5.6: Definition of k-LCFA (main changes highlighted)

it is found that all three channels (get, set and val) receive * (i.e. external values) for each of their arguments. Whilst this is correct, I would like to distinguish between get (or set), which could be called any number of times from outside the definition, and val, which is only called internally, despite receiving foreign values via set. I achieve this by also constructing an escape set $E \supseteq \text{Constructor}$, which is the minimal set satisfying:

$$(\bar{\alpha} \mapsto \beta) \in S \wedge \mathcal{F}(*) \in \Phi(\beta) \implies \forall \mathcal{B}(S, \hat{\Gamma} \mid f : \bar{\gamma} \mapsto \diamond) \in \bigcup_i \Phi(\alpha_i).\ f \in E$$

This is computed by initialising E to Constructor and closing under:

$$\{\alpha_i \succeq \{\mathcal{B}(S, \hat{\Gamma} \mid f : \bar{\gamma} \mapsto \diamond)\},\ \bar{\alpha} \mapsto \beta,\ \beta \succeq \{\mathcal{F}(*)\}\} \subseteq S^+ \implies f \in E$$
$$\{\alpha_i \succeq \{\mathcal{F}(S, \hat{\Gamma} \mid f : \bar{\gamma} \mapsto \diamond)\},\ \bar{\alpha} \mapsto \beta,\ \beta \succeq \{\mathcal{F}(*)\}\} \subseteq S^+ \implies f \in E$$

The escape set E is useful for queue bounding, closedness detection and proving the k-LCFA technique to be sound with respect to the concrete semantics. The proof itself is given in the next section.

Returning to the examples presented earlier, this novel approach overcomes the inaccuracy of conflating call-sites while still allowing for the firing semantics. For the functional subset of the join calculus,[5] my approach collapses to conventional k-CFA for a CPS lambda-lifted λ-calculus. In particular, * represents only primitives when there is just a single instance, and if all rules are functional then it never makes use of $\hat{\Gamma}$ and always deals with $\mathcal{F}$ values. Indeed, the **load.channel** rule collapses as follows to a rule very similar to Faxén's (see Section 2.5). The remaining differences are due to the imperative nature of the JCAM rather than the join calculus itself.

$$\frac{S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}[x \mapsto \alpha], \bar{b} \qquad \begin{array}{c} \exists \alpha, S', \bar{\gamma} \\ S', \hat{\Gamma} \vdash l_0, \bar{c}, [\bar{x} \mapsto \bar{\gamma}], \bar{b} \text{ where } b_0 = (l_0, [], \bar{c}) \text{ and } (f_\bullet(\bar{x})\{\bar{b}\}) \in \mathbb{T} \end{array}}{\{\alpha \succeq \{\mathcal{F}(S', \hat{\Gamma} \mid f_\bullet : \bar{\gamma} \mapsto \diamond)\}\} \cup S, \hat{\Gamma} \vdash l, (x = \textbf{load.channel}\ f_\bullet) \cdot \bar{c}, \hat{\rho}, \bar{b}}$$

5.4 Correctness of k-LCFA

I now present a proof that the k-LCFA analysis just described produces a sound approximation to the values that might occur in a concrete execution. Soundness is defined by the abstraction function ($\text{abstract}_{d,E}$) and the valid-approximation relations given in Figure 5.7—$\Phi, \alpha \rhd_{d,E} v$ along with $\blacktriangleright$ (both of which I will implicitly lift to vectors and environments), $\Phi, \hat{\Gamma} \rhd_{d,E} \Gamma$ and $\Phi, \hat{\Gamma} \rhd_{d,E} \Gamma, \Sigma$. I introduce d to specify the definition that the analysis is considering. Sub-domains corresponding to d are given using a subscript (e.g. $\mathbb{C}_d$). Note that since * can overlap with an abstract channel value f, the abstraction function gives a set of possible abstract values for a given concrete value.

This is similar to preservation proofs for type systems. However, this proof relies on doing extra work after each **emit** or **construct** in order to approximate the firing rule, which is not present as an explicit feature in the k-LCFA analysis. This 'work' is inductive and therefore it is only possible to prove soundness when the machine is started at an expected start state.

[5] This was referred to during Section 3.2 (see page 59) and equates to programs where join patterns only ever include one channel. Here I also consider it to mean there is no non-determinism due to multiple rules on one channel.

Abstraction Function: $\text{abstract}_{d,E} : \text{Value} \to \mathcal{P}(\widehat{\text{Value}})$

$$\text{abstract}_{d,E}(v) = \begin{cases} \{*\} & (v = (f,t) \wedge f \notin \mathbb{C}_d) \vee (v \text{ is primitive}) \\ \{*\} \cup \mathcal{W}_f & v = (f,t) \wedge f \in \mathbb{C}_d \wedge f \in E \\ \mathcal{W}_f & v = (f,t) \wedge f \in \mathbb{C}_d \wedge f \notin E \end{cases}$$

Valid Approximation Relations:

$$\begin{aligned} \Phi, \alpha \rhd_{d,E} v &\iff \exists c \in \text{abstract}_{d,E}(v).\mathcal{B}(c) \in \Phi(\alpha) \\ \Phi, \alpha \blacktriangleright_{d,E} v &\iff \exists c \in \text{abstract}_{d,E}(v).\mathcal{F}(c) \in \Phi(\alpha) \\ \Phi, \hat{\Gamma} \rhd_{d,E} (M, \theta) &\iff \forall ((f,t), \bar{v}) \in M. \begin{cases} \Phi, \hat{\Gamma}(f) \rhd_{d,E} \bar{v} & f \in \mathbb{C}_d \\ \forall i. * \in \text{abstract}_{d,E}(v_i) & f \notin \mathbb{C}_d \end{cases} \\ \Phi, \hat{\Gamma} \rhd_{d,E} \Gamma, \Sigma &\iff \forall (\Gamma', \Sigma'). \Big(\Gamma, \Sigma \twoheadrightarrow^* \Gamma', \Sigma' \implies \Phi, \hat{\Gamma} \rhd_{d,E} \Gamma' \Big) \end{aligned}$$

Figure 5.7: Valid Approximation Relations

5.4.1 Constraint Generation

I start by proving that the generated constraints are sound, given the meaning defined by their model.

Theorem 1 *For any mapping Φ and constraint set S, analysis of the definition d is sound—i.e. whenever the initial analysis conditions hold and the mapping Φ satisfies the model of constraints:*

$$\begin{aligned} &\forall (\overline{f_i(\overline{x_i})\{\bar{b}\}}) \in \mathbb{T}.\ S, \hat{\Gamma} \vdash l_0, \bar{c}, [\overline{x_i} \mapsto \hat{\Gamma}(f_i) \mid 1 \leq i \leq n], \bar{b} \textit{ where } b_0 = (l_0, [], \bar{c}) \\ &\wedge \forall f \in \text{Constructor}. \forall i. \{\hat{\Gamma}(f)_i \succeq \{\mathcal{F}(*)\}\} \subseteq S \\ &\wedge (\Phi, \hat{\Gamma} \models S) \end{aligned}$$

we have:

$$\Phi, \hat{\Gamma} \rhd_{d,E} (\{((\texttt{@main}, 0), \bar{v})\}, 1), (\_, \_, [\textbf{finish}], \_, \_)$$

where $\bar{v}$ is correctly typed (so contains either primitives or channel values in definitions disjoint to those in the program)—i.e.

$$\forall i.\ * \in \text{abstract}_{d,E}(v_i)$$

and E satisfies the criteria previously stated (in Section 5.3) given Φ and S.

The proof proceeds by structural induction over the reachable states (Γ, Σ) of the concrete machine. The base case is the starting state, and I then show that any step made by the concrete semantics $\twoheadrightarrow$ preserves my induction hypothesis. This splits into four cases (fire, emit, construct and load) that correspond to the rules in Figure 3.13. As already mentioned, this proof does extra work after **emit** and **construct** instructions, since in order to be able to prove that the abstraction covers firing steps, I need to 'save' some properties. My induction hypothesis is therefore much stronger than the property I am trying to prove. It consists of three parts:

- The first is that the global environment is always correctly abstracted:

$$\Phi, \hat{\Gamma} \rhd_{d,E} \Gamma \tag{1}$$

- Secondly, it must be the case that the state $\Sigma = (t, l, \bar{c}, \rho, \bar{b})$ is covered by the constraint set whenever l and $\bar{b}$ are from the definition d. Specifically, every value in the concrete environment ρ must be in the foreground ($\mathcal{F}$) somewhere in the analysis, and that point of the analysis must also abstract the remainder of the environment, although not necessarily in the foreground:

$$\begin{aligned} l \in \mathrm{Label}_d &\implies \forall x.\ \exists \hat{\rho}.\ (\Phi, \hat{\rho}(x) \blacktriangleright_{d,E} \rho(x)) \wedge (\Phi, \hat{\rho} \rhd_{d,E} \rho) \\ &\qquad\qquad \wedge (S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}, \bar{b}) \\ l \notin \mathrm{Label}_d &\implies \forall x. * \in \mathrm{abstract}_{d,E}(\rho(x)) \end{aligned} \tag{2}$$

- Finally, I require that any messages in the concrete environment $\Gamma = (M, \theta)$ have been considered in the foreground ($\mathcal{F}$) by some part of our analysis. So for each channel $f_\bullet \in \mathbb{C}_d$ and rule $\overline{f_i(\overline{x_i})}\{\bar{b}\}$ such that $\exists i.\ f_i = f_\bullet$ with $b_0 = (l_0, [], \bar{c})$:

$$\begin{aligned} \forall((f_\bullet, t), \bar{v}) \in M &\implies \exists \bar{\gamma}.\ (\Phi, \bar{\gamma} \blacktriangleright_{d,E} \bar{v}) \\ &\qquad \wedge S, \hat{\Gamma} \vdash l_0, \bar{c}, [\overline{x_j} \mapsto \hat{\Gamma}(f_j) \mid j \neq i] + [\overline{x_i} \mapsto \bar{\gamma}], \bar{b} \end{aligned} \tag{3}$$

The base case and induction steps for each of the four cases are then as follows.

Base Case

(2) holds trivially since the JCAM starts off with an empty concrete local environment ρ.

$@\mathtt{main} \in \mathbb{C}_d$: For all rules $@\mathtt{main}(\bar{x})\{\bar{b}\}$ with $b_0 = (l_0, [], \bar{c})$:

$$\begin{aligned} &\forall i. * \in \mathrm{abstract}_{d,E}(v_i) \wedge \{\hat{\Gamma}(@\mathtt{main})_i \succeq \{\mathcal{F}(*)\}\} \subseteq S \\ &\qquad\qquad \wedge S, \hat{\Gamma} \vdash l_0, \bar{c}, [\bar{x} \mapsto \hat{\Gamma}(@\mathtt{main})], \bar{b} \wedge \Phi, \hat{\Gamma} \models S \\ \implies &\forall i. * \in \mathrm{abstract}_{d,E}(v_i) \wedge \mathcal{F}(*) \in \Phi(\hat{\Gamma}(@\mathtt{main})_i) \\ &\qquad\qquad \wedge S, \hat{\Gamma} \vdash l_0, \bar{c}, [\bar{x} \mapsto \hat{\Gamma}(@\mathtt{main})], \bar{b} \end{aligned}$$

This implies both (1) and (3).

$@\mathtt{main} \notin \mathbb{C}_d$: (3) holds trivially and $\forall i. * \in \mathrm{abstract}_{d,E}(v_i)$ gives (1).

Load Channel

The environment Γ of the concrete machine is unchanged, so I need only consider (2). The concrete step for the machine is:

$$\frac{c_0 = (x = \textbf{load.channel}\ f)}{p \vdash \Gamma, (t, l, c_0 \cdot \bar{c}, \rho, \bar{b}) \twoheadrightarrow \Gamma, (t, l, \bar{c}, \rho[x \mapsto (f, t)], \bar{b})}$$

Note that $l \in \mathrm{Label}_d \iff f \in \mathbb{C}_d$. I case split on this:

$l \in \text{Label}_d$**:** Assuming (2) for the initial state:

$$\begin{aligned} &\forall y.\ \exists \hat{\rho}.\ (\Phi, \hat{\rho} \rhd_{d,E} \rho) \land (\Phi, \hat{\rho}(y) \blacktriangleright_{d,E} \rho(y)) \\ &\qquad\qquad\qquad\qquad \land (S, \hat{\Gamma} \vdash l, c_0 \cdot \bar{c}, \hat{\rho}, \bar{b}) \\ \implies &\forall y.\ \exists \hat{\rho}, \alpha.\ (\Phi, \hat{\rho} \rhd_{d,E} \rho) \land (\Phi, \hat{\rho}(y) \blacktriangleright_{d,E} \rho(y)) \\ &\qquad\qquad\qquad\qquad \land (S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}[x \mapsto \alpha], \bar{b}) \\ &\qquad\qquad\qquad\qquad \land \exists w \in \mathcal{W}_f \land (\alpha \succeq \{\mathcal{F}(w)\}) \in S \end{aligned}$$

By our assumptions, we have $\Phi, \hat{\Gamma} \models S$.

$$\begin{aligned} \implies &\forall y.\ \exists \hat{\rho}, \alpha.\ (\Phi, \hat{\rho} \rhd_{d,E} \rho) \land (\Phi, \hat{\rho}(y) \blacktriangleright_{d,E} \rho(y)) \\ &\qquad\qquad\qquad\qquad \land (S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}[x \mapsto \alpha], \bar{b}) \\ &\qquad\qquad\qquad\qquad \land \exists w \in \mathcal{W}_f \land \mathcal{F}(w) \in \Phi(\alpha) \end{aligned}$$

Note that $f \in \mathbb{C}_d \implies w \in \text{abstract}_{d,E}((f,t))$.

$$\begin{aligned} \implies &\forall y.\ \exists \hat{\rho}, \alpha.\ (\Phi, \hat{\rho}[x \mapsto \alpha] \rhd_{d,E} \rho[x \mapsto (f,t)]) \\ &\qquad\qquad \land (\Phi, \hat{\rho}[x \mapsto \alpha](y) \blacktriangleright_{d,E} \rho[x \mapsto (f,t)](y)) \\ &\qquad\qquad \land (S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}[x \mapsto \alpha], \bar{b}) \end{aligned}$$

$l \notin \text{Label}_d$**:** Since $f \notin \mathbb{C}_d \implies * \in \text{abstract}_{d,E}((f,t))$.

$$\begin{aligned} &\forall y.\ * \in \text{abstract}_{d,E}(\rho(y)) \land * \in \text{abstract}_{d,E}((f,t)) \\ \implies &\forall y.\ * \in \text{abstract}_{d,E}(\rho[x \mapsto (f,t)](y)) \end{aligned}$$

This gives both cases of (2).

