A PARALLEL FUNCTIONAL LANGUAGE COMPILER FOR MESSAGE-PASSING MULTICOMPUTERS

Sahalu B. Junaidu

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews

1998

A Parallel Functional Language Compiler for Message-Passing Multicomputers

Ph.D. thesis submitted

by

Sahalu B Junaidu

School of Mathematical and Computational Sciences
University of St Andrews

Th
D 04

Godiya tā tabbata ga Allāh Ubangijin halittu.

Acknowledgements

All praises are due to Allāh (God) for His boundless bounties and, in particular, for giving me the good health and the strength of determination to enable me to carry out this work.

I register my profound gratitude to my Parents who have trained me, taught me, amongst other things, to take life easy while encouraging me to learn to face, on my own, the multi-faceted challenges of life.

Financial sponsorship from the Federal Government of Nigeria is gracefully acknowledged, with particular mention of the effort of the Nigerian Education Attaché in London, Alhaji Sani B Idris. I'm grateful to my Head of Department, Umar M Makarfi, for his effort and encouragement.

I'm much indebted to Gerald Ostheimer for introducing me into the field of parallel programming, for his ideas and help and for his encouragement during the course of this research.

To Kevin Hammond for helping me out with many intricacies of C programming, for guiding and polishing my thoughts on parallelism and parallelism measurements and for his continuous willingness to entertain half-baked ideas without prior arrangements. I am grateful to my supervisor, Tony Davie, for his help and understanding, to Ron Morrison for his help and for allowing me to use his personal printer with which coloured pages of this thesis were printed and to the other members of the Department—the Secretaries and Scientific Officers, especially—who have helped in one form or another for this work to come to fruition.

I'm grateful to Philip W Trinder and Hans-Wolfgang Loidl for the many discussion we have had and for the use of their parallel programming tools.

I record my indebtedness to my friend and colleague, Mansur D Liman, for his assistance, encouragement and advice. I acknowledge the support—through Abdul-Ilāh Al-Arfaj and Abdul-Muhsin Al-Arfaj—of the International Islamic Relief Organisation, Makkah, Saudi Arabia. I'm also grateful to the Hebdomadar—the students' welfare officer—St Andrews University, for his assistance.

I'm very grateful to the members of the St Andrews University Islamic Society—both past and present—for their Brotherly company and for helping me keep afloat spiritually, morally and otherwise. To AbdurRahman Ishāq (*Ndagi Audu*) for his sincere advice and for 'running me around' in his car for shopping and other things. And to my many friends who have prayed for me and whose confidence in me seemed greater than my own.

Finally, and on a more personal note, I acknowledge the support, patience and understanding of my wife Nafisat and our two young daughters, Yusra and Hājara, and for maintaining a good and happy home.

Abstract

The research presented in this thesis is about the design and implementation of **Naira**, a parallel, parallelising compiler for a rich, purely functional programming language. The source language of the compiler is a subset of Haskell 1.2. The front end of Naira is written entirely in the Haskell subset being compiled. Naira has been successfully parallelised and it is the largest successfully parallelised Haskell program having achieved good absolute speedups on a network of SUN workstations. Having the same basic structure as other production compilers of functional languages, Naira's parallelisation technology should carry forward to other functional language compilers.

The back end of Naira is written in C and generates parallel code in the C language which is envisioned to be run on distributed-memory machines. The code generator is based on a novel compilation scheme specified using a restricted form of Milner's π-calculus which achieves asynchronous communication. We present the first working implementation of this scheme on distributed-memory message-passing multicomputers with split-phase transactions. Simulated assessment of the generated parallel code indicates good parallel behaviour.

Parallelism is introduced using explicit, advisory user annotations in the source program and there are two major aspects of the use of annotations in the compiler. First, the front end of the compiler is parallelised so as to improve its efficiency at compilation time when it is compiling input programs. Secondly, the input programs to the compiler can themselves contain annotations based on which the compiler generates the multi-threaded parallel code. These, therefore, make Naira, unusually and uniquely, both a parallel and a parallelising compiler.

We adopt a medium-grained approach to granularity where function applications form the unit of parallelism and load distribution. We have experimented with two different task distribution strategies, deterministic and random, and have also experimented with thread-based and quantum-based scheduling policies. Our experiments show that there is little efficiency difference for regular programs but the quantum-based scheduler is the best in programs with irregular parallelism.

The compiler has been successfully built, parallelised and assessed using both idealised and realistic measurement tools: we obtained significant compilation speed-ups on a variety of simulated parallel architectures. The simulated results are supported by the best results obtained on real hardware for such a large program: we measured an absolute speedup of 2.5 on a network of 5 SUN workstations.

The compiler has also been shown to have good parallelising potential, based on popular test programs. Results of assessing Naira's generated unoptimised parallel code are comparable to those produced by other successful parallel implementation projects.

Declarations

I, Sahalu B Junaidu, hereby certify that this thesis, which is approximately 83K words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

date 30/3/98 signature of candidate

I was admitted as a research student in February 1992 and as a candidate for the degree of PhD in February 1993; the higher study for which this thesis is a record was carried out in the University of St. Andrews between 1992 and 1998.

date 30/3/98 signature of candidate

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St. Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date 30/3/98 signature of supervisor

In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

date 30/3/98 signature of candidate

Contents

List of Figures

Chapter 1

Thesis overview

1.1 Goals and contributions

The work reported in this thesis is part of an SERC-funded research project on the development of a parallel compiler technology for functional programming languages. The two broad aims of the project are to:

1. Develop compiler technology to generate scalable, topology-independent parallel code from functional programs.

2. Apply this technology to our implementation of compilers themselves and investigate a layered methodology of compiler building which will supply large amounts of data parallelism in addition to the process parallelism normally expected.

In his PhD thesis, Ostheimer [Osth93] proposed a framework for evaluating implicitly parallel functional programs, a software-based load bounding scheme and a processor architecture model which supports the evaluation model efficiently.

Part of the research presented in this thesis builds on Ostheimer's work and part of it is focussed towards realising the second goal of the project. The aim of the thesis was therefore to develop a pilot compiler for the project, parallelise it and implement the back end to generate multi-threaded parallel code using the ideas espoused by Ostheimer as a basis. Accordingly, the research starts by developing and implementing a sequential compiler, from scratch, called Naira, for a rich functional language. The main contributions of the thesis are itemised below.

- *Design and implementation of a parallel functional language compiler, Naira.* A complete sequential compiler was first crafted (Chapter 4) including stream I/O and runtime system support (Chapters 5, 6) before the parallelisation proceeded as mentioned in the other contributions enumerated in this section. Naira is the second largest parallel Haskell program ever written, about 6K lines compared with Lolita with 47K lines, and the largest to be written explicitly with parallelism in mind.

- *Extension and implementation of many compile-time program analyses.* After lexical and syntax analyses, the research extended, implemented and parallelised four main other compiler phases—pattern matching, lambda lifting, type inference and intermediate language optimisations (Chapter 4).

- *Extensive application of the parallel programming technology of Trinder et al [THLP98].* A wealth of experience has been built from using evaluation strategies in small-sized programs where the actual workings of the technology has been explored, in the parallelisation of the benchmark programs of Chapter 8, and in applying the technology to parallelise Naira itself (Chapter 7).

- *Design and implementation of a parallel name-server.* Allied to the use of evaluation strategies to exploit parallelism is the use of a parallel name-server which creates unique names to enable otherwise data-dependent computations to proceed in parallel (Chapter 4).

- *Compiling a lazy, purely functional language via π-calculus.* Naira is the first parallel compiler (for lazy functional languages) that generates parallel code using compilation rules specified using an asynchronous π-calculus (Chapter 5).

- *Design and implementation of the* **process** *and* **value** *annotations.* The design, implementation and demonstration of the use of the **process** and **value** annotations, as a vehicle for specifying parallelism and strictness in user programs, have successfully been realised in this research (Chapters 5, 8).

- *Extension of Ostheimer's π-calculus-based compilation scheme for a first-order functional language to cover an expressive higher-order functional language.* Ostheimer's work was first extended to cover a complete first-order language by adding compilation rules for code-generating case-expressions, individual modules and complete programs. It was then significantly enhanced by adding rules for higher-order functions. A working implementation of the complete rules is first provided in this thesis (Chapters 1, 2, 5, 8).

- *Generating multi-threaded parallel code based on the extended compilation scheme.* Based on Naira's sequential code generator, multi-threaded parallel code is generated using the annotation information in the intermediate language and the quality of the generated code assessed (Chapters 5, 8).

- *Achieved good absolute speedups on both simulated and real hardware.* The parallel compiler is successfully assessed using the latest technology both on simulators and on real hardware. A wall-clock speedup of 2.5, and a relative speedup of 2.7 have been measured on a network of five SUN workstations. To our knowledge, this absolute speedup has not been achieved for a similarly large, irregular, parallel lazy functional program as Naira.

1.2 Source language

The language base of our compiler is a subset of standard Haskell 1.2 [HPW92][1], a strongly-typed, higher-order, non-strict purely functional language. The features of standard Haskell omitted in this implementation are the following

- Type classes.
- Fixity declarations.
- The **renaming** clause in **import** declarations.
- We assume modules are not mutually recursive.
- I/O requests on binary files[2] and channels (see Chapter 6) because they are not used by our compiler; they can be added (in similar way as I/O on text files is handled) if required.

Clearly, the most significant syntactic omission is type classes which are essentially used to implement *ad hoc* polymorphism uniformly. These omissions are made solely to simplify the implementation (of overloading) and do not affect the main concern of the thesis—exploitation of parallelism. We discuss our implementation of some *ad hoc* operations on structured objects in Chapter 6 so as to make up for the omission of type classes.

We find it convenient to refer to this language simply as Haskell. References to Haskell without qualification will, therefore, henceforth refer to the subset under consideration in the implementation. The standard Haskell language will be referred to as *full* Haskell.

[1]The latest edition of Haskell, version 1.4 [PeHa+97], is not used because the implementation was well underway before the birth of Haskell 1.4.

[2]As a matter of fact, we are not aware of any released Haskell compiler which supports binary files.

The front end of the compiler is written entirely in Haskell which is the language we compile. The compile time transformations implemented (see Section 1.4 below) eventually produce an intermediate language 'middle end' which is quite suitable for code generation. The back end is written in C and our principal target is, following the tradition of, for example [Peyt92, LaHa92], to generate code in the C language which can then be handled in the usual way by a C compiler.

1.3 Annotations for parallelism

Following the example set by Burton and others, e.g., [Burt84, Huda91, THLP98, Acht91], our approach to parallel programming is to annotate the source program to indicate our intention for parallelism explicitly. Accordingly, the programmer assigns explicit parallelism (and strictness) annotations to components of tuples, arguments to function/constructor applications, and to the expression right-hand sides in non-combinator definitions. Basic arithmetic and Boolean operators are strict by default and so their arguments need not be annotated.

We make use of two built-in single-argument combinators, `process` and `value`, to partition programs statically. These combinators correspond, to a certain degree, to the `P` and `I` annotations in [Kess96], the `par` and `seq` strategies in [THLP98] and the `FORK` and `SUSP` abstract instructions in [Chak94], etc. The operational semantics of these combinators is that an application of `process` to an expression provides a hint to the compiler that the expression may be evaluated concurrently with an enclosing expression while the application of `value` to an expression indicates that the expression must be evaluated before passing it to an enclosing expression. An expression without either of these tags is evaluated when, and only when, its value is demanded by an enclosing expression (i.e., by need).

We define a Haskell data type, `Mexp`, for modal expressions (i.e., expressions that have annotations attached to them),

$$\begin{aligned}\texttt{data}\ \mathit{Mexp} &= \mathit{Need}\ \mathit{Exp}\\ &\mid \mathit{Process}\ \mathit{Exp}\\ &\mid \mathit{Value}\ \mathit{Exp}\end{aligned}$$

which is used to reflect the programmer-inserted source annotations in our intermediate language. The code generator makes use of this information to generate the appropriate code for a given expression.

In order to give a more concrete flavour of our annotations, consider the two function calls `(f (process g)(process h))` and `(f (value x) (value y))`. First, `g` and `h` must both be function applications in order for the effect of `process` to apply. In the call `(f (process g)(process h))`, the calls for g and h will be created and distributed to different processors of the machine and all three calls for `f`, g and h can proceed in parallel (see Section 5.5.4 for a more detailed example). For the call `(f (value x) (value y))`, however, `x` and `y` can be any expression forms (but not annotated, see the `Mexp` type)—they will be evaluated on the same processor on which `f` is called *before* the call is made (Section 8.3.1 gives a detailed explanation of these annotations and their implications). An important point to note is that, `process` causes the evaluation of its argument to *weak head normal norm* (WHNF) while `value` causes the evaluation of its argument *hyper-strictly* to full *normal form.*

With this introduction to the `process` and `value` annotations, we can now consider the runtime behaviour of our compiler on the following realistic but simple program:

Example 1.1

$parmap\ f\ [\,] \quad = [\,]$

$parmap\ f\ (h{:}t) = process(f\ h)\ {:}value(parmap\ f\ t)$

$take\ 1\ (parmap\ g\ [3,\ 1/0,\ 7])$

The `value` annotation in the recursive call to `parmap` forces the evaluation of the spine of the entire list and the use of `process` causes the operation `f` to be applied to all the elements of the list *in parallel.* This particular encoding of `parmap`, therefore, essentially realises data-parallelism. The value of this program depends on the strictness of g: if g is a strict function the program returns undefined otherwise the singleton list `[3]` is returned as expected. If the `value` annotation is replaced by `process`, however, the program terminates regardless of the strictness of g since the latter annotation does not force evaluation of its argument (beyond *weak head normal form*) before the argument is demanded.

With these annotations, we create a more flexible parallel programming environment as in GpH, Glasgow Parallel Haskell, than for example using the `letpar` construct[3] of [LKID89, Chak94] since we use the standard let expression and simply annotate those bindings in the let that we wish to be parallel tasks.

Notice also that `process` and `value` annotations are used to maximise exploitable parallelism while `need`, the default, is used to maximise safety. We also point out that we make use of annotation strategies at two levels in the implementation. First, we use them in the compiler's source code, to parallelise the compiler itself (Chapter 7) and, secondly, they are used in the source

[3]The `letpar` construct is added as a new construct in the language. It has similar syntax to the ordinary `let` except that its definitions are all parallel tasks [LKID89, Chak94].

programs to be compiled using the compiler. These annotations do not specify process mapping decisions, in contrast to, for instance, those in [Huda91, Acht91], so that task placement is implicit and is handled dynamically.

1.4 Compiler structure

The overall structure of the Naira compiler is depicted in Figure 1.1. The front end of the compiler consists of five phases namely, lexical/syntax (the analysis phase), pattern matching, lambda lifting, type checking and abstract syntax tree (AST) optimisation phases. The analysis phase inputs the (possibly annotated) source program and produces an AST representation for it.

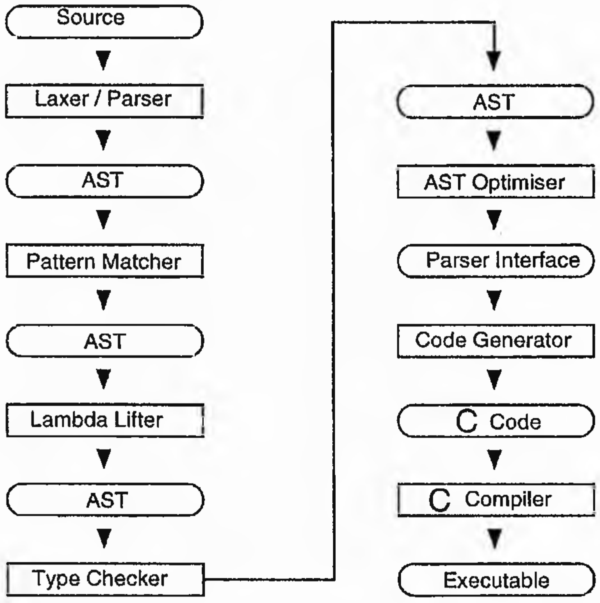

Figure 1.1: Structure of the Naira compiler

The pattern matching compiler takes the AST input, compiles the syntactic sugar inside patterns and produces an AST closer to the underlying computation model. The lambda lifter then inputs the resulting AST and turns functions into *supercombinators* [Hugh82, John87, PeLe91]. Lambda lifting is particularly beneficial in a parallel implementation since supercombinator invocations can be distributed across the processors of the parallel machine without worrying about access to free variables. The AST is then type checked and further simplified to produce an optimised intermediate language.

The intermediate language, which is given in the form of a Haskell data type (Section 4.3), is transformed to a corresponding C structure by the C parser interface (Section 5.5.2) for input to the C code generator. The code generator (Section 5.5) generates multi-threaded C code to be handled by a C compiler. Generating C has the benefits of enhancing portability because C is implemented on a wide variety of platforms [Peyt92]. Another benefit is that of getting a good code generator which is important for RISC machines.

In comparison to the highly optimised functional language [illegible]ers that are widely available, like the GHC, Glasgow Haskell compiler [PHH+93] and HBC, the [illegible] ML compiler [AuJo89b], our research compiler performs a smaller number of compile time transformations as depicted in Figure 1.1 (see also Section 4.2).

1.5 Runtime organisation

Efficient graph reduction requires a good node representation with modest memory requirements. Although the most compact representation of the graph nodes is [illegible] best [illegible] ideal node design should aim for efficiency with respect to space usage, node access and [illegible] [illegible]ion while remaining flexible for future development without requiring substantial changes to the node layout.

We represent graph nodes by three kinds of heap-allocated objects—function frames, constructed cells and suspension objects. Function frames have a simple flat structure (as recommended by Appel in [Appe92]). Constructed cells, which represent aggregate data values, are associated with family tags which distinguish them from their siblings. Data constructors also contain layout information[4] used by the evaluator and the garbage collector. A suspension object, which can be viewed as a thread descriptor, contains a status flag indicating its evaluation status (cf. [Peyt92, BHY88, FaWr87]).

As mentioned in Section 1.3, parallelism is introduced explicitly using annotations. We support a medium grain of parallelism where function calls form the units of granularity based on which tasks are created and distributed. This is based on the expectation that the underlying target architecture has an integrated network interface (along the lines of STAR:DUST [Osth91]) which supports very fast thread switching to tolerate high communication latencies. Parallel tasks are distributed across the machine processors using an active distribution scheme. That is, a processor does not have to request work before it is given any; a random task distributor, similar to that in [Kess96], sends tasks to processors where these tasks may be buffered before they are eventually evaluated (cf., [HMP94, Kess96, Chak94]).

Requests for data values are always issued by sending messages and the amount of information

[4]All three types of object contain house-keeping information for use by the evaluator and garbage collector.

carried in a request message is exactly what is required for the evaluation of the target object and communicating the result—i.e., it consists of a frame pointer, a code pointer inside the frame, pointers to the function's argument values and the return addresses of the requesting processes. We employ the notification model of inter-thread communication and synchronisation: using the tags in objects' cells to act as locks to ensure mutual exclusion and to avoid evaluating an object repeatedly (cf., [Peyt92, Iann88, NiAr89, CSS+91]).

Our target architecture is that of a distributed memory MIMD multiprocessor (the results reported here include those obtained on other machine architectures as well). We therefore adopt a message-passing communication interface between the machine's processors, the usual communication mechanism in multiprocessors that have no shared memory. Since message-passing is very pervasive in the implementation, our compiler generates a very large number of threads for an input program to provide each processor of the machine with many threads to execute so that a processor can switch very rapidly between them (rather than possibly staying idle) in response to remote memory reference latencies or synchronisation. Therefore the architectural requirements for a machine to support our model efficiently are that, it should support fast dynamic thread scheduling, provide tolerance to long-range communication latencies and support cheap and rapid switching between multiple executable threads (cf., [NiAr89, ArIa87, CSS+91]).

1.6 Thesis structure

The remainder of this thesis is structured as follows:

Chapter 2 Presents a short history of computers and the evolution of methodologies for programming them—leading to discussion of functional languages: their computational basis, sequential implementation methods and their costs.

Chapter 3 Discusses the motivation behind parallel computers, outlines the main issues addressed in a parallel programming system and surveys parallel functional language implementations related to our research.

Chapter 4 Presents the design and implementation of the front end of the compiler: symbol tables, lexer/parser, pattern matcher, lambda lifter, type checker and resulting intermediate language.

Chapter 5 Presents the design and implementation of the back end of the compiler: concrete representation of data in the heap, messages and message-passing, compilation rules and the code generation process.

Chapter 6 Describes our implementation of a stream I/O mechanism used by our compiler to communicate with the outside world. This chapter also covers the implementation of comparison operations on structured objects (since type classes are omitted) as well as scheduling issues.

Chapter 7 Describes the parallelisation of four main phases of our compiler namely, the pattern matcher, lambda lifter, type checker and the AST optimiser aimed at improving the efficiency with which Naira compiles input programs.

Chapter 8 Presents the second part of the compiler assessment, based on typical popular benchmark programs that serve as the basis for appraising many other parallel functional language implementations, like [AuJo89a, Mara91, Huda91, KLB91].

Chapter 9 Summarises the thesis, its main contributions and the limitations of work presented. The chapter also suggests future optimisations, further research directions and concludes.

1.7 Authorship

The work reported in this thesis can be broadly classified into three parts: the development of a sequential compiler which pilots the research, making the compiler run in parallel and making it a parallelising compiler for input programs.

The development of the compiler (front end) was successfully realised with the support of Tony Davie, Gerald Ostheimer and Norman Paterson at St Andrews with whom many fruitful discussions were held.

For parallelising the compiler, we made use of the tools and adopted the parallelisation methodology developed by researchers on Glasgow Parallel Haskell [HLT95, THLP98, Loid96]. In particular, we made heavy use of *GrAnSim* [HLP95, Loid96], a highly tunable state-of-the-art simulator developed at Glasgow and St Andrews Universities, in the performance measurements and evaluation of the compiler.

In chapter four of his PhD thesis, Ostheimer [Osth93] considered a first-order functional language described by the syntax shown in Figure 1.2. Based on this language Ostheimer presented a compilation scheme specified using Milner's π-calculus and wrote an initial sequential code generator based on these compilation schemes. Ostheimer also described two restrictions imposed on the general synchronous π-calculus required to ensure asynchronous message passing communication. He outlined and gave a draft code for a runtime system to be used by our compiler and suggested the use of the `process` and `value` annotations.

Prog	::=	$\mathrm{Def}_1 \cdots \mathrm{Def}_n$ Exp
Def	::=	f $\mathrm{id}_1 \cdots \mathrm{id}_n$ = Exp
Exp	::=	id
Exp	::=	**apply** f $\mathrm{Exp}_1 \cdots \mathrm{Exp}_n$
Exp	::=	**let** $\mathrm{id}_1 = \mathrm{Exp}_1, \cdots, \mathrm{id}_n = \mathrm{Exp}_n$ **in** Exp
Exp	::=	**if** Exp_1 **then** Exp_2 **else** Exp_3
Exp	::=	**cons** Exp_1 Exp_2
Exp	::=	$\mathrm{Exp}_1 + \mathrm{Exp}_2$

Figure 1.2: Ostheimer's first-order language.

Building on Ostheimer's work, we started by debugging his sequential code generator and runtime system to bring them to a reasonable working status. We then extended these adding I/O and module facilities and finally generated multi-threaded parallel code based on the `process` and `value` annotations. As described in Section 6.4, we wrote a parallel simulator which takes snapshots of the compiler's execution activities and generate graphical profiles from the collected statistical data. The programs used to generate the PostScript graphs are adapted (and used to cater for our data form) from a collection of Perl scripts in the GrAnSim Toolbox [Loid96].

The actual experiments described in Chapters 7 and 8 were done by myself, acknowledging the support of Kevin Hammond for straightening my thoughts and providing useful clues round the many software engineering hurdles encountered. All of the text in the thesis is written by me and is a record of my own research in the light of the aforementioned contributions by others.

Chapter 2

Background

The purpose of this chapter is to first give a brief historical overview of programming computers so as to put into perspective and to motivate the development of functional programming languages. Having exposed the ever increasing demands for an effective means of human computer interaction and the need for functional languages, we highlight the basic philosophy of functional programming side-by-side with programming in conventional imperative languages.

The chapter also gives a brief outline of the computational theory, the λ-calculus, on top of which functional languages are built. We also find this a convenient place to discuss the computational model, Milner's π-calculus, which underlies our compilation scheme. Finally we review the various proposals for implementing functional languages as well as highlighting their costs.

2.1 Introduction

The history of the development of computing goes as far back as the times when efforts were first made to develop the concept of numbers and counting. However, the era of modern computers is said to have begun in the late 1940s with the advent of the von Neumann computers which internally store programs that control their operations.

The art of programming these machines has, since then, been growing from strength to strength in order to meet the increasing demands of programmers to communicate with them effectively. The first most efficient, but most cumbersome, programming technique was to program directly in (binary) machine language. This soon became impracticable, because it was unmnemonic, unreadable and extremely tedious. The next step forward was the development of symbolic assembly languages in which the instructions could be represented by (mnemonic) symbols and decimal notation. Programming at this level was also found to be unsatisfactory because programmers' demands like the ability to use code written by other programmers were not always met due to the differences in notation and lack of an efficient way to link program pieces together [Samm69, Maye88].

As the cost of software production began to escalate, contributed by the tedium of assembly language programming, the need to develop new programming techniques became more acute. This necessitated and led to the development of high level programming languages which abstract away from the peculiarities of a particular machine and which enable the programmer to express complex instruction sequences and data structures directly. Although high level languages did not initially receive wide spread acceptance, good performance by early optimising compilers, notably the Backus' Fortran compiler [Back81], reversed the trend in favour of these languages. In fact, the interest in high level languages led to an explosion in their development by the late 1960s with many languages simply incorporating their designer's particular interests or concerns.

In spite of the tremendous success of high-level programming languages, there was still the quest for programming at a higher level of abstraction. Programmers were looking for simpler and more natural ways to express their algorithms with minimum bearing on the underlying computational model, trading program execution speed for simplicity.

In 1978 John Backus, the author of the first optimising compiler of Fortran, expressed a dissatisfaction with the conventional programming languages in his Turing award lecture [Back78]. He pointed out that with the explosive growth in the number of programming languages and the associated claims of improvements of the subsequent languages over their predecessors, these languages still retain the major language features which contribute to their weaknesses. These languages are criticised for their word-at-a-time style of programming, close coupling of semantics to state transition and lack of useful mathematical properties.

An important observation on the design of imperative languages is their intimate relationship with the underlying machines, the von Neumann computers, on which they run. For example, one of the design requirements of Fortran was to map to machine language with minimal loss of efficiency. The von Neumann computer has a profound influence on the nature of these languages and they are sometimes called von Neumann languages to emphasise their close marriage. In fact, imperative languages are essentially an abstraction of the von Neumann computer using variables to simulate the computer's storage cells, control statements to mimic its jump and test instructions, and assignments to act as the machine's fetch, store and arithmetic operations.

An alternative solution to the problems associated with conventional imperative languages requires the discovery of a new kind of language framework which supports a powerful methodology that helps programmers to think about programs. This framework should support a clear separation of concerns; the art of programming should be a distinct activity from that of handling administrative tasks like prescribing to the machine how a given problem is to be solved and how the machine's memory is to be managed [Read89]. The languages should be simple, expressive, extensible, less error-prone, elegant and susceptible to program transformations and optimisations. Functional programming languages, which have succeeded in breaking out of the mould of imperative programming, are claimed to have a wealth of these properties salient in them [Read89, FiHa88].

The following sections introduce functional languages, the theory underlying them, the strategies for implementing them and finally highlights their strength and weaknesses.

2.2 Functional languages

Functional languages are general purpose, high level programming languages supporting programming at a higher level of abstraction than conventional imperative languages like Fortran and C. Programming in functional languages is usually a *descriptive* or *declarative* activity which involves specifying 'only' *what* is to be computed while imperative programming is *prescriptive*, specifying also the *how* of the computation steps.

According to Sarkar [Sark91], program execution issues could be broadly classified as low level and high level. Low level execution issues involve register usage, low order versus high order byte ordering, instruction selection etc. and high level issues involve the specification of algorithms and data structures. With this classification, it is obvious that the functional programmer must specify only the high level issues of *how* programs are executed. On the other hand, imperative programming requires specifying the details of both low level and high level issues of program execution. Hence, the imperative programmer is forced to *overspecify* the control flow and data flow in the computation.

A major distinction between modern functional languages and their imperative counterparts is that the former do not allow *assignments* (i.e., destructive updates) to memory locations. Alternatively, functional languages only use *declarations* (which are technically different from single assignments) whereby a variable's value in a program once declared, does not change. The lack of assignments facilitates higher level programming since the concern of programming are separated from that of low level housekeeping of recycling memory locations enforced by repeated assignments. The absence of assignments in functional languages serves as an important prerequisite which confers these languages with a useful mathematical property, *referential transparency*[1].

[1]Imperative languages are sometimes referred to as being *referentially opaque* because they support functions

This property ensures that since there are no side-effects, the value of an expression in a program depends only on the values of its syntactically correct constituent expressions and not, for example, on the order in which the expressions are evaluated.

Functional languages are often classified, on the basis of their semantics, into *strict*, *non-strict* and *lenient*. Eager evaluation is usually used to implement strict semantics while *lazy evaluation* is the implementation technique often used to implement non-strict semantics. *Lenient evaluation* combines non-strictness with eager evaluation.

A function is strict in an argument x if, whenever the value of x is undefined, the result of the function is also undefined. A strict function is a partial function which is strict in at least one of its arguments. A non-strict function is a partial function that may be defined even when one of its arguments is not defined. Strict functional languages are therefore those that support strict functions while non-strict languages are those that support non-strict functions. Lenient languages combine the features of both strict and non-strict languages: they support functions which can return results even when their computation may not terminate. In other words, given a function application, eager evaluation first evaluates all the argument expressions and *then* evaluates the function's body.

Lazy evaluation starts evaluating the function body, evaluating the function's arguments as and only when they are used. Lenient evaluation starts the evaluation of the function in parallel with the evaluation of all the arguments of the function.

Lazy evaluation enables functional languages to express algorithms involving potentially infinite data structures succinctly. Such algorithms are awkward to express in a language without lazy evaluation. This is an important language feature although debate on whether its virtues outweigh its costs (due to associated space overheads) has been lively.

2.3 Computational foundation

In this section we briefly describe the theory which underlies the implementation of our pilot compiler in particular as well as the implementation of functional languages generally. This theory is Church's *lambda calculus*. After reviewing the main features of this calculus, we also touch on two other formalisms—term rewriting and graph rewriting systems—which could be adopted as the basis for implementing functional languages.

Lambda calculus is a simple mathematical theory developed by Alonzo Church [Chur41]. It

which refer to global data and make destructive updates on the data. Such side-effecting operations can cause the value returned by a function (making the destructive updates) to change even though its arguments may be the same each time it is called. This is demonstrated by a simple example in [FiHa88].

was initially intended to be a foundation of mathematics in the 1930s before the advent of digital computers. The λ-calculus is well studied and, although it is an austere language containing only a few syntactic constructs with clean semantics, it is powerful enough to express all functional programs. In fact, the expressive power of this simple language is shown to be the same as the expressive power of any computing engine. In other words, the λ-calculus is *Turing complete*: it can express any function that can be computed by a computing device.

The λ-calculus underlies the computational model of functional programming languages. The efficient implementation of modern functional languages on today's computer architectures depends on how efficiently the operations in the λ-calculus can be implemented on these machines. That is, given an implementation of the λ-calculus, a functional language can be implemented by translating it into the λ-calculus. This is why functional languages are often considered as mere syntactic sugar coating the λ-calculus. Chapter 4 of this thesis describes the translation of Haskell into an enriched λ-calculus and Chapter 7 presents the parallelisation of the translation process.

A λ-calculus expression is 'executed' by *reducing* it. There are three basic reduction rules in this calculus: α-reduction, β-reduction and η-reduction. β-reduction is the most important operation in the λ-calculus and the efficient implementation of functional languages depends on the efficient implementation of this operation. The reduction process proceeds by continuously selecting and reducing a *redex* (reducible expression). (We shall return to this point in Section 2.5). A good introduction to λ-calculus can be found in many books including [FiHa88, Bare84, Peyt87].

A redex can be reduced to *weak head normal form* (WHNF), in which only the topmost redex is reduced, or to *normal form*, in which the resulting expression is canonical and contains no further redexes. The main modes of evaluating an expression involve performing reductions until either normal form or WHNF is reached, depending on the semantics of the language (see the next section). Since a λ-expression can have multiple redexes, it is often useful to arrange a consistent way of selecting the redexes. This is because the efficiency and the termination property of the reduction process can be affected by the choice.

There are two most common strategies for performing reductions: *normal order* and *applicative order*. Normal order selects and reduces the leftmost outermost redex while applicative order always selects and reduces the leftmost innermost redex. Normal order reduction strategy is more expressive than applicative order because a sequence of reductions in the former may terminate with an answer (i.e., a reduced expression) while the sequence of reductions, for the same λ-expression, using the latter strategy may diverge without returning an answer. Examples of these cases can be found in [Read89, Peyt87, FiHa88, Loud93]. Conversely, there are no λ-expressions for which applicative order reduction may terminate while normal order diverges. This is why normal order is considered as an 'optimal' reduction strategy although it may be less efficient in some cases.

If the sequence of reductions for a λ-expression is known to terminate with an answer, the choice of reduction order for its evaluation is immaterial and the Church-Rosser theorem [ChRo36] guarantees the uniqueness of the result using either strategy. The referentially transparent nature of functional languages, which is so beneficial for simultaneous evaluation of expressions and which we alluded to above, is a consequence of this important result.

Some implementations of functional languages are based on normal order reduction strategy and some are based on applicative order reduction. There are also implementations which are based on a hybrid of both strategies as we expatiate in Section 2.5 below.

While recognising that the λ-calculus is very suitable for studying the basic semantics of functional languages, some researchers like [PlEe93], are of the opinion that the λ-calculus is not very suitable for studying implementation aspects of functional languages. Two other models that can serve as the computational basis for functional languages and their implementation are *Term Rewriting Systems* (TRSs) [HuOp80, Klop92] and *Graph Rewriting Systems* (GRSs) [Stap80, BvEG$^+$87a,b]. TRSs form a computational model based on pattern matching (i.e., syntactic equality test) on terms that do not contain variables. GRSs extend TRSs with the notion of sharing and in which the terms (trees) are replaced by directed graphs. TRS are related to the λ-calculus and can be regarded as an extension of combinatory logic. However, TRSs have more declarative power than both the λ-calculus and combinatory logic since, for example, non-determinism can be expressed in TRSs but not in λ-calculus or combinatory logic [PlEe93].

Computation in a TRS is specified by a set of rewrite (or reduction) rules. These rules are similar to function definitions in a functional language except that they are only defined globally. Plasmeijer and van Eekelen [PlEe93] claimed that it is easier to translate functional languages into the rewrite rules of TRSs than into the λ-calculus. This is why they said functional languages are closer to TRSs and that TRSs may be a more suitable basis for implementing these languages.

In contrast to functional languages, there is no ordering in the reduction rules of TRSs and also no reduction order is specified explicitly that determines how a given term is to be evaluated. However, and although TRSs can express non-deterministic computations, the lack of these important constraints mean that general TRSs are *non-confluent* (i.e., lack the Church-Rosser property) and hence lose the guarantee of uniqueness of normal forms. Furthermore, general TRSs do not have a decidable *normalising strategy*[2] which will terminate with the normal form of any term that is known to have a normal form. This can be recovered from by restricting general TRSs

[2]In fact the left-most outer-most (or normal order) strategy that is normalising in λ-calculus is not normalising even for orthogonal TRSs.

to obtain *orthogonal* TRSs that guarantee confluency and for which normalising strategies exist. Unfortunately, normalising strategies for general orthogonal TRSs are hard[3] to find and are not efficiently implementable. The only practical solution is to extend TRSs with priorities for selecting the rewrite rules, which then become very close to the usual top-down, left-to-right semantics of pattern matching in functional languages. The language Concurrent Clean [NSvEP91] (see Section 3.3) is based on rewriting systems of this form.

2.4 Milner's π-calculus

The motivation behind this section is to mention briefly the theory which underlies our compilation scheme (see Section 5.5) in a similar way as we described the λ-calculus which underlies functional languages. We therefore outline the basic concepts of the π-calculus, how it relates to both the λ-calculus and TRSs outlined above, and finally how it is used to specify our compilation rules.

The π-calculus [Miln92] is the result of a search for an algebraic framework which would capture the essence of the notion of concurrent processes. It can be seen as an extension of the theory of the *Calculus of Communicating Systems* (CCS) [Miln80] and other similar process algebras in that channel names (references) are the subject of communication. This calculus is designed to allow the direct description of systems, at a higher level of explanation, which change their configuration dynamically. *Processes* and *Channels* are the basic entities of π-calculus. Processes are the 'terms' or 'expressions' of the calculus and channels are the media through which the processes interact by sending values through and receiving values from.

Like the λ-calculus, the π-calculus is computationally complete and has only a few more syntactic rules than the λ-calculus. Therefore the π-calculus approaches λ-calculus in economy of expression. There are six syntactic rules in the π-calculus as shown in Figure 2.1 (in which P, Q are processes and x, y are channels).

The send operation describes the action of transmitting the value y along the channel x to the process P. The receive process in turn specifies the action of reading y from the channel x and then performs the action P. The empty process, O, is required to ground the syntactic rules. $P|Q$ specifies the parallel combination of two processes, P and Q, which act side by side to each and interact in whatever way they are designed to. The replication process, $!P$, is a short hand for $P|!P$, which means as many parallel composition of the process P as are desired. The restriction combinator restricts channels for use only by specific processes. Restriction is, in fact, just a

[3] Maranget [Mara92] showed that there is an optimising reduction leading to normal form for labelled orthogonal term rewriting systems, T^lRSs. Kennaway *et al* [KKSdV90] obtained this result for term graph rewriting systems via translation between term graph rewriting and infinitary term rewriting [KKSdV93].

$$\begin{array}{lcll}
P & ::= & \bar{x}y.P & \textit{send action} \\
 & | & x(y).P & \textit{receive action} \\
 & | & O & \textit{empty process} \\
 & | & P|Q & \textit{parallel composition(has the lowest syntactic precedence)} \\
 & | & !P & \textit{replication} \\
 & | & (x)P & \textit{restriction}
\end{array}$$

Figure 2.1: Syntax of Milner's π-calculus.

distillation of the notion of local variable declaration in programming languages. This process can be used to model security issues in computer systems [Miln96]. There is no separate rule for sequential composition in the calculus since it is, indeed, a special case of parallel composition. That is, since P and Q act side by side to each other in $P|Q$, sequential composition is a special case in which the only interaction occurs when P finishes and Q starts[4].