Emit

This concrete step transfers values from the local environment ρ to the global environment Γ, specifically the marking M:

$$\frac{c_0 = (\textbf{emit}\ x(\bar{y})) \qquad \bar{v} = \rho(\bar{y}) \qquad (f_\bullet, t') = \rho(x)}{p \vdash (M, \theta), (t, l, c_0 \cdot \bar{c}, \rho, \bar{b}) \twoheadrightarrow (M + ((f_\bullet, t'), \bar{v}), \theta), (t, l, \bar{c}, \rho, \bar{b})}$$

First I will prove a small corollary, that for any f:

$$f \in E \implies \forall i.\ \mathcal{F}(*) \in \Phi(\hat{\Gamma}(f)_i)$$

f could have been placed in E in two ways:

$f \in \text{Constructor}$**:** The initial assumptions include:

$$(\Phi, \hat{\Gamma} \models S) \land \forall f \in \text{Constructor}. \forall i. \{\hat{\Gamma}(f)_i \succeq \{\mathcal{F}(*)\}\} \subseteq S$$

Which immediately gives what is required.

$\exists i, \bar{\alpha}, \beta.\ (\bar{\alpha} \mapsto \beta) \in S \land \mathcal{F}(*) \in \Phi(\beta) \land \mathcal{B}(w) \in \Phi(\alpha_i)$**:** By assumption, $\Phi, \hat{\Gamma} \models \bar{\alpha} \mapsto \beta$ holds, so we have that $\mathcal{F}(*) \in \Phi(\hat{\Gamma}(f)_i)$ as required.

I can now case split on $l \in \text{Label}_d$.

$l \notin \text{Label}_d$: By (2) it holds that:

$$* \in \text{abstract}_{d,E}((f_\bullet, t')) \wedge \forall i.\ * \in \text{abstract}_{d,E}(v_i)$$

It is therefore either the case that $f_\bullet \notin \mathbb{C}_d$ or $f_\bullet \in E$:

$f_\bullet \notin \mathbb{C}_d$: Since $\forall i.\ * \in \text{abstract}_{d,E}(v_i)$ and it can assumed (1) holds for (M, θ), it must also be that (1) holds for the new environment $(M + (f_\bullet, t'), \bar{v}), \theta)$. In this case, (3) is unaffected.

$f_\bullet \in E$: I have already shown that this implies:

$$\forall i.\ \mathcal{F}(*) \in \Phi(\hat{\Gamma}(f_\bullet)_i)$$

With our knowledge of $\bar{v}$, it is therefore clear that both (1) and (3) hold.

(2) trivially holds for this first case.

$l \in \text{Label}_d$: First I will show that (2) continues to hold:

$$\begin{aligned}
&\forall x.\ \exists \hat{\rho}.\ (\Phi, \hat{\rho}(x) \blacktriangleright_{d,E} \rho(x)) \wedge (\Phi, \hat{\rho} \rhd_{d,E} \rho) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge (S, \hat{\Gamma} \vdash l, c_0 \cdot \bar{c}, \hat{\rho}, \bar{b}) \\
\implies &\forall x.\ \exists \hat{\rho}.\ (\Phi, \hat{\rho}(x) \blacktriangleright_{d,E} \rho(x)) \wedge (\Phi, \hat{\rho} \rhd_{d,E} \rho) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge (S, \hat{\Gamma} \vdash l, \bar{c}, \hat{\rho}, \bar{b}) \wedge (\hat{\rho}(\bar{y}) \mapsto \hat{\rho}(x)) \in S
\end{aligned}$$

This gives (2) and I now split into two more cases:

$w \in \text{abstract}_{d,E}((f_\bullet, t')) \wedge \mathcal{F}(w) \in \Phi(\hat{\rho}(x))$: For all rules $(\overline{f_i(\overline{x_i})}\{\bar{b}\})$ with $b_0 = (l_0, [], \bar{c})$ and $\exists i.\ f_i = f_\bullet$:

$$\begin{aligned}
\implies &\forall x.\ \exists \hat{\rho}.\ (\Phi, \hat{\rho}(x) \blacktriangleright_{d,E} \rho(x)) \wedge (\Phi, \hat{\rho} \rhd_{d,E} \rho) \\
&\qquad \wedge \Phi, \hat{\Gamma} \models \mathcal{I}(w, \hat{\rho}(\bar{y})) \\
\implies &\forall x.\ \exists \hat{\rho}.\ (\Phi, \hat{\rho}(x) \blacktriangleright_{d,E} \rho(x)) \wedge (\Phi, \hat{\rho} \rhd_{d,E} \rho) \\
&\qquad \wedge \Phi, \hat{\Gamma}(f_\bullet) \rhd_{d,E} \bar{v} \\
&\qquad \wedge \exists S', \bar{\gamma}.\ (\Phi, \bar{\gamma} \blacktriangleright_{d,E} \bar{v}) \\
&\qquad\qquad \wedge S', \hat{\Gamma} \vdash l_0, \bar{c}, [\overline{x_j} \mapsto \hat{\Gamma}(f_j) \mid j \neq i] + [\overline{x_i} \mapsto \bar{\gamma}], \bar{b}
\end{aligned}$$

This satisfies both (1) and (3).

$* \in \text{abstract}_{d,E}((f_\bullet, t')) \wedge \mathcal{F}(*) \in \Phi(\alpha)$: This implies that:

$$\forall \mathcal{B}(S', \hat{\Gamma} \mid f_\bullet : \bar{\gamma} \mapsto \diamond) \in \bigcup_i \Phi(\beta_i).\ f_\bullet \in E$$

And therefore that:

$$\forall i.\ * \in \text{abstract}_{d,E}(v_i)$$

I can therefore reuse the proof from above for $l \notin \text{Label}_d$.

Construct

Again this step transfers values to the environment Γ:

$$\frac{c_0 = (\textbf{construct}\ f(\bar{x})) \qquad \bar{v} = \rho(\bar{x})}{p \vdash (M, \theta), (t, l, c_0 \cdot \bar{c}, \rho, \bar{b}) \twoheadrightarrow (M + ((f, \theta), \bar{v}), \theta + 1), (t, l, \bar{c}, \rho, \bar{b})}$$

Since $f \in E$, I can reuse variants of the relevant parts of the **emit** proof above.

Fire

A firing causes values to be transferred from the marking M to a local environment ρ:

$$\frac{\begin{array}{c}\Delta = \{((f_i, \alpha), \bar{v_i}) \mid 1 \le i \le n\} \\ (\overline{f_i(\overline{x_i})}\{\bar{b}\}) \in p \qquad b_0 = (l_0, [], \bar{c}) \qquad \rho = [\overline{x_i} \mapsto \overline{v_i} \mid 1 \le i \le n]\end{array}}{p \vdash (M + \Delta, \theta), (\_, \_, [\textbf{finish}], \_, \_) \twoheadrightarrow (M, \theta), (\alpha, l_0, \bar{c}, \rho, \bar{b})}$$

Since this makes M smaller, I only need to show that (2) still holds. I again case split on whether $l_0 \in \mathrm{Label}_d$.

$l_0 \in \mathrm{Label}_d$**:** By (1) and (3), I have:

$$\begin{aligned}
&\forall i.\ \exists \bar{\gamma}.\ S, \hat{\Gamma} \vdash l_0, \bar{c}, [\overline{x_j} \mapsto \hat{\Gamma}(f_j) \mid j \neq i] + [\overline{x_i} \mapsto \bar{\gamma}], \bar{b} \\
&\qquad \wedge(\Phi, \bar{\gamma} \blacktriangleright_{d,E} \overline{v_i} \wedge (\Phi, \hat{\Gamma}(f_i) \rhd_{d,E} \overline{v_i}) \\
\Longrightarrow\ &\forall i.\ \exists \bar{\gamma}.\ (\Phi, [\overline{x_j} \mapsto \hat{\Gamma}(f_j) \mid j \neq i] + [\overline{x_i} \mapsto \bar{\gamma}] \rhd_{e,D} \rho) \\
&\qquad \wedge S, \hat{\Gamma} \vdash l_0, \bar{c}, [\overline{x_j} \mapsto \hat{\Gamma}(f_j) \mid j \neq i] + [\overline{x_i} \mapsto \bar{\gamma}], \bar{b} \\
&\qquad \wedge(\Phi, \bar{\gamma} \blacktriangleright_{d,E} \overline{v_i} \wedge (\Phi, \hat{\Gamma}(f_i) \rhd_{d,E} \overline{v_i}) \\
\Longrightarrow\ &\forall x.\ \exists \hat{\rho}.\ (\Phi, \hat{\rho}(x) \blacktriangleright_{e,D} \rho(x)) \wedge (\Phi, \hat{\rho} \rhd_{e,D} \rho) \\
&\qquad \wedge S, \hat{\Gamma} \vdash l_0, \bar{c}, \hat{\rho}, \bar{b}
\end{aligned}$$

$l_0 \notin \mathrm{Label}_d$**:** This also means that $\forall i.\ f_i \notin \mathbb{C}_d$. Combining this with (1), I have:

$$\forall i, j.\ * \in \mathrm{abstract}_{d,E}(v_{i,j})$$

It is therefore the case that (2) holds in both cases.

It follows by structural induction that $\Phi, \hat{\Gamma} \rhd_{d,E} \Gamma$ holds for all reachable states whenever the constraint set has been properly constructed and Φ is a valid solution to it.

5.4.2 Closure Algorithm

Although I have shown the constraint generation rules and the model of constraints to be sound, I still need to show that the closure algorithm produces a Φ that satisfies the model.

Theorem 2 *The transitive closure algorithm for computing a solution to a constraint set S produces a valid solution Φ. Specifically:*

$$\begin{aligned}
\big(\Phi(\alpha) = \{\mathcal{F}(c) \mid (\alpha \succeq \{\mathcal{F}(c)\}) \in S^+\} \sqcup \{\mathcal{B}(c) \mid (\alpha \succeq \{\mathcal{B}(c)\}) \in S^+\}\big) \\
\Longrightarrow \Phi, \hat{\Gamma} \models S
\end{aligned}$$

I again proceed using structural induction—this time on the set of constraints S. The induction step is split into five cases, representing the five variants of constraints.

Base Case. $\Phi, \hat{\Gamma} \models S$ trivially holds when $S = \{\}$.

Case 1. $(\alpha \succeq \{\mathcal{F}(c)\}) \in S$.

$$\begin{aligned} &\implies (\alpha \succeq \{\mathcal{F}(c)\}) \in S^+ \\ &\implies \mathcal{F}(c) \in \Phi(\alpha) \\ &\implies \Phi, \hat{\Gamma} \models \alpha \succeq \{\mathcal{F}(c)\} \end{aligned}$$

Case 2. $(\alpha \succeq \{\mathcal{B}(c)\}) \in S$. Holds by a similar argument to Case 1.

Case 3. $(\alpha_1 \succeq \mathcal{F}(\alpha_2)) \in S$.

$$\begin{aligned} &\forall c.\ (\alpha_1 \succeq \mathcal{F}(\alpha_2)) \in S \wedge \mathcal{F}(c) \in \Phi(\alpha_2) \\ &\quad\implies \{\alpha_1 \succeq \mathcal{F}(\alpha_2),\ \alpha_2 \succeq \{\mathcal{F}(c)\}\} \subseteq S^+ \\ &\quad\implies (\alpha_1 \succeq \{\mathcal{F}(c)\}) \in S^+ \\ &\quad\implies \mathcal{F}(c) \in \Phi(\alpha_1) \end{aligned}$$

And one can show that a similar result holds for $\mathcal{B}(c) \in \Phi(\alpha_2)$, which gives us that $\Phi, \hat{\Gamma} \models \alpha_1 \succeq \mathcal{F}(\alpha_2)$.

Case 4. $(\alpha_1 \succeq \mathcal{B}(\alpha_2)) \in S$. Holds by a similar argument to Case 3.

Case 5. $(\bar{\alpha} \mapsto \beta) \in S$. Firstly, I deal with if $\mathcal{F}(*) \in \Phi(\beta)$:

$$\begin{aligned} &\quad(\bar{\alpha} \mapsto \beta) \in S \wedge \mathcal{F}(*) \in \Phi(\beta) \wedge \mathcal{B}(w) \in \Phi(\alpha_i) \\ &\implies \{\bar{\alpha} \mapsto \beta,\ \beta \succeq \{\mathcal{F}(*)\}, \alpha_i \succeq \{\mathcal{B}(w)\}\} \subseteq S^+ \\ &\quad\vee \{\bar{\alpha} \mapsto \beta,\ \beta \succeq \{\mathcal{F}(*)\}, \alpha_i \succeq \{\mathcal{F}(w)\}\} \subseteq S^+ \\ &\implies \mathcal{E}(w) \subseteq S^+ \end{aligned}$$

Secondly, I consider $\mathcal{F}(w) \in \Phi(\beta)$.

$$\begin{aligned} &\quad(\bar{\alpha} \mapsto \beta) \in S \wedge \mathcal{F}(w) \in \Phi(\beta) \\ &\implies \{\bar{\alpha} \mapsto \beta,\ \beta \succeq \{\mathcal{F}(w)\}\} \subseteq S^+ \\ &\implies \mathcal{I}(w, \bar{\alpha}) \subseteq S^+ \end{aligned}$$

The nature of $\mathcal{I}(w, \bar{\alpha})$ means that it may generate further emission constraints. However, since the domain of flow variables is finite, there are also a finite number of constraints so the closure algorithm will terminate. The above combine to give a logical expression identical to the definition of $\models$ itself:

$$\begin{aligned} \Phi, \hat{\Gamma} \models \bar{\alpha} \mapsto \beta \iff &\Big(\mathcal{F}(w) \in \Phi(\beta) \implies \Phi, \hat{\Gamma} \models \mathcal{I}(w, \bar{\alpha})\Big) \\ &\wedge\Big(\mathcal{F}(*) \in \Phi(\beta) \implies \forall \mathcal{B}(w) \in \bigcup_i \Phi(\alpha_i). \Phi, \hat{\Gamma} \models \mathcal{E}(w)\Big) \end{aligned}$$

By cases 1–4, one can assume that $\Phi, \hat{\Gamma} \models S'$ for any S' that does not include emission constraints. Provided that we take the maximal fixed point for $\models$ (i.e.