π-calculus terms are divided into three classes: guarded terms, terms that express concurrent behaviour and restricted terms. Guarded terms (the first two in the syntax of Figure 2.1) have the form $g.P$, where g is a guard and P is a term. Process composition, $P|Q$, is the principal term expressing concurrent behaviour with replication and empty process (the degenerate composition of no processes) being allied to it. The third class of terms has only one form: the last term in the syntax above [Miln92].

Reduction is possible in the π-calculus when there is a pair of a receive and a send action ready to communicate via a common channel. Communication takes place by cancelling receive and send actions substituting the value to be sent for any free occurrences in the receiving process of the variable to be received. Consider the term

$$\bar{x}z|x(y).\bar{x}y$$

with matching send and receive actions. Notice that the receive action $x(y)$ is followed by a send action, $\bar{x}y$, containing a single occurrence of y, of the variable being received. Reduction on this term yields

$$O|\bar{x}z$$

which is equivalent to $\bar{x}z$. In contrast, the term $\bar{x}z|(x)x(y)$ has no reduction because the send and

[4]This can be likened to the synchronisation behaviour in the λ-expression $(\lambda x.P)Q$ where Q remains passive until P activates it by passing control to it (i.e., assuming normal-order evaluation).

receive actions operate on different channels (the restriction $(x)x(y)$ creates a new channel name x and makes it private to (or whose scope is) $x(y)$). See [OsDa93, Miln92] for more examples.

The send and receive actions, as specified above, both represent synchronous operations where the process P cannot proceed until another process is ready to, respectively, receive or send a value along the channel x. In order to adapt the general synchronous behaviour of the π-calculus to asynchronous message-passing architectures, the send and replication actions can be restricted as follows. Ensure that send actions do not guard non-empty processes and permit replication only of those processes guarded by receive actions. Thus the two syntactic rules are specialised to $\bar{x}y$ and $!x(y).P$ respectively [BrOs95].

Both the λ- and π-calculi can be regarded as specialised term rewriting systems. While the λ-calculus is well studied with an agreed mathematical interpretation, research is still relatively more active in the study of the π-calculus. In terms of their dynamic behaviour, the λ-calculus has sequential, hierarchical control while π-calculus has a concurrent, heterarchical control behaviour [Miln93, p. 83]. The basic rule of computation in the former is function application (or β-reduction) and that in the latter is process interaction which passes a single datum between processes. While function application is neither commutative nor associative, parallel composition is both commutative and associative. This property coupled with the concurrent, heterarchical control behaviour of the π-calculus gives the main difference between these two term rewriting systems, namely that λ-calculus is deterministic and the π-calculus is non-deterministic.

$$
\begin{array}{lll}
[\![c]\!]o & \overset{def}{=} & \bar{o}c \\
[\![x]\!]o & \overset{def}{=} & (r)(\bar{x}r \mid r(v).\bar{o}v) \quad (alternatively : [\![x]\!]o \overset{def}{=} \bar{x}o) \\
[\![\lambda x.M]\!]o & \overset{def}{=} & (f)(\bar{o}f \mid !f(x,y).[\![M]\!]y) \\
[\![M\,N]\!]o & \overset{def}{=} & (m)(x)([\![M]\!]m \mid m(f).(\bar{f}(x,o) \mid bind(x,N))) \\
[\![M\,N]\!]o & \overset{def}{=} & (m)(n)(x)([\![M]\!]m \mid m(f).([\![N]\!]n \mid n(v).\bar{f}(x,o).store(x,v))) \\
[\![M\,N]\!]o & \overset{def}{=} & (m)(n)(x)([\![M]\!]m \mid [\![N]\!]n \mid m(f).n(v).\bar{f}(x,o).store(x,v)) \\
[\![M\,N]\!]o & \overset{def}{=} & (m)(n)(x)([\![M]\!]m \mid [\![N]\!]n \mid m(f).(\bar{f}(x,o) \mid n(v).store(x,v))) \\
bind(x,N) & \overset{def}{=} & x(r).(n)([\![N]\!]n \mid n(v).(\bar{r}v \mid store(x,v))) \\
store(x,v) & \overset{def}{=} & !x(r).\bar{r}v
\end{array}
$$

Figure 2.2: Ostheimer's encoding of λ-calculus into π-calculus.

Milner [Miln92] gave two different translations of the λ-calculus in the π-calculus. One of these is sequential and the other is parallel. Ostheimer, in [Osth93], gave another encoding of the λ-calculus in the π-calculus which combines both sequential and parallel behaviour within a single framework. In addition Ostheimer's scheme faithfully represents call-by-need with sharing, whereas Milner's 'lazy' scheme does not. These encodings are faithful to the computational behaviour of the λ-calculus in the sense that if the λ-calculus term being simulated terminates with a value, the corresponding π-calculus term delivers a representation of this value at the designated channel. Figure 2.2 shows the Ostheimer's encoding which serves as the basis of our compilation scheme.

A λ-calculus term M is encoded as a function, $[\![M]\!]$, which maps λ-calculus names directly to names of π-calculus channels. In order to be able to represent call-by-need reduction strategy, these names will not stand for values, rather they provide access to values upon request [BrOs95]. The π-calculus term $[\![M]\!]o$ uses the argument channel o as a link where $[\![M]\!]$ is to deposit its value. The first three rules in Figure 2.2 are, respectively, for constants, identifier reference and λ-abstractions. The next four encodings are for call-by-need, call-by-value, parallel call-by-value and call-by-process respectively. The last two rules are used to specify environment operations.

The rule for constants is obvious: to deposit the value of a constant into a channel, we simply put it there via a send operation. The value of an identifier x is obtained by sending a request r (i.e., a `sendEval` message to ensure that x is evaluated, see Section 5.4) along x. The value v of x will eventually be received on r. The same effect can be achieved more succinctly, perhaps less intuitively, in a continuation-passing style manner by sending o directly to x.

The encoding for abstraction shows that the representation of the function value, f, is immediately available at o. The replicated term (note that the replication is necessary because a function may be applied more than once) represents an 'activation server' which allows a function value to be applied to an argument. A process needing to apply a function sends an 'activation' (i.e., a pair of channels) consisting of (access to) the argument x and and the place where the result is required.

The rule for lazy function application specifies the evaluation of the function M and then applying it (i.e., sends a pair of channels, a request channel x and a destination channel o where the result is required, along f) to a suspended form of the argument N. When the value of N is required, a request for it is sent along x. The rule for *bind* therefore receives the (expected) request, evaluates N, satisfy the original request and store the value for future requests. Future requests for N (which are potentially many as indicated by the replication in the rule for *store*) are serviced by sending the stored value immediately along the request channels.

The rules for call-by-value, parallel call-by-value and call-by-process are substantially similar. For call-by-value, two concurrent processes are started (which are serialised via synchronisation on m) one computing M and the other receiving its value and *then* (i.e., when M's value is available) two similar subprocesses are started to compute N. When the values of M and N are available (on f and v, respectively), the function is activated and the environment entry binding x to v is established. In the parallel call-by-value rule, the computation of the argument N is not guarded by a receive action for f and the computations of M and N are started concurrently instead. When both values are received computation proceeds as in the call-by-value case. The difference between the parallel call-by-value and call-by-process encodings is that the activation $\bar{f}(x, o)$ is not guarded by the input action receiving the argument value v in the latter. Thus in the call-by-process case the function can be applied while the argument computation is still going on.

With this brief introduction to the theories which underlie our implementation, we now focus on and review the different approaches taken when implementing functional languages.

2.5 Implementation techniques

As the λ-calculus underlies functional programming languages, implementing the reduction operations outlined in the preceding section amounts to the implementation of functional languages. Functional languages usually introduce additional syntactic constructs to embellish the basic λ-calculus for the programmers' convenience. These extensions could be 'pure' or 'impure'; they are pure if they can be viewed as abbreviations of some λ-calculus constructs and impure otherwise [Gord88].

Lazy evaluation is one of several techniques that can be used to implement non-strict semantics. Normal order evaluation is an alternative technique. In a pure functional language, though, lazy and normal order evaluation will usually implement the same non-strict semantics, except that lazy evaluation will be more efficient in some cases. Similarly, *eager evaluation*, of which applicative order evaluation can be considered a restricted form, is one of several techniques that can be used to implement strict semantics [Cling95]. Lenient languages are non-strict but non-lazy. Eager evaluation can also be used to implement a non-strict semantics, which is essentially what happens in parallel implementations of lazy functional languages exploiting *speculative parallelism*.

There are two widely used implementation techniques for functional languages; one based on the use of environments and the other based on graph reduction. An environment is a data structure which holds variable-value associations. In the context of λ-calculus reductions, the environment holds the associations between bound variables in λ-abstractions and their corresponding argument expressions. An advantage of using an environment is that when implementing β-reduction or function application, the function body is left unchanged as late as possible, by storing the bound variable with its associated value in the environment. The value of a bound variable is remembered by looking it up from the environment when needed. Delaying substitution in this way, as opposed to the direct substitution naturally suggested by the λ-calculus, is potentially useful since the graph of the function's body may become significantly bigger after the substitution.

Environment-based implementations are those that use explicit environments of this form to evaluate λ-expressions. The first environment-based reduction machine, the SECD machine, was proposed by Landin [Land64]. The SECD machine is an abstract machine which has a simple definition as a set of four data structures and a set of rules describing how the data structures are transformed, a step at a time, based on their contents. The original SECD machine was most naturally suited as an applicative order reducer but can be adapted to support lazy evaluation with suspensions as in [Hend80, DaMc89]. Cardelli's FAM [Card83] is an optimised version of the SECD machine used to implement standard ML and allows very fast function application and the use of true stack [Card84]. Abramsky [Abra82] extends the SECD machine to handle multi-programming with concurrent processes.

A major disadvantage with environment-based implementations is the overhead of maintaining the environment and looking up variables' values from it. The environment could also contain associations which may not be required. Davie and McNally [DaMc89] describe an abstract machine, called CASE, which is an optimised version of the SECD machine, in which the environment is flattened to minimise the cost of environment accesses and which removes unwanted associations. For parallel implementations, the environment-based approach may not be desirable since the environment will form a bottleneck inhibiting parallel accesses to variable values.

Copy-based or graph reduction is an alternative implementation technique for functional languages. Expressions to be reduced in this model are represented by directed graphs. The leaves of the graph hold constant values, built-in functions and variables while nodes hold applications and λ-abstractions. Reduction proceeds by performing successive transformations of the graph replacing apply nodes with the corresponding values of their function applications. The graph structure, in contrast to an environment, makes sharing common subexpressions easier to express by simply using pointers. It also enables a relatively more efficient way of implementing normal order reduction and more natural way of parallel evaluations since communication is completely mediated by the graph [FiHa88, Peyt87].

Interpreters based on graph reduction work by continuously applying the λ-calculus reduction rules to transform the graph until it reaches WHNF. Non-trivial interpreters like Gofer [Jone94] introduce many optimisations to improve the efficiency of the naïve reduction process. For example, argument expressions are evaluated once (as opposed to the number of times they occur in the abstraction's body when ordinary reductions are used) and all occurrences of a bound variable share the computed value. However, in order to understand the performance of an implementation better, the reduction process has to be compiled since compilers expose weaknesses much better than interpreters.

Implementations based on compiled graph reduction, however, are faced with the problem of handling free variables which occur inside local functions. For a function `f` the compilation process aims at compiling a sequence of instructions for `f` which, when executed, constructs an instance of `f`'s body. However, if `f` is locally defined and contains free variables its body does not only depend on its arguments but also on the values of its free variables. Thus, it is not possible, for such locally defined functions, to statically generate the sequence of instructions required to implement them. Another major problem with graph reduction systems is to do with graph copying costs: function bodies need to be copied each time they are applied.

Maintaining an explicit environment as discussed above, is one way of coping with this problem. One other solution is to adopt a variable naming convention which avoids the problem [Bare84] or to use a calculus which does not contain variables at all [Turn79b]. Another approach is to use a program transformation technique, called *lambda lifting* [John87, FiHa88, Peyt87, PeLe91] so that the free variables of each local function definition are passed as extra parameters to the function.

The approach which abstracts away variables is based on a result in mathematical logic which says that variables as they are used in logic and ordinary mathematics are not strictly necessary. Combinatory logic [CFC58] is the theory which underlies the implementations based on this technique of abstracting all variables from a program. Any λ-expression can be expressed in terms of applications of a fixed set of combinators of the logic, e.g., the `S`, `K`, `I` combinators. In real implementations, this fixed set of combinators is enlarged by adding a few others for efficiency considerations. Turner [Turn79b] first proposed the use of these combinators for implementing functional languages. The computation technique is simple since it has a fixed set of graph transformation rules and the combinators have relatively simple form. Its disadvantage is that the combinator forms get complex even for simple λ-expressions and the execution steps are very small.

Instead of using a fixed set of combinators the program can be transformed to obtain a generalised set of combinators based on the programmers' functions defined in the source program. Johnsson [John87] introduced a lambda lifting algorithm which transforms a program into a set of combinator definitions. Hughes, in [Hugh83], independently described an optimised λ-lifting process which maintains laziness properties and also coined the term *supercombinators* to describe the resulting variable-free definitions. A supercombinator is a function whose free variables have been added as extra arguments to it.

With λ-lifting, the problem posed by free variables in local function definitions is solved and the resulting supercombinators give relatively bigger units of computation compared to the SKI combinators. It is therefore possible after λ-lifting (since the problem of free variables is then solved) to compile a sequence of code which constructs an instance of the supercombinator's body when executed. This technique is often used as a basis for efficient implementations of functional languages, notably [AuJo89].

A similar program transformation technique is *lambda hoisting* [Take88] which transforms a program into a fully lazy normal form (FLNF) suitable for fully lazy evaluation. Functions in the FLNF program may contain local definitions and the functions are in a more general form than supercombinators generated by Johnsson's style λ-lifting. In lambda hoisting, each maximal free subexpression is treated as a local definition and all local definitions in a function are collected into a single `whererec`-clause [KaTa92]. That is, in a FLNF program, maximal free subexpressions are not passed as extra parameters to functions thereby reducing the cost of parameter passing operations. Ordinary lazy evaluation of the FLNF program results in the implementation of full laziness of the original program. FLNF programs are compiled into an intermediate code for the fully lazy functional machine which is then transformed into code for conventional machines.

Yet another way of tackling this problem of access to non-local names inside functions, which is similar but different from preceding proposals, was introduced and implemented by Peyton Jones [Peyt92]. In this proposal, which is one of the characteristics that makes the STG machine design different from existing machines, the free variables of a local function are identified, but the function is left in place without floating it to the top level. When a closure for a function with free variables is entered (by making a particular register, the *Node* register, points to the closure), the code for the closure can access the free variables by indexing directly from the *Node* register. The advantage of this approach is that the movement of values from the heap to the stack is minimised since the free variables of a function are not pushed onto the argument stack during function call. This function call mechanism turns out to be very similar to the optimised environment lookup operations implemented in the CASE machine of Davie and McNally [DaMc89].

2.6 Costs of functional languages

The flexibility and user-friendliness offered by functional languages, by supporting programming at a higher level of abstraction and divorcing their design from exploiting hardware peculiarities, place a significant burden on the compiler and the run time system. The high level language features introduced by these languages, to extend the computational model underlying them, for user convenience, must be compiled out in order to achieve good performance. Imperative languages run more favourably than functional languages on today's von Neumann computers due to the mismatch between functional languages and these machines, although fast and competitive implementations of functional languages now exist and are rapidly offsetting this advantage [PHH+93, AuJo89b, NSvEP91, Hart94].

Although lazy evaluation offers tremendous expressive power (for instance the convenient expression of circular computations and manipulation of potentially infinite data structures) that is impossible to achieve otherwise, it hinders efficient implementation in terms of memory usage and also requires program analyses (e.g., strictness analysis) to improve performance. Some programs (like schedulers and similar system programs) are sometimes required to be non-deterministic and, since non-determinism is more difficult to express in functional languages than in imperative languages, language extensions are usually introduced to express non-determinism. Finding language constructs to express the kind of non-determinism needed is quite easy, and several examples have been implemented and used successfully, such as the "merge" operator of [AbSy85, Jone84], and the resource managers of [ArBr84]. The difficulty is rather with the maintenance of the language's desirable algebraic properties [Kell89].

Other difficulties associated with functional languages concern the difficulty of expressing non-trivial forms of I/O and algorithms that are intuitively state-bound. Functional programs are also not susceptible to traditional debugging tools of imperative languages. Debugging lazy functional programs is generally more difficult because of their demand-driven evaluation mechanism which leads to a rather non-intuitive execution order.

2.7 Summary

In this chapter we have given a brief account of the emergence of stored-program computers and also described the various ways of programming them starting from the earliest machine language programming, assembly language, to the higher level programming languages. We touched on the fact that although the introduction of high level programming lead to tremendous improvements over assembly programming, the quest for simpler and cleaner ways of programming persisted leading to the development of functional languages. We have described some of the salient features of functional languages that make programming in them easier and more expressive. We have also described the computational basis and implementation techniques of these languages. We have also covered the computational model, Milner's π-calculus, which serves as the basis for our compilation scheme described in Chapter 5. The chapter finally mentioned the costs usually paid by functional programmers in return for relieving them from prescriptive programming.

Chapter 3

Parallelism issues & related research

3.1 Introduction

In the preceding chapter we have discussed the motivation for the development of better ways to program computers so that better performance can be achieved as well as saving precious programmer time. Our presentation so far has concentrated on the issues of programming sequential machines containing a single processor in which one instruction is executed at a time. The processing speed on these machines depends on how fast data can be piped through the hardware. Although an instruction can be executed very fast in sequential machines, as fast as ten billionths of a second on some of them (at the time of writing in 1997), they are still judged to be not fast enough!

In this chapter we review the factors motivating the need and construction of new computer architectures and describe the critical issues in parallel computing and the main proposals for handling them (Section 3.2). In Section 3.3 we give a brief account of parallel functional programming, our chosen paradigm, explaining why it is more promising than conventional parallel imperative programming. In Section 3.4 we survey research projects on parallel implementations of functional languages especially those closely related to the work presented in this thesis. Section 3.5 summarises the material presented in this chapter.

3.2 Parallel computing issues

Many application areas, such as databases, simulations, image and signal processing, have problems that heavily depend on large amounts of data and require the constant manipulation of that data. Traditional sequential computers, although powerful, cannot offer the best performance required for some of these applications. This suggests the need either for powerful special-purpose computers like database machines or for still faster general purpose computers which can perform large, complex tasks more speedily and more efficiently.

A possible approach to satisfying this need could be to reinforce the existing sequential machines, especially since their compilers and tools have been well developed over many years. Possible reinforcements may involve providing more memory to these machines, improving their computational power and making use of the latest component technology to make their communications pathways larger and faster [PSU95]. It turns out that there are fundamental limitations which will inhibit this. One of these is the cost factor because single high-performance processors are extremely expensive and it is increasingly expensive to make them faster. Another limitation is to do with the (internal) communications speed and computational power increases as single processor performance is reaching its asymptotic limit [MCC96].

One of the approaches taken is to construct parallel supercomputers consisting of many conventional 'off the shelf' processors since these conventional processors are fairly fast and relatively inexpensive. With these sequential processors, which may be one or two generations older than the fastest, best available processors in the market, together with appropriate support for managing parallelism, it is possible to achieve good performance and also to work on problems which are impossible to handle with traditional sequential computers. An alternative approach taken by other researchers is to develop and implement computer architectures which are radically different from the traditional von Neumann computers and which can deliver the expected performances.

Since a parallel computer consists of a collection of many processors, there is a requirement that these processors be interconnected in some way in order to co-ordinate their activities for a desired performance. Other requirements include the development of parallel algorithms and suitable environments to manage parallel activities. That is, a parallel machine *per se* is not enough to guarantee performance improvement: the required efficiency can only be achieved with a combination of language, compiler and architectural provisions [ArNi89].

In the following six subsections we review the main issues of attention in a parallel programming environment namely, machine architecture, problem decomposition, granularity, work distribution, communication and parallelism control. As the possible alternatives solutions to these issues are discussed we point out our choices and justifying those decisions.

3.2.1 Architectural support

Researchers in large scale scientific and engineering computations have consistently demanded an increase in the number of instructions executed per unit of time. Their demands have always exceeded what could be provided by the most advanced computer architectures and therefore, new architectures which depart radically from von Neumann model were designed and constructed.

Although there are different methods used to classify computers, it seems there is no single characterisation which fits all designs. Perhaps the most widely used classification is that described by Flynn [Flyn72] which uses the relationship of program instructions to program data. Flynn categorised computers into four types based on the instruction-data relationship. These are SISD (Single Instruction, Single Data), SIMD (Single Instruction, Multiple Data), MISD (Multiple Instruction, Single Data) of which there are no practical examples, and MIMD (Multiple Instruction, Multiple data).

SISD is the most common conventional von Neumann computer that executes one instruction stream at a time. Most non-supercomputers fall under this category and their limitation is that the number of instructions that can be issued in a given unit of time is limited (performance on these architectures is frequently measured in MIPs—million of instructions per second).

SIMD is subdivided into vector or pipelined processors and array parallel processors. SIMD is also a von Neumann architecture with more powerful instructions that may operate on groups of data at the same time. A vector processor overlaps the operations on a vector of operands by means of a pipelining technique. An array processor duplicates the number of processors and applies them simultaneously to a vector of operands. These machines are usually used to exploit parallelism in data structures (i.e., data parallel algorithms). Debugging in this machine model is easier because of the single threading and the user does not have to think in terms of synchronisation. Another advantage is that given the cost of communication primitives, it is easy to understand the efficiency of the program. The disadvantage of SIMD designs lies in the difficulty of writing programs and in the limited compositionality due to the inherent limitation of single threadedness [Arvi92].

MIMD machines are divided into shared memory and distributed memory types. In shared memory machines the constituent processors (in addition to their local memories) share a common memory with each other while the processing elements are autonomous in a distributed system without shared memory. In other words, each processor acts independently from its peers in the MIMD architecture; executing its own instruction stream either sharing a common memory or with its own local memory [Perr87, MCC96]. In order to coordinate tasks for multiple processors working on the same problem, some form of inter-processor communication is required to convey information and data between processors and to synchronise node activities. In shared memory architectures only one processor can access the shared memory location at a time and in distributed memory architectures data is shared across a communications network using message passing. The advantage of MIMD machines is that multiple instructions sequences can execute simultaneously whereby each processor can perform any operation regardless of what the other processors are doing. However, unlike in the SIMD model, synchronisation is needed to co-ordinate the operation of the processors.

Another computer model is dataflow multiprocessors [NiAr89]. These machines are a radical departure from the von Neumann computers and may consist of multiple processors. Dataflow architectures are specialised MIMD machines tailored for the efficient exploitation of fine-grained parallelism. Program execution is driven by the availability of data rather than by control flow. When data for several instructions is available at a time, these instructions can potentially be executed in parallel provided there are no side-effects or multiple assignments to variables. Implementations based on these architectures [Trau91] recently seem to be shifting emphasis from the traditional view of exploiting instruction-level parallelism in dataflow architectures, in favour of a multi-threaded style where a collection of dataflow instructions is treated as a sequential thread [Well92].

The effective exploitation of parallelism requires the programmer to have a deep knowledge of both the program as well as the characteristics of the target architecture. Also, what constitutes useful parallelism depends on the hardware characteristics because moving a good parallel program to another platform may require the whole program to be revised, in order to get good performance [Brat94] on the new architecture. Therefore the issues of topology, processors and memory specification, which lead to an unfortunate binding between a program and the target platform, must be put into focus in a parallel programming system especially if portability is desired at low cost.

The compiler developed in this thesis is tailored towards distributed memory architectures with integrated network interface and without shared memory. This choice is influenced by the fact that Ostheimer's encoding of λ-calculus into the π-calculus (Section 2.4), which forms the basis of our compilation scheme, was aimed at such architectures (as in STAR:DUST [Osth91]). A given program is compiled into a large number of threads so that the high communications latency of our target architectures can be tolerated using fast context switches.

3.2.2 Extracting parallelism

One of the great challenges to parallel computing is the difficulty of programming parallel computers. Some say this is the 'only' reason why parallel machines are not widely in use today

[Arvi92]. Another important reason contributing to the difficulty of parallel programming is that the problems that need to be solved or the algorithms to solve them may not have any (inherent) parallelism and hence may not be parallelisable. One source of this difficulty is that existing code (for sequential machines) cannot be expected to run without change on a parallel machine and to expect better performance therefrom. Another issue in a parallel program is that the programmer has to be concerned about how the processing elements will compete for shared resources and how they co-operate with each other (see Section 3.2.4 below).

Therefore to realise the benefit of parallel machines, it is necessary to decide beforehand how parallelism is to be obtained. Two main strategies are usually adopted, namely *implicit* or automatic parallelisation and *explicit* techniques. In the implicit approach the programmer ignores the issues of specifying parallelism and relies on the compiler and the runtime system to decide on where and how to exploit parallelism. In the explicit approach, the programmer takes part in the parallelisation exercise where explicit knowledge about the parallelism within an algorithm is mapped to the implicitly parallel constructs of the programming language.

Automatic parallelisation is more widely used in those functional languages (which are implicitly parallel, see Section 2.2) although there are automatic parallelisers for imperative languages as well, like PTRAN [Sark91]. Implicit parallelism in functional languages does not actually mean parallelism 'without tears' or parallelism for free; some effort must still be invested to make parallelism exploitation worthwhile and to make the parallelism explicit.

Automatic transformation of a functional program, written in a strict functional language, into a parallel program can be very simple [Hamm94, Schr93]. However, the resulting parallel program often contains a large number of very fine-grained tasks with high runtime overheads. Implicit parallelism in lazy functional languages is normally obtained by using a strictness analyser to identify expressions whose evaluation contribute to the final program value. Strictness analysis is usually implemented using abstract interpretation techniques [Peyt87, HGW89, Burn87a,b, Burn91], abstract reduction [Nöck93] or projection analysis [WaHu87, Burn90]. The basic idea is that, given an input program, a parallelising compiler uses a strictness analyser to determine statically the expressions that are needed. The output program is automatically decorated with annotations to reflect this decision. This approach is very attractive and, if successful, it is a noble achievement since it provides useful parallelism while requiring minimum programmer involvement. Unfortunately, Schreiner [Schr93] observes that,

> *the problem of selecting the useful parallelism in functional programs is (almost) as difficult as detecting the possible parallelism in imperative programs.*

Finding needed expressions to achieve good parallel performance is also a difficult task especially in a language where functions are first class objects and neededness, or more broadly, optimal compilation strategies are generally undecidable [Hamm94, Szym91]. Automatic parallelisers can therefore only effectively handle simple cases where the parallel algorithm can be trivially obtained from the sequential algorithm but the hard cases require the programmer to design parallel algorithms.

In functional languages, there is a variety of ways of explicitly specifying parallelism; some are extended with impure features which are capable of introducing side-effects and some are extended only with pure features. Some functional languages are extended with imperative features such as message passing primitives to obtain parallelism [Webe89, MoTo92]. The MIT dataflow project [ANP89, BNA91] extends the functional language Id with I-Structures and M-Structures with deterministic and non-deterministic semantics respectively. On the other hand, some researchers introduce semantics-preserving annotations in source programs to indicate where parallel computations are desirable [Huda91, AuJo89a, HaPe92, Acht91, Kess96].

Another interesting idea employed to guide compilers for better exploitation of parallelism is the concept of *skeletons* [Cole89]. Skeletons are essentially higher order functions which capture patterns of parallel computation like pipelining and divide-and-conquer algorithms. Research on algorithmic skeletons has focussed on functional languages [Darl93, Brat94, Kuch94, Kess96], where the higher order functional skeletons can be expressed most elegantly. A related approach is used by some researchers to exploit *data parallelism*. In contrast to the usual approach, however, this approach exploits parallelism in data structures rather than in control structures. It is achieved by performing an operation, possibly defined as a skeleton, on all elements of a large data structure at the same time. Projects employing this approach include POD comprehension of Hill [Hill94], the bi-directional fold and scan of O'Donnell [O'Don93] and the data parallel language, NESL, of Blelloch [Blel93].

Following a popular tradition e.g., [AuJo89a, HaPe92, Kess96], parallelism is extracted explicitly by the programmer in our compiler by the use of annotations which the compiler can ignore if there are no resources available to exploit the specified parallel behaviour. It is important that these annotations are advisory rather than mandatory since otherwise they can change the semantics of the program by generating too much parallelism for the machine to handle.

3.2.3 Granularity

We see from the last section that the programmer must play a role in the parallelisation exercise in order to obtain good parallel performance for non-trivial applications. One crucial aspect of decomposing a given problem into efficiently manageable sub-parts is granularity. *Task granularity* (or simply granularity) refers to the basic units of work into which a problem is broken down and which are candidates for execution by the various processing elements on a parallel machine. Three levels or 'grains' of parallelism, fine grain, medium grain and coarse grain, are usually identified and it is hard to say which is better in a given situation: the right level of granularity depends on the properties of the underlying architecture in an implementation.

The ideal situation is to decompose the problem into enough (possibly fine grained) tasks to keep the processors busy most of the time while at the same time keeping the cost of communication as low as possible. These are conflicting objectives for if the tasks are too small the cost of communication (required to co-ordinate the results) is bound to be high. In the worst case it may even dominate the computation. If the tasks are too large, however, parallelism may be lost since there may not be enough tasks to go round the processing elements. This may lead to a situation where some of the processors will lie idle, hence underutilising the machine capacity. Research [BuRa94] has shown that fine grained tasks allow greater flexibility in programs, and have better worst-case scheduling properties than coarse grained tasks, but carry much greater overhead on an architecture built from conventional processors [HMP94] with high communication costs.

There are two main strategies which are taken to address the problem of granularity. User annotations, in addition to specifying parallelism, can also be used to determine statically the sizes of the tasks (e.g., using cut-off values) which should be created at run time when there is spare capacity [HaPe92]. In some implementations the compiler performs some complexity analyses at compile time to determine how much work is involved to justify task creation. In general, however, complexity analyses cannot find the required information to make these decisions since the answers may depend on the input data [Peyt89].

The preceding discussions suggest that a medium grain of parallelism is desirable which minimises the cost of task creation and management, keeps all processors busy and in which more computation is performed than communication. Our function level task granularity, which lies somewhere between the two extremes of fine-grained tasks in traditional dataflow architectures and the coarse-grained tasks in skeleton-based approaches, has the potential of providing reasonable balance between computation and communication.

3.2.4 Work distribution

When the problem has been partitioned into tasks, the next important decision is how to distribute the tasks onto the processing elements of the machine. This decision is also crucial because performance could suffer badly if poor distribution and scheduling strategies are used.

Work distribution decisions can be taken statically (i.e., automatically by the compiler or using explicit programmer annotations) or dynamically by some adaptive policies which make decisions based on the load status of the machine. Work distribution algorithms can be distinguished as *passive*, *active* or a hybrid of both [KuWa90]. In passive strategies, idle processors poll other processing elements for work to do while the distribution of tasks in active strategies is initiated by the processors creating the tasks. The algorithms used range from simple schemes like that proposed in [BuSl81] to more sophisticated schemes like the *gradient model* presented in [LiKe87], the *drafting algorithm* suggested in [NXG85] and a version of the *bidding algorithm* discussed in [Hwan82]. Research [EaLa86, Gold88] has shown that the simplest scheduling strategies often perform very nearly as well as (and occasionally better than) more complex ones [Peyt89].

Hudak [Huda91] uses source-level annotations to statically specify task distribution and uses a diffusion scheduler in their Alfalfa implementation [HuGo85]. The use of annotations to create static "process networks" in Caliban [Kell89] amounts to some significant user control on process creation and distribution. Culler *et al* [CSS$^+$91] describe an implementation in which program partitioning, scheduling and register management issues are left under compiler control. Hammond and Peyton Jones [HaPe92] have described several scheduling strategies and presented comprehensive measurements based on their implementations.

The ability of a task placement strategy to distribute work to its immediate neighbours is an important consideration (on many distributed memory systems) since there is less overhead associated with transporting tasks between processors that are physically close to each other than between those remote to each other. Care must be taken to ensure that the exploitation of locality does not lead to an imbalance in load sharing among the processors of the machine. Thus, a middle course is also desirable here since locality and processor utilisation have a mutually repelling effect.

Going by the results of Eager and others referred to above, we adopt an active task distribution strategy which offloads parallel processes randomly to the processors of the machine. We have also experimented with another variant of eager distribution scheme which is deterministic as detailed in our experiments in Chapter 8.

3.2.5 Communication and synchronisation

Parallel tasks that are co-operating or competing with each other need to be properly managed to make them realise the presence and purpose of one another so that correct behaviour can be achieved and duplication of effort can be avoided. The issue of communication, like that of distribution, is influenced by the underlying physical machine architecture. In shared memory machines, the tasks selected for reduction are often placed in a centralised global store, usually called a *task pool*, from where idle processors request work. Mattson [Matt93] has shown that having a centralised task pool is a bad design decision which may lead to the creation of *hot-spots*. He therefore suggests the use of distributed task pools (like those maintained by the GRIP IMUs) since they avoid the problem of hot-spots. In distributed memory machines the tasks selected for reduction are distributed directly from one processor to another. In some implementations a task offloaded to another processor for reduction contains all the information required for its evaluation and for the distribution of its result.

There are two major ways of organising communication and synchronisation between tasks in a parallel programming system namely the *notification model* (i.e., fork and join) and the *evaluate-and-die model* [Peyt89] which may involve task coalescing. In both these modes, an object to be evaluated is held in memory either in evaluated or unevaluated form or it may be under evaluation. When the value of an evaluated object is requested, the value is fetched immediately but if the object is under evaluation, the requesting task becomes *blocked*, its continuation added to a record of blocked tasks associated with the requested object undergoing evaluation. When the computation of the object's value completes, each blocked task is reawakened by notifying it with the value. While a task blocks, the processor on which it was executing can be employed to do some other useful work. The interesting difference between the two models concerns the relationship between a parent task and its unevaluated children: the notification model blocks after creating the child tasks while the evaluate-and-die model goes ahead and evaluates a child task when it needs the child's value and finds it unevaluated. Evaluating a child task by its parent in this way has the benefit of increasing granularity and locality and avoids the overhead of communication due to task switching when the parent blocks.

The evaluate-and-die model is often used in architectures with low communication latencies although it is also suitable for distributed memory architectures. In our distributed memory model data and information is communicated amongst the processors using asynchronous message passing and synchronisation is achieved using the notification model. Notice that since we compile a program into multiple threads, the costs of high latency communication can be offset by the requesting processor context switching speedily to do some useful work, in the mean time, before receiving the reply to its long range request.

3.2.6 Controlling parallelism

In some programs so much parallelism can be generated that the processing power of a machine becomes overwhelmed or that the memory requirements of the program exceed what the machine can tolerate. In these circumstances, therefore, it is necessary to impose some control to inhibit the creation of spurious parallel tasks.

The following program, based on the naïve Fibonacci function (and which counts the number of calls to the Fibonacci function), is a classical example used to indicate the possibility of parallelism explosion in a program.

nfib n = **if** n <= 1 **then** 1 **else** 1 + *nfib*($n-1$) + *nfib*($n-2$)

This function illustrates a divide-and-conquer algorithm in which the problem of computing `nfib`, for n greater than 2, is divided into 2 independent simpler problems which can be solved and their results combined to form the overall result. However, if the two independent recursive calls in the definition are made in parallel, the number of tasks created will grow exponentially, the same as the value returned by `nfib`. This function can be hand-tuned to minimise the number of tasks created or the runtime system can use some control mechanism to throttle task creation.

Load control mechanisms can be implemented either in software or in hardware. Software solutions could be specified statically (using programmer annotations or compiler transformation) or dynamically by the runtime system. The Manchester Dataflow project [RuSa87] uses a hardware solution to control parallelism explosion. Hammond *et al.* [HMP94] described the effect of two dynamic spark control strategies on granularity and presented performance measurements of these methods. Ostheimer [Osth93] proposes a load bounding algorithm integrated into a compilation scheme for functional languages. *k-bounded loops* are used by researchers at MIT [CuAr88] to restrict the number of concurrent iterations of a loop.

For the experiments reported in Chapter 8 we use an *ad hoc* thresholding mechanism to increase granularity and minimise the number of parallel tasks created. This is achieved by varying the number of threads each virtual processor executes from its context store whenever the processor's turn to run comes. We have, as a future research issue on this compiler, a plan to implement Ostheimer's proposed algorithm so that we can provide a more systematic parallelism control mechanism.

3.3 Why parallel functional programming?

As discussed in Section 2.2, functional languages prohibit assignments (which cause side-effects) and provide facilities which support higher-level control and data abstractions. Parallel functional programs are also comparatively easier to write than parallel imperative programs since functional programs are always determinate and the exploitation of conservative (i.e., non-speculative) parallelism does not change the semantics of a program [Peyt89]. Under this parallelism regime, an arbitrary subtask (like a function call) can always be assigned to a processor since the interleaved or parallel evaluation of expressions cannot change a program's value.

In conventional imperative languages, parallelism is often introduced explicitly using special constructs[1] in the language. Other systems exploit parallelism in an *ad hoc* way by making calls to parallel library routines. For example, PTRAN (parallel Fortran) extends Fortran with two control structures `PARALLEL LOOP` and `PARALLEL CASES` which can be added automatically by the compiler or explicitly by the programmer in the source code.

In contrast to imperative languages, parallel programming using functional languages is simpler because the elegant semantics of functional languages frees the programmer of the need to manage[2] parallelism explicitly. For example, in parallel imperative programming communication and synchronisation interfaces must be defined between parallel tasks to ensure that they interact correctly. The programmer is also responsible for enforcing protection on data and preventing deadlock. For example, parallel Fortran's statements for parallelism, `PARALLEL LOOP` and `PARALLEL CASES`, need to contain `PRIVATE` declarations for scalar variables to avoid data conflicts in loop iterations and among parallel tasks [Sark91]. The new language features that must be introduced to specify and manage parallelism makes programming much harder and the parallel programs are also more difficult to understand.

Language features are required to regulate situations where processes are required to interact in conventional imperative languages. For instance Pascal can be extended with *wait* and *signal* operations [Hans75] to queue and resume processes on a monitor respectively. In functional languages however, the underlying lazy evaluation model ensures that process synchronisation is implicit and therefore special measures need be taken only for efficiency considerations.

Deadlock in functional programs can occur only in erroneous programs in which some expressions strictly depend on their own results [Peyt89] while in imperative languages the programmer must ensure that tasks do not deadlock each other. Furthermore, since functional programs are determinate and new language constructs are not required to manage parallelism, parallel programs may be debugged on sequential hardware and it is no more difficult to reason about parallel programs than it is to reason about sequential ones.

In the next section we present a short review of parallel implementation projects on functional languages especially those that are closely related to our research.

[1]We point out that annotations in parallel functional languages are not language features in this sense. This is because such annotations are usually hints to the compiler which can be ignored using some kind of throttling mechanism. For the case when annotations are directives to the functional compiler, however, the distinction becomes somewhat blurred.

[2]Although parallelism needs to be specified by the user as described above, no special measures (except for efficiency considerations) need be taken to protect data that is shared by concurrent tasks [Peyt89].

3.4 Related work

As we have mentioned in Section 3.2.2, parallelism can be exploited in functional languages implicitly with minimum programmer support or explicitly with more programmer involvement. The implicit and explicit approaches are actually two extremes and there is a continuum of ways of exploiting parallelism between these two extremes. Various forms of computation contain implicit parallelism of some sort; at the high level in the algorithm and/or at the instruction level. An ideal programming language for implicit parallel programming should preserve parallelism available in an algorithm and facilitate the subsequent extraction of the parallelism from the program. Fortunately, functional languages satisfy these criteria and, by virtue of the Church-Rosser theorem referred to earlier, are used as a vehicle for implicit parallel programming.

There are three different approaches to exploiting implicit parallelism in functional languages namely, automatic parallelisation, use of programmer annotations and the use of skeletons. As pointed out in Section 3.2.2, automatic parallelisers are unable effectively to uncover and efficiently exploit the useful parallelism in a program. Programmer annotations are used to express varying degrees of control of a program execution. The annotations used by some implementations are mere hints to the compiler while others use annotations that are mandatory to the compiler in the sense that the compiler cannot choose to ignore the actions indicated by such annotations. The latter form of annotation leads to explicit parallel programming and, the former, although not totally implicit, is somewhere in the continuum between implicit and the explicit approaches. The skeleton approach is based on defining a fixed recipe of higher order functions which capture known patterns of parallel computation. The main disadvantage of this approach is that the fixed set of skeletons can hardly express all the parallelism in an application. Furthermore, the supported skeletons, which have to be implemented on every platform, may not be suitable for parallelising some problems.

In this section we present a review of previous and current research projects on parallel implementation of functional languages. Our review concentrates more on implementations that resemble ours, i.e., those based on the exploitation of implicit parallelism using advisory annotations. In Section 3.4.1 we review some implementations based on automatic parallelisers. In Section 3.4.2 we describe implementations based on programmer annotations including those implementations based on a combination of automatic parallelisers and annotations. Section 3.4.3 highlights other approaches.

3.4.1 Automatic parallelisers

Alfalfa and Buckwheat

Hudak and Goldberg [HuGo85a,b, GoHu86] developed a parallelising compiler based on *serial combinators* in their Alfalfa project. A serial combinator is a refined supercombinator which has no concurrent substructure. Their language base, Alfl, is a weakly typed functional language which requires runtime type checking performed by the Alfalfa back end. Alfl contains no constructs for specifying parallelism or synchronisation by the programmer; the compiler uses first order strictness analysis to automatically decompose and dynamically distribute the user's program for parallel execution. In contrast, Naira compiles a statically typed language and the task of specifying parallelism is left to the programmer using source annotations.

The Alfalfa implementation is based on a heterogeneous abstract machine model, borrowing ideas from dataflow and reduction machines research on the one hand and conventional compiler technology for sequential machines on the other [HuGo85b]. In order to support lazy evaluation, higher order functions and the creation of a function call on a remote processor by another function (which may require one call to suspend, due to data dependency, waiting for the value of another call), it is necessary to allocate *closures* on the *heap* rather than on the *stack*. Alfalfa performs *collecting interpretation* [Huda86] to detect function invocations which can be executed using sequential graph reduction. Two definitions are generated for each such function; one for sequential stack-based execution in which the function arguments are evaluated and one for sequential graph reduction in which some of the arguments may be unevaluated.

The target machine is the Intel iPSC (short for Personal SuperComputer), a distributed memory, hypercube-networked multiprocessor with very large communication overhead. All communication between processors is via message passing and tasks are offloaded dynamically using a heuristic *diffusion scheduler*. The scheduler runs on each processor and its operation is guided by the load information of neighbouring processors: each processor only sends work to its neighbours. Serial combinators form the units of granularity and task distribution and since they do not have concurrent substructures it is ensured that no available parallelism in a program will be lost. Comparing Alfalfa to our own work, we adopt a function-level granularity except that we do not exploit parallelism inside function bodies and thus maintain coarser grains of parallelism than their serial combinators. We make use of two variations of an active task distribution regime which deterministically or randomly offloads tasks to processors.

The high communication overhead of the iPSC multiprocessor degraded the performance and

Goldberg [Gold88] retargeted the compiler to the Encore Multimax, a shared memory multiprocessor in his Buckwheat project. In Buckwheat, a two-level task queue is maintained where each group of processors has a direct access to its primary queue with a secondary queue shared among all the processors. The secondary queue is accessed only when the primary queue is empty of full. Improved performance was measured in Buckwheat compared with Alfalfa.

The PAM project

Loogen *et al* [LKID89, HKL91] described a distributed implementation of programmed graph reduction on an OCCAM/transputer system. This research can be viewed as a continuation of Hudak's and Goldberg's work in which a functional program is automatically translated into a system of parallelised combinators. This introduces parallelism in PAM at two levels; statically and dynamically. This is implemented using a strictness analyser, based on the evaluation transformer method [Burn87a], to generate an intermediate program with an explicit `letpar`-construct to indicate subexpressions to be executed in parallel. Because static parallelisation of applications of functional parameters is not feasible, since such functions are only 'known' at runtime, higher order functions are parallelised dynamically. There are two sources of generating parallel processes in this serial combinator system: those resulting from the execution of `letpar`-constructs and a parallel process will also be generated for the delayed execution of a combinator application in a non-strict argument position. As in Naira, a parallel process corresponds always to a combinator application which is completely specified by the combinator name, the list of arguments and the return address. A parallel process in this case, however, contains two further descriptors; the *evaluator* with which the parallel task has to be evaluated and the *kind* of activation.

The implementation is based on a parallel abstract machine, PAM, which consists of a finite number of identical processors. Each processor has a modular structure, consisting of two independent communication and reduction units. The modular structure simplifies the formal specification of the parallel machine [Loog87, Loog88] and also decentralises the parallel execution of the program by separating the overhead of parallelism from the reduction process. Inter processor communication is achieved via message passing through the interconnection network. In order to minimise the communication costs, the compiler uses some heuristics to estimate the complexity of expressions to determine those that are worth parallelising. Their *process*, *answer* and *request* messages correspond to our *SendMessage*, *SendNotify* and *SendEval* messages respectively, described in (Section 5.4). Their graph representation consisting of *task nodes*, *argument nodes* and *data nodes* are correspondingly similar to our *function frames*, *suspension objects* and *constructed objects*. In contrast, they implement a simple passive task distribution mechanism in which idle processors send work-request messages to their neighbours.

Each combinator is compiled into a sequence of machine code with two different entry points and which incorporates the evaluation transformer information associated with the combinator. The two entry points correspond to the activation modes of the combinator; a potential parallel activation of the combinator's arguments before the second entry point is passed which immediately leads to the evaluation of the combinator's body. The first entry point can be short circuited when it is known that the evaluation of the combinator's arguments has already been initiated. This implies that their implementation is not capable of exploiting vertical parallelism where the evaluation of a function body proceeds in parallel with the evaluation of the function's arguments. In addition to this *horizontal parallelism* our compilation scheme of Chapter 5 nicely expresses the possibility of exploiting vertical parallelism.

Although the code for PAM is interpreted in the current preliminary implementation, their experiments on small programs show good speedup results. Their benchmark programs consist of those that manipulate data structures, like matrix multiplication, and those that work on simple integer values, like `nfib`. Speedup of up to 9 and up to 11 was measured on 12 processors in the two categories, respectively. However, as this is a preliminary stage in the implementation, it is too early to draw final conclusions before non-trivial benchmarks are considered and before optimisations are performed to reduce the cost of message passing overheads.

The disadvantage of this implementation is that a new combinator is introduced for each expression abstracted for parallel execution, unless the expression is already a combinator application. Oracle functions are used, in addition to the evaluation transformer information, to analyse expressions and determine those that are worth parallelising. Most of the time in this compiler is spent in computing the abstract interpretation of combinators.

The HDG-Machine project

Kingdon *et al* [KLB91] described the parallel implementation of functional languages on distributed memory machines. The implementation employs the evaluation transformer model of reduction [Burn87a] which uses the information about how functions use their arguments so that the function and their arguments could be evaluated in parallel — thereby saving the cost of building a data structure for the arguments in the heap. That is, parallelism is introduced via annotations for evaluation transformers only (without user annotations) and the granularity of tasks can be as small as possible. Source functional programs are compiled into code for an abstract machine which are then macro-expanded into Transputer assembly code.

The HDG-Machine [LeBu89] is a parallel abstract machine for the execution of functional programs. In contrast to conventional abstract machine designs, where the machine's state is usually defined as a tuple and the operational semantics defined by means of state transitions, this machine is specified in a functional language. The resulting specification, apart from being much easier to write and read, has the added advantage that it is executable and can be debugged and proved for correctness formally.

As in the ZAPP [BuSl82] and the GRIP systems (see Section 3.4.3), three different (heap-allocated) task pools are distinguished in the HDG-machine: *migratable*, which holds newly created tasks, *active* which holds tasks received from other processors and *blocked* task pool which stores tasks waiting for the results of other tasks. Only migratable tasks can be distributed to other processors and the distribution is carried out using a passive scheme: when the active task pool is empty, local migratable tasks are moved into it. If the migratable task pool is also empty, migratable tasks are requested from a neighbouring processor's pool. In order to prevent tasks cycling around the machine, requested tasks from a remote processor are placed in the *active* task pool of the requesting processor which is entrusted with its execution. This is also very similar to the single steal rule adopted in ZAPP.

The HDG-machine was implemented on a fully connected network of four T800-25 Transputer nodes with each transputer emulating one processor from the abstract machine. Initial implementation of this machine with a naïve code generator gave experimental results better than expected and which compare favourably with mature implementations like the LML compiler. However, the implementation has been tested only on small example programs and since the implementation lacks a garbage collector it cannot be tested using large programs. The large node layout used in the implementation is expected to introduce problems for "real" applications.

Other automatic parallelisers

Other parallelising compiler projects include the ADAM and EVE project [Loid92] and the FP compiler project [WaBa90]. The ADAM and EVE project compiles a non-strict functional language EVE to dataflow graphs which are then code generated into assembler code of an abstract hybrid dataflow/controlflow machine, ADAM. The target machine is a conventional machine extended with two constructs for the creation and synchronisation of parallel tasks. The FP compiler was targeted to a synchronous SIMD system and the compiler generates code capable of exploiting data-parallelism.

3.4.2 Annotation-based parallelisation

Chalmers LML project

Augustsson and Johnsson [AuJo89a] described the implementation of Lazy ML on a parallel graph reduction machine based on a parallel abstract machine, the $< \nu$,G>-machine. This machine is based on an earlier sequential G-machine which underlies their lazy ML compiler [AuJo89b]. As in the GHC implementation on GRIP (see below), parallelism is introduced using spark annotations, inserted explicitly by the programmer in the source, which the runtime system may ignore if there are insufficient resources available. Also as in the GHC, the evaluate-and-die model of synchronisation is used where a parent process evaluates an expression itself when it needs the expression's value and the evaluation of the expression has not been started by another process.

LML programs are compiled into a small intermediate language from which $< \nu$,G>-machine code is generated. As in the HDG-machine project, native code is finally generated from the $< \nu$, G>-code by macro expansion. The $< \nu$,G>-machine was implemented on a commercial shared memory multiprocessor, the Sequent Symmetry^{TM}, a bus-based multiprocessor which supports multiple executing threads of control, and incorporates the stack frames associated with these threads in the heap-allocated graph structure. In contrast to GRIP the CPUs of the Sequent Symmetry^{TM} do not have local memories attached (but do have caches) and a memory reference into the heap has the same cost as a reference into a stack, since they reside in the same shared memory [Augu91].

Representation of the graph nodes in this implementation is similar to ours, maintaining a tag to change the status of a node, except that they have to guarantee exclusive access to a node, when concurrently executing processes need to interact, in order to handle non-atomic instructions arising from atomic $< \nu$, G>-machine instructions. Although sparks are advisory and may therefore be lost without affecting the final result of the program, the spark pool, like in GRIP and unlike in GAML (see below), needs to be guarded by mutual exclusion, to avoid losing or duplicating sparks and in order to improve the runtime behaviour of the program.

An initial implementation of the $< \nu$,G>-machine (which produces quite simple code that is far from optimal) presents measurements showing real speedup compared to a fast compiled implementation based on the conventional G-machine. The experiments, based on a few small benchmark programs, using 15 processors measured speedups ranging between 5 and 11. Better parallel performance is expected when the code generator is improved to use registers rather than stack locations to speed up both allocation and garbage collection times.

The MaRS project

Researchers at Centre d'Etudes et de Recherches de Toulouse [CCC+89] developed a parallel graph reducer, MaRS (Machine à Réduction Symbolique), whose main design purpose is to minimise the cost of interprocessor communication and process creation by hardware means. The language base in this project is called MaRS_List, a pure Lisp type of language, that supports higher order functions and which has strictness annotation possibilities. Parallelism is made explicit in this language using a *process*-annotation similar to the *future* construct of MultiLisp [Hals85]. A MaRS_Lisp program is compiled into a combinatory object code for MaRS consisting of an enlarged set of the basic SKI combinator system called *indexed combinators* [CDL87, LCD+86] which are designed for reduction on special-purpose VLSI processors. The indexed combinators come in different flavours (some are used for optimisations only) including parallel versions generated based on the parallelism annotations from the source.

MaRS is a scalable (i.e., its power can be increased by adding more hardware resources without reprogramming), modular, specialised machine built around a special interconnection network and avoids any centralised mechanisms. The implementation aims to keep programming simple and relegates to hardware mechanisms the instantaneous regulation of processes and their distribution to processors. Therefore, task distribution and parallelism control are carried out by the interconnection network in this machine. MaRS is composed of five specific types of VLSI processors: Reduction Processors (RPs, supporting very fast context switches), Memory Processors (MPs, for managing the shared logical program graph), I/O Processors, Arithmetic Processors for floating point computations and Communication Processors (CPs) which make up the interconnection network.

The representation of graph nodes in memory adopted by this implementation is very similar to ours: the default status of a graph node (i.e., for a non-basic value expression without parallelism annotations) passes successively, and in this order, through a *Non-reduced*, *Under-reduction* and *Reduced* states corresponding to our `SUSPENDED`, `BUSY` and `READY` states (see Section 5.3) respectively. This default life-cycle of a node can be short-circuited by directly allocating nodes in `BUSY` or `READY` states as the case may be. Some of the indexed combinators may involve a Parallel version (P-version) combinator whose reduction leads to the creation of a parallel process, provided the machine is not saturated. Creating a new parallel process (upon reduction of a P-version combinator) requires sending an allocation message from the current RP to an MP. When the new under-reduction graph node is allocated, the allocating MP then sends two messages. One message is sent to transmit the address of the newly allocated node to the requesting RP (to enhance sharing of the subgraph node) and a second message is sent to a free RP to reduce the new node. Node allocation messages from an RP to an MP and process allocation messages from an MP to an RP do not carry destination addresses; they are routed using a *memory charge* and an *execution charge* information, respectively, kept by each MP and RP. Evaluation of a subprocess created in non-reduced status (on reducing combinators other than the P-version) is entrusted to the parent process (i.e., evaluate-and-die synchronisation), which will otherwise have to wait for the result. This model of synchronisation is especially important in MaRS because waiting processes are very costly since they cannot leave the resources they occupy to other processes that would like to start [Vran90]. It is clear from the foregoing that, like our implementation, this implementation adopts an active process distribution policy.

Performance evaluations of the MaRS prototype have been conducted by using a simulator. The simulation exhibits good absolute performances on typical simple benchmarks. In particular, fine-grained simulations have shown that the distribution mechanism based on *charge information* is very effective and allows a uniform distribution of nodes and processes among MPs and RPs. A disadvantage of the MaRS implementation is that their use of a fixed set of combinators and their interpretation on special purpose hardware makes their approach not readily portable to traditional hardware. Although the cost of communication is small, based on results of their simulations, it remains to be seen whether interpreting the indexed combinators can compete with compiled supercombinator execution adopted by most functional language implementations.

Concurrent Clean on ZAPP and PABC

Koopman *et al* [KvEN+90] described the sequential implementation of a lazy functional language, Clean [BvEL+87], on a stack-based graph reduction abstract machine, the ABC machine. Clean is a subset of a general graph rewriting language LEAN (Language of East Anglia And Netherlands) [BvEG+88] which is aimed at being an intermediate language between functional languages with much richer syntax and various abstract machines. The Concurrent Clean language [NSvEP91] is an extension of Clean with *process annotations* for specifying parallelism in the source program. A parallel ABC machine [NPS91], based on the sequential ABC machine, is an abstract machine used for creating and describing the implementations of Concurrent Clean. In this section we describe two implementations of Concurrent Clean; one on a generalised version of ZAPP and the other on the PABC machine.

There are two classes of annotations in Concurrent Clean, *local* and *global* annotations. Local annotations are the process annotations given explicitly by the programmer which are put on the right hand sides of rewrite rules. Global annotations are generated automatically by a strictness analyser and are placed in the definition of new types and in the type specifications of functions. A strictness annotation in the type specification of a function changes the reduction order of *all* applications of a function. Process annotations, on the other hand, are called local because they only change the evaluation order of a *specific* function application. Local annotations are of two forms, a *parallel* annotation and an *interleaved* annotation. An interleaved annotation creates a process on the current processor and executes in interleaved fashion with other processes on this processor. The parallel annotation creates a parallel process on a random remote processor unless this is not possible in which case it is treated as an interleaved process. There is also a version of the *parallel* annotation which specifies the destination processor, similar to Hudak's annotations [Huda91]. Concurrent Clean is therefore, perhaps, the most complete annotation-based language.

Early implementations of ZAPP [McBSl87] (see Section 3.4.3) required rewriting parts of the ZAPP kernel for each new application. In contrast, CCOZ, the implementation of Concurrent Clean on ZAPP [McBSl90, GMS93], uses the language primitives provided by Concurrent Clean to annotate expressions so that no modifications to the ZAPP elements are required for new applications. If a process is to be evaluated on a remote processor the subgraph (rooted by the annotated node) representing it is copied to the remote processor where it will be evaluated (and the result graph is copied back to the original processor, when demanded). On the other hand, the graph of an interleaved process is not copied but is shared. The implementation adopts the evaluate-and-die model of synchronisation where a reducer can pre-empt task creation. That is, if a reducer requests the value of a subgraph to be reduced as a separate task which has not yet been scheduled for execution, the requesting reducer reduces the subgraph itself. Concurrent Clean programs are compiled directly into Transputer assembly code. As in the Burton and Sleep implementation, the CCOZ implementation also uses transputers to implement the ZAPP elements.

Dynamic control of parallelism in the CCOZ implementation is achieved at both hardware and software levels. The depth-first scheduling of tasks in the process tree, supported by the ZAPP architecture, restricts the number of parallel tasks created at any given instance and prevents the exponential growth of parallelism at runtime. Although the evaluate-and-die model of synchronisation does not avoid task creation, it can be viewed as playing a part in the parallelism control since it increases data locality, granularity and saves copying and communication costs. Two versions of code, one for parallel and the other for sequential execution, are generated for each group of rewrite rules (that define a function) which contains task annotations. The CCOZ system decides, based on a heuristic function, which performs a simple threshold test, whether to execute the parallel code sequence or the sequential code.

Although the CCOZ implementation was not complete, the early experimental results they presented for simple divide and conquer functions showed good relative and absolute performance, which is further enhanced by the use of programmable granularity control. However, for programs that require graph copying when sending a task to a remote reducer, like their matrix multiplication benchmark, the performance was not as good and needs to be improved.

Each processor in the PABC machine contains zero or more reducers running interleaved with each other. There are two more reducers in addition to these: an operating system and a communication process which collaborate to handle I/O and interaction between processors. The operating system schedules the interleaved reducers and manages communication requests coming from the reducers or the communication process. The communication process, with the help of the operating system, handles communication between parallel reductions executing on different processors [PlEe93]. If a process wishes to communicate with another which does not reside on a neighbouring transputer (i.e., which is not physically connected to the transputer hosting the first process), the message is routed to the destination via some other transputers. All communication is therefore via message passing and the PABC machine has instructions that enable an arbitrary graph to be sent to any processor in the network. In contrast to CCOZ (which directly generates transputer assembly code), this implementation of Concurrent Clean on PABC generates the transputer object code via the PABC abstract machine code.

The PABC machine was implemented on a ParSyTec transputer architecture [Kess91, Kess93, PlEe93, Kess96] consisting of 32 T800 transputers connected in a grid topology. Each transputer supports primitives for starting and stopping processes, context switching and hardware support for a round robin scheduling mechanism. The scheduler automatically allocates reducers at two priority levels: low priority processes are automatically assigned slices of time within which to run, while high priority processes are not time-sliced so that the cost of context-switching can be minimised. Context-switches occur between basic blocks (e.g., after executing jump instructions or after I/O calls in some systems) and are avoided during garbage collection and within basic blocks otherwise the possibility of performing certain optimisations within the basic blocks will be destroyed.

Experiments in the Concurrent Clean implementation demonstrates that using a random process allocation, compared to using the HDG load balancing heuristic, does not give significantly worse execution times [Kess96]. Compared to the sequential implementation of Clean on a Motorola processor [SNvGP91], stack handling in the transputer-based parallel implementation is less efficient. As in the $< \nu$, G>-machine, execution begins with heap-allocated initial stacks of size 0 bytes which are automatically reallocated and resized—which requires performing boundary checks on the stacks—as execution proceeds. In the sequential implementation, however, programs usually use more memory since all the three stacks are allocated outside the heap. Overall, their experiments based on popular benchmark programs showed a reasonable performance considering the fact that a random process distribution was used.

GHC on GRIP

The Glasgow research team provides, amongst other things, a parallel implementation of Standard Haskell on GRIP based on GHC, the Glasgow Haskell Compiler [PHH+93]. GHC is a state-of-the-art sequential compiler modified with support for parallel language primitives and a sophisticated runtime system to support parallelism. Standard Haskell programs are compiled into a small intermediate functional language from which C code is generated [Peyt92].

The GRIP (Graph Reduction in Parallel) machine [PCSH87] is a purpose-built, shared-memory machine designed to perform supercombinator graph reduction efficiently. [illegible] of up to 20 printed circuit boards, each of which comprises of up to four processing elements, one Intelligent Memory Unit (IMU) and a fast communication subsystem. Each processor element consists of a Motorola 68020 CPU, a floating-point co-processor and 1Mbyte of local private memory. The GRIP boards are interconnected using a high bandwidth bus which provides a fast, low-cost, low-latency access path to the shared IMUs. The whole machine is connected to a UNIX host and one of the PEs is designated as the system manager which is responsible for resource allocation within GRIP, the other PEs behave like supercombinator reduction machines. The IMUs, the most innovative feature of GRIP, hold the program graph and are used to provide fast access to the large shared memory.

GRIP supports the implementation of standard Haskell whose subset we compile. The program to be executed is, as in Naira, expected to contain parallelism annotations informing the runtime system that certain expressions may be evaluated in parallel provided there is enough resources available. A *task*, which can be *sparked* by a PE based on the annotation information and the machine load status, forms the unit of parallelism in this machine and such a task is associated with a sub-graph node to be reduced later. Hammond *et al* [HMP94] presented strategies for controlling task creation in GRIP in order to improve task granularity and minimise communication overheads.

The implementation adopts the *evaluate-and-die* model of inter-thread communication and synchronisation which avoids context-switching costs and also dynamically increases granularity by absorbing a child task into its parent when the parent gets round to evaluate the child task.

GRIP runs parallel Haskell programs with substantial absolute speedup over the same program running on a uniprocessor Sun with a comparable microprocessor [HaPe90].

GAML project

Maranget [Mara91] describes a parallel implementation of GAML, a lazy functional language very close to LML, on a shared memory multiprocessor, the Sequent Balance. As in GHC, Chalmers LML and the Naira compiler, parallelism is introduced using programmer annotations. In addition to the user parallelism annotation, GAML incorporates a strictness analyser which automatically inserts complementary strictness annotations. This implementation has three different types of annotations: strictness annotations on function arguments, fork annotation given by the programmer and annotations on variables whose bound expressions have already been evaluated.

GAML is compiled into a small intermediate language (a subset of GAML) which is then compiled into G-code. As in the Chalmers compiler, function definitions occur only at the top level after lambda lifting while functions can occur at expressions level as well in our intermediate language as in Peyton Jones' STG language. Two entry points are generated for a function, for instance like in Hogen *et al* [HKL91]. In this case one entry point assumes the strict arguments of the function are already reduced while the other must arrange to reduce its (possibly non-canonical) strict arguments. The aim of the strictness analyser in GAML is to optimise the latter entry points whenever appropriate. In Hogen *et al* , the first entry point leads to a potentially parallel activation of the arguments before the second entry point is passed which leads to the evaluation of the combinator body.

The runtime organisation of GAML is quite similar to that of GRIP with both implementations identifying two pools of runnable tasks. These pools are called *fork* and *RUN* thread pools in GAML and, correspondingly, *sparked* and *unblocked* threads pools in GRIP. The fork/sparked thread pools contain newly created tasks that are not yet 'stolen' by an idle processor and the RUN/unblocked thread pools contain suspended threads waiting for the values of other threads. Whereas GRIP IMUs check sparked threads (to see whether WHNFs or locked closures are referred to by their thread descriptors) before adding them to the sparked pool, GAML allows a non-locked access mechanism to its forked task pool and like our compiler, evaluation tags are used (as locks) to prevent several machines from reducing the same subgraph.

The cost of the locking mechanism is avoided by distinguishing shared nodes, which are the only ones that really need updating, from unshared nodes. This non-locked access mechanism is crucial for the efficiency of the co-operating parallel G-machines. Storage for the forked threads pool is statically allocated and with the non-locked access makes its implementation very cheap. The RUN pool, however, needs exclusive access and that once started, a reduction must be completed to avoid the whole computation stopping before completion. GAML does not impose a scheduling policy to prefer either a sparked thread or a RUN thread (since both contribute to the final program result, assuming the usual conservative parallelism regime). However, the experiments in GAML indicate that the scheduler should look at the RUN pool first; to avoid creating new processes and to schedule existing runnable processes which waste stack and heap resources. Our multi-threaded model of computation corresponds to a FIFO scheduling strategy, without differentiating between sparked and unblocked threads, since each PE stores incoming messages in a queue in its context store and consumes them in succession. Also as in GHC and LML implementations, a parent process in GAML evaluates its child task when it requests the child's value and found it unevaluated.

GAML performs a source to source transformation, using a system variable which is set by the runtime system or by a processing element, to control the machine load at runtime. Experiments in the GAML implementation found that forked expressions should not be delayed too much. This helps to keep the number of suspended tasks low and avoid reducing forked tasks by their parents. Their implementation provides performance measurements, based on simple programs, comparable to those of [AuJo89b, HaPe90]

Id project at MIT

Id is a high level, non-strict but non-lazy functional language with fine-grained parallelism and determinacy implicit in its operational semantics [ArNi90]. A goal of the Id project is to have both parallelism and storage management issues implicit in the language. That is, Id programs are written free of any annotations which direct where to exploit parallelism and how to manage storage resources. This goal has not been realised yet in the current implementation and the compiler and the runtime system depend on user help for better efficiency. The language is, further, extended with two data structures, I-Structures and M-Structures, to increase its expressive power. I-Structures are an array-like, single assignment parallel data-structuring mechanism and M-Structures are updatable data structures with fine-grain synchronisation [ANP89, BNA91].

Id programs are compiled into dynamic dataflow graphs, a parallel machine language, directly executed on a novel multiprocessor architecture, the MIT Tagged-Token Dataflow Architecture (TTDA). TTDA consists of a number of dataflow processing elements and I-structure storage units interconnected by an n-cube packet network. This compiler was later retargeted to generate object code for TTDA's successor, Monsoon [HCAA94] which is a small, experimental shared memory dataflow multiprocessor consisting of eight processing elements.

The implementation, as in [AuJo89a, KLB91], generalises the stack of activation frames of a sequential execution model into a tree of activation frames so that an arbitrary number of activations can run in parallel. The runtime system consists of a frame manager and a heap manager which allocates and deallocates activation frames and dynamic storage respectively. The frame manager partitions work on code-block (i.e., at function activation level of) granularity and distributes the work in a round-robin fashion. Each processor has its own round-robin counters and the work distribution decisions could therefore be made locally. As in Naira and in [CSS+91] and unlike [DaRe81, HaPe92], a task is distributed over the network together with the data it requires, thus removing the need for remote data accesses during the execution of a task.

Because Id attempts to exploit parallelism at all levels (i.e., at processor, thread and instruction levels), it tends to expose too much parallelism, often exhausting machine resources, especially frame memory if some measure of control is not taken. Thus the user can supply *bounded loop* annotations to specify the amount of parallelism needed in loops and the compiler generates code accordingly. The compiler also analyses programs to determine the lifetimes of objects and to insert storage deallocation commands to avoid the difficulty of explicit storage management and reduce the cost of runtime garbage collection [Hick93]. Performance measurements on Monsoon are quite encouraging: for their benchmark programs, up to seven-fold speedups recorded on eight processors with a naïve frame management and work distribution strategy used by the runtime system.

3.4.3 Other approaches

ZAPP

ZAPP (Zero Assignment Parallel Processor) is an abstract machine originally proposed by Burton and Sleep [BuSl82] as a work distribution mechanism on a distributed memory parallel architecture. The machine is specially tailored to exploit parallelism in divide and conquer algorithms and operates by creating a virtual tree of parallel processes. The effectiveness of the ZAPP mechanism was first demonstrated, using Transputer arrays, by McBurney and Sleep [McBSl87] for a number of simple applications.

A general divide and conquer paradigm of problem solving can be captured by the following higher order function which, given a complex problem to solve, divides it into simpler subproblems which are solved and their results combined to obtain the overall result.

```
dc primitive split combine solve prob =
        if primitive prob then solve prob else
        combine (map solve (split prob))
```

where `primitive` is a Boolean-valued function which tests whether a problem is 'basic' and can be solved without subdividing it into simpler subproblems, `split` decomposes a problem into simpler subproblems, `solve` solves a given problem and `combine` collects the solutions to subproblems and builds the composite solution to the initial problem.

For a particular application, the programmer defines the functions `primitive`, `split`, `solve`, and `combine` whose code is broadcast to all ZAPP elements before execution begins. The initial problem and data are then injected into a single ZAPP element from which the divide and conquer processes are offloaded to the other nodes as execution proceeds. The ZAPP elements communicate with each other via message passing and the load distribution process is balanced dynamically by all the elements using local information: each element sends processes to its immediate neighbours only. The system adopts the notification model of task communication and synchronisation: when a parent process divides a non-trivial task into simpler tasks and spawns the simpler tasks, it waits for their results and when all results arrive, the parent combines them to obtain the result of the initial non-trivial task. The parent task then sends this result to its parent and so on until the result of the whole program is obtained.

A ZAPP system is made up of a number of processors, called ZAPP *elements*, connected together and each element consists of a conventional von Neumann processor with its private memory. ZAPP is a distributed system which does not support a physical shared memory and each element executes a *ZAPP kernel.* The kernel supports a virtual tree of processes and a parallel implementation of the `dc` function as a virtual tree generator. ZAPP maintains three different task pools: *pending*, *fixed* and *blocked.* Pending processes are those spawned by a `dc` call and which are not employed by some element yet and they are the only processes that can be offloaded to other processors. Fixed processes are those busy evaluating or offloaded and blocked processes are those suspended and waiting for some other process to return. The task pool organisation is very similar to ours except that in our implementation spawned tasks are distributed using an active scheme and we also ensure that only the newly spawned tasks can be offloaded and there is no task migration (of fixed and blocked processes).

The only source of parallelism in ZAPP is based on the parallel interpretation of the `dc` function as supported by the ZAPP kernels and all the individual codes for the user-defined functions operate sequentially. In contrast to the HDG and $<\nu$, G$>$-machine implementations (which macro-expand abstract machine code into target machine code), the ZAPP implementation employs true code generation to obtain transputer code directly (without compiling via an abstract machine code).

Distributable processes (and thus the units of granularity) are always in the form of higher order function calls of the dc combinator and such processes are defined completely by the data alone. That is, messages distributing tasks for evaluation contain the complete data defining a subproblem for the dc function call and do not include associated state information which can add to the overhead of offloading processes. The implementation uses a simple passive distribution scheme where idle processors may steal a task from one of their immediate neighbours based on a *single steal* policy which ensures that a stolen task must not be stolen again.

The ZAPP virtual tree architecture has been implemented in OCCAM and run on a small variable-topology system of 5 transputers for specific applications. The overall result of the applications studied in the implementation showed that for suitably large process trees real absolute speedups where consistently observed, and poor results were nearly always associated with very small problem sizes. One of their experiments involved running a series of `nfib` benchmarks on a 40 transputer network resulting in a relative speedup of 39.9. A disadvantage with ZAPP design is that there is at most one physical communication involved to send a parallel process. While this has the potential of minimising communications costs, it risks keeping the machine load unbalanced since it restricts possible diffusion of work throughout the physical network. It is unclear whether the benefits outweigh the disadvantages in general.

The FAST project

The FAST (Functional Programming for ArrayS and Transputers) project, was a collaborative research between Southampton University, Imperial College and Meiko Ltd of Bristol. Its aim was to provide an implementation of a pure, lazy functional language on a transputer array. The project also involved researchers from the University of Amsterdam with whom the Southampton team developed sequential compiler technology [HGW89, GHW90] and performance analysis techniques [HGW91, HaLa92]. Researchers at Imperial college and Amsterdam used the Southampton compiler as a basis for investigating various parallel implementation techniques [CHK+92, CHK+93, VrHa92].

An important component of the FAST system is a highly optimising compiler for Haskell 1.0 [HuWa90] on a single transputer [GHW90]. This compiler, which underlies the parallel implementation (see below), contains a number of analyses and syntheses specified within a single formal framework called *flow graphs*. Source programs are compiled into flow graphs which are then translated by a flow graph compiler into a severely restricted, single assignment subset of C called functional C, with call by value semantics [LaHa92]. The functional C code is then compiled into code for an abstract machine, KOALA, that uses an explicit call stack, which brings all (heap) pointers under the control of the garbage collector. KOALA is then translated back to C; disassembling a complete KOALA program into a single (giant) C function. The resulting "globally" optimised C code supports efficient garbage collection algorithms like two-space copying and generation scavenging which require all pointers into the heap to be known. As in the Naira implementation as well as [Peyt92, ScGr91, TAL91], this implementation relies on a C compiler for the final native code generation.

Cox *et al* [CHK+92, CHK+93] described the implementation of Caliban [Kell89] on a Meiko Computing Surface, containing 32 Inmos T800 transputers. Caliban is an annotation language with annotations that allow explicit control over process placement and communication on a distributed memory machine: a `moreover` clause is used to partition a program into separate node expressions each of which is distributed and evaluated in parallel on separate processors. Nodes that need to communicate are determined using data dependency analysis and an `Arc` annotation is used for consistency checking of communicating nodes. Caliban can be built on different languages and this implementation is based on Haskell 1.0 [HuWa90].

The declarative annotation in Caliban is used to describe a "process network" showing processes evaluating named expressions, linked by arcs showing where communication occurs. Functions called *network forming operators* (NFOs) can be defined in the host functional language to generate these annotations. Programs containing NFOs are later simplified, by partial evaluation, to remove all NFO calls and to transform the program into *annotation normal form* describing which data is to be computed separately. A form of λ-lifting is then performed to determine which placed streams need to communicate with each other. The implementation can make scheduling decisions based on the needs of the whole computation rather than on process basis since Caliban collects all the annotations controlling the parallel computation into the single `moreover` declarative description. A *network extraction* transformation is then performed to replace the `moreover` annotation with a call to a special system primitive, `procnet`, which implements the runtime parallelism. A standard functional program results after network extraction which is compiled using the Southampton compiler described in [GHW90].

In contrast to our implementation and many others, for instance [AuGo89a, Mara91], the annotation in Caliban is a directive (since it degenerates to a call to a system primitive) rather than advisory and the `moreover` annotation also provides the programmer with more control over parallel program execution: process partitioning, placement and communication. Programming with NFOs in Caliban is very much like that of using algorithmic skeletons [Cole89], except that Caliban allows the programmer to write new NFOs while skeletons are restricted to those provided by the system. The para-functional programming approach proposed by Hudak [Huda91] seems to implicitly embody the notion of a process network except that the decisions for mapping are explicit in the source and process interaction is not. In Caliban on the other hand, mapping the logical process network to actual processors is automatic (by a post-compilation phase) and process interaction is explicit in the source [Kell89].

Although generating a single C function from a complete KOALA program stresses most C compilers and a lot of extra code is generated to manipulate the tag bits present in each data value, the compiler showed better absolute performance than the (the parallel version of the) LML compiler [AuJo89b] on non-trivial benchmark programs. The results also show that better code results when compiling via C than in the native code generator (which had no garbage collector).

LISP derivatives

There are many projects dedicated to the implementation of parallel dialects of Lisp. A characteristic feature of these implementations is that most of the functional languages they implement include some constructs that may cause side-effects to occur. They also usually use the traditional environment-based implementation method and are targeted at shared memory multiprocessors. These projects include the implementation of Multilisp on a Concert machine [Hals85], Portable Standard List on the BBN Butterfly [SKL88] and the implementation of Mul-T on an Encore Multimax shared-memory machine [KHM89].

3.5 Summary

In this chapter we have outlined the limitations of sequential computers and motivated the need for parallel computer architectures and parallel programming. We identified the major issues of parallel programming that characterise the efficiency of a parallel implementation. We discussed these issues, namely architectural parameters, program partitioning, grain size, load distribution, communication and load management, mostly in the context of functional languages with different proposals for handling them.