Initialisation:

$$\lfloor f \rfloor = \min_{r \in \mathbb{T}_{\text{construct}}} \{\lfloor r^{\bullet}(f) \rfloor\}$$

$$\lceil f \rceil = \begin{cases} \infty & \text{if } f \in (E \setminus \text{Constructor}) \\ \max_{r \in \mathbb{T}_{\text{construct}}} \{\lceil r^{\bullet}(f) \rceil\} & \text{otherwise} \end{cases}$$

Computation: $\forall r \in (\mathbb{T} \setminus \mathbb{T}_{\text{construct}})$

$$\begin{aligned} {}^{\bullet}r(f) > \lfloor r^{\bullet}(f) \rfloor &\implies \lfloor f \rfloor = 0 \\ {}^{\bullet}r(f) < \lceil r^{\bullet}(f) \rceil &\implies \lceil f \rceil = \infty \end{aligned}$$

Figure 5.8: Simple algorithm for computing queue bound of f

allow as many solutions as possible to be considered valid), then it is the case that $\Phi, \hat{\Gamma} \models \bar{\alpha} \mapsto \beta$. This is desirable as it allows solutions that are less conservative. Such solutions assign fewer possible values to each flow variable, and are therefore more precise.

Therefore, by induction the closure algorithm gives a valid solution for all constraint sets.

5.5 Queue Bounding

The potential benefits of queue bounding were shown in Section 4.4. Recall that the motivating example was the memory cell encoding. I showed that the `val` channel in this could be replaced by a memory location and status flag. Indeed, in *fast mode*, it could even be removed from patterns. To do this automatically, an analysis is needed to bound the possible queue lengths for each channel f. The result of this is a pair $(\lfloor f \rfloor, \lceil f \rceil) \in (\mathbb{N}_0 \times \mathbb{N}_0^{\infty})$ giving the minimum and maximum queue size. I use helper functions inspired by Petri-net notation:[6]

$$\begin{aligned} {}^{\bullet}\_ &: \mathbb{T} \to (\mathbb{C} \to \mathbb{N}_0) && \text{(input count)} \\ \_^{\bullet} &: \mathbb{T} \to (\mathbb{C} \to (\mathbb{N}_0 \times \mathbb{N}_0^{\infty})) && \text{(output range)} \end{aligned}$$

The first is defined by the number of occurrences of a channel in the left-hand-side pattern of a transition rule. The second requires analysis of the transition rule body's control-flow graph using the LCFA results and range arithmetic—incorporating dominator analysis to detect loops and prevent counting to ∞.

The queue bounds of a channel f can then be approximated by the simple algorithm in Figure 5.8. This starts by initialising each queue's range by the possible effect of constructor transitions. If any transitions use up messages on a channel without replacing them, the lower limit is reduced to zero, and similarly transitions causing a net increase raise the upper limit to infinity. A more accurate solution would consider the interaction between channels in a similar manner to *boundness* checking, or *invariants* for Petri-nets, but I leave this for future work. My approach accurately (with respect to $\_^{\bullet}$) finds channels

[6] Remember that $\mathbb{T}$ gives the set of transition rules, while the set of channels is $\mathbb{C}$.

with a constant queue length, so can still replace queues with memory locations in many situations. However, it will rarely detect *cell*-style channels—i.e. $(\lfloor f \rfloor, \lceil f \rceil) = (0, 1)$.

Enabling the queue bounding optimisations presented in Section 4.4 is then relatively straightforward. The cell scenario is achieved simply by adding the results as annotations. The memory optimisation in fast mode also requires a `head` annotation to indicate that a channel is only emitted to by head emits. This can be inferred by a dataflow analysis. Initially, all channels with equal upper and lower bounds are assumed to be `head`. I then iterate, relegating any channel that with emissions that might follow emissions to a non-head channel.

5.6 Closedness of Definitions

In Section 4.3, I discussed how instances of closed definitions could be stack-allocated to improve performance. The stated requirement for a definition to be closed is that its instances do not require subsequent messages after construction. This can be checked by considering whether any channels escape to instances that are not 'internal' to the instance—i.e. whether the definition fully encapsulates part of the program. For example, consider `fib`: both `a` and `b` escape according to my LCFA. However, they escape to instances created by `fib`—no channels escape to the `k` channel value passed into the constructor from outside. The LCFA approaches can be refined to consider this by splitting the 'non-this' instances (along with the wildcard `*`) in two: *outer* instances that constructed the 'this' instance; and *inner* instances that 'this' itself constructs. Correspondingly, I can also distinguish between the escape sets E_{outer} and E_{inner} of channels escaping to these distinct sets of instances.

This amounts to a lot of duplication to provide identical analyses of $*_{\text{inner}}$ and $*_{\text{outer}}$. The only new case introduced is when these wildcards interact (i.e. one is emitted to another). Whenever this occurs, it implies that the inner and outer instances may communicate directly. In this case, it is no longer possible to distinguish between the two sets, so I conflate $*_{\text{inner}}$ and $*_{\text{outer}}$ back to a single wildcard.

For my purposes, primitive values are not interesting, so need not be tracked at all, but if they were required in the results they would have to form their own abstract value as they fit into neither the inner nor the outer set.

With this change made, closed definitions are simply equivalent to those for which:

$$E_{\text{outer}} \setminus \text{Constructor} = \{\}$$

Results indicating how often this technique detects closed definitions are given in Section 6.5.

5.7 Worked Examples

I will now illustrate the techniques from this chapter with two worked examples. The first is a combination of the motivating 'handshake' examples from Section 5.3 which demonstrates the application of foreground call-strings and the increased accuracy provided by k-LCFA. The second is the detection of closedness in the `fib` benchmark. This uses the slightly finer grained wildcards, $*_{\text{inner}}$ and $*_{\text{outer}}$, and is detectable with 0-LCFA.

```
transition @start() {
  %a = load.channel %a
  %b = load.channel %b
  ... also for: i, j, k, l, p, q, r and s
  emit %a(() %i, (()) %p)
  emit %a(() %j, (()) %q)
  emit %b(() %k, (()) %r)
  emit %b(() %l, (()) %s)
  finish
}

transition %a(() %x, (()) %m) %b(() %y, (()) %n) {
  emit %m(() %x)
  emit %n(() %y)
  finish
}

transition %a(() %x, (()) %m) %b(() %y, (()) %n) {
  emit %m(() %y)
  emit %n(() %x)
  finish
}
```

Figure 5.9: Handshake JCAM source code

5.7.1 Foreground call-strings example ('handshake')

The handshake examples showed that 0-LCFA is not always able to get the best results. The full JCAM source that I will work through is shown in Figure 5.9. Channels `i` to `l` take no arguments (i.e. void-channels), while `p` to `s` take a single void-channel argument each. Using void-channels causes the values to be tracked, whereas a primitive value would be abstracted to `PRIM`. Unlike most examples in this thesis, I have included explicit `load.channel` commands. I will initially use $\alpha_{\mathtt{x}}$ to indicate the flow variable associated with a local variable `x`. However, more than one flow variable is required per variable during the closure stage of the analysis, and these will be written α_1, α_2, etc. The first constraint generated from `@start` will be of the form $\alpha_{\mathtt{a}} \succeq \mathcal{F}(\cdots \mid \mathtt{a} : \alpha_{\mathtt{x}}, \alpha_{\mathtt{m}} \mapsto \diamond)$. Filling in the remainder of the flow closure for channel `a` requires both transitions involving `a` to be analysed, with `a` in the foreground. The first transition generates the following constraint:

$$\begin{aligned} [\alpha_{\mathtt{x}}] &\mapsto \alpha_{\mathtt{m}} \\ [\hat{\Gamma}(\mathtt{b})_{\mathtt{y}}] &\mapsto \hat{\Gamma}(\mathtt{b})_{\mathtt{n}} \end{aligned}$$

In turn, the second transition gives:

$$\begin{aligned} [\hat{\Gamma}(\mathtt{b})_{\mathtt{y}}] &\mapsto \alpha_{\mathtt{m}} \\ [\alpha_{\mathtt{x}}] &\mapsto \hat{\Gamma}(\mathtt{b})_{\mathtt{n}} \end{aligned}$$

Producing the equivalent for `b`, and the other emission constraints in `@start`, gives the following set of constraints for the complete definition:

$$\begin{aligned}
\alpha_{\mathtt{a}} &\succeq \mathcal{F}(\{[\alpha_{\mathtt{x}}] \mapsto \alpha_{\mathtt{m}}, [\hat{\Gamma}(\mathtt{b})_{\mathtt{y}}] \mapsto \hat{\Gamma}(\mathtt{b})_{\mathtt{n}}, [\hat{\Gamma}(\mathtt{b})_{\mathtt{y}}] \mapsto \alpha_{\mathtt{m}}, [\alpha_{\mathtt{x}}] \mapsto \hat{\Gamma}(\mathtt{b})_{\mathtt{n}}\}, \hat{\Gamma} \mid \mathtt{a} : \alpha_{\mathtt{x}}, \alpha_{\mathtt{m}} \mapsto \diamond) \\
\alpha_{\mathtt{b}} &\succeq \mathcal{F}(\{[\hat{\Gamma}(\mathtt{a})_{\mathtt{x}}] \mapsto \hat{\Gamma}(\mathtt{a})_{\mathtt{m}}, [\alpha_{\mathtt{y}}] \mapsto \alpha_{\mathtt{n}}, [\alpha_{\mathtt{y}}] \mapsto \hat{\Gamma}(\mathtt{a})_{\mathtt{m}}, [\hat{\Gamma}(\mathtt{a})_{\mathtt{x}}] \mapsto \alpha_{\mathtt{n}}\}, \hat{\Gamma} \mid \mathtt{b} : \alpha_{\mathtt{y}}, \alpha_{\mathtt{n}} \mapsto \diamond)
\end{aligned}$$

$$\begin{array}{cc}
\alpha_{\mathtt{i}} \succeq \mathcal{F}(\{\}, \hat{\Gamma} \mid \mathtt{i}) & \alpha_{\mathtt{p}} \succeq \mathcal{F}(\{\}, \hat{\Gamma} \mid \mathtt{p} : \alpha_{?} \mapsto \diamond) \\
\cdots & \cdots \\
\alpha_{\mathtt{l}} \succeq \mathcal{F}(\{\}, \hat{\Gamma} \mid \mathtt{l}) & \alpha_{\mathtt{s}} \succeq \mathcal{F}(\{\}, \hat{\Gamma} \mid \mathtt{s} : \alpha_{?} \mapsto \diamond)
\end{array}$$

$$\begin{aligned}
[\alpha_{\mathtt{i}}, \alpha_{\mathtt{p}}] &\mapsto \alpha_{\mathtt{a}} \\
[\alpha_{\mathtt{j}}, \alpha_{\mathtt{q}}] &\mapsto \alpha_{\mathtt{a}} \\
[\alpha_{\mathtt{k}}, \alpha_{\mathtt{r}}] &\mapsto \alpha_{\mathtt{b}} \\
[\alpha_{\mathtt{l}}, \alpha_{\mathtt{s}}] &\mapsto \alpha_{\mathtt{b}}
\end{aligned}$$

Since no transitions match on the channels `p`, `q`, `r` or `s`, the flow variable within the flow closure for them is never used. I therefore use a shared placeholder $\alpha_?$ rather than introducing distinct flow variables for each. Expanding the four emission constraints, using fresh flow variables for the arguments to `a` and `b`, gives the following extra constraints:

$$\begin{array}{rlrl}
\alpha_1 &\succeq \mathcal{F}(\alpha_{\mathtt{i}}) & \alpha_3 &\succeq \mathcal{F}(\alpha_{\mathtt{j}}) \\
\alpha_2 &\succeq \mathcal{F}(\alpha_{\mathtt{p}}) & \alpha_4 &\succeq \mathcal{F}(\alpha_{\mathtt{q}}) \\
\hat{\Gamma}(\mathtt{a})_{\mathtt{x}} &\succeq \mathcal{B}(\alpha_{\mathtt{i}}) & \hat{\Gamma}(\mathtt{a})_{\mathtt{x}} &\succeq \mathcal{B}(\alpha_{\mathtt{j}}) \\
\hat{\Gamma}(\mathtt{a})_{\mathtt{m}} &\succeq \mathcal{B}(\alpha_{\mathtt{p}}) & \hat{\Gamma}(\mathtt{a})_{\mathtt{m}} &\succeq \mathcal{B}(\alpha_{\mathtt{q}}) \\
[\alpha_1] &\mapsto \alpha_2 & [\alpha_3] &\mapsto \alpha_4 \\
[\hat{\Gamma}(\mathtt{b})_{\mathtt{y}}] &\mapsto \alpha_2 & [\hat{\Gamma}(\mathtt{b})_{\mathtt{y}}] &\mapsto \alpha_4 \\
[\alpha_1] &\mapsto \hat{\Gamma}(\mathtt{b})_{\mathtt{n}} & [\alpha_3] &\mapsto \hat{\Gamma}(\mathtt{b})_{\mathtt{n}}
\end{array}$$

$$\begin{array}{rlrl}
\alpha_5 &\succeq \mathcal{F}(\alpha_{\mathtt{k}}) & \alpha_7 &\succeq \mathcal{F}(\alpha_{\mathtt{l}}) \\
\alpha_6 &\succeq \mathcal{F}(\alpha_{\mathtt{r}}) & \alpha_8 &\succeq \mathcal{F}(\alpha_{\mathtt{s}}) \\
\hat{\Gamma}(\mathtt{b})_{\mathtt{y}} &\succeq \mathcal{B}(\alpha_{\mathtt{k}}) & \hat{\Gamma}(\mathtt{b})_{\mathtt{y}} &\succeq \mathcal{B}(\alpha_{\mathtt{l}}) \\
\hat{\Gamma}(\mathtt{b})_{\mathtt{n}} &\succeq \mathcal{B}(\alpha_{\mathtt{r}}) & \hat{\Gamma}(\mathtt{b})_{\mathtt{n}} &\succeq \mathcal{B}(\alpha_{\mathtt{s}}) \\
[\alpha_5] &\mapsto \alpha_6 & [\alpha_7] &\mapsto \alpha_8 \\
[\alpha_5] &\mapsto \hat{\Gamma}(\mathtt{a})_{\mathtt{m}} & [\alpha_7] &\mapsto \hat{\Gamma}(\mathtt{a})_{\mathtt{m}} \\
[\hat{\Gamma}(\mathtt{a})_{\mathtt{x}}] &\mapsto \alpha_6 & [\hat{\Gamma}(\mathtt{a})_{\mathtt{x}}] &\mapsto \alpha_8
\end{array}$$

$$\begin{aligned}
[\hat{\Gamma}(\mathtt{a})_{\mathtt{x}}] &\mapsto \hat{\Gamma}(\mathtt{a})_{\mathtt{m}} \\
[\hat{\Gamma}(\mathtt{b})_{\mathtt{y}}] &\mapsto \hat{\Gamma}(\mathtt{b})_{\mathtt{n}}
\end{aligned}$$