We have reviewed several implementation projects (see Table 3.1) that exploit parallelism in implicitly parallel functional languages. In most of the work reviewed, parallelism is introduced

PROJECT	Function entry points	Target architecture	Extracting parallelism	Intermediate code	Load distribution	Parallelism control
Naira	1	Distributed memory	Advisory annotations	C	Active scheme	Quantum scheduling
Alfalfa/ Buckwheat	2	Distributed, Shared memory	Automatic (directive) annotations	Abstract machine code/C	Passive scheme	Heuristics
PAM	2	Distributed memory	Automatic (heuristics)	PAM-code	Passive scheme	Heuristics
HDG	1	Distributed memory	Automatic	HDG-code	Passive scheme	Heuristics
HBC	1	Shared memory	Advisory	<*v*,G>-code	Passive scheme	Spark annotations
MaRS	1	Shared memory	Hybrid	Combinator object code	Active scheme	Hardware mechanisms
FAST	1	Distributed memory	Directive annotations	C	Declarative annotations	moreover annotations
ZAPP	1	Distributed memory	dc combinator	Transputer assembly	Passive scheme	ZAPP kernels
CCOZ	2	Distributed memory	Hybrid	Transputer assembly	Active scheme	Software and hardware
PABC	2	Distributed memory	Annotations	PABC-code	Active scheme	Transputer hardware
GRIP	2	Distributed memory	Advisory annotations	C	Passive scheme	Annotations, spark strategies
GAML	2	Shared memory	Hybrid	G-code	Passive scheme	System call
Id	1	Distributed memory	I-Structures	Dataflow graphs	Active scheme	Annotations (loop bounds)

Table 3.1: Summary of related work

using annotations added by the compiler or by the human programmer. The Concurrent Clean language (described in Section 3.4.3) has arguably the most comprehensive annotations that allow graph copying, graph sharing, task placement and scheduling. For a number of implementations, the program is first translated into an abstract machine code of some sort which is then code-generated or macro-expanded to code for the underlying concrete machine. A popular trend recently is to compile into a high level imperative language (usually C) so that the best sequential compiler technology can be exploited. Most of the implementations adopt function-level granularity except that the functions may have a restricted structure in some cases (like the serial combinators of [HuGo85a] and the indexed combinators of [LCD$^+$86]).

In some implementations, like the STG-machine [Peyt92], two different entry points are generated for functions to minimise the cost of function invocation. In our implementation, we generate a single entry point for each function since general function applications have already been specialised in the intermediate language for faster, cheaper function calls (see Section 4.3.2). Tasks are distributed lazily or actively in other implementations and, synchronisation is implemented by means of some locking or tagging mechanisms.

It is quite tricky to asses these issues in isolation of other implementation issues except to say that lazy task distribution and evaluate-and-die synchronisation model seem to be in wider use. Another observation from our review is that better performance seems to accrue, generally, from shared-memory than from distributed-memory implementations. Of the implementations reported here, only the GHC researchers [HMP94] gave a detailed description and measurements of their load control mechanisms.

Chapter 4

The compiler front end

4.1 Introduction

This chapter presents the design, organisation and implementation of the front end of our functional language compiler, Naira. The back end is discussed in chapter five. One of the main motivations for the development of this compiler is to explore the prospects and problems of parallelising a modern functional language compiler.

After giving an overview of the compiler structure, the next section starts by describing the basic data structures used to represent our various symbol tables and the abstract syntax tree for a program. The section then delves into some detail on the organisation and implementation of the main phases of the compiler. Section 4.3 describes the intermediate language resulting from the analyses of the preceding section. Section 4.4 describes the representation for modules and the chapter is summarised in Section 4.5. Parallelisation issues are addressed in Chapter 7.

4.2 Compiler structure

Recall from Section 1.4 that the front end of our compiler consists of five major phases—analysis, pattern-matching, lambda lifting, type checking and optimisation—as shown again in Figure 4.2. This structure is the same as the basic multi-level structure of other production compilers of lazy functional languages such as the Glasgow and Chalmers Haskell compilers [PHH+93, AuJo89b], and thus, the parallelisation technology we present in the following chapters should carry forward to these and other functional languages compilers.

The aim of this section is to elaborate in some details the organisation and implementation of

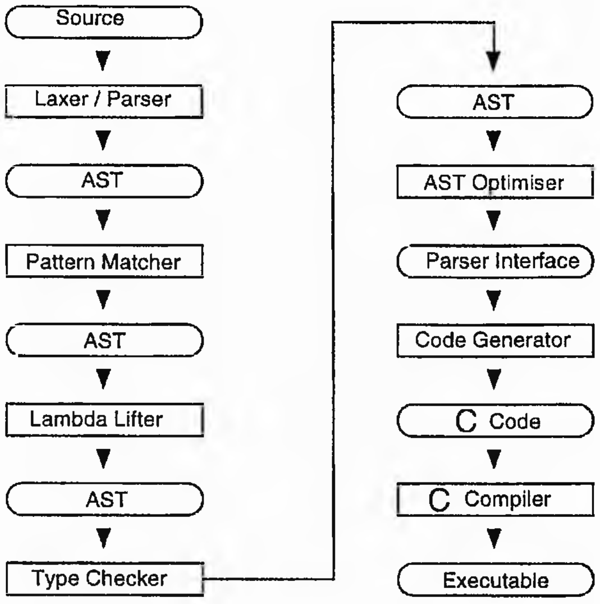

Figure 4.1: Structure of the Naira compiler

each of these phases. These are described in the next five subsections. The design and implementation of the back end of the compiler is presented in Chapter 5.

4.2.1 Symbol tables

One of the most substantial tasks of a compiler is to build and maintain symbol tables. This requires operations for updating a symbol table with information about a new entity, interrogating a symbol table for the attribute(s) of an entity and deleting entity/attribute(s) associations from a symbol table.

In order to make these operations moderately efficient, we maintain a binary tree representation for all the symbol tables in our compiler. Two general purpose binary tree types are defined that are used to structure the symbol tables in the compiler. `AssocTree a b` is used to represent symbol tables holding entity-attributes associations of entries.

```
data AssocTree a b = NL                                  --a leaf
                   | ND a b (AssocTree a b)(AssocTree a b) --a branch point
```

The type variable a is identified with an entity's name, usually a string, and b with the entity's attribute(s). Since the binary trees are sorted, an ordering relation must be defined on values of the type a. We describe our built-in implementation of such an ordering relations in Section 6.2, since we do not support type classes.

```
data Tree a    = Tip
               | Node  a  (Tree a) (Tree a)
```

`Tree a` (which is also sorted) is used to represent a collection of elements where there is no notion of entity-attribute association, like a list of definitions. Although `AssocTree a b` could be used to represent such values as well, the resulting representation will be less efficient because any information held in `b` (of `AssocTree a b`) is redundant.

We maintain the following symbol tables:

- **Fixity symbol table.** This associates each arithmetic operator with its associativity and fixity, as defined in Haskell. It has type `AssocTree` and is represented as follows

  ```
  AssocTree Name (Fixity, Associativity)
  type Name          = String
  type Fixity        = Int
  data Associativity = Non | Left | Right
  ```

  The constructors `Non`, `Left`, and `Right` are used to represent non-associativity, left-associativity and right-associativity respectively while fixities range from 0 to 9. We do not support user-defined fixity declarations, so this symbol table contains only the information about the basic arithmetic operators we support (see Section 4.3.1). The fixity symbol table is only used in the parser and could be garbage collected after parsing.

- **Pattern-matching symbol table.** This symbol table associates each constructor with its arity and the names of its sibling constructors. It has the form

  ```
  AssocTree Name (Arity, Siblings)
  type Arity    = Int
  type Siblings = [String]
  ```

  It initially contains information about the constructors `NIL`, `:`, `True`, `False` for the built-in types for lists and Booleans. It is extended with the information for the constructors of user-defined data types after parsing and before pattern matching compilation. This symbol table is used by the pattern-matching compiler after which it can safely be garbage collected.

- **Renamer symbol table.** Before lambda lifting (Section 4.2.4), there is a requirement to rename all identifiers bound locally so that there is no risk of name-clashes during λ-lifting and in subsequent passes. Note that the renamer affects only locally defined identifiers since any inadvertent redefinition of top level identifiers, which is of course an error, is caught earlier during parsing.

In this symbol table each identifier is associated with a uniquely generated integer to be attached to its name to make it distinct (see Example 4.2 , Section 4.2.4). The binary tree for this association is described by the type `AssocTree String Int`. After lambda lifting the table becomes garbage.

- **Constructor-tag symbol table.** This is used to associate a constructor with a 'family tag' (a small integer) with which the constructor will be identified in the intermediate language. It is described by the form `AssocTree Name Tag` (where `Tag` is a synonym for `Int`) and it initially contains the required information for the constructors of our built-in types. It is used during the **case** expression optimisation of Section 4.3.4 and can be dropped thereafter.

- **Type environment.** This is used by the type checker to store the association between identifiers and their types. It has the type `TExpList` defined as follows

  data *TExpList* = [*TypeExp*]
  data *TypeEnv* = *TEnv* [(*String*, *TypeExp*)] *TExpList*

  and is designed in such a way that it is easy to distinguish generic type variables from non-generic ones (in a similar way to [Read89]). The list `[(String, TypeExp)]` associates an identifier with its assumed type such that all type variables in the associated type are implicitly generic except those which occur in some type in `TExpList` as well. That is, to associate an identifier `id` with a generic type variable `tv` (as in **let**-bound variables), `tv` is not added to the list `TExpList`. Thus, `TExpList` essentially records non-generic variables.

  Type expressions are represented as follows:

  data *MyInt* = *Nl* --base case
  | *Cn Int MyInt* --constructed integers
  data *TypeExp* = *Untyped* --for 'null' type
  | *TVAR MyInt* --for type variables
  | *TCON Name* [*TypeExp*] --for type constructors

  Clearly, the type environment is potentially a very large data structure and as the type inference proceeds the type environment must be updated to ensure type consistency. Instead of updating the type environment after every type deduction step, which is costly to do functionally [Read89, Paul91], we use a substitution environment, described below, to keep a separate record of the modifications to be made. Note that this does not avoid carrying out substitutions; they are applied lazily since they are expensive to perform.

The type environment initially holds type information for built-in identifiers (of built-in operators and constructors). It is later updated with the type information contained in user-defined data types. When type checking a binding, the type environment is also extended with the intermediate (assumed) types for identifiers in the definition.

- **Exported bindings/type association.** This symbol table is used to associate each exported top level binding with its inferred type and arity. It has the form `AssocTree Name (TypeExp, Arity)` and is used to write the interface files of modules. Notice that only the static information of these values (together with exported type definitions) should appear in interface files in accordance with the information hiding principle of Haskell.

- **Substitution environment.** A substitution, in our setting, is a function from type expressions to type expressions. It can be represented as a finite collection of associations

$$t_1/t'_1, ..., t_n/t'_n$$

in which the t_i are distinct and no t'_i is the same as the corresponding t_i [ReCl90]. So we define a type for substitution as:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Subst} & = & \mathit{OK}\ (\mathit{AssocTree}\ \mathit{TypeExp}\ \mathit{TypeExp}\ ) \\ & | & \mathit{ERR}\ \ \mathit{String} \end{array}$$

The two constructors `OK` and `ERR` are used to build valid and invalid substitutions respectively. `ERR String` is used to force the propagation to the top level of an error occurring during type checking. `String` is the message written to alert the programmer of what has gone wrong. Notice that without `Error String` some errors may not be forced because of lazy evaluation.

Substitutions are used to record changes to be made to the type environment rather than updating it (by applying the substitution to each type in the type environment) whenever types are refined. When the type `t` of an identifier `id` is retrieved (from the type environment) the most recent substitution is applied to it to obtain a refined type `t'`. Generic type variables inside `t'` must be instantiated (by generating unique type variables) so that these generic variables are unaffected by any constraints applied to the new type variable instances.

Before any compile-time transformation can be performed on the bindings inside a module (after parsing), the static information in the interface files of all the modules imported by the current module must first be collected and the necessary symbol tables updated. We therefore parse the interface files corresponding to the imported modules to extend the symbol tables and to report any conflicting imports.

4.2.2 Lexer and parser

The lexical analyser is written in Haskell in conventional style, taking the input string, analysing it and breaking it down into a stream of tokens, recognising keywords, identifiers and literals. Each token is a substring of the source that takes one of the seven forms specified by the data declaration `Token`.

```
type Row   =  Int
type Col   =  Int
data Token =  Null Row Col
           |  VarId   String Row Col
           |  ConId   String Row Col
           |  VarSym  String Row Col
           |  ConSym  String Row Col
           |  Literal String Row Col
           |  Special String Row Col
```

The two type synonyms `Row` and `Col` are used to specify, respectively, the line number and the column at which the token starts on that line. Each token, except `Null`, is therefore completely described by its name and position; the line and column numbers on which it occurs. Token positions are used for error reporting as well as to ensure that programs containing the offside rule are correctly parsed.

`Null` is built from the empty string passed back as the last token in order to record the position of the end of the source text. `VarId` builds variable identifiers beginning with small letters (including reserved identifiers), `ConId` constructs identifiers beginning with upper-case letters. `VarSym` builds variable symbols and `ConSym` builds constructor symbols beginning with a colon. `Literal` builds a representation for ground literals (integers, characters and strings) and `Special` for brackets and commas.

Our top-down recursive descent parser, based on the principles in [DaMo81, ASU85, Hutt92], inputs the resulting tokens from the lexical analyser and produces an abstract syntax tree representation of the program. The syntax tree is then processed by subsequent passes of the compiler each of which transforms it into simpler form as outlined in the following sections.

4.2.3 Pattern matching compilation

Patterns are a notational convention employed by modern high level programming languages (especially functional languages) to increase their ease of use. Functional languages provide these high level facilities to define functions using equations and pattern matching. As with all good things, the convenience of programming using patterns is not completely free; they carry an implementation cost as they have to be compiled out. Pattern matching compilation is the program transformation process whereby these embellishing features are removed. That is, functions defined using equations and pattern-matching are transformed into equivalent single-equation definitions containing `case` expressions with simple variable patterns. Although `case` expressions are also an embellishing feature, since they are not part of the basic λ-calculus syntax, programs involving them are more efficient than those using the equational definitions instead. The transformed program ensures that patterns are not evaluated more than once and that they are evaluated at most once, when required, in line with the spirit of full laziness of functional languages [Peyt87, PeLe91, Hugh83].

There are different algorithms proposed for compiling pattern matching in functional languages notably those by Augustsson [Augu85], Wadler in [Peyt87], Cardelli [Card84] and the 'best-fit' pattern matching algorithm of Field [FiHa88]. The algorithm on which our pattern matching compiler is based is an extension of the algorithm proposed by Wadler. Maranget [Mara94], as an extension of similar work [Lavi91, PuSu90, Mara92, SRR92, Kenn90], described two techniques for compiling lazy pattern matching. One of the schemes described is based on the pattern matching technique using backtracking automata adapted to the world of lazy pattern matching. These lazy pattern matching schemes, which adapt the evaluation order to each set of patterns and guarantee termination whenever possible, are incorporated into the GAML compiler [Mara91] for Lazy ML.

In our implementation of the pattern matching compilation algorithm, a number of 'preprocessing' operations are applied to the patterns and expressions of a definition to simplify the AST before applying the algorithm. These tidying-up operations include the following:

- literal patterns (in left-hand sides of bindings and in user-given case expressions) are replaced with variable patterns using conditional expressions. This eliminates all literals from patterns and ensures that patterns are either variables or constructors only, as expected by our pattern matching compiler.

- user-given `case`-expressions containing literal patterns at top-level are transformed into equivalent (possibly nested) conditional expressions.
- anonymous lambda-abstractions, which do not occur at top-level in the right-hand sides of bindings, are named using the **let** construct.
- partial applications of constructors and built-in functions are saturated.
- conformality transformation [Peyt87] is performed on refutable pattern bindings.
- pattern matching transformation is applied to locally defined functions.

Our aim is to produce a simple and optimised intermediate language so that the code generator and the runtime system are not made too complicated. The underlying pattern matching compilation algorithm we implement expects the patterns to consist of either variables or constructor patterns with variable subpatterns. It is also our aim that all patterns in the resulting AST after the pattern matching compilation to be of this form because we will eventually identify all constructors (in the patterns in `case` expressions) with small integers in the intermediate language (see Section 4.3.4). We therefore translate equations involving literal patterns into equivalent ones in which the literal patterns are replaced with variable patterns before invoking the pattern matching compiler. Since user-given `case` expressions may contain literal patterns, such expressions have to be transformed into equivalent ones without literal patterns.

Unnamed lambda-abstractions which occur as part of other expressions are uniquely named (using **let** expressions) so that they become subject to future transformations (e.g., lambda lifting since they may contain free variables). The expressiveness of the program may also be increased as a result of this in the sense that **let**-bound variables such as `i` in

$$\textbf{let } i = \text{e} \textbf{ in } E$$

are attributed generic types while λ-bound variables like `i` in the equivalent expression, `(`λ`i.E)e` , provided they both type check, are attributed non-generic types. In similar fashion, auxiliary definitions are introduced using the **let** construct so as to saturate partial applications of constructor functions and built-in functions.

A conformality transformation (on pattern bindings) [Peyt87] is also implemented so that explicit error messages are generated when pattern matching in a pattern binding fails. To minimise the cost of this transformation, it is only performed on the so called sum-constructor patterns; exactly those that can be refuted.

As the patterns and expressions are tidied, pattern matching compilation for local definitions proceeds simultaneously with that for the top level definitions. A name-supply parameter to the compilation functions ensures that distinct names are generated for the auxiliary functions used to name anonymous λ-abstractions, partial applications of constructors and built-in functions, variables introduced while translating literal patterns, etc. This name-supply is one facility we employ to aid our annotations for parallel evaluation, as described in Chapter 7.

4.2.4 Lambda lifting

Lambda lifting, a term coined by Johnsson [John87] (but earlier invented by Hughes in his super-combinator transformation [Hugh83]), is the process of transforming a program containing local functions with free variables into ones where these functions are turned into combinators (i.e., functions without free variables). This is done by transforming each function by passing in its free variables as extra parameters. Lambda lifting provides a convenient way of solving the problem of accessing the free variables occurring in the body of a local function definition. Lambda lifting is especially beneficial in a setting like ours, for parallel machines with distributed memory, since communication costs for accessing a function's arguments will be minimised. This is because accesses to a combinator's arguments will be local operations into its frame. Without lambda lifting, however, the free variables to the function may reside on arbitrary processors which may involve long-range communication to access the free variables thereby incurring higher communication costs.

Many implementations of (lexically scoped) imperative languages compilers solve this problem of accessing non-local names in procedures by maintaining a *display* mechanism [ASU85, DaMo81] to access variables on the stack. Davie [Davi79] presents an alternative technique which keeps all variables on the heap and accesses them using a pair of registers (holding access to local and non-local variables) loaded from a data structure called a *block descriptor.*

Alternative solutions to lambda lifting in functional languages implementations include environment based approaches, as in [DaMc89], the use of Turner's combinators [Turn79b], Takeichi's lambda hoisting [Take88] and so on, as described in Section 2.5.

Our implementation of lambda lifting is influenced by the algorithm of Johnsson in [John87] except that we do not lift the resulting combinators to the top level. This has the advantage of minimising the number of top level combinators and the overhead of handling them. It also provides scope for optimising the combinators if they are left local (e.g., inlining them) when the compiler can spot that they are used only once.

Our implementation of the lambda lifting phase consists of three sub-transformations:

- scope analysis;
- dependency analysis;
- lambda lifter proper.

The scope analyser performs two functions: it computes a dependency 'graph' and renames identifiers to make them distinct. The dependency graph is an association between a bound identifier and the identifiers on which it depends.

We demonstrate our lambda lifting process using the following example:

Example 4.1

$$\begin{array}{l} \textbf{let } f\ x = x + y \\ \quad\ y\ \ = 5 \\ \textbf{in let } y = 3 \\ \qquad\quad g\ z = f\ z + y \\ \quad \textbf{in } g\ (f\ 7) \end{array} \qquad (1)$$

The definitions and occurrences of y in the outer and the inner **let** must be made distinct so that there is no ambiguity when turning g into a combinator.

The dependency analyser breaks the bindings in a **let** expressions into minimal dependency groups to simplify subsequent passes. Dependency analysis enhances efficiency and polymorphism since it allows only mutually recursive definitions to be grouped together and thereby enables the assignment of generic types to non-recursive **let**-bound variables. More seriously, a program may fail to type check if dependency analysis is not performed first [Peyt87].

$$\begin{array}{l} \textbf{let } y2 = 5 \\ \textbf{in let } f1\ x = x + y2 \\ \quad \textbf{in let } y3 = 3 \\ \qquad \textbf{in let } g4\ z = f1\ z + y3 \\ \qquad\quad \textbf{in } g4\ (f1\ 7) \end{array} \qquad (2)$$

Our implementation of a dependency analyser is based on the graph algorithms described in [Peyt87, Paul91, Sedg90]. It involves sorting the dependency graph (computed by the scope analyser) into mutual recursion groups or strongly-connected components and then sorting these components topologically into dependency order. Nested **let** expressions are built from the topologically sorted classes. Equation (2) shows this example after scope and dependency analyses.

Notice that the y bound by the inner and the outer lets in (1) are renamed to y2 and y3 respectively. The resulting expression is cascaded with each binding now standing on its own since none of them are mutually recursive.

```
let y2          = 5
in let f1 y2 x  = x + y2
   in let y3    = 3                          (3)
      in let g4 y2 y3 z = f1 y2 z + y3
         in g4 y2 y3 (f1  y2 7)
```

The final lambda lifting process gathers free variables for functions, forms and solves equations to obtain the solutions (i.e., the transitive closure of the relation that collects free variables) to these functions. A substitution is then performed to replace each reference of a function identifier f in an expression with the application of f to its free variables. After lambda lifting, our working example is transformed to the program in equation (3).

Notice here that since y2 was free in f1 (2) it is passed as extra argument to f1. Substituting f1 y2 for the reference of f1 in the right-hand side of g4 in (2) exposes y2 as an additional free variable of g4. Accordingly, y2 and y3 are added as extra parameters to the reference of g4 in the body of the expression. Notice also that, as explained earlier, none of the definitions is lifted to the outer level. We discuss the parallelisation of the lambda lifting process in Section 7.5.

4.2.5 Type checking

A type is a family of values classified by a given rule. The classification technique of data values according to their usage and checking that no values are misused is known as *typing* the data [Read89]. Programming languages usually impose some notion of a type discipline ranging from weaker to stronger typing requirements which are checked either during compilation or during program execution.

Most modern functional languages support a strong, static, polymorphic type system, often an extension of Milner's [Miln78], which automatically attributes types to program values. That is, types are inferred at compile-time (static type checking) with some functions capable of operating on values of different types (polymorphism) and ensuring that well-typed expressions cannot lead to erroneous computations. This is an invaluable feature and, in fact, can be indispensable in large software, since the types provide partial specification of the program ensuring that a number of errors can be found early during program development, thereby making programs more secure and reliable.

Our implementation of a polymorphic type checker is based on algorithms and ideas expressed in [Miln78, Dama85, Card85, Read89, FiHa88, ReCl91]. The implementation entails extending the basic Milner-Damas type system with constructs to facilitate type checking bindings which involve patterns and case expressions.

To type check a given module M, the compiler collects the static information (types and arities) of the values imported by M (from the interface files of the imported modules) and including any type and synonym definitions. The type and synonym definitions within M must also be parsed and the relevant symbol tables built before the type inferencing commences.

When this information is gathered Naira proceeds to infer the types for the definitions implemented by M in such a way that parallelism can be exploited at both function and expression levels when annotations are added later (see Chapter 7). As the type of each definition is inferred, its user-given type signature (if any) is scrutinised to ensure that it is an instance of the compiler inferred type. Finally an interface file for M is written which contains the static information of the definitions it exports.

In comparison with the other transformations in the compiler, the type checking process is the most computationally expensive. We elaborate on the details of parallelising the type checker in Chapter 7.

4.3 Intermediate language

The transformations in the previous sections together with the AST optimisation described in this section culminate in the intermediate language representation for expressions shown in Figure 4.2. The intermediate representations for bindings are described in Section 4.3.6.

4.3.1 Ground types and primitive functions

We support integer, Boolean and character literals as the basic types. A type `String` is defined as an abbreviation for a list of characters. Integer, character and string literals are built, respectively, by the constructors

IntLit *Int*
CharLit *Char*
StringLit *String*

Values of the basic types share a common concrete representation as heap objects containing two cells. One of the cells contains a tag which distinguishes literals from other (aggregate) values

```
data Exp   =  IntLit Int
           |  CharLit Char
           |  Id String
           |  PrimMinInt      Exp
           |  PrimMaxInt      Exp
           |  PrimCharToInt   Exp
           |  PrimIntToChar   Exp
           |  PrimNegInt      Exp
           |  PrimNotBool     Exp
           |  PrimPlusInt   Exp Exp
           |  PrimMulInt    Exp Exp
           |  PrimDivInt    Exp Exp
           |  PrimEquals    Exp Exp
           |  PrimLEquals   Exp Exp
           |  PrimRemInt    Exp Exp
           |  PrimAndBool   Exp Exp
           |  PrimOrBool    Exp Exp
           |  Constr FamilyTag [Mexp]
           |  Select Exp Index
           |  Case Exp [(Int,Exp)]
           |  If Exp Exp Exp
           |  Define [Def] Exp
           |  Close  Name Arity [Mexp]
           |  Call      Name [Mexp]
           |  Apply Exp [Mexp]
           |  Let [Def] Exp
           |  Error Exp
type FamilyTag =  Int
type Index     =  Int
```

Figure 4.2: Intermediate code for expressions.

and the other cell stores a pointer to the value of the literal. A garbage collector[1] distinguishes constants from pointers by inspecting their family tags: basic values have a tag of 0 and other values have small positive integer tags.

PrimNegInt Exp
PrimNotBool Exp
PrimIntToChar Exp
PrimCharToInt Exp
PrimPlusInt Exp Exp
PrimMulInt Exp Exp
PrimDivInt Exp Exp
PrimEquals Exp Exp
PrimLEqual Exp Exp
PrimRemInt Exp Exp
PrimAndBool Exp Exp
PrimOrBool Exp Exp

Applications of primitive functions are distinguished from application of user-defined functions to improve the efficiency of the basic arithmetic operations since primitive function applications require strict semantics. This avoids building intermediate suspension objects for the argument expressions of these functions. Some of the primitive functions we support are shown above; others are defined in terms of these. Built-in functions are always fully applied in the intermediate language. Partial applications are transformed by the compiler using the **let** construct as mentioned in (Section 4.2.3) so that the code generation process is simplified. For example, the partial application of the binary addition function to one argument,`(+) x`, transforms to

let $f\ y\ x = x + y$
in $f\ y$

where `f` and `y` are unique names that are generated by the compiler.

4.3.2 Function application

The intermediate language specialises a general function application into one of three forms depending on the structure of the function being applied and the number of arguments available.

[1] Naira does not (yet) support a full-fledged garbage collector. We currently use a simple storage manager which reclaims space for frames whose associated function invocation returns with a basic value or a HNF object (see Section 5.3.1).

The three forms are represented by the intermediate forms:

> *Call Name* [*Mexp*]
> *Close Name Arity* [*Mexp*]
> *Apply Exp* [*Mexp*]

`Mexp` is the data type for expressions with evaluation modes as described in Section 1.3. `Name` and `Arity` represent the name of the combinator being represented and its arity respectively. If the function expression is a known supercombinator applied to the exact number of arguments, `Call` is used to build the application. If the number of arguments supplied is less than the combinator's arity, `Close` is used to represent the application recording the combinator's arity. This arity information will be used to determine, at runtime, when the code of the combinator can be executed as more arguments are provided. When the function being applied is an arbitrary closure-valued expression or an unknown combinator identifier (e.g., a function argument of a higher order function) whose arity is not statically determinable, `Apply` is used to build a representation for such applications.

Specialising function applications in this way simplifies the runtime system by minimising the number of *argument satisfaction checks* [Peyt87] performed, at runtime, for every function which takes one or more arguments.

4.3.3 Constructed objects and selector functions

When type inference is performed, we no longer need to refer to constructors by their string names; we can instead identify them by family tags. A family tag is a small integer which distinguishes each constructor of a type from its siblings. Although tuple constructors have no tags, since every tuple type has exactly one constructor, we nevertheless assign each tuple a tag of 1 for uniform handling of constructors at runtime (see Sections 5.3.3 and 6.4). The intermediate representation for tuples and other constructors is

> *Constr FamilyTag* [*Mexp*]

The compiler assigns tags to constructors in a data type beginning from 1. For example, given the data definition

```
data Tree a  = Tip
             | Node a (Tree a) (Tree a)
```

The compiler assigns the tag 1 to `Tip` and 2 to `Node`. Identifying constructors using small integers in this way makes the code generation easier and improves efficiency. This is discussed further in Section 4.3.4 where we present further transformations on case expressions.

Select Exp Index

A selector function is defined which selects the components of aggregate objects. The first component of `Select` evaluates to a constructed object and strong typing ensures that only the 'right' values are passed to `Select`.

4.3.4 Case expressions

A transformation pass prior to the pattern matching compilation of Section 4.3.3 ensures that literal patterns in `case` expressions are compiled out. That is, after pattern matching compilation all patterns in case expressions are either simple variables or constructors whose subpatterns are variables. With this proviso, `case` expressions in the intermediate language are represented by the structure:

Case Exp [(*Int*,*Exp*)]

The first component of `Case` evaluates to a structured object and each expression in the alternatives list, `[(Int,Exp)]` is associated with a corresponding constructor tag. The value of the case expression is the value of the expression in the alternative list whose associated tag equals that of the case scrutinee expression. A default expression has a corresponding default tag of 0.

The transformation optimising the representation of case expressions is summarised thus:

$$\begin{array}{lcl}
\texttt{case e of} & & \texttt{case (Select e 0) of} \\
\quad \texttt{Con}_0\ \texttt{v}_{01}\ \ldots\ \texttt{v}_{0n}\ \texttt{->}\ \texttt{e}_0 & & \quad \texttt{0 -> e'}_0 \\
\qquad \ldots & \Longrightarrow & \quad \ldots \\
\quad \texttt{Con}_k\ \texttt{v}_{k1}\ \ldots\ \texttt{v}_{kn}\ \texttt{->}\ \texttt{e}_k & & \quad \texttt{k -> e'}_k
\end{array}$$

Occurrences of the variables $\texttt{v}_{ij}$ of the constructor $\texttt{Con}_i$ in the corresponding expression $\texttt{e}_i$ are replaced by selector expressions over the case-scrutinee expression, `e`. Consider the following example:

Example 4.2

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Tip} \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)(\mathit{Tree}\ a)$$

$$f\ \mathit{Tip} = \mathit{Tip}$$
$$f\ (\mathit{Node}\ t\ l\ r) = \mathit{Node}\ t\ (f\ r)(f\ l)$$

The pattern matching compiler transforms this example roughly[2] to

$$\begin{aligned} f\ u = \textbf{case}\ u\ \textbf{of} & \\ &\mathit{Tip} \rightarrow \mathit{Tip} \\ &\mathit{Node}\ t\ l\ r \rightarrow \mathit{Node}\ t\ (f\ r)\ (f\ l) \end{aligned}$$

and then after type checking `Tip` will be identified with 1 and `Node` with 2 to transform the `case` expression to

```
Case (Select (Id "u") 0)
     [(1, Constr 1 []),
      (2, Constr 2 [Need(Select (Id "u") 1),
                    Need(Call "f" [Need(Select (Id "u") 3)]),
                    Need(Call "f" [Need(Select (Id "u") 2)])]
```

Notice how the whole pattern `(Node t l r)` is identified by the tag of `Node`, the integer 2, in the resulting intermediate code. Its subpatterns `t`, `l` and `r` are represented, respectively, by `Select (Id "u") 1`, `Select (Id "u") 2` and `Select (Id "u") 3` in the expression corresponding to the pattern. Notice also that applications of `f` are wrapped with the `Call` constructor (as explained in Section 4.3.2) and its arguments as well as those of the constructor all have the default lazy evaluation tag since there were no annotations specifying otherwise.

The benefits of this transformation are that it:

- simplifies the code generator (e.g., fewer variables to deal with);
- avoids allocating the constructors Con_i;
- avoids the overhead of maintaining an environment (as in Peyton-Jones [Peyt92]) for the values of the variables v_{ij} which may occur in the expressions e_i.
- avoids the need to perform constructor re-use transformation of Santos [Sant95] while at the same time gaining the benefit of that transformation.

[2]Details have been suppressed to maintain readability.

4.3.5 Other expressions

The other expressions namely, conditional expressions, qualified expressions and the compiler-generated `Error` expression (for error reporting) translate straightforwardly into the following intermediate representations.

If Exp Exp Exp
Let [Def] Exp
Define [Def] Exp
Error Exp

Qualified expressions defined using **let** and **where** translate into a representation built by `Let` or `Define` or a cascade of the two. Just as top level bindings are distinguished into combinators and non-combinators across modules so are local bindings distinguished amongst themselves. `Define` contains only local combinator bindings while `Let` contains local variable bindings. This provides yet another simplification for straightforward code generation. `Error` is a representation for an error message to be reported at runtime; this may arise from error messages explicitly given by the user in the source (by calling the built-in `error` function) or automatically generated by the compiler for user definitions in which pattern matching fails. Code generated for `Error` simply displays the error message and terminates the execution of the program (without consuming the rest of the requests).

4.3.6 Bindings

Functions and patterns can be bound at top level as well as in nested scopes using the **let** and **where** constructs. The intermediate language distinguishes function bindings from non-function bindings and also distinguishes exported top level bindings from those used locally. The following `data` definition describes the intermediate language forms of these bindings.

$$\begin{aligned}\textbf{data}\ \mathit{Def} &= \mathit{Fundef\ Name\ [String]\ Exp}\\ &\mid \mathit{Efundef\ Name\ [String]\ Exp}\\ &\mid \mathit{Bind\ Name\ Mexp}\\ &\mid \mathit{Ebind\ Name\ Mexp}\end{aligned}$$

`Fundef` is used to build representations for local combinators and top level combinators which are not exported while `Efundef` builds representations for top level exported combinators. Similarly, `Bind` constructs unexported top level bindings and local bindings while `Ebind` constructs exported bindings.

Function definitions are transformed into combinators through lambda lifting (Section 4.2.4). Exported combinators are exported together with their arity information so that the importing modules can specialise applications of the combinators it imports (see Sections 4.3.2 and 4.4).

4.4 Modules

A complete program in Haskell consists of one or more modules. A module is a self-contained program unit which can import/export entities from/to other modules in a program. Our implementation does not support mutual dependencies between modules; the import/export dependencies form only a directed acyclic graph.

Modules are compiled separately in the order specified by their import/export dependencies. Modules can only be defined at top level and no two modules may be defined in a single file. A module may implicitly or explicitly export the entities it imports or implements. Only a subset of the entities exported by a module need be imported and it is an error to import an entity through more than one route (we omit the **renaming** construct of the full Haskell language).

```
type Program   =  [Module]
data Module    =  Mod String [(String, [Imports])] [Def]
data Imports   =  Comb String Int
               |  Val String

type Dialogue  =  [Response] -> [Request]
data Request   =  ReadFile String
               |  WriteFile String String
               |  AppendFile String String
               |  ReadChan String
               |  AppendChan String String
               |  GetArgs
               |  GetProgName
data Response  =  Success
               |  Str       String
               |  StrList   [String]
               |  Failure String
```

The type `Module` above describes a module's representation in the intermediate language. The components of the `Mod` constructor correspond, respectively, to the module's name, some information about the modules it imports and the names of the values it implements. The second component of `Mod`, `[(String,[Imports])]`, describes a data structure in which an imported module is associated with the values imported from it. The type `Imports` distinguishes imported combinators from other values. Imported combinators include their arity information so that their applications in the importing module can be specialised (see Section 4.3.2) so as to simplify the runtime system.

Following the Haskell 1.2 standard, one of the modules constituting a program must be called `Main` and must implement and export the combinator `main`. The value of the program is the value of the `main` identifier and `main` must have type `Dialogue`. Chapter six presents the description of our implementation of stream I/O involving the `Request` and `Response` data types defined above.

4.5 Summary

In this chapter we have presented the design and implementation of the front end of our Naira compiler comprising the lexical analyser, parser, pattern matching compiler, lambda-lifter and type checker all of which are written in Haskell. We have also described our symbol table environment consisting of different symbol tables used at different stages of the compilation so that each can be dropped after the transformation in which it is used.

The abstract syntax tree initially produced by the parser is transformed by the subsequent phases of the compilation finally producing an intermediate language with simplified constructs suitable for efficient code generation. For example, function applications are statically specialised, where possible, into a form that allows code to be generated which causes many function calls to be performed directly, thereby avoiding the costs of argument satisfaction checks. Further literature survey revealed that this transformation which statically specialises general function applications is a reinvention of similar ideas proposed elsewhere [Hamm88, Nikh89].

Each pattern in the alternatives of a `case` expression is identified with a small integer (a family tag) and occurrences of its subpatterns in the corresponding expression are replaced by applications of a selector function on the `case` scrutinee, which can lead to a significant efficiency improvement.

The following chapter describes the design and implementation of the back end of the Naira compiler.

Chapter 5

The compiler back end

5.1 Introduction

The preceding chapter described the translation of the Haskell source program into the intermediate code of (Section 4.3). This chapter describes the design and implementation of the back end of the compiler. The compilation route we follow is summarised by (Figure 5.1).

The two main issues addressed by this chapter are: the design and implementation of a runtime support system and a code generator for the intermediate language produced by the front end. Following a growing popular trend which exploits conventional compiler technology, as in [Peyt92, GHW90] for instance, we generate code in the C language and a conventional C compiler is relied upon, as a "portable assembler", for the final native machine code generation.

The following section describes the details of the design decisions and the assumptions we make in our implementation. Section 5.3 describes our representation of data structures in the heap and

$$\begin{tikzcd}
\text{Haskell source} \arrow[d, Rightarrow] \\
\text{Intermediate code} \arrow[d, Rightarrow] \\
\text{C code}
\end{tikzcd}$$

Figure 5.1: Simplified compilation route

Section 5.4 describes our messages and message-passing communication protocols. Sections 5.5 presents the code generation process from the optimised intermediate code produced by the front end. In Section 5.6 and 5.7 we give some details about how the generated code is linked, its execution triggered and the value of the program printed.

5.2 Runtime design framework

The design theme underlying our implementation is efficient medium-grained, data-driven execution, which is widely believed to be essential for parallel processing on large-scale MIMD machines [ArIa87, Nikh89, CSS+91, Osth93]. The architectural requirements needed to support our model efficiently is that of multi-threaded, distributed memory machines consisting of a network of (conventional) processors which communicate with each other via asynchronous message-passing. Each of the processing elements in the network should have its own local memory, in the tradition of P-RISC [ArNi88] and STAR:DUST [Osth91], and there should be no global shared memory which can potentially create a bottleneck.

Simplicity of design (simple representation of heap objects, probabilistic load distribution, using environments rather than stacks, etc.) is an important consideration in our implementation since experimental results have indicated that extremely clever or very sophisticated designs may not have significant relative payoffs. For example, Appel's measurements for SML of New Jersey suggest that clever closure-representation techniques gain little and potentially lose a lot (in space complexity) [Appe92, Peyt92].

In contrast to other implementations, like that based on the STG-machine [Peyt92], all data objects/suspensions in our implementation are tagged and are allocated in the heap. Each suspension consists of a frame pointer and an entry point. The entry point specifies the thread to execute next when the object's value is required. The frame pointer, which should be viewed as a pair of a processor identifier and a local address in the processor, defines the context within which the thread is executed. Note therefore that pointers to suspensions are globally available to the machine.