Note that solving these for $\hat{\Gamma}(\mathtt{a})$ and $\hat{\Gamma}(\mathtt{b})$ gives only background ($\mathcal{B}$) values as shown below, and therefore the constraints that emit to either $\hat{\Gamma}(\mathtt{a})_\mathtt{m}$ or $\hat{\Gamma}(\mathtt{b})_\mathtt{n}$ have no effect.

$$\begin{aligned}
\Phi(\hat{\Gamma}(\mathtt{a})_\mathtt{x}) &= \{\mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{i}), \mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{j})\} \\
\Phi(\hat{\Gamma}(\mathtt{a})_\mathtt{m}) &= \{\mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{p} : \alpha_? \mapsto \diamond), \mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{q} : \alpha_? \mapsto \diamond)\} \\
\Phi(\hat{\Gamma}(\mathtt{b})_\mathtt{y}) &= \{\mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{k}), \mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{l})\} \\
\Phi(\hat{\Gamma}(\mathtt{b})_\mathtt{n}) &= \{\mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{r} : \alpha_? \mapsto \diamond), \mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{s} : \alpha_? \mapsto \diamond)\}
\end{aligned}$$

Solving the other constraints to reach the final solution is straightforward, with the following values for the remaining global $\hat{\Gamma}$ flow variables:

$$\begin{aligned}
\Phi(\hat{\Gamma}(\mathtt{p})) &= \{\mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{i}), \mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{k}), \mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{l})\} \\
\Phi(\hat{\Gamma}(\mathtt{q})) &= \{\mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{j}), \mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{k}), \mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{l})\} \\
\Phi(\hat{\Gamma}(\mathtt{r})) &= \{\mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{i}), \mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{j}), \mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{k})\} \\
\Phi(\hat{\Gamma}(\mathtt{s})) &= \{\mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{i}), \mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{j}), \mathcal{B}(\{\}, \hat{\Gamma} \mid \mathtt{l})\}
\end{aligned}$$

5.7.2 Closedness example (fib)

In this example, closedness can be detected without the increased accuracy of k-LCFA. I can therefore stick to a single flow variable per local variable. I also use a single flow variable α_{prim} for all primitive-typed variables, to avoid cluttering the presentation. The source code under analysis is shown in Figure 5.10.

The constraints generated for this are as follows:

$$\begin{aligned}
\hat{\Gamma}(\mathtt{fib})_\mathtt{x} &\succeq \mathcal{F}(\mathtt{PRIM}) \\
\hat{\Gamma}(\mathtt{fib})_\mathtt{k} &\succeq \mathcal{F}(*_{\text{outer}}) \\
[\hat{\Gamma}(\mathtt{fib})_\mathtt{x}] &\mapsto \hat{\Gamma}(\mathtt{fib})_\mathtt{k} \\
\alpha_\mathtt{temp} &\succeq \mathcal{F}(\{\alpha_{\text{prim}} \mapsto \alpha_\mathtt{m}\}, \hat{\Gamma} \mid \mathtt{temp} : \alpha_\mathtt{m} \mapsto \diamond) \\
[\hat{\Gamma}(\mathtt{fib})_\mathtt{k}] &\mapsto \alpha_\mathtt{temp} \\
\alpha_\mathtt{fib1} &\succeq \mathcal{F}(*_{\text{inner}}) \\
\alpha_\mathtt{a} &\succeq \mathcal{F}(\{\alpha_{\text{prim}} \mapsto \hat{\Gamma}(\mathtt{temp})_\mathtt{m}\}, \hat{\Gamma} \mid \mathtt{a} : \alpha_\mathtt{res1} \mapsto \diamond) \\
[\alpha_{\text{prim}}, \alpha_\mathtt{a}] &\mapsto \alpha_\mathtt{fib1} \\
\alpha_\mathtt{fib2} &\succeq \mathcal{F}(*_{\text{inner}}) \\
\alpha_\mathtt{b} &\succeq \mathcal{F}(\{\alpha_{\text{prim}} \mapsto \hat{\Gamma}(\mathtt{temp})_\mathtt{m}\}, \hat{\Gamma} \mid \mathtt{b} : \alpha_\mathtt{res2} \mapsto \diamond) \\
[\alpha_{\text{prim}}, \alpha_\mathtt{b}] &\mapsto \alpha_\mathtt{fib2}
\end{aligned}$$

There is only one occurrence of the $*_{\text{outer}}$ flow value in these constraints, and it is clear that it never interacts with other non-primitive flow values. As a result of this, no channels are ever added to the *outer* escape set—this is what allows us to deduce that the definition is closed. The *inner* escape set for these constraints consists of the channels a and b, which are passed into the two **construct** commands in order to collect results from the recursive invocations of fib.

```
transition @fib(i32 %x, (i32) %k) {
  %base = cmp ult i32 %x, 2
  br %base, label %base_case, label %recurse

base_case:
  emit %k(i32 %x)
  finish

recurse:
  %temp = load.channel %temp
  emit %temp((i32) %k)

  %x1 = add i32 %x, -1
  %a = load.channel %a
  construct @fib(i32 %x1, (i32) %a)

  %x2 = add i32 %x, -2
  %b = load.channel %b
  construct @fib(i32 %x2, (i32) %b)

  finish
}

transition %a(i32 %res1) %b(i32 %res2) %temp((i32) %m) {
  %sum = add i32 %res1, %res2
  emit %m(i32 %sum)
  finish
}
```

Figure 5.10: `fib` benchmark under analysis.

5.8 Summary

This chapter has presented a novel and accurate k-LCFA approach (Section 5.3) for the join calculus, along with a simpler 0-LCFA (Section 5.2). In addition, I have given two dependent analyses (Sections 5.5 and 5.6) that make use of this information to enable optimisations and faster implementations of the join calculus primitives to be used in common cases as described in Chapter 4. The LCFA techniques and also closedness analysis were illustrated by worked examples in Section 5.7.

Chapter 6

Evaluation and Discussion

This chapter provides a general assessment of the techniques developed by my research. Chapter 4 has already given some indication of the quantitative performance benefits that my optimisations produce. However, I now supplement those measurements with comparisons to other languages and compilers, over a wider range of benchmarks (Sections 6.2 to 6.4). I also discuss areas that my research does not cover, and the difficulties associated with these. These predominantly focus on scheduling and inlining issues (Sections 6.6.2 and 6.6.3), but also the accuracy of Chapter 5's control flow analysis (Section 6.5).

The introduction (Chapter 1) detailed the original goals of the research, and these offer guidance on success criteria. It is important to remember that Dovetail is an early prototype, and that the evaluation is therefore aiming to determine whether the JCAM approach might be viable in the future, rather than whether Dovetail itself offers adequate performance. As well as the performance aspect, a key goal was to offer a universal representation of parallelism. Although my Dovetail implementation of the JCAM only supports multi-core architectures, the mapping of other architectures was discussed in Section 3.4. Further investigation of compilation to other targets is left as future work and suitable next steps are discussed briefly in my conclusion (Chapter 7). Section 3.4 also detailed how language concepts can be expressed in the JCAM, and while that is not repeated here, Section 6.2 does validate some of those arguments with concrete benchmarks.

6.1 Test Environments

All the benchmark measurements presented in the next three sections, as well as those in Chapter 4, were taken on a machine with two 8-core processors.[1] With hyperthreading this gives 32 logical cores. For the Dovetail measurements, the worker threads were tied to separate physical cores and therefore only ever used 16 threads. Each execution was repeated 10 times in order to reduce the effect of variation between runs. In order to ensure that the results were not affected by CPUs being in power-saving states, the first set of benchmarks was run again after all others had completed and timings checked to ensure they did not differ.

The compiler and library versions used are shown in Table 6.1.

[1] The actual processors were *AMD Operton 6376*s.

Compiler	Version
GCC	4.6.3
Java	1.6.0_32
LLVM	3.4
Wool	0.1.5alpha
Boehm GC	7.4.0

Table 6.1: Compiler and library versions used for benchmarking.

Benchmark		C	Wool	Java
Fork-Join				
`fib`	$n = 40$	Sequential	✓	Sequential
`nqueens`	$n = 13$	Sequential	✓	Sequential
`quicksort`	$n = 30000000$	Sequential	✓	Sequential
Coordination				
`locks`	$n = 16$	Parallel		Parallel
`barrier`	$n = 16$	Parallel		Parallel
`rwlock`	$n = 16$	Parallel		Parallel
`queue`	$n = 1000$	Parallel		Parallel
Data Parallel				
`blackscholes`	$n = 10000$, 5000 repeats	Parallel	✓	Parallel

Table 6.2: Summary of non-JCAM benchmark variants.

6.2 Benchmarks

The next three sections evaluate Dovetail and my analysis based on the benchmarks I describe here. The range selected comes from the type of programs used to measure the performance of related work [101, 113, 39, 33]. In each case, as well as the JCAM implementation, there are also versions in each of the following existing languages where appropriate—as summarised in Table 6.2.

C compiled with GCC. It should not be possible for any system to beat sequential C code on single-thread performance. In many cases, such a simple implementation allows optimisations that cannot be considered in a parallel setting—for instance, in-place modification (see `nqueens`). For the coordination benchmarks, the pthreads library was used which offers a fair, but beatable, baseline.

Wool [38] adaptations of the C version. For benchmarks that fit the fork-join paradigm, this offers a good indication of the best performance that can be hoped for.

Java. These versions offer an indication of the performance currently considered acceptable by developers, and are useful in showing the costs of a managed language. In the fork-join cases below, the Java version is single-threaded in order to focus on the cost of the pure JVM. For the coordination cases and `blackscholes`, implementations from `java.util.concurrent` are used to compare Dovetail with standard approaches.

6.2.1 `fib`

Calculating the Fibonacci sequence is a popular and widely used micro-benchmark, and has already been used throughout Chapter 4. It is also one of the main benchmarks used in the original evaluations of Wool. It is useful as an example of fork-join parallelism where the amount of actual work done between synchronisations is minimal. This allows true comparison of the coordination overhead of different approaches.

6.2.2 `nqueens`

This is another fork-join benchmark that has been used for the evaluation of both Wool and Cilk. It performs more work in between synchronisations and requires arrays to store possible solutions. In a sequential program, allocation is not necessary as modification can be made in-place. Under the fork-join model, the arrays can be allocated on the stack. However, this is not possible in general with a join-calculus encoding, which would therefore typically use heap allocation. For comparison to the Dovetail version, my JVM version (although single-threaded) also allocates a new array on the heap.

6.2.3 `quicksort`

A final fork-join case is the Quicksort algorithm. This differs from `nqueens` as most of the work is done when 'joining' the spawned threads, rather than in those threads themselves. The partitioning at each level is also not necessarily even, so the speedup is unlikely to be linear with the number of cores. Similar sorting benchmarks have been used in the evaluation of many other systems, including Wool, Cilk and JoCaml.

6.2.4 `locks`

This evaluates the encoding of the simplest coordination primitive: the mutex. The benchmark launches n threads, each of which acquire and release a common mutex 1,000,000 times. As with `fib`, this offers an evaluation of the overheads without the effect of actual computation. This approach was used in the evaluation of the Joins Library, and a similar example of incrementing a shared counter was used for evaluating JoCaml.

6.2.5 `barrier`

The `barrier` benchmark is of a similar style to `locks`. All n threads block on a barrier 1,000,000 times. Again there is no computation between the synchronisations. The nature of the barrier means that the JCAM operations are even more closely synchronised than with the other coordination benchmarks. Indeed, this code picked up a number of subtle concurrency bugs in the Dovetail implementation which all other benchmarks missed. It is also worth remembering that this was a benchmark where the lock-free Joins Library implementation performed particularly well [113].

6.2.6 `rwlock`

The next type of coordination formulated as a specific benchmark is a reader-writer lock. This is again taken from the list of benchmarks used to evaluate the Joins Library. Each thread behaves in a similar manner to the `locks` benchmark, except that 25% of the time they acquire the write lock, and 75% of the time a read lock. This asymmetry and randomness in the benchmark is not present in the other benchmarks and tests a slightly different aspect of the implementation. It is also a characteristic that many real programs will exhibit.

6.2.7 `queue`

The final coordination benchmark is the "arbitrary-sized" queue data structure introduced in Section 3.4.2. This is a very pure encoding of a complex structure into the JCAM—no structures or arrays are used, simply join calculus definitions. The benchmark creates such a queue, along with n producing 'threads' and n consuming 'threads'. 1000 pieces of data are then sent (or received) by each over the queue. In my measurements, I use a value of 1000 for n. I compare this to Java's concurrent queue implementation, and also a standard C++ queue wrapped with a mutex.

6.2.8 `blackscholes`

My final benchmark is *Black Scholes*. This is an 'embarrassingly parallel' problem, with data parallelism available between all the threads. It is included as a single example of a program with a true real-world workload that has also been used in the evaluation of a number of other research efforts—for example, as part of the PARSEC suite of parallel benchmarks [11] (upon which my variants of the benchmark are based). This allows consideration of whether the overheads associated with the JCAM inhibit performance for normal programs. There is practically no coordination in this benchmark, apart from the final collection of results. However, the body of the computation is of course encoded into the JCAM and its continuation-passing style. The efficiency of Dovetail for sequential sections of code is therefore tested.

6.3 Sequential Performance

The baseline for the sequential performance of each benchmark is of course the C version. Alongside this, Java and Wool give an indication of acceptable overheads. The single-core results for each of the benchmarks are shown in Figure 6.1. For each case, two bars are shown for Dovetail—one making use of the *fast mode* optimisations presented in Section 4.3 and one not. For the coordination benchmarks without closed definitions, the results are essentially identical, with variations only due to experimental error. In the other cases, the significant benefits of this optimisation are apparent. The `locks`, `barrier`, `rwlock` and `queue` benchmarks are not supported by Wool, so no results are shown for Wool in those cases.