As is usually the case in other implementations, the whole program is replicated on each processor node which executes a copy of the program. To tolerate long, unpredictable communication latencies, a program is compiled into a very large number of small threads of execution, provided the application contains some parallelism, with the aim of keeping all the processors busy. Parallel tasks, each comprising of multiple threads, are distributed actively and randomly across the processing agents. Since messages are passed asynchronously, each processor is expected to contain a message buffer to queue incoming messages. In the event of long range communication, therefore, the task waiting for the result of the communication can asynchronously transfer control to one of the runnable threads in the message buffer of the resident processor. Another requirement for an architecture to efficiently support our multi-threaded model of computation is support for fast context-switching between parallel threads of execution.

The theoretical results of Eager *et al* in [Eage86] which assumes zero-cost communication, seem to indicate that a simple task distribution mechanism can be satisfactory and these results show that any algorithm is within a constant fraction of an optimal distribution strategy. It has also been known for more than two decades [Grah69] that a large collection of heuristics, the so-called list scheduling strategies, have a performance that is not far from optimal[1].

The use of environments rather than stacks in our implementation (i.e., since we allocate all objects in the heap) has the advantage of providing a convenient way of transporting large amounts of data across processors and means that complicated analyses, like Lester's *s*tacklessness analysis [Lest89], which is used to determine maximum stack sizes, are avoided. Furthermore, in our implementation, the exact frame size of each function is automatically determined by the code generator as described below.

Function applications form the basis of program decomposition and grain of computation. In other words, function invocations form the unit of parallelism and load distribution. Each function invocation is associated with a unique function frame and the function calls for parallel computations are distributed using an active distribution scheme in the sense that work is distributed to the processors as it becomes available and without the processors needing to request work.

The code associated with a function invocation is executed on the processor which holds the frame of the function so that accesses to the locations in the function frame are local operations and therefore very fast.

Our execution model is completely data-driven in the sense that when a message is received the data packaged inside the message drives the execution of the handler thread specified by the message. A thread in our case refers to an ordinary sequence of C code and there could be an arbitrary number of threads within the context of a single function frame. Threads are executed sequentially until they terminate, without pre-emption. Our support for light-weight threads in this way increases the likelihood of creating useful parallelism in a program capable of exploiting the computing power of parallel machines effectively.

[1]These strategies, however, only work with a multiprocessor in which the cost of assigning a parallel task to any processor is zero [Hofm94].

5.3 Data representation

All objects in this implementation are tagged and allocated in the heap. We identify three families of objects, namely function frames, constructed cells and suspension cells. A unique function frame is associated with each function invocation. Tuples and other constructors have a uniform concrete representation; tuples are also associated with integer tags to facilitate a uniform treatment of constructors (see Section 6.2). Suspension objects are tagged according to their state of evaluation. In the next section we describe our tagging system on heap objects followed by a description of the three families of heap objects.

5.3.1 Runtime tags

The purpose of this section is to distinguish the different kinds of 'tags' that we associate with heap objects and to highlight the need and purpose of each of these tags.

We have already explained, during the intermediate language optimisation of Section 4.3.4 and elsewhere, that each constructor of a type is associated with a small integer tag which distinguishes it from the other constructors in its family. Notice that this *family tag* (called the *structure tag* in [Peyt87]) is only associated with constructors and different constructors from different types can have the same integer tag. In addition to the benefits enumerated in Section 4.3.4, family tags are used at runtime in the implementation of comparison operations over values of user-defined data types (see Section 6.2).

The next set of tags used in our implementation are those that signify the state of evaluation of heap values. There are three different evaluation tags corresponding to `SUSPENDED` (i.e., tag 0), `BUSY` (i.e., tag 1), and `READY` (i.e., tag 2) evaluation states (see Section 5.3.4). Unlike family tags, each heap value (constructor or otherwise) is in one of these evaluation states at any given time. Notice that these evaluation states are closely related to but different from *evaluation modes* in the intermediate language (Section 5.5.3) arising from parallelism annotation in the source program. Evaluation modes specify what expression should be eagerly evaluated, resulting in such values short-circuiting the default `SUSPENDED` state and assuming the `BUSY` state first before finally (hopefully, given termination) entering the `READY` state.

Lastly, there is a pair of tags used to distinguish basic values from constructed values. We assign a tag of zero to all basic values (integers, characters and Booleans) while aggregate values have non-zero tags. By ensuring that family tags start from 1 within each type, this separation of values into two categories is realised. This provides a uniform treatment of values which is advantageous when implementing comparison operations on aggregate values of algebraic types.

The classification is very similar to the pointer and non-pointer tags in Peyton Jones [Peyt87] but there are differences because all values are accessed by following pointers (i.e., values are boxed) in this implementation.

This classification of heap objects into basic values and aggregate values (cf. pointer stack and basic value stacks in stack-based implementations such as that of LML)) serves as a good aid for memory management: whenever the result of a function call returns with a basic value, the associated activation frame can safely be freed. The memory reuse enhanced by this representation leads to space savings of up to 25% in some benchmark programs. In fact we were unable to run to completion some of our benchmark programs from our benchmark suite (see Chapter 8) prior to the incorporation of this dynamic memory manager. Notice that it is incorrect to free an activation frame which returns with a WHNF value. This is because of the lazy semantics of Haskell; there may be some live data inside the returned value which may be needed later in the execution process after the return.

5.3.2 Function frames

A function frame is physically represented by a pointer to a contiguous block of heap-allocated storage. The first component of this frame contains the address of the destination to which the result of the function application will be sent when it becomes available. The next fields are pointers to the function's arguments followed by space to store transient local data since the frame serves as a workspace for the function call. Each function invocation is associated with a unique function frame and frames are allocated by simply calling the UNIX *malloc* function, for example as in Hartel *et al* [HGW94] and Hicks *et al* [HCAA94]. Experiences from the implementation of the functional language KIR [Klug94] suggest that managing some program-specific heap from within the code can be expected to improve performance by 10% [Scho96]. Our implementation ensures that each request for space allocation is successful before continuing. Figure 5.2 shows the outline of a function frame in the heap.

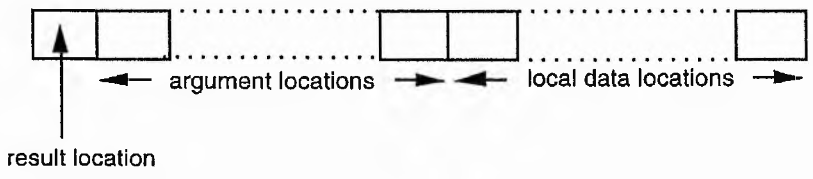

Figure 5.2: Structure of a function frame

Since all functions in the intermediate language have been transformed into combinators (i.e., functions without free variables), the generated code does not have to be concerned with free variable accesses and since code for a function call is executed on the processor holding the frame, access to the function's arguments are local operations. This is an important benefit of performing lambda lifting in a parallel implementation since the free variables may otherwise have to be accessed remotely.

The size (number of slot locations) of a frame is determined by the compiler at code generation time. The frame size of a combinator is obtained by adding 1 to the number of arguments of the associated function and the number of cells that will be required by the expression in the combinator's body. We illustrate this using a simple example given below.

Example 5.1

$f\ x\ y\ = x\ /\ y$

In this example, a function frame of length 5 is allocated for `f` by the current version of the compiler. The five slots store: the continuation of the function calling `f`, one slot for each of the two arguments, a cell to hold a synchronisation count indicating whether both components of the division operation have arrived and a slot to store the operand which is first to arrive. When the second argument arrives, the division is performed immediately, and the result location written.

Notice that the continuation of the calling function, a pair of frame pointer and code label, must be deposited in the frame of the called function (along with the arguments) so that when the latter returns it knows the requester and therefore notifies it with the result.

Since communication is overlapped with computation in our set-up and in particular since a function does not block when it calls another, the dynamic call structure of the computation is, in general, described by a tree of activation frames, often called a *cactus or seguara stack.*

5.3.3 Constructed cells

Data structures are, by default, non-strict in Haskell. Therefore a suitable representation for their graph must be arranged. As outlined in Section 4.3.3, tuples and other constructed objects translate into the intermediate form

```
Constr n [MExp]
```

where `n` is a small integer used to distinguish a constructor from its siblings. `[Mexp]` is the list of modal expressions representing the components of the constructor. Each tuple constructor is assigned a 'family tag' of 1 (see Section 6.2). Figure 5.3 shows the structure of all constructors in the heap.

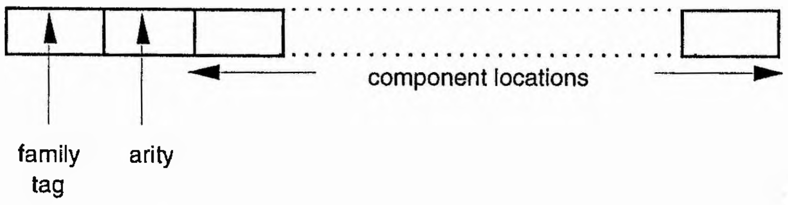

Figure 5.3: Structure of a constructed object

As part of their layout information, the representation of structured objects includes a size information field for garbage collection and for equality and related tests.

5.3.4 Suspension cells

A suspension object lies in the heap in one of three evaluation states: as a suspended or unevaluated thunk, in a busy evaluating state or in a ready evaluated (WHNF) form. These states of computation are represented by small integers. A suspension is represented by a contiguous area of storage space containing three storage cells as depicted in (Figure 5.4).

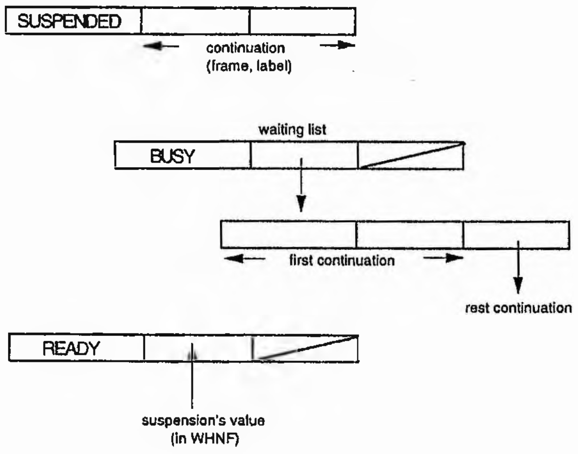

Figure 5.4: Structure of suspension objects

The first storage slot holds the status flag of the suspension; `SUSPENDED`, `BUSY` or `READY`. `SUSPENDED` is the default tag in a non-strict language since values are passed to functions, or stored in data structures in unevaluated form. Any value which does not have a parallelism annotation passes through these evaluation states, if it ever needs to be evaluated, in the order `SUSPENDED`, `BUSY` and `READY`.

The second and third fields of a SUSPENDED object contain a frame pointer and an entry point for the next sequence of code to be executed in the frame when the suspension's value is requested. A BUSY suspension contains a *waiting list* of the computations that requested the value of the suspension. The third cells in the BUSY and READY suspensions are shown to emphasise the metamorphosis of these suspensions from the SUSPENDED state.

A task in the BUSY state in this implementation corresponds to a black-holed[2] process in the parallel implementations of the GHC and HBC runtime systems. When the value is computed, the BUSY suspension is physically overwritten with a pointer to the value and all the waiting processes are notified with a copy of the computed value. Overwriting the BUSY suspension also involves changing its evaluation tag to READY. A READY suspension contains the (WHNF) value of the suspension. In relation to the *Running*, *Runnable* and *Blocked* tasks described in the simulator of Sections 6.4, a value under evaluation by Running or Blocked task is in BUSY state and that to be evaluated by a Runnable task is in the SUSPENDED evaluation state.

Mutating a suspended object, upon execution of its code, into a BUSY object is the synchronisation mechanism used to ensure that suspensions are evaluated by one process only and that other processes can always share the result. Note that changing the evaluation status of an object is accomplished in a single indivisible operation in order to achieve correct behaviour.

5.4 Messages and communication

All requests and responses for values in the heap are accomplished by sending messages. Different messages have different number of arguments and argument types. Following the classification of [GoHu86], we can identify three classes of messages, namely, *dialog*, *evaluation* and *storage* messages. Dialog messages are used to implement the stream I/O used in our system (see Chapter 6). Evaluation messages are those dealing with computations and storage messages are those dealing with storage management in the system.

Corresponding to each message is a message handler which accesses the message's data, performs the operation required to accomplish the message and terminates. Notice that a message handler may spawn an arbitrary number of messages before its execution completes. Each message handler is a parameterless function, in the tradition of Peyton Jones [Peyt92].

To give a flavour of our messages and message-passing communication, we describe the standard

[2] In a (sequential) unthreaded world, a black hole indicates a cyclic data dependency, which is an error resulting in deadlock and non-termination. In the threaded world, however, a black hole may simply indicate that the desired expression is being evaluated by another thread. In the latter case, therefore, the requesting thread simply blocks and waits for the black hole to be updated [Peyt94].

messages involved during the computation of a suspension's value. These messages are:

```
sendEval(dest, context, label)
sendUpdate(dest, value)
sendNotify(dest, label, value)
sendMessage(dest, label, msgData)
```

`dest` is the destination processor which holds the target object whose value is sought. `context` is the current working environment of the sender and `label` is the address of the next thread to execute. `value` is the computed value of the suspension. `msgData` is the data packaged with the message which is consumed by the handler specified by `label`.

`sendMessage` is the generic message form which all other messages take before they are scheduled. The others are specialised commonly used message forms. For example, `sendUpdate(dest, value)` first takes the form `sendMessage(dest, &doUpdate, b_value)` and then the `doUpdate` handler is entered to perform the update. `b_value` is a single-celled heap object which boxes the pointer to the value as expected by the `doUpdate` handler.

When `sendEval` is sent to an unevaluated suspension (i.e., in `SUSPENDED` state), its status flag is flipped to `BUSY`, a waiting list set up to store the `(dest,label)` continuation of this requester and the code to evaluate the suspension's value entered. Notice that because of our design requirements of non-blocking transactions, the source processor should not block (assuming source and destination processors are distinct: the communication is non-local), rather, it should switch context to one of the several dormant threads in its context store. The same applies to subsequent requests of the suspension's value before the computation of the value completes.

When the suspension's value becomes available the status flag is flipped to `READY`, to avoid re-evaluation when the value is requested in the future, and the computations that requested the value are all notified, by way of `sendNotify`. The handler for `sendEval` now immediately services any request for the suspension with (a pointer to) the value since the suspension would have physically been updated with a pointer to the value.

5.5 Code generation

This section discusses the compilation scheme which forms the basis of our code generator. The code generator takes as input the optimised intermediate code (after some preprocessing as described in Section 5.5.2) described in Section 4.3 and outputs standard C code. The compilation scheme presented here is an extension of the basic compilation rules given by Ostheimer in [Osth93] for a small first-order language. The main features added are therefore those dealing with general (higher order) function applications and modules.

In Section 5.5.1 we describe the symbol tables created and used during the code generation. Section 5.5.2 describes a parser which interfaces the compiler's front and back ends written in Haskell and C respectively. Section 5.5.3 through 5.5.6 describe the code generation process, respectively, for modal expressions, ordinary expressions, bindings and modules.

5.5.1 Code generator's symbol tables

We maintain a symbol table (in addition to those described in Section 4.2.1) as part of the implementation of the code generator. This symbol table associates identifiers with small non-zero integers and supports the managing of frames to aid the implementation of the **let** construct. It also provides support for 'stacking' symbol tables (i.e., setting up a new symbol table that completely hides the current one until the new symbol table is released).

Global names (i.e., those which can be exported and which are either defined locally of imported) are entered into the symbol table using negative integer entries. These entries specify the module in which the names are defined and each global name is attached to the name of the module which defines it, so as to aid the readability of the generated code (see symbol table for module names below). Argument identifiers for each function are numbered beginning at 1. These integers are used to refer to the suspensions corresponding to the associated names in the frame created by the function handler. Unique positive integers are generated for the names of local bindings during code generation (see Example 5.2).

The symbol table is currently organised as a linked-list of symbol table frames. In order to support the frame management mentioned earlier, each function definition (global or local, which can potentially have a block of local bindings) is associated with its own table frame which hides the argument variables in this function. When the code generation of the current function completes, the associated table frame is dropped. Notice that when the handler for a supercombinator is activated (see Section 5.5.5), the return address component of the message data is placed in the first location (location 0). Our symbol table then ensures that for each function frame, the ith frame location holds a pointer to the value of the ith argument expression. Furthermore, this systematic integer association with variables facilitates the determination of the exact frame size for each function by the code generator (see Example 5.2).

An auxiliary symbol table is also maintained which stores the corresponding name/integer association for the current module and for the other modules it imports. For a module `M` with integer identifier i in the module names symbol table, each (value) identifier implemented by `M` is associated with the negative integer $-i$ in the environment symbol table. When a global variable identifier is used in the current module—when it is looked up from the environment symbol table—the associated negative integer is used as the key to find its parent module name, in the modules table, which will be attached to the identifier's name, like `main_from_Main` for the global identifier **main** defined in the module `Main`.

Example 5.2

$$\begin{aligned} f\ x\ y = \textrm{let}\ & g\ x = x\ /\ y \\ & z\ \ \ \ = x \times 3 \\ \textbf{in}\ & g\ z \end{aligned}$$

As explained above, the global name `f` is associated with a negative integer in the symbol table and its parameters `x` and `y` are associated with the integers 1 and 2 respectively. Note that our lambda lifter will pass `y` as additional argument to g and change the application `(g z)` to `(g y z)` in the intermediate code and so on. The code generation function for expressions is passed an integer parameter used to keep count of the frame size for the function in which the expression is part. `z` and g are therefore assigned the integers 3 and 4, the frame locations 5 and 6 are generated for the suspensions of `y` and `z` in the call `(g y z)`. The frame size for `f` is therefore 7 while that for g is 5 (as explained in Example 5.1).

5.5.2 A parser interface

As discussed in Chapter 4 the front end of the compiler produces an intermediate code structured as a Haskell data type. A parser is written which processes this intermediate code and builds the corresponding C structures in a way that is suitable for input to the code generator. The runtime system manipulates tagged heap objects which are represented using different data structures, in a similar manner to the representation adopted by Hartel *et al* in [HGW94]. The C data structure, `Exp`, shown in Appendix A corresponds directly to the Haskell data type `Exp` of Section 4.3. Corresponding structures for programs, modules, bindings and import/export entities are given in the following subsections.

As part of the runtime support for our graph reduction model, an allocation function is defined for each substructure which reserves space for the structure in the heap. Each allocator initialises the fields of the structure and returns a pointer to the structure it allocates. Each structure is distinguished from its siblings using a small integer value.

The C function in Figure 5.5 shows a sample allocation routine which claims a node from the heap for conditional expressions.

```
extern Exp *makeIfExp (Exp *condition,
                       Exp *consequent,
                       Exp *alternative)
{
    Exp *e = (Exp *) safeMalloc (sizeof (Exp));

    e -> tag = IF;
    e -> fields.ifExp.condition   = condition;
    e -> fields.ifExp.consequent  = consequent;
    e -> fields.ifExp.alternative = alternative;
    return e;
}
```

Figure 5.5: Sample allocator for conditional expressions

This completes our discussion for the basic run time support of the compiler and we now focus on the details of the code generation process.

5.5.3 Compiling modal expressions

Modal expressions are expressions in the intermediate language that contain `Process` and `Value` annotations (evaluation modes). Evaluation modes are associated with arguments to functions, constructors, components of data structures and to the expressions forming the right-hand sides of non-function bindings. These tags arise from the programmer's annotations in the source program. As outlined in Section 1.3, the operational meaning of these modes is that `(Process e)` causes `e` to be evaluated in parallel with surrounding context, `(Value e)` forces `e` to be passed by value and the default `(Need e)` suspends the evaluation of `e` until its value is requested.

Accordingly, code is generated, using the $\mathcal{C}$ compilation function of the next section, to build suspensions for `Need`-annotated expressions. For each `Value`-annotated expression, which is supposed to be passed by value, its code is immediately followed by termination (i.e., returning a NULL code pointer), thus forcing this code to be executed immediately to bring the value of the expression to normal form before passing it.

Notice that `Process` is normally attached to function applications since as we pointed out earlier, function applications form the basic units of work distribution. Annotating other forms of expressions with `Process` will behave as if the expression has the default `Need` annotation. Notice also that the primary effect of the `Process` annotation is to determine where (i.e., either on the current processor or on another one) a task is to be offloaded. It neither alters the evaluation order nor the evaluation degree of its argument expression.

Code is therefore generated for `Process`-annotated expressions which determine the processor on which the associated call frame is allocated and the code executed.

In the compilation schemes presented in the next section, the termination of an executing thread is indicated by a horizontal line at the bottom of the code sequence.

5.5.4 Compiling expressions

As mentioned at the start of this section, this implementation adopts the compilation scheme developed in [Osth93] for a first-order language and extends it to cover a higher-order language using the `Call`, `Close` and `Apply` mechanisms described below. Five compilation functions were developed, $\mathcal{P}$ for compiling programs, $\mathcal{M}$ for compiling modules, $\mathcal{D}$ for definitions, $\mathcal{C}$ for modal expressions and $\mathcal{C}'$ for ordinary expressions.

Each of these compilation functions takes a piece of abstract syntax and produces a sequence of C statements. The $\mathcal{C}$ scheme normally invokes the $\mathcal{C}'$ scheme to perform the bare code generation while the former wraps up this code with the necessary pieces of code reflecting the original programmer annotation from the source program.

Notice that the compilation rules for top level expressions in this section, using the $\mathcal{C}'$ function, have no finishing horizontal line for termination. This is because such a termination has been 'factored' and used at a single place instead; in the rule for compiling bindings (Section 5.5.5).

- **Literals**

  All values, including literals, are boxed (i.e., their concrete representations in the heap contain two or more storage cells, see for example [Peyt87]) in our implementation. The compilation rule for literal values (integers, characters and Boolean) is given in Figure 5.6.

$$\boxed{\begin{array}{l} \mathcal{C}' \ [\![\text{ const }]\!] \text{ result} = \\ \qquad \textbf{update}(\text{result, const}) \end{array}}$$

Figure 5.6: Compilation rule for `Literals`.

This compilation rule corresponds to the π-calculus action which transmits the value of the constant on the designated channel. Therefore following the operational semantics of the

π-calculus specification, $[\![c]\!]o \overset{def}{=} \bar{o}c$, this compilation rule simply expands to a sequence of code which boxes the literal and sends the boxed object back to the requesting process.

If the literal value is part of the computation of a bigger expression in the right-hand side of some binding, a `sendNotify` message is sent to restart the computation. If, on the other hand, the literal value is the entire right-hand side of a binding, like 3 in `v = 3`, then a `sendUpdate` message is generated. Recall that the handler for an update message dispatches a number of `sendNotify` messages to the computations waiting for the value of the suspension being updated.

- **Identifiers**

  An identifier is referred to by first sending a `sendEval` message to it to ensure that it is evaluated as specified by the compilation rule of Figure 5.7.

```
C′ ⟦ id ⟧ result =
          eval(id, L)
      L:  update(result, msg.value)
```

Figure 5.7: Compilation rule for `identifiers`.

  If the identifier being referenced is unevaluated, its evaluation is initiated and if it is evaluated its value is immediately sent back to the requesting process. This compilation rule corresponds to the π-calculus process for name reference, $[\![x]\!]o \overset{def}{=} (r)(\bar{x}r \mid r(v).\bar{o}v)$ in which x, v and o correspond to *id*, *msg.value* and *result* in the compilation rule. As explained in Section 2.4, a resuest for x is sent along the new channel r (corresponding to the `eval` message to *id*). The value is eventually sent back along r and deposited on the output channel o. The direct correspondence can be seen in the compilation rule with the request via `eval` and the response via `update`.

- **Function applications**

  The intermediate language specialises general function applications into one of `Call`, `Close` or `Apply` as described in Section 4.3. In what follows, we give the compilation rules for the three extreme cases of function application in which all argument expressions to the function application have the same kind of annotation.

  Figure 5.8 shows the compilation rule for lazy function application. The identifier `call`

```
$\mathcal{C}'$ [[ call f Mexp$_1$ ··· Mexp$_n$ ]] result =
            fp [s$_1$] := suspended_node (L$_1$)
                  ⋮
            fp [s$_n$] := suspended_node (L$_n$)
            go to L
    L$_1$:  $\mathcal{C}$ [[ Mexp$_1$ ]] s$_1$
            ________
                  ⋮
            ________
    L$_n$:  $\mathcal{C}$ [[ Mexp$_n$ ]] s$_n$
            ________
    L:      call (f, fp [result], fp [s$_1$], ···, fp [s$_n$])
```

Figure 5.8: Compilation rule for `lazy function call`.

which surrounds the application corresponds to the constructor `Call` in the intermediate language of Section 4.3 and it is used in order to emphasise the correspondence between the compilation rule and the intermediate language representation.

As explained in Section 5.2, each function invocation is associated with a unique function frame (the workspace for the call) and `fp` here is used to represent such a frame in which the argument suspensions are stored. The labels L_i are the code addresses for the computation of the argument expressions.

Notice that as in the encoding for abstraction in Section 2.4, the function value is readily available and code is generated based on this rule which allocates a function frame, suspends the arguments and activates the function application. Recall from Section 2.4 that the corresponding π-calculus specification of call-by-need is

$$[\![MN]\!]o \overset{def}{=} (m)(x)([\![M]\!]m \mid m(f).(\bar{f}(x,o) \mid bind(x,N)))$$

Comparing this with the compilation rule of Figure 5.8, we see that in the compilation rule the number of arguments is generalised and each of these arguments is a modal expression—expression containing annotation information. Notice that there is no prospect for parallelism in this rule between the evaluation of the function arguments and the function call or even among the computations for the argument expressions.

The compilation scheme of Figure 5.9 specifies the rule for parallel call-by-value function application. As in the previous rule, the direct correspondence between the compilation rule of Figure 5.9 and the corresponding π-calculus specification

$$[\![MN]\!]o \overset{def}{=} (m)(n)(x)([\![M]\!]m \mid [\![N]\!]n \mid m(f).n(v).\bar{f}(x,o).store(x,v))$$

is clear. Even though the function f value is available the function invocation is not started until the arguments are evaluated as indicated by the variable `count` in the rule.

```
C' [[ call f Mexp_1 ··· Mexp_n ]] result =
        count := n
        fp [s_1] := busy_node (L_1)
              ⋮
        fp [s_n] := busy_node (L_1)
        C [[ Mexp_1 ]] s_1
              ⋮
        C [[ Mexp_n ]] s_n
        __________
L_1:    count := count - 1
        if (count = 0) goto L_2
        __________
L_2:    call (f, fp [result], fp [s_1], ···, fp [s_n])
```

Figure 5.9: Compilation rule for `parallel call-by-value`.

In contrast to the previous rule for lazy function call, this rule provides scope for the parallel evaluation of the argument expressions. Notice, however, that since granularity is at the level of function application, the parallelism comes about only when some of the arguments involve function applications, leading to the creation of a parallel task on another processor.

The compilation rule of Figure 5.10 is used to generate code for a parallel function application. The main difference between this rule and that of Figure 5.9 (as is the difference in the corresponding π-calculus encodings) is that the function invocation is not 'guarded' by the completion of the computation of the function arguments. The name `waiting_node` is used in Figure 5.10 to indicate that each suspension is waiting to be updated with the result of a function call.

```
C′ ⟦ call f Mexp_1 ··· Mexp_n ⟧ result =
        fp [s_1] := waiting_node ()
            ⋮
        fp [s_n] := waiting_node ()
        C ⟦ Mexp_1 ⟧ s_1
            ⋮
        C ⟦ Mexp_n ⟧ s_n
        call (f, fp [result], fp [s_1], ···, fp [s_n])
```

Figure 5.10: Compilation rule for `parallel call-by-process`.

This provides the most opportunity for parallelism: both horizontal parallelism (parallel evaluation of function arguments) and vertical parallelism (parallel execution of a function with the evaluation of its arguments) can be exploited.

Our compilation rules for dealing with general (i.e., mixed parameter mode) function applications, constructor applications and variable bindings combine aspects of the above three rules. However, because of the different possible ways[3] a programmer may place annotations, it is not possible to capture the general combined effect of the above three rules without prior knowledge of which argument expressions take what annotations.

Compiling general annotated function application

Assuming some specific annotations[4] for the argument expressions, we can express general function application as in Figure 5.11. Clearly, this is just one of many possibilities and, we therefore opt to use the compilation scheme for lazy function application to specify the rules for closure creation and closure application, while bearing in mind that they are compiled using all the three rules based on the annotations on the argument expressions.

[3]There are 3^n different possible representations for annotations (including default ones) in a function with n arguments.

[4]The assumptions are: the first i arguments are call-by-need, the next $i+1$ to k arguments are call-by-value and the last collection of arguments ($k+1$ to n) are call-by-process.

```
$\mathcal{C}'$ [[ call f Mexp$_1$ $\cdots$ Mexp$_n$ ]] result =
          count := k - j + 1, ($j = i + 1, m = k + 1$)
          fp [s$_1$] := suspended_node (L$_1$)
               ⋮
          fp [s$_i$] := suspended_node (L$_i$)
          fp [s$_j$] := busy_node (M)
               ⋮
          fp [s$_k$] := busy_node (M)
          fp [s$_m$]:= waiting_node ()
               ⋮
          fp [s$_n$] := waiting_node ()
          $\mathcal{C}$ [[ Mexp$_j$ ]] s$_j$
               ⋮
          $\mathcal{C}$ [[ Mexp$_k$ ]] s$_k$
          $\mathcal{C}$ [[ Mexp$_m$ ]] s$_m$
               ⋮
          $\mathcal{C}$ [[ Mexp$_n$ ]] s$_n$
          __________
    M:    count := count - 1
          if (count = 0) goto N
          __________
    L$_1$:   $\mathcal{C}$ [[ Mexp$_1$ ]] s$_1$
          __________
               ⋮
          __________
    L$_i$:   $\mathcal{C}$ [[ Mexp$_i$ ]] s$_i$
          __________
    N:    call (f, fp [result], fp [s$_1$], $\cdots$, fp [s$_n$])
```

Figure 5.11: Compilation rule for 'general' function call.

Example 5.3. Consider the call of the Takeuchi function (Section 8.3.3).

$tak\ (process(tak\ (x\ -\ 1)\ y\ z))\ (process(tak\ (y\ -\ 1)\ z\ x))\ (process(tak\ (z\ -\ 1)\ x\ y))$

Notice that since the three arguments are all `process`-annotated, this is a call by process application and code is generated for it using the compilation rule of Figure 5.10. Figures 5.12 and 5.13 show a sketch of the code generated for this application.

From the complete definition of the Takeuchi function (Section 8.3.3) and the compilation rule for combinators (Section 5.5.5), we see that the code generated for the expression right-hand side of combinators is ended by a `return NULL` (as in the `tak_from_Main` in Figure 5.12) to trigger the evaluation of the combinator's body immediately. When `tak` is called the thread labelled `tak_from_Main` is executed which allocates a function frame for the call, uses the frame pointer and the PID to make a global address, stores the pointers to the function's arguments in the frame and finally terminates. This termination causes the two `sendEval` messages (requesting the values of x and y for the computation of $x <= y$) to be handled. Depending on which of these two messages is handled first, execution continues at L_5 or L_6 using the value in `frame[4]` for synchronisation since the operator '<=' requires its arguments in the right order. When the value of the conditional is computed, execution continues at L_2.

The value of the conditional is received at L_2 and if this value is TRUE a jump is made to L_3 where the value of z is returned otherwise the rest of the code in L_2 (for the call to `tak` in this example) is executed. With the aid of the compilation rule of Figure 5.10, the code generated for this call is straightforward: it creates three *waiting suspensions* (each waiting to be updated with the value of a function call) for the three parallel argument computations, suspends the three arguments of the first parallel call (the last three suspensions in L_2) and jumps to L_9. At L_9, a message is built and sent to one of the machine's PEs which will be responsible for the evaluation of the first child task. The suspensions for the three arguments to the second child task are constructed in L_9 before jumping to L_{18} where the message for the call is built and sent. Notice that very similar operations are performed in L_{18} and L_{27} for the second and third child tasks, respectively (see Figure 5.13), as done in L_9 for the first child. The code in L_8 creates the message for the parent `tak` call and terminates. Notice that this is the first termination since starting from L_2. By the time this termination occurs, the four calls to `tak` would have been distributed to four PEs of the machine (assuming the machine has >= 4 PEs) which will of course be invoked in parallel.

```
extern void    * tak_from_Main()
{
    frame = makeFrame(tak_from_Main_size_1);
    fp = (Global) MAKE_GLOBAL(currentPE, frame);
    frame[0] = msg[0];                              /* result location for the call */
    frame[1] = msg[1];                              /* first argument suspension */
    frame[2] = msg[2];                              /* second argument suspension */
    frame[3] = msg[3];                              /* third argument suspension */
    sendEval((Global) frame[1], fp, &L_5);
    sendEval((Global) frame[2], fp, &L_6);
    frame[4] = (Value) 1;
    return NULL;                                   /* terminate to execute the function body */
}

static void    * L_2()
{
    Global          tVal = (Global) msg[0];
    if (tVal[1]) return &L_3;
    frame[6] = (Value) makeWaitingSuspension(fp);      /* suspension for the first child task */
    frame[7] = (Value) makeWaitingSuspension(fp);      /* suspension for the second child task */
    frame[8] = (Value) makeWaitingSuspension(fp);      /* suspension for the third child task */
    frame[9] = (Value) makeSuspension(fp, &L_10);
    frame[10] = (Value) makeSuspension(fp, &L_11);
    frame[11] = (Value) makeSuspension(fp, &L_12);
    return L_9;
}

static void    * L_9()
{
    {
        Message         m = makeMessage(4);
        m[0] = frame[6];
        m[1] = frame[9];
        m[2] = frame[10];
        m[3] = frame[11];
        sendMessage((Global) getNextPID(), tak_from_Main, m);
    }
    frame[15] = (Value) makeSuspension(fp, &L_19);
    frame[16] = (Value) makeSuspension(fp, &L_20);
    frame[17] = (Value) makeSuspension(fp, &L_21);
    return L_18;
}
```

Figure 5.12: First code extract for the call of tak in Example 5.3

For the parallel call by value case (which is a restricted form of the parallel call by process just described, see Figure 5.9) the same execution pattern obtains except that there will be an additional termination after sending the message for the third child task to ensure that

```
static void    * L_18()
{
    {
        Message         m = makeMessage(4);
        m[0] = frame[7];
        m[1] = frame[15];
        m[2] = frame[16];
        m[3] = frame[17];
        sendMessage((Global) getNextPID(), tak_from_Main, m);
    }
    frame[21] = (Value) makeSuspension(fp, &L_28);
    frame[22] = (Value) makeSuspension(fp, &L_29);
    frame[23] = (Value) makeSuspension(fp, &L_30);
    return L_27;
}

static void    * L_27()
{
    {
        Message         m = makeMessage(4);
        m[0] = frame[8];
        m[1] = frame[21];
        m[2] = frame[22];
        m[3] = frame[23];
        sendMessage((Global) getNextPID(), tak_from_Main, m);
    }
    return L_8;
}
static void    * L_8()
{
    {
        Message         m = makeMessage(4);
        m[0] = frame[0];
        m[1] = frame[6];
        m[2] = frame[7];
        m[3] = frame[8];
        sendMessage((Global) currentPE, tak_from_Main, m);
    }
    return NULL;                                /* terminate after creating the four tasks */
}
```

Figure 5.13: Second code extract for the call of tak in Example 5.3

the execution of the three child tasks is completed before the parent call starts. When the arguments are not explicitly annotated, a jump is made from L_2 (immediately the three suspensions for the three argument calls are built) to L_8 to make the parent call. Again, this is a direct translation of the compilation rule of Figure 5.8.

- **Closure creation** The compilation rule specifying the code generated for a function presented with fewer arguments than its arity is given in Figure 5.14. In this rule the function f has arity n but is supplied k arguments ($k < n$).

```
C′ ⟦ close f Mexp₁ ··· Mexpₖ⟧ result =
            fp [s₁] := suspended_node (L₁)
                 ⋮
            fp [sₖ] := suspended_node (Lₖ)
            go to L
    L₁:     C ⟦ Mexp₁ ⟧ s₁
            ________
                 ⋮
            ________
    Lₖ:     C ⟦ Mexpₖ ⟧ sₖ
            ________
    L:      c := make_close (f, n-k, k, fp [s₁], ···, fp [sₖ])
            update (fp [result], c)
```

Figure 5.14: Compilation rule for '`closure creation`'.

Code is generated to allocate a heap object for the closure storing the code label to jump to when the rest of the outstanding arguments arrive, the number of pending arguments, the number of arguments available (if any) and pointers to the available arguments. The result location is updated with this newly created closure which we further explain in the rule for closure application below. When the closure is later supplied with the remaining arguments, it is turned into a `Call` and the function call is performed. If the additional arguments supplied are fewer than the closure requires, the closure is replaced with a new extended closure with these additional arguments. If the additional arguments are more than closure needs to saturate it, an `Apply` structure is built. The `Apply` structure is a generalisation of the one compiled by the above compilation scheme where `f` could now be an arbitrary (closure-returning) expression.

- **Closure application** Code is generated for a structure built using `Apply` (i.e., higher-order function application/closure application) based on the rule given in Figure 5.15.

```
C' [[ apply Exp Mexp_1 ··· Mexp_k ]] result =
            fp [s_1] := suspended_node (L_1)
                ⋮
            fp [s_k] := suspended_node (L_k)
            go to L
    L_1:    C [[ Mexp_1 ]] s_1
            ________
                ⋮
            ________
    L_k:    C [[ Mexp_k ]] s_k
            ________
    L:      c := make_apply (fp [result], k, fp [s_1], ···, fp [s_k])
            fp [t]  := mk_apply_susp (c)
            C' [[ Exp ]] t
```

Figure 5.15: Compilation rule for `closure application`.

Notice that `Apply` arises from an application of a functional argument, which the compiler does not know and whose exact arity is therefore unknown statically. In such cases, the function application cannot be specialised to one of the two forms covered earlier and therefore the function being applied is a closure-valued expression as opposed to a code entry point in the earlier two cases. Notice also that this is one of the costs paid for using higher-order functions because just like our optimisation fails here, similar analyses like strictness analysis, for example see Hogen *et al* [HKL91], are harder to handle in the presence of higher order functions.

Code is generated to allocate a heap object for this apply construct to store the pointer to the result location, the number k of the available arguments and the pointers to the values of the arguments of the closure application.

An *apply suspension* is built using the newly created object and the closure-valued expression is evaluated immediately. An apply suspension is essentially a specialised kind of `BUSY` suspension which is specified by an *apply object* and a `doRestApply` entry point (see Figure 5.16). The apply suspension is eventually updated with the closure value and, once updated, `doRestApply` turns the closure value into a `Call`, an extended `Close` or `Apply`. The `doRestApply` handler has enough information (from the closure and the apply object) to perform the process described above until a final `Call` is obtained that performs the function application and sends the result to the requesters.