In the non-coordination benchmarks, where some real computation is being done, Dovetail does not impose prohibitive overheads. For all these cases, the runtime is within a factor of 3 of a standard C implementation. By using the optimisations in Chapter 4, most importantly fast-mode, the majority of the costs of the join calculus message queues are avoided. However, the pattern matching logic is still required as the compiler does not understand when messages might be produced, and a transition therefore enabled. Wool scores better in these cases, and I believe that this is because the fork-join model simplifies

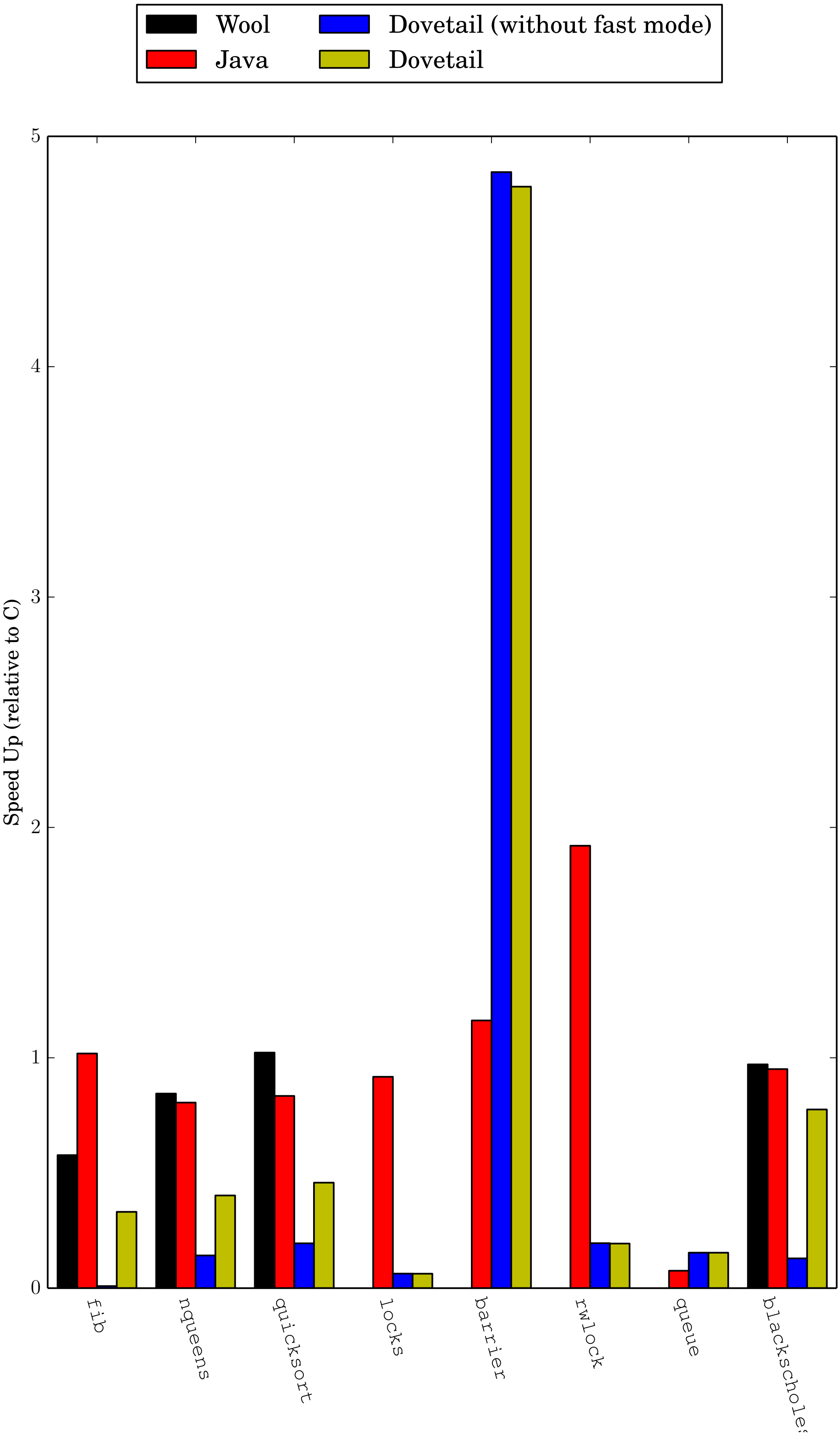


Figure 6.1: Sequential performance of Dovetail.

down to a sequential scenario better than the message matching primitives. Transition inlining is referred to throughout this thesis as a key area for future work, and I think that being able to use it to remove matching overheads in the sequential case will remove the remainder of the performance difference compared to C. It is reassuring that in the benchmark with the most realistic work load (`blackscholes`), the overhead is very low.

The results for the coordination benchmarks are harder to explain. Using Valgrind's *callgrind* tool [120] on `locks` shows that the matching operation is the main overhead. The form of each 'thread' in the JCAM version of the benchmark is as follows (`rwlock` and `queue` are of a similar form):

```
def @worker(acq,rel,k) ▷ values(1000000,acq,rel,k) & step1(),
    values(i,acq,rel,k) & step1() ▷
                    if i=0 then k()
                           else values(i,acq,rel,k) & acq(step2),
    values(i,acq,rel,k) & step2() ▷
                    values(i-1,acq,rel,k) & rel(step1)
```

Dovetail has no way of knowing that every emission to `values` will fail to match. Unfortunately, analysis to determine that the matching logic is unnecessary on emissions to `values` is beyond that presented in Chapter 5.

One feature developed by Turon and Russo in their work [113], which Dovetail has not yet implemented, is *tentative counters*. These allow message queues where a message does not contain any data (i.e. channels simply used as signals, such as `step1` and `step2` above) to be implemented more cheaply. They saw significant performance improvements when using these for the coordination-style of benchmarks where Dovetail struggles. The performance improvement if this were added to Dovetail would be less pronounced, as for the benchmarks concerned my 'cell' optimisation already removes many of the memory access overheads that the counters are designed to avoid. The challenge in integrating these counters with Dovetail is an engineering one, as they need to fit in with the specialised emission function approach that allows many of my other optimisations.

Although Figure 6.1 makes the performance of `barrier` on the JCAM look remarkably good, the explanation is actually that the performance of C and Java has changed significantly for the worse. The wall clock time for the JCAM is consistent between `locks`, `rwlock` and `barrier`, as one would expect since in each case 16 threads are looping and performing coordination 1,000,000 times, whereas the times for C and Java increase by about two orders of magnitude.

6.4 Scaling to Multiple Cores

The results in Figure 6.2 show that Dovetail performs very competitively in a benchmark exhibiting closedness that does not require memory allocation (i.e. `fib`). In the case of `nqueens`, the performance is still reasonable—however, each recursion requires allocation to represent the new state of the chessboard. Both C and Wool can make this allocation on the stack, while in the JCAM case it is made on the heap. The trailing off in performance scaling can therefore at least partially be attributed to the memory allocation cost.[2] However, in fast-mode, arrays that are passed to a closed definition and otherwise do not

[2] Although the Java version also allocates on each recursion, this becomes more costly when multiple threads are doing allocation.

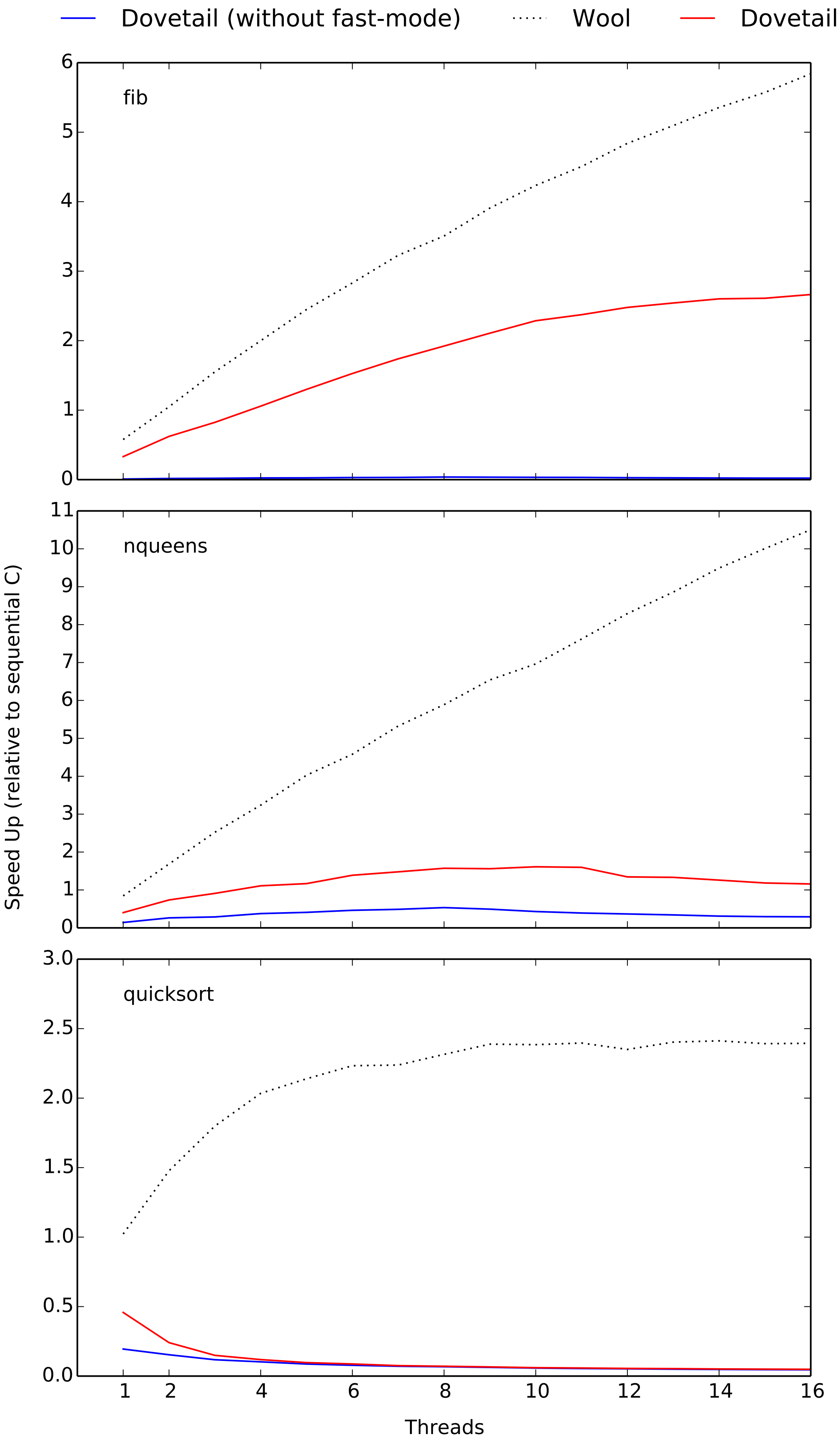


Figure 6.2: Multicore performance of Dovetail on fork/join benchmarks.

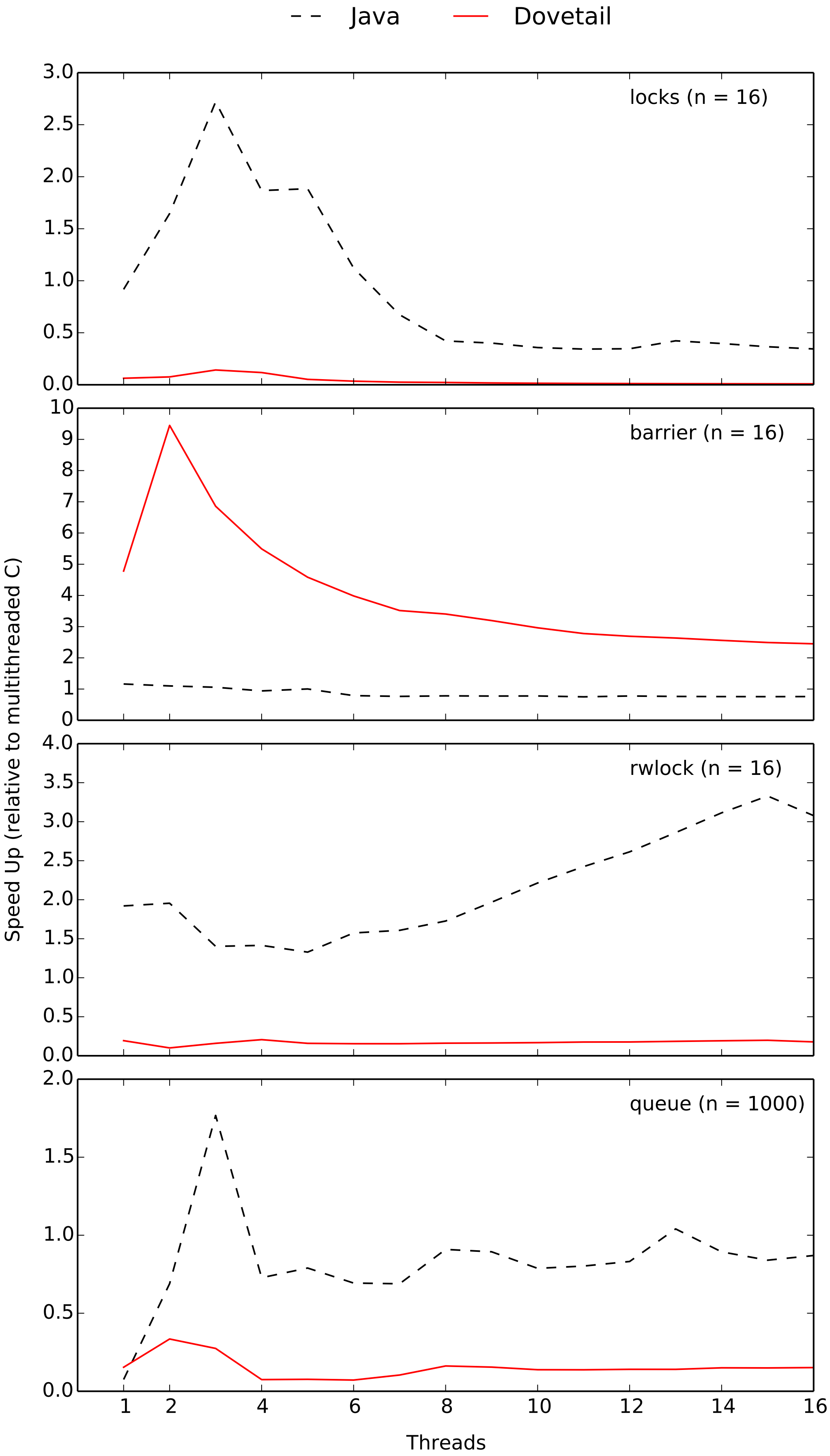


Figure 6.3: Multicore performance of Dovetail on coordination benchmarks.