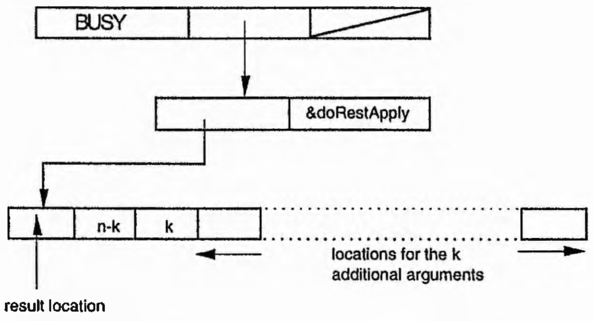

Figure 5.16: Structure of an *Apply suspension*

The operational semantics of the message handler `doRestApply` is similar to that of the `DO` instruction emitted by the < ν,G>-machine compilation rules for function applications in the LML compiler project [AuJo89a].

Before considering an example, we remark that since built-in functions are strict in their arguments, in this implementation, code is compiled for calls to these functions to evaluate their arguments eagerly and to perform the arithmetic specified by the function.

Consider the definition of the function ap and the associated call in Example 5.4 below.

Example 5.4

$ap\ f\ x = f\ x$

$succ\ n = n + 1$

$ap\ \ succ\ \ 3$

Since the compiler cannot determine `f` statically and while ap has arity 2, it generates the intermediate language structures, respectively, for `(f x)` and `(ap succ 3)`.

```
Apply (Id "f") [(Need (Id "x"))]
Call "ap" [(Need (Close "succ" 1 [])), (Need (IntLit 3))]
```

The code generated for `(f x)` builds a heap structure which we depict in the graph of Figure 5.17. The number 0 and 1 in the figure respectively stand for the number of arguments carried inside the closure value and the number of pending arguments. The code for `f` is then entered and the apply suspension updated with the closure value.

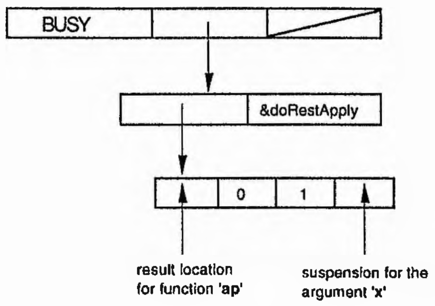
BUSY
&doRestApply
0
1
result location for function 'ap'
suspension for the argument 'x'

Figure 5.17: Apply suspension for the application `(f x)`

Therefore when `ap` is called the suspension of Figure 5.17 is updated with the closure for `succ`. The update handler will notify the `ap` continuation with the closure value and `doRestApply` will 'see' that the closure expects one argument as supplied by the environment and therefore `Calls succ` on the integer `3` and updates the result location of `ap` with the answer `4`.

- **Constructors**

  Like function applications, argument expressions to constructors can have parallelism annotations. Accordingly, the compilation scheme for constructors is very similar to that of function applications and it is specified as in Figure 5.18. Code generation for constructors is much easier than that for function applications because the compiler has transformed partial applications of constructors into saturated ones (surrounded by an explicit function definition). The code therefore constructs the object (see Figure 5.3) and sends an update or notify message, as the case may be, to the result node with a pointer to this cell.

- **Conditionals**

  Code generated for conditionals is based on the compilation scheme of Figure 5.19. Code is generated to force the evaluation of the conditional expression and the corresponding alternative is selected based on the value of the conditional. After executing the code for the alternative, execution continues at L_3, the code label for the expression following the conditional.

```
$\mathcal{C}' [\![$ con t Mexp$_1$ $\cdots$ Mexp$_n$ $]\!]$ result =
        fp [s$_1$] := suspended_node (L$_1$)
            ⋮
        fp [s$_n$] := suspended_node (L$_n$)
        go to L
L$_1$:  $\mathcal{C} [\![$ Mexp$_1$ $]\!]$ s$_1$
        ________
            ⋮
        ________
L$_n$:  $\mathcal{C} [\![$ Mexp$_k$ $]\!]$ s$_n$
        ________
L:      c := make_con (t, n, fp [s$_1$], $\cdots$, fp [s$_n$])
        **update** (fp [result], c)
```

Figure 5.18: Compilation rule for `constructors`.

```
$\mathcal{C}' [\![$ **if** Exp$_1$ **then** Exp$_2$ **else** Exp$_3$ $]\!]$ result =
        s := busy_node (L$_1$)
        $\mathcal{C}' [\![$ Exp$_1$ $]\!]$ s
        eval (s, L$_1$)
        ________
L$_1$:  if (!msg.value) goto L$_2$
        $\mathcal{C}' [\![$ Exp$_2$ $]\!]$ result
        goto L$_3$
L$_2$:  $\mathcal{C}' [\![$ Exp$_3$ $]\!]$ result
L$_3$:
```

Figure 5.19: Compilation rule for `conditionals`.

- let expressions

The intermediate language transforms a `let` expression into one of two forms (built with either `Let` or `Define`, as explained in Section 4.3.5) or into a cascade of both. The `Let` form contains only variable bindings in its definition list while `Define` contains only function bindings. The compilation rule for the `Let` constructor is given in Figure 5.20.

```
$\mathcal{C}'$ [[ let $\text{id}_1$ = $\text{Mexp}_1$, ···, $\text{id}_n$ = $\text{Mexp}_n$ in Exp]] result =
         fp [$\text{id}_1$] := suspended_node ($\text{L}_1$)
                  ⋮
         fp [$\text{id}_n$] := suspended_node ($\text{L}_n$)
                  go to L
    $\text{L}_1$:  $\mathcal{C}$ [[ $\text{Mexp}_1$ ]] $\text{id}_1$
         ________
                  ⋮
         ________
    $\text{L}_n$:  $\mathcal{C}$ [[ $\text{Mexp}_n$ ]] $\text{id}_n$
         ________
    L:   $\mathcal{C}'$ [[ Exp ]] result
```

Figure 5.20: Compilation rule for `Let`.

Note that the id_k are identifiers in the source program while they are indices into the associated activation frame of the function in the generated code.

This compilation rule is very similar to the code for first-order function application. In this case, however, each identifier of the **let** is used to bind the suspension of its corresponding expression. Notice that as we pointed out earlier, the right-hand side of some of these bindings could contain parallelism annotations and may therefore be in similar mixed modes to function arguments and constructor arguments covered earlier. Code is therefore generated which creates suspensions and directly executes the code for explicitly annotated expressions, bypasses the code for implicitly annotated expressions and executes the code for the expression in the `Let` body.

As we pointed out earlier, the code generator sets up a local environment (to hide an outer one for the outer enclosing block) for the identifiers in the **let** and eventually falls back to the outer environment on dropping the local one and returning from the **let**.

For the case of locally defined combinators, their compilation is governed by the rule in Figure 5.21.

```
$\mathcal{C}'$ [[ define $\mathrm{Def}_1$, ···, $\mathrm{Def}_n$ Exp]] result =
            goto L
            $\mathcal{D}$ [[ $\mathrm{Def}_1$ ]]
               ⋮
            $\mathcal{D}$ [[ $\mathrm{Def}_n$ ]]
    L:      $\mathcal{C}'$ [[ Exp ]] result
```

Figure 5.21: Compilation rule for `Define`.

Code is generated for `define` which directly jumps to the label `L` that addresses the code for the body, bypassing the code for the locally-defined supercombinators (see Section 5.5.5).

- case expressions

```
$\mathcal{C}'$ [[ case Exp (1, $\mathrm{Exp}_1$), ···, (n, $\mathrm{Exp}_n$) ]] result =
            fp [s] := busy_node(L)
            $\mathcal{C}'$ [[ Exp ]] s
            eval(fp [s], L)
            ―――――――
    L:      i := get_tag(msg.value)
            goto $L_i$
    $L_1$:     $\mathcal{C}'$ [[ $\mathrm{Exp}_1$ ]] result
            goto M
               ⋮
            goto M
    $L_n$:     $\mathcal{C}'$ [[ $\mathrm{Exp}_n$ ]] result
    M:
```

Figure 5.22: Compilation rule for `case`.

Code generation for case expressions is also greatly simplified because of the compile-time transformation of Section 4.3.4 which compiled out constructor patterns in case alternatives. That is, since constructors in the intermediate language are identified by small integers and their subpatterns (which are all variables) expressed in terms of the case-scrutinee expression using a selection function, the generated code does not require maintaining an environment for variables in the patterns of case alternatives. The compilation rule for case expressions is shown in Figure 5.22.

This compilation scheme is an optimised generalisation of the compilation scheme for conditional expressions. Code is generated which evaluates the discriminating expression of the case expression, extracts its tag, `i`, and takes a multi-way jump to the label $\mathtt{L}_i$, addressing the code sequence of the first expression whose associated tag equals `i`. When the execution of the code for the case expression completes, with the execution of the code at $\mathtt{L}_i$, the program execution continues at `M`, the code block following that of the case.

5.5.5 Compiling bindings

The compiler identifies and distinguishes those bindings in a module that are exported from those that are not exported. The code generator uses this information to generate the appropriate storage class declarations (extern or static) for the names of these bindings in the resulting C code. Combinator bindings are also distinguished from other value bindings and all this is reflected in the intermediate language.

```
𝒟 ⟦ f id₁ ··· idₙ = Exp ⟧ result =
        frame := make_frame(f_size)
        fp := make_global(frame)
        frame [result] := msg.result
        frame [id₁]    := msg.id₁
                  ⋮
        frame [idₙ]    := msg.idₙ
        C′ ⟦ Exp ⟧ result
        ________
```

Figure 5.23: Compilation rule for `bindings`.

For non-function bindings, code is generated in a similar way as for the same kind of bindings in let described above. A difference being that top level bindings may be exported. Code is generated for combinators as specified by the rule in Figure 5.23.

Each combinator compiles into a sequence of code which when executed performs the computation specified by the combinator. In other words, each supercombinator is translated into a handler which is eventually activated by a message to perform the function application. Whenever activated, the handler first allocates a new frame of the exact size, `f_size`, to hold the data required for the call before executing the code for the supercombinator's body. The function `make_global` combines the local frame pointer `frame` with the PID (of the machine on which the function is called) to make the global frame pointer `fp` which is inherited by all suspensions arising from the right-hand side expression of f. The code for the right-hand side expression of a combinator is followed by termination so that the computation specified by the function can be performed. This is why the compilation function C' in the preceding cases is not ended by termination (putting which will be superflous). Notice that the creation of a unique frame for each activation is necessary since the function may be called with different parameters and there may be an arbitrary number of parallel activations of the function. This directly corresponds to the π-calculus action $!f(x,y).[\![M]\!]y$, except that the π-calculus specification contains a single argument function and also abstracts over the frame environment. The compilation rule on the other hand, generalises the number of arguments and makes the working environment explicit.

5.5.6 Compiling modules

A module may import entities from other modules as well as export the entities it defines or imports. The intermediate language representation of a module includes an association between each imported module with an explicit list of values imported from that module as specified by the Haskell source for the module. Each imported value is further specified according to whether it is a combinator or a non-combinator value. The structure representing an imported combinator includes the arity information of the combinator as specified by the `Imports` type in Figure 5.24.

```
type Program  =  [Module]
data Module   =  Mod String [(String, [Imports])] [Def]
data Imports  =  Comb String Int
              |  Val String
```

Figure 5.24: Intermediate representation for `modules`.

A Haskell source module gives rise to a corresponding C module together with an associated header file. The header file, as expected, contains the declarations for the parameterless entry points in the code and the frame size macros for the combinators all of which are generated automatically.

The Haskell type in Figure 5.24 describes the intermediate code for modules (as presented in Section 4.4). The compilation scheme for a module is given by the rule in Figure 5.25.

$$
\begin{array}{l}
\mathcal{M}\ [\![\ \textbf{mod}\ \text{M importList Def}_1 \cdots \text{Def}_n\ ]\!] = \\
\qquad \text{mk\_extern\_decls(importList)} \\
\qquad \text{mk\_decl}\ [\![\ \text{Def}_1, \cdots, \text{Def}_n\ ]\!] \\
\qquad \mathcal{D}\ [\![\ \text{Def}_1\ ]\!] \\
\qquad \quad \vdots \\
\qquad \mathcal{D}\ [\![\ \text{Def}_n\ ]\!] \\
\qquad \text{M\_init()}
\end{array}
$$

Figure 5.25: Compilation rule for `modules`.

This rule is used to generate code for a module separately from the code for the other modules in the program. The 'macro' `mk_extern_decls` creates 'extern' declarations for the imported values from other modules. `mk_decl` generates similar declarations for the top level definitions implemented by the module. Code is then generated for the definitions in `M` and the module `M` initialised. Initialising a module consists of initialising its top level suspensions (i.e., for the top level non-function bindings defined by the module) as well as initialising the modules it imports. Notice that local definitions in the source are hoisted to the top level in the generated C code instead of embedding their code inside that of the **let** expression.

We now give our final compilation rule for complete programs. A complete program consists of a collection of modules as indicated by the type `Program` in Figure 5.24. The compilation scheme for programs is shown in Figure 5.26.

Notice that as our modules are not mutually recursive (see Section 4.4), the modules $\texttt{Mod}_i$ in the compilation scheme are in some import/export dependency order and that $\texttt{Mod}_n$ is the **Main** module which must define the combinator **main**.

5.6 Starting program execution

In the previous section we have specified the compilation rules for modules and programs and we now focus on the issues of linking a compiled program and triggering its execution. Note that the intermediate representation generated for each of the constituent modules of a program contains enough information for that module to be code generated and compiled separately. The program

```
$\mathcal{P}$ [[ prog $\text{Mod}_1 \cdots \text{Mod}_n$ ]] =
            s := suspended_node (N)
            eval (main, L)
            ________
    L:      print (msg.value)
            ________
            $\mathcal{M}$ [[ $\text{Mod}_1$ ]]
                 ⋮
    N:      $\mathcal{M}$ [[ $\text{Mod}_n$ ]]
            ________
```

Figure 5.26: Compilation rule for `programs`.

is therefore linked, in the standard way, using a C compiler.

In accordance with the Haskell syntax, the module `Main` defines the single-parameter combinator **main** with type (see Section 6.1.1)

$$main :: [Response] \to [Request]$$

and which specifies the program's value. The runtime system therefore prepares and sends the message

```
sendMessage(dest, main_from_Main, msgData)
```

to one of the machine's processors to execute the code for **main**. What is the message data for this function application? It is clear from the type for **main** that when its code is executed, it will build and update its result location with a WHNF request list. We therefore need some specialised message handler will arrange that execution continues after this update.

Before sending the message for the function call above, we set-up two `BUSY` suspensions which serve as the data for the message. The first suspension called a *waiting suspension*, has one continuation in its waiting list while the second (which is a `BUSY` suspension for the request list to be built) is more specialised with an empty waiting list.

The waiting suspension, which is a description of the next thread to be activated when the function call returns, is associated with an entry point, `receiveRequests`, which governs the main computation as described below. When the above message is sent, it activates the handler for the

main combinator which creates the required frame, places the waiting suspension component of the data item (i.e., the second `BUSY` object) into the result location of the frame, places the other message data in the next frame locations and enters the code for the body of **main**.

The waiting suspension is eventually updated with the lazy requests list leading to the activation of the associated continuation. In other words, `receiveRequests` receives the (WHNF) request list value, evaluates the first request and builds the corresponding response. If this is successful, `receiveRequests` then processes the next request, builds and updates the responses list and so on until all the requests are processed or an error message occurs to prematurely terminate the program execution.

Consuming and responding to the program's requests may involve performing some I/O operations, like writing some data to or reading some data from some I/O media. We discuss the issue of printing program results in the next section. The details of I/O implementation are discussed in Chapter 6.

5.7 Printing program result

A program's value can be a simple data item or an aggregate of such values. The programmer may require the value of the program to be displayed on the terminal or written to a file, for instance. To do this, the printing routine must ensure that each node it comes to handle is in evaluated form. The evaluator must therefore be called on each node before it is printed.

The current version of the compiler adopts a simple solution to the printing problem. As noted earlier the implementation of type classes may be necessary in order to obtain a systematic implementation of such overloaded functions as **show**. Our solution involves restricting the class of possible program values that can be printed as described by the following type signatures.

type *Display a* = *a* → *String*
showInt :: *Display Int*
showChar :: *Display Char*
showBool :: *Display Bool*
shows :: *Display String*
showList :: *Display* [α]
showTuple :: *Display* [α]

These six functions specify the only program values admissible for printing in the current state of Naira; integer, character and Boolean values and lists or tuples of these (i.e., the type variable α in `showList` and `showTuple` ranges over these basic types). The user must therefore specify the correct program value using one of these display functions. Values of user-defined types could be included by (possibly) adding type tags to the data objects at runtime.

5.8 Summary

In this chapter we have presented the design and implementation of the back end of our Naira compiler. The design space distinguishes three classes of objects in the heap; function frames, constructor cells and suspension cells. All objects in the implementation reside in the heap and communication is achieved via message passing with tolerance for high-latency memory requests.

A new unique function frame is associated with each function invocation. Constructors (tuples and sum constructors) have a uniform concrete representation to facilitate their manipulation by the runtime system. We have described the code generator which translates the intermediate code into C. We have also described how the execution of a compiled program can be driven and the program result printed.

The main research contributions as implemented and described in this chapter are itemised as follows:

- Extension and implementation of our compiler's π-calculus-based compilation rules
- Design and implementation of our compiler's sequential code generator
- Design and implementation of our compiler's multi-threaded parallel code generator
- Design and implementation of our compiler's runtime support system

Chapter 6

I/O & simulation issues

In chapter five we presented a detailed description of the runtime system of our compiler; how objects are represented in the heap, how our message passing system works and how we generate multi-threaded parallel code from our π-calculus inspired compilation rules. In this chapter we describe our implementation of other aspects of the runtime system using the facilities described in the preceding chapter.

The rest of the chapter is organised as follows. In the next section we review the motivation for, and the various ways of, realising I/O in (mainly functional) programming languages pointing out the style we adopt in our compiler, presenting the operations we support and describing how we implement those operations. In Section 6.2 we describe how we implement comparison operations on values of user-defined types which have the same structure as our I/O constructors. Section 6.3 covers scheduling issues and in Section 6.4 we describe a quasi-parallel simulator used in our experiments. A brief summary of the chapter follows in Section 6.5.

6.1 I/O in general

The essence of performing Input/Output operations in a programming language is to enable programs in the language to communicate with the outside world by manipulating I/O resources like files, keyboards, screens and so on. I/O commands, which input data from and output data to files and terminals, are a necessity in imperative programs. However, in an interactive environment, like that provided by an ML system, one can often do without these commands [Wiks87]. When programming 'in the large' or working with large data sets, however, the more traditional mode of working is preferred where data is stored in and retrieved from files.

The notion of I/O usually conjures up an image of state, side-effects and sequencing because traditional implementations of I/O in imperative languages are usually state-based, side-effecting and sequential. Even in functional languages some implementations like those of ML [HMT88] and ALFL [Huda84], use I/O 'functions' which have or can lead to side-effects and hence violate referential transparency. However, since referential transparency is one of the most important generally accepted advantage of functional languages over imperative languages, purely functional I/O must be achieved, so as to preserve the semantic elegance of these languages. Such functional I/O will be required to be as universal, flexible and efficient as I/O in imperative languages.

There are two main styles of incorporating purely functional I/O in modern functional languages. Some I/O solutions are based on the notion of *streams* and others use some form of *environments* (these are also called *side-effecting* I/O systems [Gord93]). The name *stream* was coined by Landin [Land65] and since then stream-based I/O systems have been developed which essentially transform an input stream into an output stream. A stream is basically a lazy list of data objects and this approach has been proposed in two flavours; token stream styles like in Miranda[1], Haskell and the FUDGETS system [Turn90, HPW92, CaHa93] and continuation style [Thom90, Perr88].

Environment based approaches to I/O use functions which directly manipulate a special object, the environment, that represents the state of the world [AcHa95]. Implementations based on this approach include the monadic I/O of Peyton Jones and Wadler [PeWa93] and others [BWW90, AcHa95].

We adopt the lazy stream model for I/O in our implementation. This chapter is concerned with the discussion of the organisation and implementation of this I/O model in our compiler.

6.1.1 I/O operations supported

The stream model of implementing purely functional I/O is based on the view that a program communicates with the outside world by sending request messages to the operating system and consuming a stream of response messages from the operating system. Accordingly, a Haskell program has type `Dialog` which is defined in terms of the `Request` and `Response` types defined below.

[1]Miranda is a trade mark of Research Software Limited.

```
type Dialog = [Response] → [Request]
data Request  =
          -- file system requests:
                    | ReadFile String
                    | WriteFile String String
                    | AppendFile String String
          -- channel system requests:
                    | ReadChan String
                    | AppendChan String String
          -- environment requests:
                    | GetArgs
                    | GetProgName
data Response  =
                    | Success
                    | Str       String
                    | StrList   [String]
                    | Failure String
```

Intuitively, `[Response]` is an ordered list of *responses* and `[Request]` is an ordered list of *requests*; the nth response is the operating system's reply to the nth request [HPW92]. This ordered nature of requests and responses make I/O operations sequential and hard to parallelise. We highlight how file I/O affects our experimental results in Chapter 7.

As in standard Haskell, our implementation handles most of the requests dealing with the file and channel systems of the underlying operating system. The corresponding requests on binary files are, however, not supported as the compiler does not make use of them itself. Similarly for the environment requests we implement all those that we use in the code of our compiler. Note that all the remaining requests that we do not currently support have no hindrance on our main task of parallelisation and they can be added when needed.

As for the operating system responses defined by the type `Response`, we only omit the response on binary values although we simplify the form of the error response that we generate.

6.1.2 Implementing the operations

We maintain two I/O state structures which are used to hold the relevant information that is threaded during program execution as requests are issued and corresponding responses are generated. The two state structures are called `DialogState` and `RequestState`. We now describe the need for and the use of each of these structures in turn.

The `DialogState` is used to maintain the lazy lists for requests and responses in a program. Elements of these lists must be built and maintained for the entire life-time of the program's execution in order to achieve the correct operational behaviour of the program.

When a request is issued, the pending requests to be issued are saved in the `DialogState` state until the corresponding response is generated for the current request and the lazy response list updated. The processes of issuing requests and that of generating their corresponding responses are interleaved continuously (since the ith request may depend on the first $i - 1$ responses) until all the requests specified by the **main** combinator are issued and responded to.

The `DialogState` has the following shape:

```
typedef struct {
    Global  busyResp;
    Global  restRequests;
} DialogState;
```

The first member of this structure, `busyResp`, represents the lazy response list to be incrementally built. It is initially a BUSY suspension with an empty waiting list. `restRequests` represents the lazy request list which is first evaluated to WHNF when the code for the **main** combinator is executed. The evaluation of the individual requests is triggered by `receiveRequests` (see Section 5.6) in order to build `DialogState`. The lifetime of `DialogState` is the execution of the program.

```
typedef struct {
    int counter;
    char name[256];
    Global stringSusp;
    Global restChars;
} RequestState;
```

`RequestState` is maintained to hold the data associated with the process of issuing and responding to a particular request. For instance, to handle the `AppendFile` request, we may first want to check whether the target file exists before starting to evaluate the string to be appended. `RequestState` is structured to cater for all possible requests and has the form shown above.

The `counter` member is an index used when building the array of characters `name` that stores a file or channel name in the request. `stringSusp` stores the lazy string to be written or appended (and it is empty in case of `ReadFile` and `ReadChan`). `restChars` contains the remainder of the string which is gradually evaluated and written to the designated media.

A new `RequestState` is created whenever a request is to be handled and this state is reclaimed immediately after the corresponding response is built—this is akin to the allocation and deallocation of table frames when generating code for supercombinators described in Section 5.5.4.

Notice that all the (`String`) arguments to our requests are represented as lazy lists of characters. Since responses are also constructors, the runtime system builds responses in such a way that their `String` components are represented as 'evaluated' lists of characters. For example, suppose a file 'test' contains the two-character string `"Hi"`, then the response `Str "Hi"` to the request `ReadFile "test"` will have the shape depicted in Figure 6.1. Our implementation of the stream of

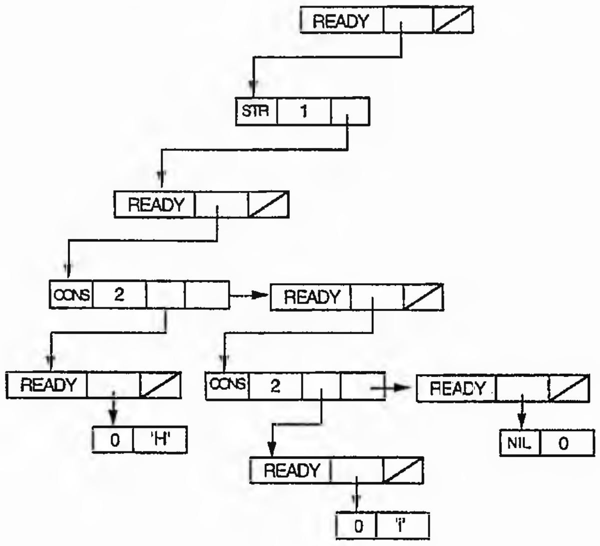

Figure 6.1: Structure of the response (`Str "Hi"`)

messages issued by a program and the corresponding stream of messages emitted by the operating system is achieved by modelling these operations in our message-passing framework. Program requests are implemented as messages, each associated with a handler which enforces the ordering on the program's requests and associated responses. That is, each message handler ensures the 'termination' of the process of handling the request in question, in a continuation passing style, before the next request can be handled. When the computation involved with the handling of a request fails to complete, because of an error, the error is reported immediately and the execution terminated without consuming any pending requests.

6.2 Comparing structured objects

As mentioned in Section 1.2 we do not support type classes, the facility in standard Haskell used for a systematic implementation of overloading. The purpose of this section is to provide a brief description of our implementation of some operations on aggregate values of user-defined types. Structured values have a long-winded representation in the heap (built from constructors as in Example 6.1 and Figure 6.1) and implementing operations on them can be tedious and delicate.

Haskell like other high-level functional languages encourages the programmer to define new types whose values could be subject to the same operations as the values of built-in types. One such operation, for instance, is an ordering relation over the constructors of a type. More mature compilers for the full Haskell language automatically derive a wealth of these operations for user-defined types whose definitions involve the **deriving** clause from the Haskell syntax. Although we do not support type classes, the mechanism used by the full Haskell compilers to implement the derived operations, our implementation supports some of these operations in a less systematic fashion. For example we implement comparison on (aggregate) data values as follows.

Recall that all data values in our implementation are boxed. That is, each data value is represented using a cell containing at least two fields with similar concrete layout to that described by Peyton Jones in [Peyt87]. To compare two values we start by evaluating them and inspecting their layout tags to find whether they are both atomic, in which case the values are compared immediately. If, however, the values are aggregates, their corresponding family tags and arities are compared, component-wise. If a decision cannot be reached based on the previous two tests, the two structures are traversed to evaluate and compare their corresponding components from left to right and in a depth-first manner. The result of the comparison is returned immediately a pair of subcomponents is encountered and on which the test criterion fails. Notice that the strong static typing of Haskell ensures that only compatible values (i.e., those of the same type) are compared. We illustrate the comparison operation using the program of Example 6.1:

Example 6.1

data *Tree a* = *Leaf a* | *Node a* (*Tree a*) (*Tree a*)

tree1 = *Node* [2..7] (*Leaf* [])(*Leaf* [8,9])
tree2 = *Node* [2..7] (*Leaf* [8])(*Leaf* [9])

Comparing `tree1` and `tree2` for equality, for instance, returns the answer `False` immediately the corresponding substructures `[]` and `[8]` are compared and without evaluating the last subcomponent pair of the top level nodes in `tree1` and `tree2`.

Notice that our alternative implementation achieves the same effect (for the operations we implement) as that provided by other implementations of standard comparison techniques based on type classes as found, for instance, in the GHC and HBC compilers. In those implementations as well as in ours, constructors of a type are ordered according to their appearance in the defining type. For instance, in the type `Tree a` above, `Leaf < Node`.

6.3 Scheduling

6.3.1 Sequential scheduling

As mentioned in Section 5.2, communication in our message passing system as in Dally's J-machine [DaCa89], for example, is completely data-driven in the sense that messages start handler threads. Handler threads are implemented as parameterless functions which access data from the associated message using global variables.

The multi-threaded code we generate is such that each parameterless function, which represents a handler thread, returns the code pointer, `ep`, of the next handler thread to which it would like to jump, rather than calling the function directly. When a thread is started it runs to completion without pre-emption and a thread wishing to terminate returns a NULL code pointer.

Our sequential scheduler is based on the tiny "interpreter" described by Peyton Jones in [Peyt92]. The interpreter, which is implemented as

```
while (TRUE) { ep = (*ep) ();}
```

nicely handles block-structured code of the form described above and we only needed to flesh it with code to enforce breaking of the interpreter loop to reflect thread termination. We achieve this by embedding the one-line interpreter inside another loop as in:

```
while (nextMessage(&fp, &ep, &msgData)) {
    frame = GET_LOCAL(fp);
    while ((ep = (*ep) ()) != NULL) {};
}
```

There can be several messages within a single handler thread which are stored in a *message (or context) store* before the thread terminates causing the messages within the thread to be handled. `nextMessage` checks the context store for the availability of messages and its three arguments are, respectively, the frame pointer which determines the context within which the handler thread ep is executed using `msgData` to access the data required to execute ep. Recall that the frame pointer, fp, consists of a processor identifier and a local address on the processor. GET_LOCAL is used to extract the local address from fp.

Notice that ep is the thread to execute next. So when a thread terminates, i.e., when the last returned ep is NULL, the outer loop will attempt to 'receive' a new message. If unsuccessful, i.e., if no more threads are available, then the program as a whole has terminated.

6.3.2 Parallel scheduling

The source annotations given by the user merely indicate the user's intention of where parallel evaluation is desired and not how the annotated expressions are mapped onto processing agents. To execute the multi-threaded code on multiple processors, the expressed parallelism must be distributed appropriately.

As mentioned in Section 5.2, frame pointers and suspension pointers are global to the machine in our design, as in the GUM implementation [THM+96]. In order to achieve the desired parallel behaviour, each of these global pointers must contain a processor identifier (PID) component which is automatically passed down (to child suspensions) and which will be used to determine the target processor for a message.

We have implemented two variations of task distribution schemes which compute PIDs and make them explicit in the code for function calls. Notice that since suspensions inherit the PIDs of their parents and since function calls are the units of task distribution, the form starting function calls, sendMessage, is the only place where we need to make PIDs explicit.

The first version of our task distributor randomly determines the PID on which to channel a message, selected from a fixed collection of PIDs, and it is based on the algorithms described by Sedgewick in [Sedg90]. The second is deterministic and offloads work to the processors of the machine fairly, in a round-robin manner.

Our experiments (see Chapter 8) using these offloaders reveal that they lead to a similar load sharing property for applications in which the parallel tasks have relatively the same grain size. Even in such applications, however, the use of the deterministic distributor usually leads to a slightly better equilibrium in work sharing amongst the processors of the machine. This is due to the fact that the random selection of PIDs is more likely is disturb the balance of work sharing—especially when the machine load is low—than in the deterministic case.

For parallel applications where the granularity of the parallel tasks is substantially different (which is usually the case for most non-trivial applications), the random task distributor is still, generally, more likely to yield inferior performance, especially in multiprocessors where the cost of allocation to one processor is more expensive than to another.

The next crucial issue is, what will be the best or optimal scheduling policy to use so that we can achieve the best performance—minimal completion time—from our parallelised program? Unfortunately, it has been shown that this is an intractable problem, in the sense that it is NP-Complete [Gare79], except if the application's structure is extremely simple, for instance when the dependencies between the tasks constitute a tree and each task takes the same amount of time [Hofm94].

We have experimented with two main scheduling policies: one at a thread level and the other at function-invocation or *quantum* level. The thread-based scheduler ensures that within each machine cycle[2], a single thread is removed (if any) from the context store of each virtual processor and executed. In the function-based scheduler, a quantum of threads are executed from each processor within a machine cycle. A quantum value in our experiments is taken to be the average number of threads per call in an application.

If the amount of work within the tasks of an application differ significantly, as highlighted above, the use of the thread-based scheduler leads to a badly skewed load sharing equilibrium. This led us to experiment with quantum schedulers by changing the quantum sizes as reported in our experiments of chapter eight.

6.4 A quasi-parallel simulator

As we have discussed in Section 6.3.1, the scheduler for our sequential compiler repeatedly removes and handles a message from a message store until the program terminates (i.e., until all messages in the message store have been handled.)

A natural way of building a simulator for a parallel machine is to extend the above representation to allow for multiple message stores each associated with a virtual processor. We use small integers, starting from zero, to represent the PIDs described in our parallel scheduler above. Since each global pointer is composed of a processor number paired with another integer specifying a local context, the processor number can be used to determine the right message store for each message.

In addition to the context store, each processor being simulated consists of three registers which hold information about its space usage, the number of function calls it has made and whether it is currently running or blocked (i.e., whether the task the processor was executing became blocked). The current PID, which is maintained as a global pointer, is used to determine the processor against

[2]A machine cycle in our experiments is a unit of time a fraction of which the machine dedicates its resources to execute a number of threads from each processor being simulated.

which the current allocation is costed. A processor is considered running provided its context store is non-empty and its PID is not held in the waiting list of some BUSY suspension (i.e., it is not currently blocked on communication), otherwise it is blocked or idle. Notice that some form of throttling mechanism will need to be adopted (see Section 3.2.6) to guard the potential overflow of context stores for applications with high amount of parallelism.

We have developed a quasi-parallel simulator, called Mizani, whose operation we now describe. As the program execution proceeds, Mizani takes a census of the computation activities of the constituent processors of the machine. As it visits each processor, it records the computation status of the processor (whether blocked on communication or busy running some task) as well as the number of tasks awaiting execution inside the processor's context store.

When all the processors are visited the accumulated number of running, runnable and blocked tasks, at this census time, are written to a statistics file. The memory in use by each processor is also recorded. The census is taken typically after every one hundred machine cycles.

The data in the statistics file is used to plot parallelism profiles as shown in Chapter 8. The PostScript graph generator[3] is based on Perl scripts which form part of the GranSim simulator [HLP95, Loid96] and which we adapt to conform to our data formats. The heap-usage data is not of much interest at the moment, as graphical representations are concerned, because it increases linearly with time due to our lacking a proper garbage collector.

The main simplifying assumption in our simulator, like in [Desc89, Roe91, RuWa95] for instance, is that communication has zero costs. It is inevitable in our message-passing system that when communication is properly costed the performance reported in our experiments in Chapter 8 will be affected. We strongly believe that the effect of costing communication on our figures will not be adverse provided quantum level scheduling is used together with efficient support for fast context switching.

6.5 Summary

In this chapter we have highlighted the fact that the I/O resources manipulated during I/O operations are, in the real world, globally accessible and manipulating them is, in essence, making assignments. This implies that the I/O resources cannot be used in the same direct, unrestricted way as they are manipulated in imperative languages. This is why functional languages are often viewed as less powerful than their imperative cousins with regards to I/O [AcHa95]. Nevertheless,

[3]Thanks to Kevin Hammond and H-W Loidl for pointing this out to me at a crucial time after several unsatisfactory attempts to generate graphical profiles using *gnuplot* program on UNIX machines.

there are two broad styles proposed for providing purely functional I/O in functional languages, which we have briefly reviewed in this chapter. We described in some detail our implementation of the model based on the notion of lazy streams. We have also described our alternative implementation of some comparison operations on values of user-defined algebraic types to compensate for our lack of support for standard Haskell type classes. Another aspect of the runtime system, scheduling of the decomposed program, is also covered here. Finally the chapter outlined the quasi-parallel simulator that we employed to evaluate our compiler.

The major research contributions, as this chapter is concerned, are two-fold. Firstly, we have designed and implemented a parallel scheduler (see analyses based on it in Section 8.3.7) which can schedule work at different levels of granularity—both at thread and at quantum levels. Furthermore, the quantum variant of the scheduler has been used to perform some kind of *ad hoc* 'granularity analysis' as a way of maximising locality and processor utilisation (Chapter 8). The other main contribution in this chapter is the development of an idealised simulator for the compiler.

Chapter 7

Parallelising the front end

As described in Chapter 1 and elsewhere, Naira is both a parallel compiler as well as a parallelising compiler. That is, we use semi-explicit parallelism and strictness annotations to parallelise the compiler itself and to parallelise input programs to the compiler.

The essence of parallelising the compiler front-end is to improve the efficiency with which Naira compiles programs (i.e., produce intermediate representations for the programs as described in Chapter 4). Annotations in the input programs on the other hand are used to generate parallel code the execution of which is simulated to test the effect of our annotations as well as to test our runtime system design parameters. This chapter is concerned with the issue of making the compiler execute in parallel and in the next chapter we consider the other parallelisation issue.

The front end of the compiler consists of 18 source modules with about 5K lines of Haskell code. The parallelisation process proceeds piecemeal, as expected, so that the effect of each parallelisation step can be analysed. In order to have a reasonable spread of input programs to test our parallelisation, we take the constituent compiler modules as our input since they implement different algorithms with varied computational costs.

There are two simulators used in our experiments to evaluate the compiler. The first simulator, *Mizani*, described in Section 6.4, is the one we wrote as part of this research work. The second is the *GrAnSim simulator* [HLP95, Loid96], which is available as part of the GHC compiler bundle.

As mentioned in Chapter 5 Naira is not supported by a garbage collector. For this reason it does not compile itself which means we cannot use it and its associated simulator to effectively evaluate the parallelisation of the front end. We therefore make use of the more robust GrAnSim simulator which can be tuned to simulate the execution of parallel Haskell programs on a variety of architectures. Naira and Mizani, on the other hand, are used extensively in the next chapter in the evaluation of the runtime system and the generated parallel code.

The rest of the chapter is organised as follows. In the next section we give a motivation for parallelising software while outlining our basic tools for parallelisation. Section 7.2 highlights the specific GrAnSim set-up used for our experiments. In Section 7.3 we report on the parallelisation of the compiler's top level pipeline. Sections 7.4 to 7.7 present results of parallelising four main phases of the compiler. In Section 7.8 we report our experimental figures for the overall parallelisation. Section 7.9 considers further parallelisation after gaining more experience parallelising the compiler. The performance of Naira on real hardware is presented in Section 7.10 and we summarise the chapter in Section 7.11

7.1 Parallelisation issues

Why do we need to parallelise software at all and what are the requirements for parallelising large software? This is a very important question and, in our opinion, a piece of software (small and large) is parallelised with the main aim of decreasing either its execution time or its resource usage or both. We set out to parallelise our compiler in this chapter with the purpose of reducing both its execution time and its resource usage.

Parallelising large-scale non-strict functional programs which have distinct stages of execution, like compilers, can be very hard. This is because parallelism in such applications may be highly irregular which makes understanding and controlling their dynamic behaviour hard. Furthermore, the distinct execution stages may not all be successfully parallelisable while this can be a precondition for good speedup.

Our parallelisation process proceeds top-down, following the methodology outlined by Trinder *et al* in [THLP98], starting from the top level pipeline and then parallelising the successive components of the program. We concentrated on parallelising four main phases of the compiler — the pattern matcher, lambda lifter, type checker and the intermediate language optimiser. We parallelised these phases in a data-oriented fashion by attaching parallelism annotations to the complex intermediate data structures used between these phases. We experimented with two common data structures — lists and binary trees — which were used to hold the intermediate parse trees.

A notable aspect that hinders the effective exploitation of parallelism is data dependency. We make use of the unique name server of Section 4.2.1 to generate new variable names at relevant points in the computation thereby breaking or minimising data dependencies and therefore exposing more parallelism.

To generate these unique names, we are faced with a range of options for choosing a name server as a few different kinds were described by Augustsson *et al* in [ARS94] and Hankin in [Peyt87]. Sequential name servers were described which can be extremely efficient except that they may either be 'impure' or that they can lead to both loss of laziness and parallelism. The parallel name servers proposed by the above authors include simple but inefficient ones and a more efficient one but part of which needed to be written in assembly code [ARS94]. Our early experiences with some name supply mechanisms suggest that a name server similar to that of Hancock in [Peyt87] is acceptable.

7.2 Experimental set-up

As mentioned in the previous section, GrAnSim has a wealth of runtime system options and tools which can be used to simulate a wide range of different parallel machine architectures. One such set-up is *GrAnSim-Light*, an idealised machine with zero-cost communication and an infinite number of processors, which the authors recommend for use in the early stages of parallelising large software. We experimented with GrAnSim-Light to familiarise ourselves with the working of GrAnSim and to get an idea of how much parallelism we can generate from our compiler.

Since the parallel machine underlying the design of Naira is that of loosely connected distributed memory message passing computers, we use a GrAnSim set-up specific for these machines so as to have a fairly accurate simulation of the compiler's behaviour. Such a GrAnSim set-up as given by Loidl in [Loid96] is:

```
./nairaOnGrAnSim <input> +RTS -bP -bp32 -bl2000 -bG -by2 -b-M
```

`nairaOnGrAnSim` is the compiled program for the front end of our compiler and `<input>` is a place-holder for an input module name. The **+RTS** means that the options which follow are to be passed to the runtime system. The first three options respectively mean generate a full GrAnSim profile on a 32-node machine with a communication latency of 2,000 cycles. The last three options mean enable bulk fetching with asynchronous communication while turning off thread migration.

The full profile enables the generation of activity profiles so that every major event in the system can be visualised, see [Loid96] for further details. A bulk fetching scheme is used to pack multiple thunks (closures) into a message packet so as to make communication in these high latency machines cost-effective. An alternative scheme, incremental (or 'lazy') fetching, in which only the immediately needed thunk is sent, is aimed for low latency systems and does not fit high latency machines because of the enormity of messages in the latter [LoHa96]. Note that packing multiple threads into one packet amounts to eager work distribution, realising the active message offloading of Naira.

Of the four *rescheduling schemes* described in [LoHa96], which decide what to do when a request is sent to another processor, we experimented with the two 'local' schemes (specified using the runtime options `-by1` and `-by2`) which involve task switching to execute a runnable thread or to turn a local spark into a thread, if no runnable thread is available locally. We did not experiment with the other pair of 'global' rescheduling schemes which involve additional remote communication to get new work after context switching. We recorded no significant difference between using either of the local rescheduling schemes. Because task migration is expensive in distributed memory machines, it is often not supported at all [Loid96] and that is why it is turned off in the above set-up.

Our experimental results reported in this chapter as well as in chapter eight are presented both in tabular and graphical forms. At each step in the experiment the tabular information shows the average parallelism and speedup obtained for each of the eighteen input modules of the compiler. The graphical information gives sample parallelism profiles for some input modules showing the number of tasks that were running, runnable and blocked during the program's execution life-span. The average parallelism and speedup figures are calculated using the following relations:

$$\begin{aligned} average\ parallelism &= \frac{\sum_{i=1}^{n} T_i}{T_{par}} \\ speedup &= \frac{T_{seq}}{T_{par}} \end{aligned}$$

$$where$$

$$\begin{aligned} n &= total\ number\ of\ paralleltasks \\ T_i &= time\ for\ executing\ task\ i\ (including\ overheads) \\ T_{seq} &= sequential\ execution\ time \\ T_{par} &= parallel\ execution\ time \end{aligned}$$

Notice that the calculated speedup is actually relative speedup since T_{seq} is based on the parallel compilation of the program (absolute speedup results if the sequential execution is based on the (optimised) sequential compilation of the program). The relationship between speedup and average parallelism is that they should be equal when the speedup is ideal[1].

7.3 Parallelising the top-level pipeline

As mentioned in Section 7.1, we follow the top-down methodology of parallelising large software as proposed by Trinder *et al.* [THLP98]. In this section we concentrate on the parallelisation

[1]That is, *average parallelism = ideal speedup* when all the parallelism exploited was actually needed (i.e., conservative with no superflous speculative work done) and there was no overhead due to communication and computation.

of the top level pipeline of our compiler before considering the parallelisation of the successive components. The overall top level pipeline of the compiler is depicted in Figure 7.1.

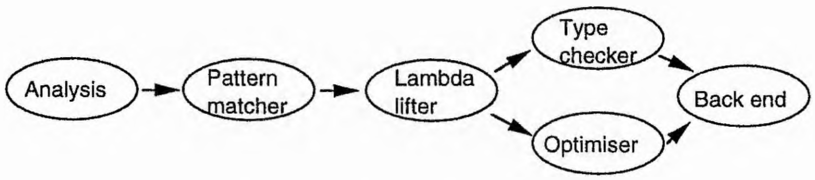

Figure 7.1: The Structure of Naira's Top-Level Pipeline.

The analysis phase consists of the lexical analyser and the parser. The next four phases respectively implement the pattern matching compiler, the lambda lifter, the type checker and the intermediate language optimiser. The splitting in the diagram after the lambda lifter indicates that the type checker and the optimiser can potentially proceed in parallel. The detailed organisation and implementation of these phases are described in chapter four. The back end is described in chapter five.

Each of these phases, starting from the analysis phase, produces a complex intermediate data structure which is input by the next phase, transformed and output for input in the next phase and so on. We parallelise this pipeline in a data-oriented fashion by defining evaluation strategies on the intermediate data structures produced by the phases. This facilitates the top-down parallelisation since the strategies define which parts of the data structures should be evaluated in parallel and without looking into the algorithms that produced the data structures. The algorithms producing the data structures are themselves parallelised in the following sections.

Notice that the choice of the 'best' combination of strategies to use in order to determine what parts of the data structures to evaluate can be quite delicate because of laziness in Haskell. This is because we have to ensure that we do not introduce too many speculative computations (created using annotations) which will result in the creation of superflous data structures. Although speculation may increase parallelism, it can adversely affect the overall performance since the response time of the program may be increased due to additional overhead.

The function, `analyseModule`, defined in Figure 7.2 implements the top level pipeline. It is called after the necessary symbol tables are built and when the compile-time analyses are ready to begin. It takes six arguments which it passes down to the functions implementing the individual transformations.

```
analyseModule fileName modName imports exports symbTabs defs  =
                        result `using` strat
 where
  (stPM,stTE,stOpt)    = symbTabs
  (dats,syns,funs)     = defs
  (aInfo,tInfo,impVals) = imports
  pMatchedDefs = mkDefs fileName stPM funs
  liftedDefs   = lLift fileName stPM pMatchedDefs
  typeList     = tcModule fileName stTE exports tInfo syns liftedDefs
  intermCode   = optimiseParseTree fileName exports stOpt aInfo liftedDefs
  result       = showModule modName impVals dats exports (intermCode,typeList)
  strat res    = parForceList funs         `par`
                 parForceList pMatchedDefs `par`
                 parForceList liftedDefs    `par`
                 parForceList typeList      `par`
                 parForceList intermCode    `par`
                 ()
parForceList = parList rnf
```

Figure 7.2: The Top-Level Compiler Function: `analyseModule`

The first two arguments are the name of the file containing the module being analysed and the module identifier. The next two arguments `imports` and `exports` contain the required information for the analyses from the imported and exported entities, respectively. The last two arguments are, respectively, the structure which holds the initial symbol table information (which is extended when the module is parsed) and the value definitions in the module on which to perform the different analyses.

The function `strat` defines the strategic code used in the parallelisation, giving a clear separation between algorithmic and parallelisation code. It sparks five parallel tasks using `par` (`c 'par' r` creates a task to evaluate `c`, then continues executing `r`), one for each of the pipeline phases shown in Figure 7.1. The `parList` strategy applies its first argument (the `rnf` strategy in this case, which reduces its argument to normal form) to each element of its second (list) argument in parallel thereby creating new tasks to reduce each element of the list to normal form in parallel. Therefore, `parForceList` forces the parallel evaluation to *normal form* of elements of a list. Trinder and others [THLP98] have given more detailed explanations of the strategies used in this chapter.

A disadvantage of using strategies in this form (consisting of the `using` combinator) is that every intermediate structure must be named. To avoid this, the researchers on GpH defined two binary combinators, (\$|) and (\$||), for sequential and parallel function application, respectively. The second argument in each case is a strategy to be applied, before or in parallel with the function application respectively.

Using these combinators, the code for `analyseModule` can be written more concisely, but perhaps less intuitively, as shown in Figure 7.3. Notice that the textual separation of algorithmic code and strategic code (specifying dynamic behaviour) is preserved.

```
analyseModule fileName modName imports exports symbTabs defs  =
    showModule modName impVals dats exports              $||
                                        parPair parForceList parForceList $
    fork (optimiseParseTree fileName exports stOpt  aInfo,
         tcModule fileName stTE exports tInfo  syns) $|| parForceList $
    lLift fileName stPM                                $|| parForceList $
    mkDefs fileName stPM                               $|| parForceList $ funs
  where (stPM,stTE,stOpt)   = symbTabs
        (dats,syns,funs)    = defs
        (aInfo,tInfo,impVals) = imports
fork (f, g) inp = (f inp, g inp)
($||) :: (a → b) → Strategy a → a → b
f $|| s = \ x → f x `sparking` s x
```

Figure 7.3: `analyseModule` rewritten using Pipeline Strategies

Our experiments revealed that as well as being less concise, the original version of `analyseModule` is also less efficient than the second version. For our 18 sample input modules, we found that the second version was up to 20% more efficient than the first. We note, however, that there were a couple of cases where the version using (\$||) was inferior.

With the parallelisation code added to the second argument of (\$||) in Figure 7.3, we measured parallelism ranging from 1.2 to 4.0 and speedups from 1.23 to 3.95 using the compiler's source modules as input. Table 7.1 summarises our experimental results for this parallelisation stage.

Input Module	*Before parallelisation*		*After parallelisation*	
	Average parallelism	**Speedup**	**Average parallelism**	**Speedup**
MyPrelude	1.0	1.00	3.6	3.46
DataTypes	1.0	1.00	4.0	3.95
Tables	1.0	1.00	3.5	3.42
PrintUtils	1.0	1.00	1.4	1.41
Printer	1.0	1.00	2.1	2.03
LexUtils	1.0	1.00	2.7	2.61
Lexer	1.0	1.00	1.5	1.46
SyntaxUtils	1.0	1.00	3.2	3.15
Syntax	1.0	1.00	1.3	1.24
MatchUtils	1.0	1.00	3.3	3.22
Matcher	1.0	1.00	2.6	2.52
LambdaUtils	1.0	1.00	2.0	1.91
LambdaLift	1.0	1.00	2.4	2.31
TCheckUtils	1.0	1.00	3.0	3.31
TChecker	1.0	1.00	1.9	1.82
OptimiseUtils	1.0	1.00	1.5	1.46
Optimiser	1.0	1.00	1.2	1.23
Main	1.0	1.00	1.6	1.61

Table 7.1: Top-Level Pipeline: Parallelism and Speedup.

As mentioned in Section 7.2, we also generate (in addition to the tabular information) graphical representations called *parallelism profiles* for our experimental data. These profiles show the number of parallel tasks and their execution status throughout the program's runtime. The vertical axes on these profiles record the number of parallel tasks and the horizontal axes record the simulated execution time. Our profiles show three different execution status—running, runnable and blocked—for the parallel tasks in the program over time as depicted by three different shades in the profiles (see, for example, Figures 7.4 and 7.5).

The parallelism profiles we obtained for the eighteen input modules divide into two categories as we now describe. Each profile in one of the two groups has very similar shape with those in the same category and Figures 7.4 and 7.5 describe representative profiles from these two groups.

Recall that as we explained earlier, symbol tables have to be built from type and synonym

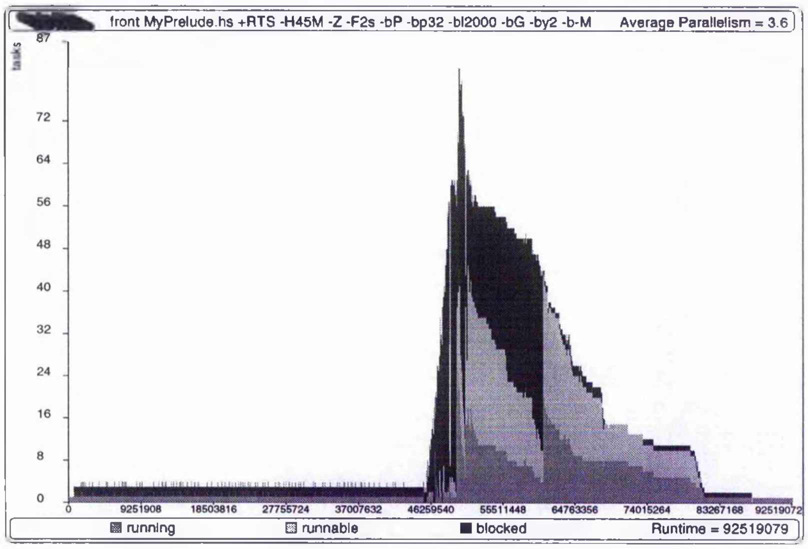

Figure 7.4: Top-Level Pipeline Compiling **MyPrelude**.

definitions (those imported as well as those defined locally) and the static information of all imported value bindings before any of the major transformations can begin. Building these symbol tables (see Section 4.2.1) can require a lot of file I/O especially if the module imports many other modules since information about imported entities has to be read from the interface files of the imported modules.

Notice the initial segments of the GrAnSim activity profiles of Figures 7.4 and 7.5. The first profile is for our standard prelude, **MyPrelude**, which does not import any other module and the second profile (i.e., Figure 7.5) is for a module which imports a few other modules in addition. Although interpreting activity profiles such as these is very difficult without the aid of specialised tools [HHLT97], we speculate, armed with experience and knowledge of our source code, that the initial sequential segments in these profiles are due to the I/O overheads.

In Figure 7.4, therefore, there is comparatively less amount of I/O (about half as much) at the start of the computation and so the five threads executing the first five phases attempted to start executing immediately. These threads blocked immediately because no initial parse tree on which they will operate has been produced yet. This blocking is depicted by the black-shaded portion in the initial segment of the profile in Figure 7.4.

In Figure 7.5 on the other hand, the initial long sequential segment, which occupies about 80% of the runtime, probably represents the I/O thread which reads information about imported

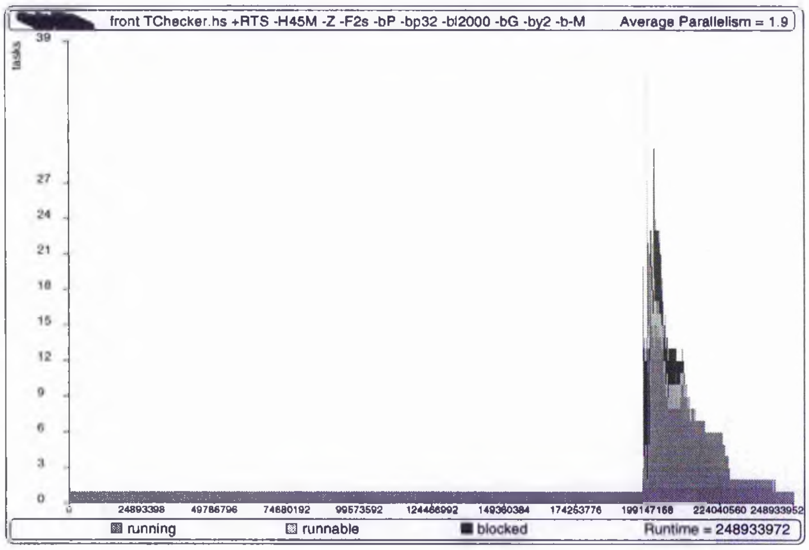

Figure 7.5: Top-Level Pipeline Compiling **TChecker**.

entities from interface files. The last portions of both the profiles of Figures 7.4 and 7.5 are similar. They both depict a raggedly-declining parallelism signifying that vast amount of parallelism is available immediately after the sequential I/O thread completed and which reduces as the results are combined in the final stages. The last sequential segment in both cases is again due to the I/O task which now writes the result of the module's analysis into two files: an intermediate language file and an interface file.

We can see from the parallelism profile of Figure 7.4 that only about half the tasks that can execute are actually running. We can also see that the number of running tasks is less than thirty two, the number of processors we are simulating. This means that not all of the virtual processors are fully utilised throughout the execution life-time of this program. This lack of processor utilisation is caused by the fact that the threads that were dormant are on busy or blocked processors and that these threads were not allowed to migrate to idle processors.

One alternative way of ensuring full processor utilisation is to run the simulations with migration enabled or on a GrAnSim-Light set-up which allows task migration and cost-free communication. Table 7.2 summarises our experimental measurements for both of these possibilities. The corresponding parallelism profile obtained when compiling **MyPrelude** using GrAnSim-Light is shown in Figure 7.6 (compare with Figure 7.4).

Comparing the values in the second and third columns of this table with the last two columns

Input	*Migration enabled*		*GrAnSim-Light set-up*	
Module	Average parallelism	Speedup	Average parallelism	Speedup
MyPrelude	4.0	3.83	4.2	4.00
DataTypes	4.0	3.97	4.2	4.10
Tables	3.5	3.42	3.5	3.46
PrintUtils	1.4	1.41	1.4	1.42
Printer	2.1	2.03	2.1	2.04
LexUtils	2.7	2.61	2.7	2.62
Lexer	1.5	1.46	1.5	1.46
SyntaxUtils	3.2	3.15	3.2	3.17
Syntax	1.3	1.24	1.3	1.24
MatchUtils	3.3	3.22	3.3	3.25
Matcher	2.6	2.51	2.6	2.54
LambdaUtils	2.0	1.91	2.0	1.93
LambdaLift	2.4	2.31	2.4	2.32
TCheckUtils	3.4	3.35	3.5	3.42
TChecker	1.9	1.82	1.9	1.83
OptimiseUtils	1.5	1.46	1.5	1.46
Optimiser	1.2	1.23	1.2	1.23
Main	1.6	1.61	1.7	1.61

Table 7.2: Top-Level Pipeline: Migration enabled and GrAnSim-Light.

of Table 7.1, we see that although the average parallelism increased only for two modules (i.e., in `MyPrelude` and `TCheckUtils`, while the values remained unchanged for the others) when migration is enabled, the runtime improved in ten out of the eighteen input modules. These runtime increases were only high enough to affect the speedup figures for `MyPredule`, `DataTypes` and `TCheckUtils` and with a slight decrease in `Matcher` due to some runtime overheads. Comparing the values in the second and third columns with those in the last two columns of Table 7.2, however, reveals that the experimental results with GrAnSim Light have, as expected, better runtime for all our inputs and that while the average parallelism increased in only four modules, the speedup increased in thirteen cases.

Compared with the experimental figures recorded by Trinder and others in their parallelisation a data-intensive transportation problem (called *Accidents Blackspots*) and Lolita (a natural language

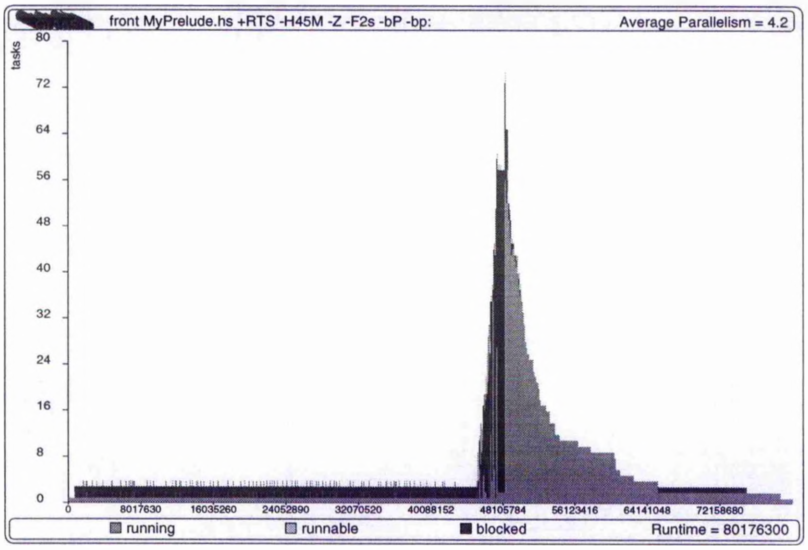

Figure 7.6: Top-Level Pipeline Compiling **MyPrelude** using GrAnSim-Light.

parser), our measured average parallelism figures are quite encouraging. Trinder reported average parallelism of 1.2 on accident black-spots [THL+96] and about 2.5 on Lolita [THLP98], at the same top level parallelisation stage.

We now turn our focus on the parallelisation of the subalgorithms of the compiler, starting from the pattern matching compiler. In each of the following subsections we analyse the parallelism extracted within a single phase in isolation and in Section 7.8 we analyse the overall parallelisation of the subalgorithms put together.

7.4 Parallelising the pattern matcher

The pattern matching compiler transforms function definitions made using equational patterns into equivalent ones involving case expressions with simple variable patterns. This transformation is primarily performed for efficiency purposes and without which patterns may be multiply evaluated.

When the definitions within a module are parsed, the pattern matching transformation can be applied to these definitions in a data-parallel fashion since there is no top level data dependencies to inhibit this. The pattern matching compiler is implemented using the function `mkDefs` defined in Figure 7.7.

The three arguments to `mkDefs` are the file name, the pattern matching symbol table (see

```
mkDefs :: Name → (AssocTree [Char] (Int, [String])) → [Def] → [Def]
mkDefs fileName env [] = []
mkDefs fileName env l =
    mkAppend $|| parallelPair parForceList $
    fork2 (checkAdjDefs fileName env,
           mkDefs fileName env) $|| parallelPair parForceList $
    partition (sameDef (head l)) $|| sparkList $ l
fork2 (f, g)(x,y) = (f x, g y)
sparkList = parList rwhnf
```

Figure 7.7: The Pattern Matching Compiler: `mkDefs`

Section 4.2.1) and the list of definitions output from the parser. Note that a single function definition may be characterised by more than one element (i.e., equation) in the third argument of `mkDefs`. `checkAdjDefs` is used to ensure that adjacent bindings (defining a single function) are correctly grouped together and any errorneous redefinitions of identifiers reported immediately.

The transformation is parallelised at different levels in order to determine a reasonable granularity level in the parallelisation. This definition of `mkDefs` introduces coarse-grained parallelism which allows the analysis of each binding to proceed in parallel with that of the others. `parallelPair` simultaneously applies `parForceList` to both components of a 2-tuple. `sparkList` causes the parallel evaluation to *weak head normal form* of elements of a list.

We attempted to extract parallelism further at three places within the top level functions. The first step makes the pattern matching compilation of the local definitions within a function proceed in parallel with that of the top level function. The second step parallelises heavily used auxiliary functions which perform some reasonable amount of computation. The third step changes the data structure used to represent the pattern matching symbol table.

In step one, we make use of a parallel name server to avoid data dependencies and to provide the opportunity that the transformation can proceed at different levels (determined by the depth of the nesting of local definitions in a function). The function `local_mkDefs` in Figure 7.8 is used to implement this.

Notice that this code differs from that of `mkDefs` in the additional name-supply argument used to facilitate parallelisation.

The second and third columns of Table 7.3 summarise the runtime and average parallelism obtained at the initial stage of the parallelisation and the last two columns represent those figures for

Input	*Initial parallelisation*		*Step one*	
Module	**Average parallelism**	**Speedup**	**Average parallelism**	**Speedup**
MyPrelude	1.1	1.07	1.1	1.12
DataTypes	1.1	1.08	1.1	1.11
Tables	1.1	1.08	1.1	1.10
PrintUtils	1.1	1.03	1.1	1.05
Printer	1.1	1.06	1.1	1.07
LexUtils	1.3	1.18	1.3	1.22
Lexer	1.1	1.09	1.1	1.12
SyntaxUtils	1.1	1.04	1.1	1.05
Syntax	1.1	1.05	1.1	1.06
MatchUtils	1.1	1.07	1.1	1.09
Matcher	1.2	1.12	1.2	1.15
LambdaUtils	1.2	1.12	1.2	1.18
LambdaLift	1.2	1.12	1.2	1.15
TCheckUtils	1.1	1.06	1.1	1.07
TChecker	1.2	1.18	1.2	1.22
OptmiseUtils	1.0	1.00	1.1	1.05
Optimiser	1.1	0.79	1.1	1.06
Main	1.1	1.10	1.1	1.12

Table 7.3: Parallelising the pattern matcher: Initial step and step 1.

step one. Comparing these two sets of results we see that introducing parallelism in `local_mkDefs` leads to some improvement in both average parallelism and speedup. While the average parallelism increased only for `OptimiseUtils` (and remaining the same for other input modules), the speedup increased for all input modules with a difference of upto about 30% for `Optimiser`. This parallelisation step therefore gives rise to a clean performance improvement over the preceding step with zero overhead for all our input modules.

Recall from Section 4.2.3 that we perform some tidying up operations on the patterns of a function before calling the pattern compiler. This is one place that we attempted to extract parallelism. Another highly used auxiliary function is that which repeatedly replaces pattern variables in the source program with compiler-generated ones during the transformation. Parallelising the functions performing these tasks gave the results in columns two and three of Table 7.4.

```
local_mkDefs :: Name → MyInt → (AssocTree [Char] (Int, [String])) → [Def] → [Def]
local_mkDefs fileNm ns env [] = []
local_mkDefs fileNm ns env l =
   mkAppend                                  $|| parallelPair parForceList $
   fork2 (checkAdjSymbDefs1 fileNm ns0 env,
          local_mkDefs fileNm ns1 env) $|| parallelPair sparkList $
   partition (sameDef (head l))          $|| sparkList $ l
  where (ns0,ns1) = split ns
```

Figure 7.8: Pattern Matcher for Local Definitions:`localmkDefs`.

Compared with the performance figures of step one (in the last two columns of Table 7.3), we record a slightly better overall performance although we incur some overhead in some cases. Although the average parallelism and speedup figures remained unchanged for most input modules, the runtimes of these modules show that we obtained better runtimes in step two.

In step three we used a list data structure, with type `[(String,(Int,[String]))]`, instead of the binary tree of association `AssocTree String (Int,[String])`, to represent the pattern matching symbol table. With this modification our experimental figures show that there is a slight improvement over the previous step. As in the previous analysis and although the average parallelism and speedup values change only in a few cases, we can see that the runtimes in step three are slightly better.

Notice from the above parallelisation steps that although we always improved the average parallelism figures, we also paid some runtime overhead for some input modules. This is because the finer the granularity of the parallel tasks that we generate the more expensive their management especially in our distributed memory set-up.

As explained earlier, we can run our simulations under GrAnSim-Light so as to find some of the sources of overhead and to find the maximum parallelism that we can extract. The last two columns of Table 7.4 summarise our experimental results on GrAnSim-Light. These results are only slightly better than those obtained using standard GrAnSim, with a maximum speedup improvement of 2%, and that this signifies that we paid very little overhead in this parallelisation.

The activity profiles generated during this parallelisation steps do not contain much useful visual information because the other phases of the compiler are running sequentially and there is not much parallelism in this phase either. For example, Figure 7.9 shows a typical parallelism

Input Module	*Step two*		*Step three*		*GrAnSim-Light*	
	Avg. paral.	**Speedup**	**Avg. paral.**	**Speedup**	**Avg. paral.**	**Speedup**
MyPrelude	1.1	1.12	1.2	1.14	1.2	1.16
DataTypes	1.1	1.11	1.1	1.11	1.2	1.13
Tables	1.1	1.10	1.1	1.10	1.1	1.10
PrintUtils	1.1	1.05	1.1	1.05	1.1	1.05
Printer	1.1	1.07	1.1	1.07	1.1	1.07
LexUtils	1.3	1.21	1.3	1.22	1.3	1.22
Lexer	1.1	1.13	1.1	1.12	1.1	1.13
SyntaxUtils	1.1	1.05	1.1	1.05	1.1	1.05
Syntax	1.1	1.07	1.1	1.06	1.1	1.08
MatchUtils	1.1	1.09	1.1	1.09	1.1	1.09
Matcher	1.2	1.15	1.2	1.15	1.2	1.15
LambdaUtils	1.2	1.17	1.2	1.17	1.2	1.18
LambdaLift	1.2	1.14	1.2	1.14	1.2	1.15
TCheckUtils	1.1	1.07	1.1	1.07	1.1	1.07
TChecker	1.2	1.22	1.2	1.22	1.3	1.23
OptmiseUtils	1.1	1.05	1.1	1.05	1.1	1.05
Optimiser	1.1	1.06	1.1	1.06	1.1	1.06
Main	1.0	1.01	1.1	1.11	1.1	1.12

Table 7.4: Parallelising the pattern matcher: Steps 2, 3 & GrAnSim-Light.

profile obtained during the parallelisation of the pattern matcher. Notice from this figure that immediately after the sequential file I/O there was a sharp thin rise of running parallel tasks created in the pattern matching phase. This is followed by a band of about three-thread high running for about 20% of the overall execution time. The remaining long sequential tail represents the execution span of the other phases which run sequentially.

7.5 Parallelising the lambda lifter

In this section we present our parallelisation of the lambda lifter. As introduced in Section 4.2.4, the lambda lifter consists of a scope analyser, a renamer, a dependency analyser and the final

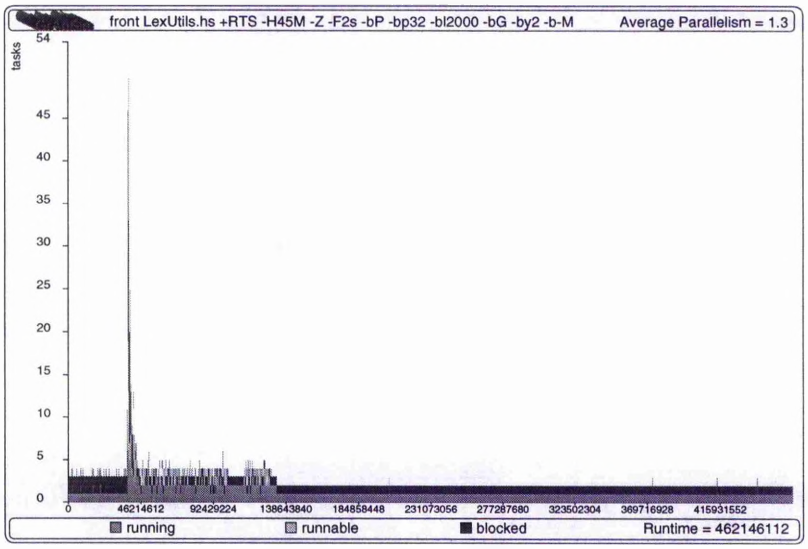

Figure 7.9: Pattern Matching Stage Compiling **LexUtils**.

'lifting' operation. Recall that this transformation is relevant only in top level recursive bindings and in function definitions which have local definitions that may take some of the arguments of the functions in the enclosing scopes as free variables.

We implement the lambda lifting transformation using the function `lLift` as defined in Figure 7.10. Notice the use of the identity function `id` which ensures that `sparkList` is applied to the result of `lifter` before being returned. `parallelTriple` simultaneously applies `sparkList` to a 3-tuple of lists.

```
lLift :: String → AssocTree String (Int, [String]) → [Def] → [Def]
lLift fileName stPM defs =
  id $|| sparkList $
  lifter.triplet1 $|| parallelTriple sparkList $
  scopeAnalysis fileName stPM [] [] initNS 1 $|| sparkList $ defs
 where triplet1 (x, y, z) = x
```

Figure 7.10: The Lambda Lifter: **lLift**

There is a two-level pipeline formed by the two main functions, `scopeAnalysis` and `lifter`, which perform the meat of the computation in the lambda lifting process. The first part of the pipeline performs scope analysis and incorporates the renamer and the dependency analyser. The second part is the `lifter` which computes the transitive closure of the free variables of a function and then performs substitutions as exemplified in Section 4.2.4.

The experimental results obtained when running our compiler with the above code as the only source of parallelism are summarised in the second and third columns of Table 7.5. This parallelisation leads to a modest performance improvement: the average parallelism has improved to 1.1 for eight input modules and the runtime is improved in all the input modules over the non-parallelised version of the program. These runtime improvements lead to speedup increases of upto 6% in the initial parallelisation.

Input Module	*Initial step*		*Final step*		*Using GRIP set-up*	
	Avg. paral.	**Speedup**	**Avg. paral.**	**Speedup**	**Avg. paral.**	**Speedup**
MyPrelude	1.1	1.01	1.1	1.01	1.1	1.02
DataTypes	1.0	1.01	1.0	1.01	1.0	1.02
Tables	1.0	1.01	1.0	1.01	1.0	1.01
PrintUtils	1.0	1.00	1.0	1.00	1.0	1.00
Printer	1.0	1.01	1.0	1.01	1.0	1.01
LexUtils	1.1	1.03	1.1	1.03	1.1	1.04
Lexer	1.1	1.03	1.1	1.03	1.1	1.03
SyntaxUtils	1.0	1.01	1.0	1.01	1.0	1.01
Syntax	1.0	1.02	1.0	1.02	1.0	1.02
MatchUtils	1.0	1.01	1.0	1.01	1.0	1.01
Matcher	1.1	1.04	1.1	1.04	1.1	1.04
LambdaUtils	1.1	1.04	1.1	1.04	1.1	1.05
LambdaLift	1.1	1.05	1.1	1.05	1.1	1.05
TCheckUtils	1.0	1.01	1.0	1.01	1.0	1.01
TChecker	1.1	1.06	1.1	1.06	1.1	1.06
OptimiseUtils	1.0	1.01	1.0	1.01	1.0	1.01
Optimiser	1.0	1.02	1.0	1.02	1.0	1.02
Main	1.1	1.04	1.1	1.04	1.1	1.04

Table 7.5: Parallelising the lambda lifter: parallelism and speedup.

The prospects of generating further useful parallelism inside function bodies lies in the feasibility of sparking two parallel subtasks to perform the renaming and dependency analysis. Another place that we experimented with the worthiness of extracting parallelism in is inside the `lifter` function.

The renamer simply associates an identifier with a small integer and since only locally defined identifiers are renamed (the parser would have reported any name clashes amongst top level identifiers), there is little amount of work performed by the renamer and dedicating a parallel task for it resulted in figures almost the same as the ones obtained in the initial parallelisation step.

The dependency analyser on the other hand performs relatively more amount of computations. However, the dependency analyser is based on graph algorithms (essentially collecting strongly-connected components) and thus there is not much parallelism that can be extracted from it either. The combined effect of parallelising the scope analyser is shown in the fourth and fifth columns of Table 7.5. Similar to our results in the pattern matching parallelisation, although the average parallelism and speedup values remain unchanged from the previous step, the runtime is improved in all but four cases.

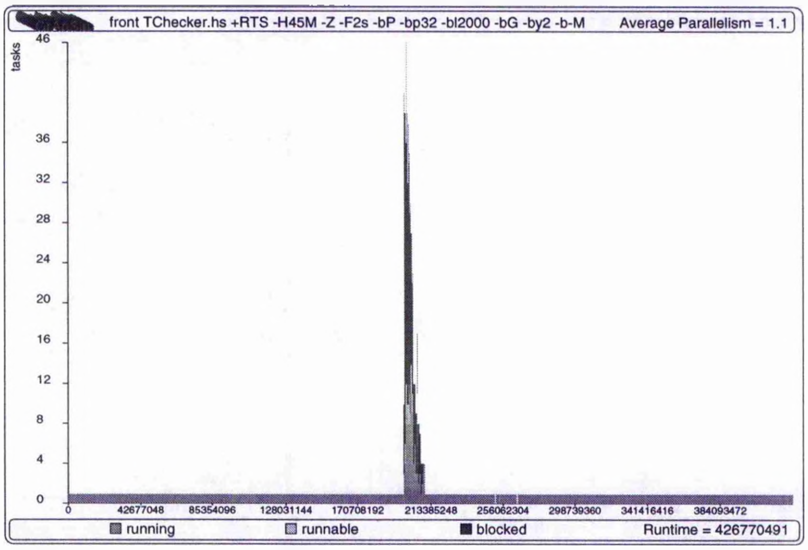

Figure 7.11: Lambda Lifting Stage Compiling `TChecker`.

In the second pass, the core lambda lifter collects free variables, forms and solves equations (reminiscent of Johnsson, in [John87]) to determine the complete set of free variables of each function. This turns out to be not computationally expensive, going by our input programs, because the number of definitions within a local binding in this pass is small since the definitions have been separated into minimal dependency groups to aid type checking [Peyt87]. This means that creating parallel subtasks within such local definitions gives rise to excessively fine-grained units of work which do not support cost-effective communication.

Running these experiments using the GrAnSim set-up for GRIP, a closely-coupled distributed memory machine, we obtained the results shown in the last two columns of Table 7.5. Comparing these with those (in columns 2 and 3) of Table 7.4 reveals that the low-latency of GRIP makes it perform better on the input programs, although the speedup changed only in four cases.

Figure 7.11 shows a typical parallelism profile obtained during the parallelisation of the lambda lifter. The amount of parallel activity depicted in this profile is quite scanty signifying the fraction of parallelism that can be generated in this phase compared to the overall parallelism.

Compared with the pattern matching compiler, there is not as much parallelism in the lambda lifter. This fact appeals to intuition because real programs can be imagined to contain a large collection of function definitions over structured data. Furthermore, the pattern matching compilation process involves more work than the lambda lifting transformation.