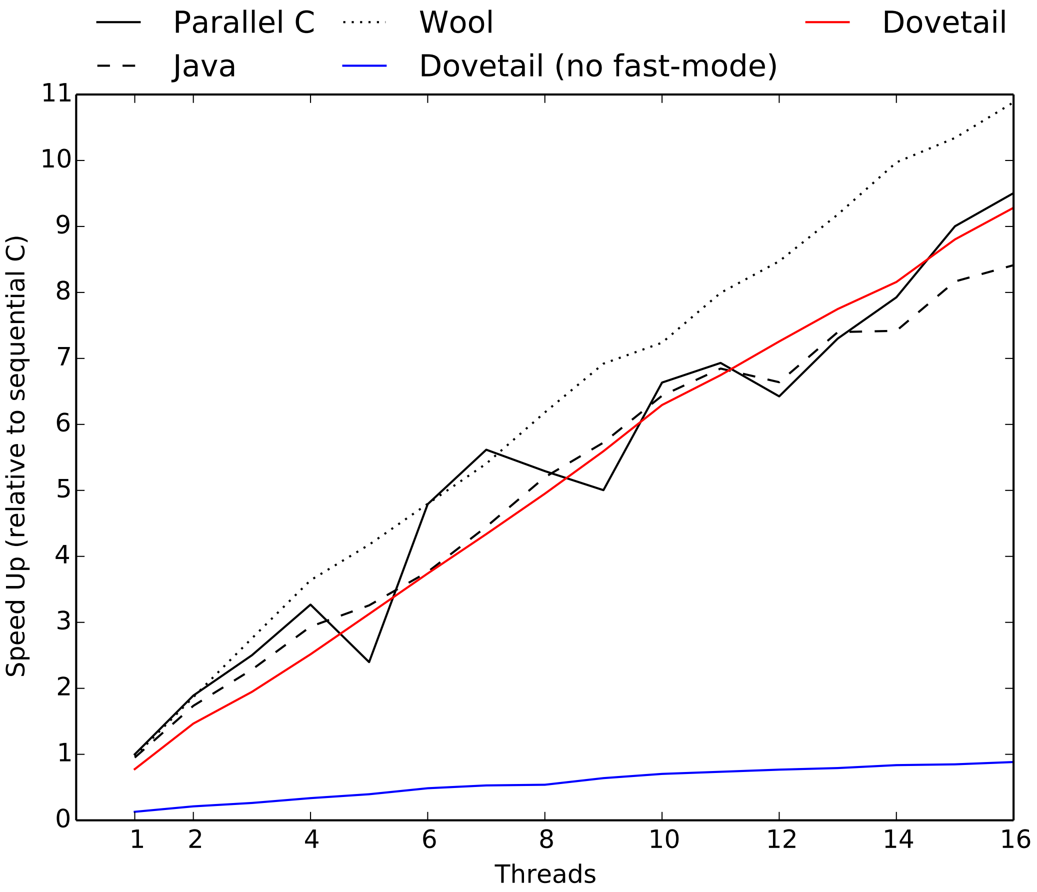


Figure 6.4: Multicore performance of Dovetail on `blackscholes`.

escape the transition (such as in `nqueens`) could be stack-allocated. Detecting this in a future compiler is achievable by incorporating standard pointer escape analysis with the closedness detection from my work.

The performance of `quicksort` is affected by cache locality. This is confirmed by running it with two threads on the same physical core (i.e. using hyperthreading), which results in a speed up compared to one thread. The array operations in Dovetail allow this benchmark to be written without allocating new arrays on each recursion, in the same way as the Wool version. However, my simple scheduler does not manage to allocate transition firings to workers in a way that maintains locality over the single shared array.

Moving on to the coordination benchmarks which have more complex communication (Figure 6.3), it is clear that, as in the Joins Library work [113], the `barrier` benchmark performs extremely well. The `locks` and `rwlock` cases struggle as they did in the sequential case. I believe that this is due to the lack of a tentative counter implementation and also the unnecessary match checking as discussed in the previous section, which also affects `queue`. In a multi-threaded case, these combine to cause high contention over the message channel queues, and therefore poor cache performance. However, it is encouraging that the performance of these benchmarks remains constant relative to the parallel C version as the number of threads increases. This suggests that the JCAM can be implemented such that it scales as well as other languages, whilst also offering better opportunities for heterogeneous implementations as discussed in Section 3.4.6.

It is worth noting that all four coordination benchmarks are programs that inherently scale badly. In each case, all n logical threads ($2n$ for `queue`) are competing for a single resource. Runtime is therefore improved when they are constrained to one hardware thread. As soon as multiple threads are used, cache locality becomes a significant problem. It is for this reason that I have presented these results as speed ups relative to the parallel C version, rather than to the single-threaded C execution time as for the other benchmarks.

The continuation-passing style of the JCAM makes it very easy for logical 'threads' within programs (e.g. the producers and consumers in `queue`) to be switched between workers, and therefore hardware cores, when they are matched upon. As mentioned in Section 2.4.3, the Joins Library distinguishes between synchronous and asynchronous channels, with normally exactly one synchronous channel appearing in a join pattern. It then ensures that a worker follows the synchronous message sends, preventing the unnecessary switches.

Finally, the performance for the `blackscholes` benchmark is given in Figure 6.4. It is unsurprising that this scales well, as it is an *embarrassingly parallel* benchmark. However, recall that it is entirely implemented with JCAM primitives. Therefore, even the sequential parts of the benchmark make use of message sends and pattern matching. The strong results therefore show that the emission function specialisation is having the desired effect. Enabling fast-mode in this benchmark allows the use of memory channels, avoids the enqueuing and dequeuing of transition matches, and also reduces memory allocation when calling a maths subroutine (for calculating the cumulative normal distribution function).

Overall, whilst there is room for improvement in a number of cases, I feel that the general trend of the results in this section is encouraging for the JCAM, and supports my claims that its implementation need not be prohibitively burdened by overheads. As well as evaluating the analysis in Chapter 5, the remainder of this chapter also discusses how further improvements in performance might actually be achieved.

6.5 Accuracy of Analysis

The performance figures in the previous two sections were attained for JCAM programs with queue bounds and closedness properties manually specified. In this section, I show that many of these are accurately determined by the techniques of Chapter 5. First, I consider closedness, since it gives rise to the most dramatic performance improvements, before considering queue bounds. I also give synthetic examples that highlight specific scenarios where the analysis might be improved.

With 1-LCFA, closedness is detected in the `fib`, `nqueens` and `quicksort` benchmarks. In the case of `blackscholes`, the use of library functions (e.g. `sqrt`) means that the analysis needs more understanding of their behaviour. I do not consider this a fault of the CFA itself, but rather the prototype implementation that I have built.

The CFA is equally proficient at detecting memory channels of the form used in numerous benchmarks to encode state internal to an instance. The queue bounding approach from Section 5.5 is poor at detecting cell channels that are emitted to outside a constructor. However, even if the queue bounding itself were more sophisticated, both `fib` and `quicksort` expose such channels to other instances of the definition in order to receive their results. The instance-local CFA approach that I have developed therefore considers them to have escaped and would never permit bounding. Inlining the child definitions (e.g. as in Section 4.5.2) allows these to be analysed correctly. Of course, in a recursive scenario such as `fib`, an infinite number of inlinings would be required to analyse all channels correctly.

At the moment, the domain of abstract instance identifiers is $\{\text{this}, \text{other}\}$. Definition inlining highlights that it is possible to give true call-strings to any instances created only in the constructor, or its descendent constructors. This would allow better accuracy without actually doing the inlining mentioned above. I also believe that it would allow recursive definitions to be understood in a way that simple inlining never could.

It is worth considering a few cases where my k-LCFA produces inaccurate results (for any k). Firstly, it is unaware of any ordering of calls that might be enforced by the program. Consider the example (`compare` is assumed to be a system call which prints “Yes” or “No” depending on whether its arguments are equal):

```
def @main()     ▷ i(a) & a(),
    a() & i(x) ▷ i(b) & b() & compare(a, x),
    b() & i(x) ▷               compare(b, x)
```

Clearly the printed message should always be “Yes”. However, the analysis cannot tell that specific calls to `i` are forced to join with each of the channels, and therefore concludes that either `a` or `b` could be passed as the second argument to `compare` on each occasion—a refinement of my approach that allows the flow variables in $\hat{\Gamma}$ to change over time might address this.

The second source of imprecision is more expected (and reminiscent of how tuples are analysed in an independent-attribute approach [61]). Consider:

```
a(k) & b(m,x) ▷ k(m,x)
c(m,x)        ▷ m(x)
```

with calls of `a(c)`, `b(p,q)` and `b(r,s)`. The call to `c` is considered while `a` is on the foreground path. It therefore receives the argument sets $\{\mathtt{p}, \mathtt{r}\}$ and $\{\mathtt{q}, \mathtt{s}\}$ for `b`, and cannot determine whether p receives argument q or s. A relational method would address this, but at some cost in algorithmic complexity.

6.6 Better Scheduling

For the shared-memory multi-core implementation presented in Chapter 4, I opted for a simple work-stealing scheduler. This resolved non-determinism simply based on the order that transitions are written in the program, rather than by considering any time estimations. Section 4.5 also conceded that transition inlining in the JCAM was not possible without some consideration of scheduling due to the need to keep the original transitions to avoid any risk of deadlock. I use this section to discuss the scheduling and inlining topics in more depth.

6.6.1 The Scheduling Problem

The notion of a JCAM program path was defined in Section 3.3.2 using pomsets along with a duration function. The problem of finding an optimal[3] execution trace allows one to consider all performance non-determinism choices together. This is not limited to placement and scheduling, but also programming model specific choices, such as *which thread should get access to the lock first* and *how many times should divide and conquer be performed for the algorithm to best match the available parallelism.*

I consider the problem in two stages. Firstly, it must be possible to find traces of minimum duration assuming $\Downarrow$ is completely defined, and therefore checking path feasibility is trivial. This corresponds to analysis of programs without conditionals. Once solved, extending this to the complete problem requires runtime analysis, since input values will affect which paths can occur. Fortunately, the assumption of confluence means that there are not any *wrong* choices (just slow ones). One can imagine using profiling data to produce probability distributions on the output channels of a transition, in a similar manner to branch predictors.

Unfortunately, even the first part is an NP-hard problem. The proof uses a reduction from the *exact cover* problem, which is very similar to the reduction from *3-dimensional matching* for AND/OR network scheduling [50]. These problems are both known to be NP-complete [62].

Theorem 3 *Checking whether a path exists between two markings is NP-hard.*

Given a set X, and a set $Y \subseteq \mathcal{P}(X)$ of subsets, an exact cover is a set $Y^* \subseteq Y$ such that for each element of X it appears in exactly one element of Y^*. Determining whether an exact cover exists for a given X and Y is known to be NP-complete [62].

I will encode this in a single join calculus definition as follows. For each $x \in X$, I include a channel $f_x \in \mathbb{C}$. I also introduce a constructor channel **start**. Now for each subset $y = \{x_1, \ldots, x_n\} \in Y$, a transition $t_y \in \mathbb{T}$ is added such that:

$$\begin{aligned} {}^{\bullet}t_y &= [\mathbf{start}] \\ t_y^{\bullet} &= [\mathbf{start}, f_{x_1}, \ldots, f_{x_n}] \end{aligned}$$

[3] Optimal would most frequently refer to minimum duration, but could also be based on other metrics such as energy consumption.

It can then be determined whether an exact matching Y^* exists by checking whether there is a path from $\{\textbf{start}\}$ to $\{\textbf{start}\} \cup X$. This proves that the problem of finding a path between markings is also NP-hard. □

6.6.2 Improvements on Basic Work Stealing

At various points in this thesis, my arguments have relied on a scheduler being able to make non-deterministic choices corresponding to the fastest execution—and clearly these need to be made quickly. This section discusses how this might be achieved in practice.

The simplest improvement to my current scheduler would be to use a static heuristic to reorder the transitions within the matching process, and possibly even remove some transitions from consideration. A major advantage of this is that it allows all the existing work on minimising overheads to be used without modification. Such a static scheduler need not be particularly fast itself and could be run on installation of the program. There are several further steps that would allow this approach to become more sophisticated.

1. Use different transition orderings on different types of processor.
2. Compile multiple copies of a definition with different orderings, and choose between these on instance construction. This would allow basic load balancing between different implementations.
3. Use garbage collection on work queues to support aborting computations (see the end of Section 3.4.3).

With just-in-time compilation, the second of these could even allow adaptation based on runtime measurements.

However, unlike in standard *work stealing*, there is also a second form of stealing that I have not yet considered—as well as taking a matched transition, individual messages could be taken by first decomposing some of the existing matches. This moves away from the greedy and *list scheduling* approaches advocated by StarPU (and most other work) and may not offer many benefits since work stealing is based on the premise that most matches are executed by the worker that created the match.

I will now consider some similar scheduling problems and how they can be related to the JCAM.

Task Graphs. Scheduling of these is a similar problem, and despite being NP-complete, effective heuristics do exist for it. StarPU is the closest to the scenario in this thesis since it considers choices between implementations of a task. They find HEFT [112] (a list scheduler) to be the most effective technique. However, this can only resolve *intra-task* choices, not *inter-task*. That is to say, there is never a choice between routes through the graph. They have also shown how this can be extended to consider memory communication costs [6].

AND/OR Graphs. These share the non-deterministic choice characteristic that appears in the join calculus, and is generally uncommon elsewhere. Typical solutions again make use of list scheduling [37].

Dataflow and Streaming. In typical task-graph scheduling problems, precedence constraints do not allow the concept of a completed task being 'used up' in the way

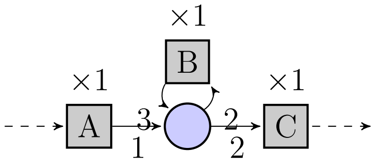


Figure 6.5: A hypergraph flow that is not a valid path (A must fire *twice* for B or C to fire).

that messages are in the join calculus, and cycles therefore behave differently. This *dataflow*-like situation is seen with *streaming models* (e.g. [111]), where the input and output rates of pipeline stages need to be matched. Adaptive runtime approaches have been shown to produce good results [108] for such pipelines, as have machine learning techniques [118].

Graph Flows. Whilst not typically associated with scheduling, flows through graphs share many similarities, with the flow into and out of each node needing to match much like producers and consumers of a channel. It would be necessary to use *hypergraphs*[4] since transitions match on multiple channels, and produce multiple messages. The *minimum cost flow* problem is most relevant since, for a given program, it should be known how much data will be input and expected as output. However, this minimises total cost rather than the critical path.

In the standard graph case, there are efficient *cost-scaling* algorithms [51] for this which could be implemented without a single 'master' scheduler. However, work on hypergraphs [25] appears to be restricted to 'B-hypergraphs' where each edge has only a single head.

Unfortunately, the standard definition of 'hyperflows' fails to describe execution paths exactly (for example, Figure 6.5). A flow would also need to be supplemented by an actual schedule for execution.

Note that since a flow will not necessarily use every edge, graph flows naturally allow non-determinism.

However, it is unclear how the above approaches can be combined. Rather than attempting to solve the problem in general, an easier first step might be to attempt extending the scheduler to a more complex situation such as GPU architectures. Even doing this with work stealing is non-trivial for the join calculus (Section 7.1).

Many of the approaches are also specific to static graphs and dependency relations. The higher-order nature of the join calculus means that any solution will require the results of the CFA as well as an estimation of which channels are more likely to be chosen as message destinations. This suggests that machine learning approaches may be the most appropriate, with the program viewed as a variety of stochastic game.

6.6.3 Transition Inlining

Whenever a λ-calculus CFA resolves the destination of a call-site to a single function, a compiler can consider inlining. Similarly, wherever my CFA resolves the destination of an

[4] A (directed) hypergraph is here a digraph where each hyperedge e has multiple heads and multiple tails—i.e. $e \in \mathcal{P}(V) \times \mathcal{P}(V)$.

emit to a single channel, one might hope to inline transitions to reduce firing overheads, as mentioned in Section 4.5. However, although inlining should reduce overheads, it may also reduce the available parallelism.

Consider the following example for expressions P and Q, having free variables x and y respectively:

```
def @f(x,y,k)          ▷ p(x,m) & q(y,n) & s(k),
    p(x,k)             ▷ k(P),
    q(y,k)             ▷ k(Q),
    m(a) & n(b) & s(k) ▷ k(a * b)
```

The behaviour of a call @f(x,y,k) is to invoke k on P * Q. If there is insufficient parallelism to fire p and q concurrently, it is preferable to inline these, together with the final multiplication a * b, to eliminate the overheads associated with passing messages and firing transitions—resulting in the optimised code of:

```
def @f(x,y,k) ▷ k(P * Q)
```

However, since channels and join patterns are in a many-to-many relation, it may be necessary to resolve multiple **emit**s before being able to inline (as in this example). There may also be a choice between multiple transition matchings (i.e. 'static' scheduling to resolve non-determinism).

The inlinings that are possible become clearest by constructing a Petri-net version of the LCFA results for the definition. In this net, places correspond to channels, and the pre-places of a transition are given by its join pattern. The post-places are given by **emit** instructions with a resolved destination, which are statically known to be executed a fixed number of times. Valid inlinings then correspond to valid transition mergings in this Petri-net—these can be represented as pomset paths [87] which might include repetitions of a single transition.

The pomset paths restrict the ordering of the original transition bodies within the merged transition's body. Any **emit** or **finish** instructions, which become internal due to inlining, should be removed, and local variables used to thread values between the original transition bodies.

The deadlock complications involved in join calculus inlining were briefly discussed in Chapter 4. A more subtle case of this occurs when the new transition matches on a channel that it might also emit. For example, inlining just the channel b in:

```
a()        ▷ b() & c()
b() & c() ▷ ...
```

gives:

```
a() & c() ▷ c() & ...
```

Assuming c does not appear elsewhere, then the former allows a() to cause a firing but not the latter, potentially causing deadlock. This therefore relies on using a more sophisticated scheduler as just discussed, and retaining the original transitions.

Note that because my CFA only considers message interaction within a single instance, transition inlining alone cannot support whole-program inlining (e.g. if p or q above were in a different definition). It therefore needs to be combined with definition inlining as discussed in Section 4.5.

6.7 Summary

This chapter has shown that Dovetail and the JCAM concept are viable in terms of performance on multi-core architectures. In Sections 6.3 and 6.4, it was demonstrated that speed ups across multiple cores could scale as well as any approach, even though there were inevitably costs compared to more established compilers.[5] For programs with 'real' computation, I do not believe that these costs would be significant, as the vast majority of the program would be made up of bounded channels and closed definitions. Section 6.5 then discussed my control-flow analysis and concluded that, while it is an encouraging first step in analysis of the JCAM, there is still plenty of scope for further analysis that would enable better performance to be achieved entirely automatically. The area that presents the greatest remaining challenges is scheduling, and implicitly inlining. While the JCAM is successful in allowing a wide range of execution choices to be expressed, it is not clear how these choices can be best resolved. Section 6.6.2 discussed possible approaches for this, but further research is needed.

[5] The benchmarks utilising arrays, `nqueens` and `quicksort`, did not scale as well and require further work on reducing memory allocation and improving locality when stealing work.

Chapter 7

Conclusion

Motivated by the widely acknowledged shift towards parallelism, and the lack of standardised compiler techniques to support this change, this dissertation has attempted to find a common higher-level view of parallel languages and architectures. Doing so has necessitated a broad sweep through different areas of compiler research. The result is an approach based on the join calculus, through which I have made the following contributions:

- **Chapter 3** developed a novel intermediate representation, the *Join Calculus Abstract Machine*, appropriate for representing the forms of parallelism present in modern languages and architectures, without losing information. Whilst the join calculus itself has existed for nearly two decades and inspired much programming language research, this is the first work to test its suitability for an intermediate representation. The *flattening* of the calculus and its condensation into just a few key primitives are also both novel.

- **Chapter 4** built an implementation (available online [21]) for this abstract machine based on recent work [113] before proceeding to look at how performance could be further improved. This isolated two key program properties that could be used to significantly decrease overheads. Whilst *queue bounding* had been touched on by previous work [66], the *closedness* of definitions offered an entirely new approach that gives huge benefits to execution speed once hardware parallelism is exhausted.

- **Chapter 5** produced two new *control-flow analyses* appropriate for the join calculus, and demonstrated how these could be used to infer the annotations required by the implementation.

- Finally, **Chapter 6** evaluated the overall performance of the machine and its suitability for various benchmarks. It also gave detailed discussions of several areas where I believe the current implementation and analyses fall down.

Using these techniques, I have argued that the JCAM is a sensible solution as a universal and fundamental representation of parallelism. As a novel model, one would also expect that the results in this work are only the beginning of what could be achieved with such a representation. This casts significant doubt on the Berkeley ‘stovepipe‘ view (i.e. that a new compiler should be written for each language-target combination), and suggests that it may be possible to pair up frontends and backends arbitrarily. Of course, any compiler targeted at a specific combination can outperform a universal approach, but the hope is that with further research the JCAM can be competitive enough that the development effort saved outweighs the performance costs.

There are three key areas for future work on the JCAM: compilation to other architectures, more accurate analysis, and better scheduling and inlining. Some of these have already been discussed in Chapter 6. However, they are summarised in the following sections, before I offer some final remarks on this work.

7.1 Compilation to Other Architectures

Whilst the design of the JCAM was made with compilation to non-uniform memory, heterogeneous architectures in mind, the prototype compiler introduced in this dissertation is restricted to shared memory multi-core targets. There are two obvious routes to supporting a wider range of hardware which I describe here. These descriptions should be treated as sketches of a solution rather than a tested or fully developed approach.

The first would be to attempt to target GPUs. The difficulty in doing this is discovering data parallel workloads—i.e. extracting SIMD from MIMD. This equates to detecting when multiple matches of the same transition rule will occur, and offloading these to a GPU. I envisage this being possible through a combination of static analysis and runtime monitoring. The static analysis would highlight points in the code where data parallel execution might be applicable. For instance, if messages are sent to a channel from within a loop, an annotation might be inserted before and after the loop. The annotation before would prevent (or perhaps just discourage) any matches generated during the loop from being immediately stolen by another worker. The annotation after the loop would then check for multiple matches of the same transition on the work queue, offloading them all to a GPU.

The other development would be to make use of the ideas in Section 3.4.6 to support distributed memory architectures, or clusters. This could be done whilst initially keeping the processing nodes themselves homogeneous. Of course, doing this also requires a more complex scheduler than my prototype. This has been discussed in Section 6.6.2 and will be recapped below.

7.2 Further Analysis

This dissertation developed the first control-flow analysis techniques for the join calculus, and made use of their results to infer several key properties (Chapter 5). However, as discussed in Section 6.5, there are areas where this CFA could be improved—for instance, inter-transition flow-sensitivity. Since this CFA has been used to drive the majority of the optimisations within my compiler, small improvements in accuracy could have a relatively large impact on overall program performance.

There are no doubt also other analyses that have not even been touched on here. One avenue that should be explored would be how existing analyses on parallel languages translate into the JCAM. For instance, work has been done with X10 on *may-happen-in-parallel* (MHP) analysis [2]. As a common substrate for parallelism, it should be possible to perform this on the JCAM. Not only will this further test the universality claim of the JCAM, but applying the MHP-equivalent to other paradigms not found in X10 may provide useful new insights into those languages.

7.3 Better Scheduling and Inlining

Inlining, which in the presence of non-determinism is effectively static scheduling, and runtime scheduling are areas that this dissertation has not explored in depth. Given the importance of inlining in the optimisation of traditional languages and intermediate representations, it is reasonable to expect that research in this area could lead to significant performance improvements. The hand-optimised inlining results given at the end of Section 4.5.2 were encouraging, and it would be valuable to be able to perform this automatically on a larger range of benchmarks. However, resolving non-determinism sufficiently is difficult and will require significant further research. A detailed discussion of the issues involved was given in Section 6.6.2.

7.4 Final Remarks

In closing, I want to offer a more pragmatic view of the next steps for the JCAM. It would be naïve not to realise that adopting an IR with such a pure application of the join calculus model is an incredibly disruptive change. However, there is of course a middle ground. Supporting the JCAM primitives alongside existing instruction sets would enable different styles of concurrency to be expressed in a common way that the compiler could optimise. Even without the analysis and optimisation described in the later parts of Chapter 4 and 5, performance would exceed that of library implementations. This would offer a far richer way of describing concurrency in managed languages. Improvement in this area is needed by both mainstream virtual machines (i.e. the JVM and .NET). It will therefore be interesting to see how progress is made over the coming years, and whether the ideas presented here offer any inspiration.

Bibliography

[1] Adve, V., Lattner, C., Brukman, M., Shukla, A., and Gaeke, B. LLVA: A low-level virtual instruction set architecture. In *36th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO-36, pages 205–216. IEEE, December 2003.

[2] Agarwal, S., Barik, R., Sarkar, V., and Shyamasundar, R. K. May-happen-in-parallel analysis of X10 programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 183–193. ACM, 2007.

[3] Alpern, B., Cocchi, A., Fink, S., and Grove, D. Efficient implementation of Java interfaces: `INVOKEINTERFACE` considered harmless. In *16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 108–124. ACM, 2001.

[4] Appel, A. W. *Compiling with Continuations.* Cambridge University Press, 1992.

[5] Armstrong, J., Virding, R., Wikström, C., and Williams, M. *Concurrent Programming in ERLANG.* Prentice Hall, 1993.

[6] Augonnet, C., Clet-Ortega, J., Thibault, S., and Namyst, R. Data-aware task scheduling on multi-accelerator based platforms. In *16th International Conference on Parallel and Distributed Systems*, ICPADS '10, pages 291–298. IEEE, 2010.

[7] Augonnet, C. and Namyst, R. A unified runtime system for heterogeneous multi-core architectures. In *Euro-Par 2008 Workshops-Parallel Processing*, volume 5415 of *LNCS*, pages 174–183. Springer-Verlag, 2008.

[8] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *15th International Conference on Parallel Processing (Euro-Par '09)*, volume 5704 of *LNCS*, pages 863–874. Springer-Verlag, 2009.

[9] Benton, N., Cardelli, L., and Fournet, C. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):769–804, September 2004.

[10] Benton, N., Kennedy, A., and Russell, G. Compiling Standard ML to Java bytecodes. In *3rd ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 129–140. ACM, 1998.

[11] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC benchmark suite: Characterization and architectural implications. In *17th International Conference on*

Parallel Architectures and Compilation Techniques, PACT '08, pages 72–81. ACM, 2008.

[12] Blelloch, G. E. NESL: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, September 1995.

[13] Blelloch, G. E. and Chatterjee, S. VCODE: A data-parallel intermediate language. In *3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 471–480. IEEE, 1990.

[14] Blelloch, G. E. and Sabot, G. W. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, February 1990.

[15] Blumofe, R. D. and Leiserson, C. E. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science*, FOCS '94, pages 356–368. IEEE, 1994.

[16] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[17] Boehm, H. Transactional memory should be an implementation technique, not a programming interface. In *1st USENIX Conference on Hot Topics in Parallelism*, HotPar '09, page 15. USENIX Association, 2009.

[18] Boehm, H., Demers, A., and Weiser, M. Boehm-Demers-Weiser garbarge collector. `http://www.hboehm.info/gc/`. [Accessed 15-September-2014. Version 7.4.0].

[19] Bunzli, D. C. and Laneve, C. A timed join calculus. Technical Report UBLCS-2002-1, Department of Computer Science, University of Bologna, February 2002.

[20] Calvert, P. Parallelisation of Java for graphics processors. *Final-year dissertation at University of Cambridge Computer Laboratory*, 2010.

[21] Calvert, P. Dovetail. `https://github.com/prc33/dovetail`, 2014.

[22] Calvert, P. and Mycroft, A. Petri-nets as an intermediate representation for heterogeneous architectures. In *17th International Conference on Parallel Processing (Euro-Par '11)*, volume 6853 of *LNCS*, pages 226–237. Springer-Verlag, 2011.

[23] Calvert, P. and Mycroft, A. Control flow analysis for the join calculus. In *19th International Conference on Static Analysis (SAS '12)*, volume 7460 of *LNCS*, pages 181–197. Springer, 2012.

[24] Calvert, P. and Mycroft, A. Mapping the join calculus to heterogeneous hardware. In *5th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, PLACES, pages 45–51, 2012.

[25] Cambini, R., Gallo, G., and Scutellà, M. G. Flows on hypergraphs. *Mathematical Programming*, 78(2):195–217, August 1997.

[26] Catanzaro, B., Garland, M., and Keutzer, K. Copperhead: Compiling an embedded data parallel language. In *16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 47–56. ACM, 2011.

[27] Catanzaro, B., Kamil, S., Lee, Y., Asanovic, K., Demmel, J., Keutzer, K., Shalf, J., Yelick, K., and Fox, A. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *1st workshop on Programming Models for Emerging Architectures*, PMEA. Citeseer, 2009.

[28] Cavé, V., Zhao, J., Shirako, J., and Sarkar, V. Habanero-Java: The new adventures of old X10. In *9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61. ACM, 2011.

[29] Chase, D. and Lev, Y. Dynamic circular work-stealing deque. In *17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28. ACM, 2005.

[30] Chen, T., Raghavan, R., Dale, J. N., and Iwata, E. Cell Broadband Engine architecture and its first implementation: A performance view. *IBM Journal of Research and Development*, 51(5):559–572, September 2007.

[31] Christadler, I. and Weinberg, V. RapidMind: Portability across architectures and its limitations. In *Facing the multicore-challenge*, volume 6310 of *LNCS*, pages 4–15. Springer, 2010.

[32] Coarfa, C., Dotsenko, Y., Mellor-Crummey, J., Cantonnet, F., El-Ghazawi, T., Mohanti, A., Yao, Y., and Chavarría-Miranda, D. An evaluation of global address space languages: co-array fortran and unified parallel C. In *10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 36–47. ACM, 2005.

[33] Conchon, S. and Le Fessant, F. JoCaml: Mobile agents for Objective-Caml. In *1st International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, ASAMA '99, pages 22–29. IEEE, 1999.

[34] Dagum, L. and Menon, R. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, January 1998.

[35] Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *6th Conference on Symposium on Operating Systems Design & Implementation*, OSDI'04, page 10. USENIX Association, 2004.

[36] *Standard ECMA-335 - Common Language Infrastructure (CLI)*. ECMA International, 5th edition, December 2010.

[37] Erlebach, T., Kääb, V., and Möhring, R. H. Scheduling AND/OR-networks on identical parallel machines. In *Approximation and Online Algorithms*, volume 2909 of *LNCS*, pages 123–136. Springer, 2004.

[38] Faxén, K.-F. Polyvariance, polymorphism and flow analysis. In *5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *LNCS*, pages 260–278. Springer-Verlag, 1997.

[39] Faxén, K.-F. Wool: A work stealing library. *SIGARCH Computer Architecture News*, 36(5):93–100, June 2008.

[40] Faxén, K.-F. Efficient work stealing for fine grained parallelism. In *39th International Conference on Parallel Processing*, ICPP '10, pages 313–322. IEEE, 2010.

[41] Faxén, K.-F. and Ardelius, J. Manycore work stealing. In *8th ACM International Conference on Computing Frontiers*, CF '11, pages 10:1–10:2. ACM, 2011.

[42] Fernández, B. M. An investigation of the join calculus abstract machine. Master's thesis, University of Cambridge Computer Laboratory, 2012.

[43] Fluet, M., Rainey, M., Reppy, J., Shaw, A., and Xiao, Y. Manticore: A heterogeneous parallel language. In *2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 37–44. ACM, 2007.

[44] Forum, T. M. MPI: A message passing interface. In *1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 878–883. ACM, 1993.

[45] Fournet, C. and Gonthier, G. The reflexive ChAM and the join-calculus. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 372–385. ACM, 1996.

[46] Fournet, C. and Gonthier, G. The join calculus: a language for distributed mobile programming. In *Applied Semantics*, volume 2395 of *LNCS*, pages 268–332. Springer, 2002.

[47] Fournet, C., Laneve, C., Maranget, L., and Rémy, D. Inheritance in the join calculus. In *20th Conference on Foundations of Software Technology and Theoretical Computer Science (FST TCS 2000)*, volume 1974 of *LNCS*, pages 397–408. Springer-Verlag, 2000.

[48] Fournet, C., Maranget, L., Laneve, C., and Rémy, D. Implicit typing à la ML for the join-calculus. In *8th International Conference on Concurrency Theory (CONCUR '97)*, volume 1243 of *LNCS*, pages 196–212. Springer-Verlag, 1997.

[49] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223. ACM, 1998.

[50] Gillies and Liu. Scheduling tasks with AND/OR precedence constraints. In *2nd Symposium on Parallel and Distributed Processing*, SPDP '90, pages 394–401. IEEE, 1990.

[51] Goldberg, A. V. and Tarjan, R. E. Finding minimum-cost circulations by successive approximation. *Mathematics of Operations Research*, 15(3):430–466, July 1990.

[52] Gudka, K. *Lock Inference for Java.* PhD thesis, Imperial College London, December 2012.

[53] Hammarlund, P., Chappell, R., Rajwar, R., Osborne, R., Singhal, R., Dixon, M., D'Sa, R., Hill, D., Chennupaty, S., Hallnor, E., et al. 4th Generation Intel® Core Processor, codenamed Haswell, August 2013. Slides presented at the 25th Symposium on High Performance Chips (Hot Chips 25).

[54] Harris, M. Mapping computational concepts to GPUs. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05. ACM, 2005.

[55] Heintze, N. Set-based analysis of ML programs. In *1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 306–317. ACM, 1994.

[56] Hewitt, C., Bishop, P., and Steiger, R. A universal modular ACTOR formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann, 1973.

[57] Hoare, C. A. R. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[58] Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[59] Jensen, K. Coloured petri nets: A high level language for system design and analysis. In *Advances in Petri Nets*, volume 483 of *LNCS*, pages 342–416. Springer, 1991.

[60] Johnson, N. and Mycroft, A. Combined code motion and register allocation using the value state dependence graph. In *12th International Conference on Compiler Construction (CC '03)*, volume 2622 of *LNCS*, pages 1–16. Springer-Verlag, 2003.

[61] Jones, N. D. and Muchnick, S. S. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In *21st Annual Symposium on Foundations of Computer Science*, FOCS '80, pages 185–190. IEEE, 1980.

[62] Karp, R. M. *Reducibility among combinatorial problems.* Springer, 1972.

[63] Khaldi, D., Jouvelot, P., Ancourt, C., and Irigoin, F. Task parallelism and data distribution: An overview of explicit parallel programming languages. In *25th International Workshop on Languages and Compilers for Parallel Computing (LCPC '12)*, volume 7760 of *LNCS*, pages 174–189. Springer, 2013.

[64] Khaldi, D., Jouvelot, P., Irigoin, F., and Ancourt, C. SPIRE: A methodology for sequential to parallel intermediate representation extension. In *17th Workshop on Compilers for Parallel Computing*, CPC '13, 2013.

[65] Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., and Kobayashi, H. Evaluating performance and portability of OpenCL programs. In *5th International Workshop on Automatic Performance Tuning*, 2010.

[66] Le Fessant, F. and Maranget, L. Compiling join-patterns. *Electronic Notes in Theoretical Computer Science*, 16(3):205–224, 1998.

[67] Leung, A., Lhoták, O., and Lashari, G. Automatic parallelization for graphics processing units. In *7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 91–100. ACM, 2009.

[68] Lewis, J. and Neumann, U. Performance of Java versus C++. *Computer Graphics and Immersive Technology Lab, University of Southern California*, January 2003.

[69] Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. *The Java Virtual Machine Specification.* Oracle, 2011.

[70] Macrakis, S. From UNCOL to ANDF: Progress in standard intermediate languages. *Open Software Foundation*, 1993.

[71] Mainland, G. and Morrisett, G. Nikola: Embedding compiled GPU functions in Haskell. In *3rd ACM Haskell Symposium on Haskell*, Haskell '10, pages 67–78. ACM, 2010.

[72] Michael, M. M. and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *15th Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275. ACM, 1996.

[73] Might, M. Abstract interpreters for free. In *17th International Conference on Static Analysis (SAS '10)*, volume 6337 of *LNCS*, pages 407–421. Springer-Verlag, 2010.

[74] Might, M. and Shivers, O. Analyzing the environment structure of higher-order languages using frame strings. *Theoretical Computer Science*, 375(1-3):137–168, April 2007.

[75] Might, M. and Van Horn, D. Family of abstract interpretations for static analysis of concurrent higher-order programs. In *18th International Conference on Static Analysis (SAS '11)*, volume 6887 of *LNCS*, pages 180–197. Springer-Verlag, 2011.

[76] Milner, R., Parrow, J., and Walker, D. A calculus of mobile processes, part I. *Information and Computation*, 100(1):1–40, September 1992.

[77] Moore, G. E. et al. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, April 1965.

[78] Munshi, A. The OpenCL specification v1.0. *Khronos OpenCL Working Group*, 2009.

[79] Murray, D. G. *A distributed execution engine supporting data-dependent control flow.* PhD thesis, University of Cambridge Computer Laboratory, 2011.

[80] Newburn, C. J., So, B., Liu, Z., McCool, M., Ghuloum, A., Toit, S. D., Wang, Z. G., Du, Z. H., Chen, Y., Wu, G., Guo, P., Liu, Z., and Zhang, D. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235. IEEE, 2011.

[81] Nielsen, M., Plotkin, G., and Winskel, G. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1):85–108, 1981.

[82] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of program analysis.* Springer-Verlag, 1999.

[83] *Compute Unified Device Architecture programming guide.* NVIDIA, 2007.

[84] Odersky, M. Functional nets. In *9th European Symposium on Programming Languages and Systems (ESOP '00)*, volume 1782 of *LNCS*, pages 1–25. Springer-Verlag, 2000.

[85] Odersky, M., Zenger, C., Zenger, M., and Chen, G. A functional view of join. Technical Report ACRC-99-016, University of South Australia, 1999.

[86] Patterson, D. Steps towards heterogeneity and the UC Berkeley Parallel Computing Lab. In *DTTC Conference*, June 2011.

[87] Pratt, V. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, February 1986.

[88] Pratt-Szeliga, P. C., Fawcett, J. W., and Welch, R. D. Rootbeer: Seamlessly using GPUs from Java. In *14th International Conference on High Performance Computing and Communication & 9th International Conference on Embedded Software and Systems*, HPCC '12, pages 375–380. IEEE, 2012.

[89] Randal, A., Sugalski, D., and Toetsch, L. *Perl 6 and Parrot Essentials.* O'Reilly, 2nd edition, 2004.

[90] Randall, K. H. *Cilk: Efficient Multithreaded Computing.* PhD thesis, Massachusetts Institute of Technology, 1998.

[91] Reinders, J. *Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism.* O'Reilly, 2007.

[92] Reppy, J. Type-sensitive control-flow analysis. In *2006 workshop on ML*, ML '06, pages 74–83. ACM, 2006.

[93] Reppy, J. and Xiao, Y. Specialization of CML message-passing primitives. In *34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 315–326. ACM, 2007.

[94] Reppy, J. H. Synchronous operations as first-class values. In *ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 250–259. ACM, 1988.

[95] Reppy, J. H. Concurrent ML: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *LNCS*, pages 165–198. Springer-Verlag, 1993.

[96] Rhodin, H. A PTX code generator for LLVM. *Bachelor's Thesis at Saarland University*, 2010.

[97] Richards, M. BCPL: A tool for compiler writing and system programming. In *Spring Joint Computer Conference*, pages 557–566. ACM, May 1969.

[98] Rose, J. R. Bytecodes meet combinators: `INVOKEDYNAMIC` on the JVM. In *3rd Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2:1–2:11. ACM, 2009.

[99] Roy, A., Hand, S., and Harris, T. A runtime system for software lock elision. In *4th ACM European Conference on Computer systems*, EuroSys '09, pages 261–274. ACM, 2009.

[100] Russell, R. M. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[101] Russo, C. The joins concurrency library. In *9th International Conference on Practical Aspects of Declarative Languages (PADL '07)*, volume 4354 of *LNCS*, pages 260–274. Springer-Verlag, 2007.

[102] Saraswat, V. A., Sarkar, V., and von Praun, C. X10: Concurrent programming for modern architectures. In *12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, page 271. ACM, 2007.

[103] Serrano, M. Control flow analysis: A functional languages compilation paradigm. In *Symposium on Applied Computing*, SAC '95, pages 118–122. ACM, 1995.

[104] Shirako, J., Peixotto, D. M., Sarkar, V., and Scherer, W. N. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *22nd Annual International Conference on Supercomputing*, ICS '08, pages 277–288. ACM, 2008.

[105] Shivers, O. *Control-flow analysis of higher-order languages.* PhD thesis, Carnegie Mellon University, 1991.

[106] Singh, S. Computing without processors. *Communications of the ACM*, 54(8):46–54, August 2011.

[107] Steel Jr, T. B. A first version of UNCOL. In *Western Joint IRE-AIEE-ACM Computer Conference*, pages 371–378. ACM, May 1961.

[108] Suleman, M. A., Qureshi, M. K., Patt, Y. N., et al. Feedback-directed pipeline parallelism. In *19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 147–156. ACM, 2010.

[109] Sutter, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr Dobb's Journal*, March 2005.

[110] Taylor, M. B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.-W., Lee, W., et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, 22(2):25–35, March 2002.

[111] Thies, W., Karczmarek, M., and Amarasinghe, S. P. StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction (CC '02)*, volume 2304 of *LNCS*, pages 179–196. Springer-Verlag, 2002.

[112] Topcuouglu, H., Hariri, S., and Wu, M.-y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[113] Turon, A. J. and Russo, C. V. Scalable join patterns. In *2011 ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA '11, pages 575–594. ACM, 2011.

[114] Valiant, L. G. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[115] van der Wijngaart, R. F., Mattson, T. G., and Haas, W. Light-weight communications on Intel's Single-chip Cloud Computer processor. *SIGOPS Operating Systems Review*, 45(1):73–83, February 2011.

[116] Von Itzstein, G. S. and Kearney, D. Join Java: An alternative concurrency semantic for Java. Technical Report ACRC-01-001, University of South Australia, 2001.

[117] Wadge, W. W. and Ashcroft, E. A. *LUCID: The Dataflow Programming Language*. Academic Press Professional, 1985.

[118] Wang, Z. and O'Boyle, M. F. Partitioning streaming parallelism for multi-cores: A machine learning based approach. In *19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 307–318. ACM, 2010.

[119] Watt, D. *Programming XC on XMOS devices*. XMOS Limited, 2009.

[120] Weidendorfer, J., Kowarschik, M., and Trinitis, C. A tool suite for simulation based analysis of memory access behavior. In *4th International Conference on Computational Science (ICCS '04)*, volume 3038 of *LNCS*, pages 440–447. Springer, 2004.

[121] Zhao, J., Nagarakatte, S., Martin, M. M., and Zdancewic, S. Formalizing the LLVM intermediate representation for verified program transformations. In *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 427–440. ACM, 2012.

[122] Zhao, J. and Sarkar, V. Intermediate language extensions for parallelism. In *Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 329–340. ACM, 2011.