7.6 Parallelising the type checker

The type checker is the most expensive phase of the compiler, both in terms of space usage and running time. This is largely because of the fact that the type checking process includes subalgorithms like unification and some operations on large data structures which themselves require significant amount of computations.

```
tcModule fileName env exports typeList syns defs  =
                              (typeList ++ topDefsTypes) `using` parForceList
   where
     (ns0,ns1)      = split initNS
     tIds           = map getDefId defs
     auxEnv         = mkTypeVars tIds ns0
     topDefsTypes = tcTopDefs fileName env auxEnv exports initSubs ns1 syns defs
```

Figure 7.12: The Type Checker: `tcModule`

Space and time profiling information, using both sequential and parallel profilers [SaPe95, HHLT97], revealed that the type checker is, in fact, more expensive than the other phases of the compiler put together. The parallelisation of Naira therefore depends significantly on how much useful parallelism can be extracted from the type checker.

The function `tcModule`, defined in Figure 7.12, is used to implement the type inferencing algorithm for a collection of definitions in a module. The first three arguments to this function are the file name, the type environment and a list of exported values whose static information is to be written into the interface file. The last three arguments contain the type information of imported values, a list of type synonyms used for type resolutions and the list of definitions in the module whose types are being inferred.

Input	*Intial parallelisation*		*Step one*	
Module	**Average parallelism**	**Speedup**	**Average parallelism**	**Speedup**
MyPrelude	3.3	3.26	3.7	3.32
DataTypes	2.2	2.21	2.5	2.21
Tables	2.4	2.33	2.6	2.33
PrintUtils	1.4	1.40	1.4	1.40
Printer	2.0	2.01	2.4	2.02
LexUtils	2.3	2.22	2.8	2.23
Lexer	1.5	1.43	2.0	1.44
SyntaxUtils	3.2	3.15	3.6	3.15
Syntax	1.2	1.23	1.5	1.23
MatchUtils	3.3	3.23	3.8	3.24
Matcher	2.1	2.07	3.3	2.10
LambdaUtils	1.7	1.63	2.2	1.63
LambdaLift	2.2	2.16	4.3	2.07
TCheckUtils	2.2	2.14	2.5	2.14
TChecker	1.8	1.71	2.3	1.70
OptimiseUtils	1.2	1.15	1.4	1.15
Optimiser	1.2	1.23	2.1	1.23
Main	1.6	1.61	2.7	1.60

Table 7.6: Parallelising the type-checker: Intial step and step 1.

As in standard polymorphic type checking algorithms, `tcModule` initially associates each bound name with an assumed type creating an auxiliary environment, `auxEnv`. These assumed types usually become specialised as unifications and substitutions are performed. Inferred types are also checked against user-declared type signatures to ensure that the declared types are not more general than or incomparable to the deduced principal types (in accordance with Haskell).

The type checker can be parallelised using a parallel name server and by distributing substitutions to avoid sequentialising the inference process. For intance, to type check two quantities d_1 and d_2, we analyse them simultaneously in the current type environment, each returning a type and a substitution record. If a variable v common to both d_1 and d_2 is assigned (possibly different) types t_1 and t_2 from these two independent operations, t_1 and t_2 will be unified in the presence of the resulting substitutions and the unified type associated with v.

```
tcLocalDefs fileName env subs ns syns []                            = ([],[],subs)
tcLocalDefs fileName env subs ns syns (VDef(IdPat id) args e:defs) =
    res `using` parTriple rnf parForceList rwhnf
  where
    (ns0,ns1)          = split ns
    (infTy,subs1)      = typeCheck fileName id (mkLam args e) env subs syns ns1
    (idsL,tysL,subs4)  = tcLocalDefs fileName env subs ns0 syns defs
    res                = (id:idsL, infTy:tysL,subs4)
```

Figure 7.13: Type Inference for Local Definitions: `tcLocalDefs`

Accordingly, our initial parallelisation step uses a simple name supply mechanism to relax data dependencies so that the type inference of top level definitions within a module can proceed in parallel. Columns two and three of Table 7.6 show our experimental figures after this initial parallelisation. These values indicate very good parallel behaviour on account of this parallelisation of the type checker. Comparing with the figures (in the last two columns) of Table 7.1, for parallelising the top level pipeline of Naira, we see that the values obtained at this stage are very close to (and for some input modules the same as) those in Table 7.1. The parallelism profiles obtained are also very similar, as the corresponding numeric values.

In line with our top-down parallelisation tradition, we now proceed to experiment with the introduction of parallelism inside the type inference of individual top level functions at decreasing levels of granularity. As in Section 7.4, we identify three main areas within which to exploit parallelism further. These are inside local definitions, on calls to frequently used functions and at other expression constructs.

In step one, we added strategic code to the function `tcLocalDefs` defined in Figure 7.13 so as to create additional parallel threads to infer the types of the locally defined identifiers.

The strategic code `res 'using' parTriple rnf parForceList rwhnf` ensures the creation of parallel tasks for `res`, the result of `tcLocalDefs`, as described in the previous paragraph.

This modification leads to a modest increase in parallel activity as can be seen by comparing the values in columns 2 and 3 with those in the last two columns of Table 7.6. The average parallelism has increased in all input modules (with an increase of up to 2.1 for `LambdaLift`) except in `PrintUtils` where it remained unchanged. The runtime also increased except in five input modules (i.e., `PrintUtils`, `LambdaLift`, `TChecker`, `Optimiser` and `Main`) where we incurred some overhead due to increased parallel activity. Note that this overhead is only significant in `LambdaLift`, `TChecker` and `Main` to affect speedup in those modules.

Notice also that while the average parallelism when compiling the `LambdaLift` module has almost doubled on account of the current parallelisation, there was actually a negative speedup, signifying that the parallelism exploited for this particular input was not useful. This is one instance that highlights the subtlety of parallelisation code; while it may improve performance on some inputs it can at the same time decrease performance for some other inputs! Figure 7.14 shows a sample profile resulting from this parallelisation step.

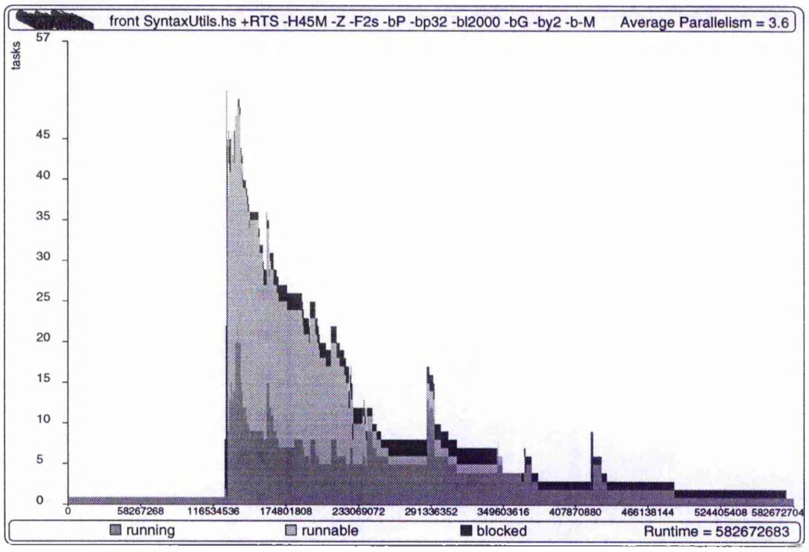

Figure 7.14: Type Checking Stage Compiling `SyntaxUtils`.

Notice from this figure that a better performance can be obtained by turning thread migration on since there are unemployed runnable threads. We shall consider this in the next steps.

Step two introduces parallelism at the unification subalgorithm. It is clear that type unification, which we implement using `mkUnify` defined in Figure 7.15, is one the most perverse operations during type inference and which can involve some reasonable amount of work.

```
mkUnify syns t1 t2 = (subs,theTy)
   where subs  = unify (OK []) (expandSynonyms syns t1)(expandSynonyms syns t2)
         theTy = mkTheType subs t1 t2
```

Figure 7.15: Type Unification:**mkUnify**.

The function **expandSynonyms** ensures that type synonyms within the types being unified are replaced before unification proceeds while **mkTheType** obtains the unified type on successful unification or generates an error message on unification failure.

We introduce parallelism here by sparking a child task to carry out the unification task when type checking a subtree while the parent task proceeds with the 'main' type checking subthread. We achieve this by adding the strategic code,

result 'sparking' sequentialPair rnf resOfUnify

to each call of **mkUnify** inside the type checker. **result** is the result of the inference step of which **resOfUnify**, the result returned by **mkUnify** is a part.

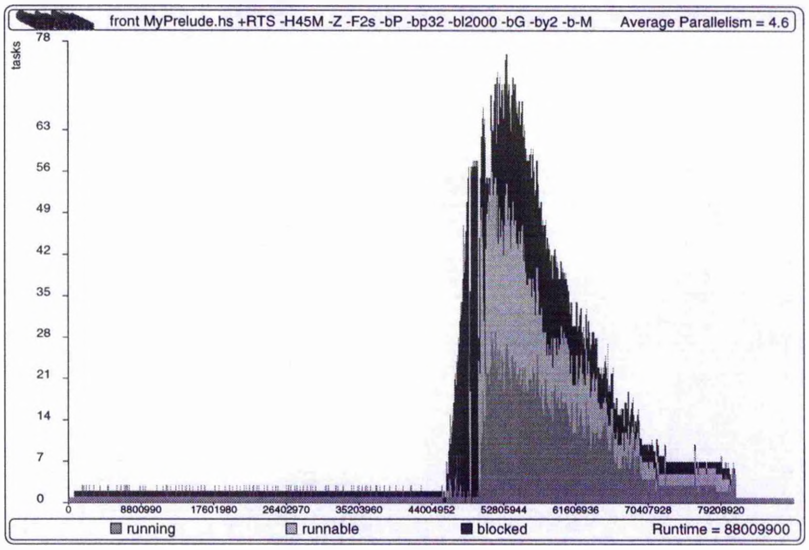

Figure 7.16: Type Checking Stage Compiling **MyPrelude**.

We found that better performance can be obtained by using **parallelPair rnf**, instead of the **sequentialPair rnf**, to make **unify** and **mkTheType** proceed in parallel.

Notice that 'factoring' the `parallelPair rnf` strategies from the type checker and using only one of them inside the definition of `mkUnify` (while using none of these strategies in the type checker) will degrade performance. This is because using the strategy inside `mkUnify` alone will amount to creating parallel tasks inside a substructure whose enclosing structure may only be lazily demanded. Using the strategies in both the type checker and `mkUnify` on the other hand can be less efficient since it will lead to duplicate applications of the same strategy on the same data structure. The best option, as per our experiments, is to apply strategies in as outer level as possible.

Input Module	*Step two*		*Step three*		*Migration enabled*	
	Avg. paral.	**Speedup**	**Avg. paral.**	**Speedup**	**Avg. paral.**	**Speedup**
MyPrelude	4.6	3.64	6.1	3.57	6.2	3.67
DataTypes	3.6	3.05	4.2	2.55	6.4	3.86
Tables	2.9	2.49	5.1	2.71	6.3	3.39
PrintUtils	1.5	1.41	1.9	1.23	2.2	1.43
Printer	2.5	2.02	3.1	1.95	3.3	2.04
LexUtils	4.1	2.58	7.3	2.40	8.3	2.60
Lexer	2.1	1.45	4.6	1.40	4.7	1.46
SyntaxUtils	3.7	3.17	4.3	3.11	4.4	3.16
Syntax	1.5	1.23	1.8	1.23	1.8	1.24
MatchUtils	3.5	2.89	6.0	3.17	6.1	3.24
Matcher	3.1	1.89	5.1	1.93	6.4	2.43
LambdaUtils	2.5	1.81	3.2	1.87	3.3	1.91
LambdaLift	4.3	1.97	6.8	2.06	7.6	2.30
TCheckUtils	2.9	2.33	4.4	2.58	5.8	3.42
TChecker	2.4	1.71	2.7	1.64	2.8	1.71
OptimiseUtils	1.5	1.21	3.2	1.41	3.3	1.46
Optimiser	2.0	0.79	4.5	0.75	4.7	0.79
Main	2.7	1.56	5.2	1.51	5.3	1.61

Table 7.7: Parallelising the type-checker: Steps 2, 3 & 4.

The values in the second and third columns of Table 7.7 summarise our experimental results for this step. Compared with the values obtained in the previous step (recorded in the last two columns of

Table 7.6), this parallelisation step leads to significant improvement in parallel performance. All the runtime and average parallelism figures in step two are better than those in step one except for the modules `MatchUtils`, `Matcher` and `Optimiser`. Also while the speedup increased in most of the input modules, going up by upto 74% for `DataTypes`, it also went down in five others, by upto 44% in `Optimiser`. Figure 7.16 shows the activity profile at this stage for the module `MyPrelude`, which corresponds to that of Figure 7.4 at the top level parallelisation stage.

For step three of parallelising the type checker, we create a child task to perform composition of substitutions in similar manner as we described for `mkUnify` above. As we indicated at the start of this section and by the fact that we do not continuously apply substitutions to the type environment after every inference step (see Section 4.2.1 for details of our representation of the type and substitution environments), substitution compositions are performed very often.

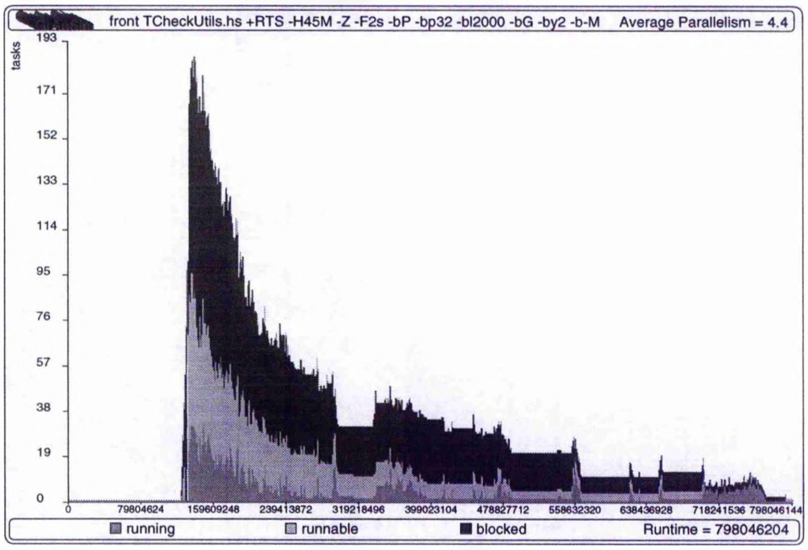

Figure 7.17: Type Checking Stage Compiling `TCheckUtils`.

We have also tried extracting parallelism further inside the parse trees for conditional, function application and case expressions. Our results are recorded in the fourth and fifth columns of Table 7.7. Comparing with the results of the previous step, we see (from the actual runtimes) that although the recorded average parallelism increased in all cases, most of the parallelism introduced in this step was not useful. This is because we incurred some runtime overheads in ten of the eighteen input modules while gaining speedup increases only in seven other modules.

Figure 7.17 shows the activity profile for the module `TCheckUtils` after the changes in this step. Notice from this figure that there is some scope for improving the parallel behaviour since there are some runnable threads that could not run because the processor on which they are is busy and that they were not allowed to migrate by the experimental set-up.

We rerun these experiments using our set-up for typical distributed memory machines having no shared memory but this time enabling thread migration. The values in the last two columns of Table 7.7 are obtained from this. Notice the increase in average parallelism of upto 2.2 although the performance goes down slightly for some inputs and that we pay some runtime penalty in some other cases. On the whole this version of the experiment with thread migration enabled gave better result than the previous case in which no thread migration was allowed. This shows that thread migration can be beneficial in loosely coupled multicomputers as it is in multiprocessors with shorter latencies.

7.7 Parallelising the optimiser

This section describes the parallelisation of the parse tree simplifier and optimiser. As mentioned in Section 4.3, the optimiser specialises general function applications (using arity information of combinators so that argument satisfaction checks are short-circuited at runtime for the specialised applications) and transforms case-expressions to a simplified form better suited for code generation. The optimiser is implemented using the function `optimiseParseTree` defined in Figure 7.18.

```
optimiseParseTree fileNm exptNames stOpt aInfo defs = defs1
  where
    (defs1,comb_arity_tree) = optimiseTopDefs fileNm exptNames stOpt cs_as1 defs
    cs_as1 = aInfo ++ comb_arity_tree
```

Figure 7.18: The Optimiser:`optimiseParseTree`.

The arguments to `optimiseParseTree` are: `fileNm`, the file containing the module under scrutiny, used mainly for error diagnostics purposes; `exptNames`, names of exported (value) bindings from this module, only whose static information is written into the interface file of the module to ensure information hinding; `stOpt`, a symbol table holding associations between constructor names and their corresponding family tags (see Section 4.2.1 for more details); `aInfo`, an association list holding the arity information of values imported from other modules and `defs`, the (parse tree of) values implemented by this module which are being optimised.

There is no data dependency between the analysis of one definition in the module and another once their arities are known. We can therefore simplify the parse trees of the definitions within a module in parallel with each other. The arity information of all functions (imported or locally defined) is needed in the optimisation of the parse tree for each function in the module. We therefore spawn a child task for the arity collection to proceed simultaneously with the optimisation. The second and third columns of Table 7.8 show the result of this initial parallelisation step.

Input	*Initial parallelisation*		*Final parallelisation*	
Module	**Average parallelism**	**Speedup**	**Average parallelism**	**Speedup**
MyPrelude	1.0	1.03	1.1	1.03
DataTypes	1.0	1.03	1.0	1.03
Tables	1.0	1.03	1.1	1.03
PrintUtils	1.0	1.01	1.0	1.01
Printer	1.0	1.02	1.0	1.02
LexUtils	1.1	1.06	1.1	1.05
Lexer	1.1	1.04	1.1	1.04
SyntaxUtils	1.0	1.01	1.0	1.01
Syntax	1.0	1.02	1.0	1.03
MatchUtils	1.0	1.02	1.0	1.03
Matcher	1.1	1.06	1.1	1.05
LambdaUtils	1.1	1.07	1.1	1.06
LambdaLift	1.1	1.07	1.1	1.06
TCheckUtils	1.0	1.02	1.0	1.02
TChecker	1.1	1.10	1.1	1.09
OptmiseUtils	1.0	1.02	1.0	1.02
Optimiser	1.0	0.77	1.0	0.77
Main	1.0	1.04	1.1	1.03

Table 7.8: Parallelising the optimiser: parallelism and speedup.

Other places in which we explored parallelism further include inside local definitions, arity collection of functions and the case expression optimisation. We recorded slight improvement in parallel behaviour for some inputs by creating a child task at each of these three places with performance degrading as parallelism is introduced at lower levels than these.

Useful as the parse tree optimisation transformation is, it is clear from the description in Section 4.3 that it is cheaply implementable. Table 7.8 shows the experimental results we obtained as a result of the parallelisation. This table shows that because this phase is not computationally intensive introducing parallelism inside the analysis of individual definitions does not achieve much improvements and can lead to some loss of performance. For our input programs, we can see that the speedup was better in more modules in the initial step than in the final much finer grained step. The activity profiles obtained here are very similar to those obtained in Section 7.5.

7.8 Combined parallelisation

In the five preceding sections we have presented the parallel behaviour of our compiler by first parallelising its top level pipeline and then parallelising four of its component phases each in isolation with the rest. The aim of this section is to turn on all the strategic code used to parallelise the individual phases of the compiler, rerun our experiments and observe its performance.

Table 7.9 shows the results obtained when all the parallelism annotations in the compiler phases are enabled. Compared with the results for the final parallelisation of the top level pipeline (in the last two columns of Table 7.1) we record a good increase in average parallelism for all modules (up to a factor of two) and modest speedup increases (up to 94% for `DataTypes`) for all but the input module `Optimiser` where we incurred some runtime costs.

Also comparing these results with those of the final parallelisation of the type checker (fourth and fifth columns in Table 7.7) we see a much better speedup for all the input programs. The average parallelism recorded for some modules in Table 7.7 (namely for the inputs `LexUtils`, `SyntaxUtils`, `Syntax`, `LambdaLift`, `OptimiseUtils`, `Optimiser` and `Main`) is much higher than those of the same input modules in Table 7.9 indicating that, in the light of the comparative speedup figures, some of the parallelism we generated during the parallelisation of the type checker was not useful.

Although the experimental results of Table 7.9 are relatively better, as in our comparisons in the preceding paragraphs, they fall short of realising the combined speedup we had hoped for—that obtained from the type checker and that obtained from the top level pipeline. This is probably hindered by the fact that the pieces of strategic code inside the phases interfered with each other and, consequently, affecting the overall performance.

Our experiences with adding parallelisation code on top of the ones inside the phases revealed that it is quite hard to understand and predict the performance of the compiler and that small changes in the parallelisation code can lead to significant changes in parallel behaviour for some inputs. Nevertheless, all the results we obtained as a result of varying the combinations of our parallelisation code at this level do not differ significantly from that of Table 7.9.

Input Module	*Combined parallelisation*	
	Average parallelism	**Speedup**
MyPrelude	7.2	3.88
DataTypes	7.5	4.20
Tables	6.8	3.48
PrintUtils	2.5	1.54
Printer	3.2	2.12
LexUtils	5.4	2.68
Lexer	4.1	1.55
SyntaxUtils	4.2	3.22
Syntax	1.6	1.26
MatchUtils	6.0	3.25
Matcher	6.2	2.58
LambdaUtils	3.5	2.28
LambdaLift	3.4	2.48
TCheckUtils	5.3	3.66
TChecker	2.8	2.08
OptimiseUtils	2.9	1.49
Optimiser	2.3	0.83
Main	3.0	1.69

Table 7.9: Combined parallelisation results summary.

Careful study of the parallelism profiles reveals that file I/O and parsing account for a significant part of the remaining sequential component to the computation (and therefore by Amdahl's law represents a major limitation on further parallelisation). Other areas we want to investigate further include the creation of symbol tables and the printing process that writes the intermediate language and the inferred type information into files. We explore these and other issues in the following section.

7.9 Further parallelisation

In the previous section we summarised results of our experiments after activating all annotations in all the compiler phases while modifying and experimenting with various different combinations of annotations. In this section we summarise results obtained after re-examining more closely the algorithms on which the individual phases were based.

Input module	**Average parallelism**			**Speedup: 8 processors**		
	Ideal	SMM	DMM	Ideal	SMM	DMM
MyPrelude	6.2	4.7	4.5	4.32	3.34	3.31
DataTypes	6.5	4.5	4.8	5.18	3.71	3.96
Tables	7.8	4.9	5.1	4.84	3.14	3.27
PrintUtils	2.7	2.5	2.4	2.74	2.65	2.50
Printer	4.2	3.8	3.8	4.25	3.96	3.90
LexUtils	10.9	6.3	5.8	4.95	3.47	4.45
Lexer	5.9	4.8	4.7	4.38	3.69	4.18
SyntaxUtils	8.4	5.0	4.7	8.13	5.81	5.53
Syntax	1.9	1.8	1.8	1.40	1.39	1.35
MatchUtils	5.2	4.0	3.5	5.30	4.20	3.61
Matcher	6.7	4.1	4.2	4.64	3.74	3.59
LambdaUtils	4.1	3.6	3.3	2.53	2.22	2.09
LambdaLift	6.7	4.5	4.5	6.11	4.03	4.39
TCheckUtils	7.3	4.9	5.6	6.47	4.68	5.32
TChecker	3.1	2.6	2.7	1.89	1.81	1.76
OptmiseUtils	3.1	3.0	2.8	3.68	3.61	3.33
Optimiser	3.5	3.4	3.4	4.79	4.79	4.70
Main	3.9	3.6	3.5	2.93	2.67	2.66

Table 7.10: Further parallelisation results summary: 8 processors.

This undertaking was motivated by a few observations, one of which was the fact that our experimentation with different evaluation strategies lead to no significant overall performance improvements. Another consideration is the need to broaden the scope of our compiler's performance by simulating it on shared memory architectures as well, taking full advantage of the tunability of GrAnSim.

In our search for culprits—those parts of the computation that thwart our parallelisation efforts—we evidently looked no farther than inside the type checker because it is more expensive than all the other parts of the compiler. Our main finding was that composition of substitutions, which is performed quite often in Naira, forms the main bottleneck in the parallel performance of the compiler. Interestingly, our research found that this is a notoriously famous problem [Hamm90]. We revised our implementation of this algorithm and fine-tuned our strategic code resulting in substantial performance improvements as summarised in Table 7.10.

The first important difference we point out between the experimental set-ups giving the results in Tables 7.9 and 7.10 is that a 32-node GrAnSim was used in the former (as in most other experiments reported earlier) while an 8-node one was used in the latter. Another difference is that task migration was always enabled for our experiments in this section, contrary to most other experiments in the preceding sections.

Comparing the results of Table 7.9 with the corresponding distributed memory results in Table 7.10 we see that the performance is now much better: Even though run on eight processors, we measured better speedups in Table 7.10 for thirteen input modules (with speedup increasing by a factor of two for the input modules `Lexer`, `OptimiseUtils`, and `Optimiser`).

The results obtained on the idealised machine (GrAnSim-Light) in Table 7.10 are, as expected, better than those obtained from the shared memory and the distributed memory machine emulations. Comparing the shared memory and the distributed memory results we see that the shared memory results are only negligibly better (with higher average parallelism in ten cases and higher speedups in twelve cases). Perhaps this is because the incremental fetching used in the shared memory set-up is less efficient than the bulk fetching used in the distributed memory set-up or that it reflects we made a good parallel decomposition of the program giving low communication overheads.

Further investigation in a bid to find the cause of this insignificant difference between the shared memory and the distributed memory results revealed that we have actually made judicious use of evaluation strategies and have successfully parallelised the compiler leading to cost-effective communication. We achieved this by experimenting with the four rescheduling schemes and the two fetching strategies of [LoHa96] without recording any significant performance variation in the simulated results, hence our conclusion.

7.10 Naira on a network of workstations

Our experimental results so far (presented in this chapter) have been based on the GrAnSim simulation of typical parallel machine architectures. The essence of this section is to report our experimental results of measuring Naira on a network of SUN workstations running under Solaris 2 (SunOs 5.5.1) operating system.

The experiments were carried out using GUM [THM+96], a portable parallel implementation of Haskell, which sits on top of PVM[2](Parallel Virtual Machine). A parallel Haskell program under GUM (Graph reduction for a Unified Machine model) implies multiple processes running on multiple processors running under a PVM framework. The salient features of GUM are that it does not support thread migration and threads are distributed lazily, although data distribution is performed somewhat eagerly[3]. GUM uses a fair thread scheduler whereby runnable threads are executed in following a round-robin policy and communication is achieved via message passing.

Before presenting our experimental results on real hardware, we provide our simulated results for a LAN-connected machines which have a much higher communication latency than typical distributed memory machines simulated results on which we have presented in the preceding sections. According to Hammond *et al* [HLP95, LoHa96], a latency of 5–10 cycles corresponds to a typical shared-memory machine, fast distributed-memory machines such as GRIP or the Meiko CS2 have latencies in the 100–500 cycle range, while typical distributed memory machines such as the IBM SP2 have latencies of 1,000–5,000 cycles. Fully distributed machines connected over an ethernet LAN would typically have latencies of 25,000–100,000 cycles.

Table 7.11 summarises our GrAnSim-based results simulating a loosely-coupled network of computers with varying communication latencies. Because the aim here is to simulate GUM, thread migration was disabled in the experiments on which this data is based.

In comparison to our earlier results in the preceding sections and, in similarity with Hammond's analysis of a Ray-tracing application on high latency (about 100,000 cycles) GrAnSim [HLP95], we record a parallel slowdown with different high-latency GrAnSim set-ups as summarised in Table 7.11. The mean speedup is not as good (lower by 23%) when the latency increased from 50K cycles to 85K cycles and is lower by 32% when the latency increased from 85K cycles to 120K cycles. The corresponding figures with thread migration enabled (not shown in the table), on the other hand, are 41% and 30% respectively.

We now introduce a summary of the results of our experiments on GUM. We have conducted the experiments under different network conditions and with varying numbers of processors. First, Table 7.12 shows the result obtained when different versions of Naira's code (compiled under different conditions) are executed. The runtime figures in this table are the wall-clock (real) timings taken to compile each input module. These times (measured in seconds) are averaged over different runs of the programs.

The second column of Table 7.12 records the runtime of each module for the sequential version of the compiler when compiled with the full optimising sequential compiler, GHC. The third column

[2]We used the latest version of pvm, version 3.3.11, obtainable from `http://netlib2.cs.utk.edu/pvm3/index.html`.

[3]By speculatively packing some 'nearby' reachable graph into the reply message for a closure request.

Input Module	*Varying Latencies for Distributed Machines*					
	50,000 cycles		*85,000 cycles*		*120,000 cycles*	
	Avg. paral.	**Speedup**	**Avg. paral.**	**Speedup**	**Avg. paral.**	**Speedup**
MyPrelude	3.2	2.50	2.5	2.13	1.7	1.41
DataTypes	2.0	1.64	2.8	2.39	1.7	1.63
Tables	3.8	2.80	2.7	1.78	2.6	1.85
PrintUtils	1.8	1.93	1.9	2.06	1.7	1.93
Printer	2.2	2.30	2.7	2.80	1.8	1.88
LexUtils	4.6	3.40	3.3	2.38	3.7	3.15
Lexer	2.6	2.88	2.2	2.84	2.2	2.78
SyntaxUtils	3.5	4.04	3.6	4.56	2.7	3.19
Syntax	2.9	1.81	2.3	1.39	2.2	1.34
MatchUtils	3.1	4.06	1.8	1.87	1.6	1.98
Matcher	3.1	3.27	2.8	2.77	2.0	1.50
LambdaUtils	2.6	1.61	2.3	1.57	1.7	1.09
LambdaLift	2.7	2.49	2.6	2.58	1.9	2.00
TCheckUtils	4.4	4.15	3.8	3.69	3.6	3.53
TChecker	2.2	1.36	1.8	1.12	2.0	1.31
OptmiseUtils	2.3	2.69	1.9	2.30	1.8	2.19
Optimiser	2.6	3.71	2.3	3.88	2.0	3.66
Main	2.5	1.89	2.7	2.05	2.1	2.02
MEAN	2.8	2.66	2.5	2.43	2.1	2.11

Table 7.11: Simulated results on 8-node loosely-coupled distributed memory machines with varying latencies.

gives the runtime of compiling each module when the sequential version of the compiler (Naira) is compiled for parallel execution and run under GUM on a single processor. The last column contains the efficiency figures based on the figures in columns two and three of the table.

An advantage of the tabular information in Table 7.12 is that we can isolate the overhead of parallelism incurred in GUM based on the sequential execution times. The parallel runtime system imposes a more-or-less fixed percentage overhead on every program regardless of its use of parallelism [THM$^+$96]. There are also overheads introduced by every spark site in the parallel

Input Module	**Sequential**	**Seq. for par. exec.**	**Efficiency**
MyPrelude	14.4	16.3	88%
DataTypes	14.6	17.0	86%
Tables	16.6	17.9	93%
PrintUtils	11.5	9.6	120%
Printer	16.2	17.1	95%
LexUtils	15.3	17.9	86%
Lexer	15.1	13.3	113%
SyntaxUtils	68.5	71.7	96%
Syntax	67.9	69.4	98%
MatchUtils	19.0	20.7	92%
Matcher	22.1	24.0	92%
LambdaUtils	13.1	13.4	98%
LambdaLift	30.9	30.0	103%
TCheckUtils	37.1	37.3	99%
TChecker	17.7	15.2	117%
OptmiseUtils	28.8	31.7	91%
Optimiser	39.7	47.6	83%
Main	29.2	32.8	89%
MEAN	26.5	27.9	95%

Table 7.12: Results summary of **Naira**'s runtimes on GUM.

program (due to the extraneous costs of closure creation and entry). The time behaviour of a program running under GUM is further complicated because of the use of PVM. Each PE task is typically a Unix process, and at the mercy of the Unix process scheduler. In some configurations, such as a network of workstations, there is also competing network traffic that affects overall performance [THM+96].

The third column of Table 7.12 shows that the overhead imposed by the runtime system and PVM on all the input modules can be quite high. This is probably exacerbated by the fact that the individual modules are not big enough to simulate 'real' inputs as can be seen from the sequential execution times. The figures in the last column of this table show that a slowdown on one processor of upto 30% can be expected due to parallelism overhead.

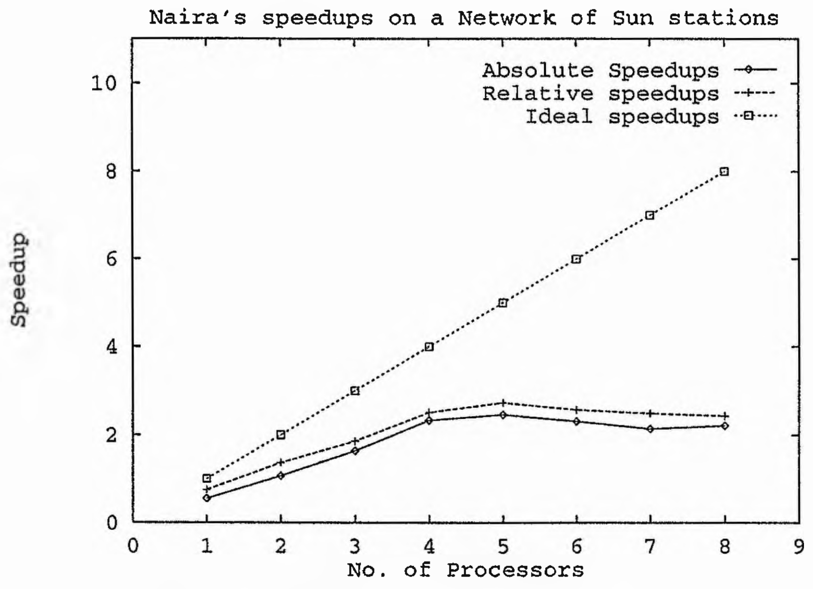

Figure 7.19: Speedup summary of Naira on GUM.

Figure 7.19 shows the speedups obtained on GUM (in relation to pure sequential compilation and execution) on different numbers of processors. The 'parallel machine' used here was a set of SUN workstations (Sun 4/65s), each with 32MByte of RAM, and connected to a common Ethernet segment. The input program used in these experiments is a bigger module created by combining the source code of three constituent modules of Naira—MyPrelude, DataTypes and Tables. The speedups shown in this figure are average speedups obtained over different runs and under different network and processor conditions. Several runs were made at different times in an attempt to ameliorate the effect of scheduling accidents to which the execution is highly susceptible since GUM does not support thread migration.

As stated earlier, the parallel code has some overheads that are not present in the sequential Glasgow Haskell implementation, such as the need to test every closure when it is entered in order to determine if it is already being evaluated by another parallel task. As a result of this, the parallel code on a single processor slows down by 25% to make the parallel system only 75% as efficient as the sequential implementation. We measured a wall-clock speedup of 2.46, and a relative speedup of 2.73 on a network of five workstations as depicted in the graphs of Figure 7.19. These results are in agreement with those obtained using the GrAnsim simulator which predicted a speedup of 3.01. Better speedups would be expected if the machine resources are dedicated solely to our experiments and if thread migration, which has been demonstrated (in this thesis and elsewhere [HaPe92, BuRa94]) to be necessary for good parallel performance, is implemented in GUM.

7.11 Summary

We have presented the parallelisation, performance measurements and analysis of the front end of our compiler in this chapter. We have achieved this by making use of the parallelisation methodology developed by researchers on Glasgow Parallel Haskell. Our performance measurements were carried out using GrAnSim, a fairly accurate state-of-the-art simulator developed at Glasgow and St Andrews Universities. Most of our experimental results were obtained using a GrAnSim set-up specific to distributed memory message passing architectures on which the compiler is aimed to run. Good speedups were also recorded for Naira as analysed in the preceding section.

Using a name server similar to that described by Hankin in [Peyt87], we were able to break data dependencies and expose some good deal of parallelism. We experimented with an ordered binary tree to represent some of our symbol tables and to store the collection of definitions within a module.

Level	**Mean average parallelsim**			**Mean speedup**		
	Ideal	SMM	DMM	Ideal	SMM	DMM
Top-level	2.5	2.3	2.4	2.42	2.24	2.26
Pattern matcher	1.3	1.3	1.2	1.12	1.11	1.10
Lambda lifter	1.2	1.2	1.2	1.11	1.11	1.11
Type checker	3.3	2.5	2.5	3.21	2.28	2.29
Optimiser	1.2	1.2	1.2	1.11	1.10	1.10
Overall	5.5	4.0	4.0	4.36	3.50	3.55

Table 7.13: Mean performance figures on an 8-node GrAnSim with latency of 2K cycles.

We found out that while `AssocTree` may provide faster accesses, there is a lot of overhead associated with building the ordered tree. For example, there can be a lot of processing required to rebuild an ordered tree when an association is deleted[4] from a tree of substitutions. Using a list in place of the association tree, `AssocTree`, leads a significant reduction in Naira's runtime overhead since all the (mainly) string comparisons needed to insert an association in the tree are avoided. Furthermore, we found it convenient to use a list data structure to hold the definitions within a module so that we can use the predefined evaluation strategies over lists that came with the GrAnSim simulator.

Our parallelisation process proceeded top-down, parallelising the top level pipeline of the com-

[4] An entry in an association tree of substitutions needs to be deleted when combining substitutions and when a variable is found to be associated with different types in the substitutions being composed.

piler phases before delving into the parallelisation of the individual algorithms in the constituent phases. The phases were themselves parallelised systematically, in stages, analysing the effect of each parallelisation step. Four phases of the compiler were parallelised separately so as to provide an informed assessment of the potential parallelism within each phase. The parallelisation code inside these phases was finally activated and the overall behaviour of the compiler analysed.

Our experimental results revealed that the most productive phase turned out to be the type checker. This is because it dominates the overall compilation process. Other phases are relatively cheap, and therefore give less overall improvement. Table 7.13 gives a concise summary of our experimental results on eight processors. The table shows the mean average parallelism and mean speedup obtained (at six different levels of the compiler) from an idealised, a shared memory and a distributed memory machine set-up. The overall performance is quite good with speedups ranging from 1.35 to 5.53 (average of 3.55 for our eighteen input modules) on GrAnSim emulating a distributed memory machine with eight processors.

Our experimental results on GUM (i.e., real machine hardware) in the last section strongly support the GrAnSim-based simulated results. Most importantly, these results are the best achieved so far for a large, irregular parallel Haskell program. The largest parallel Haskell program (Naira is the second) in the world, Lolita, developed by a team of able researchers, achieves an absolute speedup of 0.9 on 4 processors while Naira achieves an absolute speedup of 2.5 on 5 processors (Section 7.10)

Chapter 8

Evaluating parallelism in the compiled code

8.1 Introduction

In the previous chapter we have described and presented measurements on the parallelisation of the front-end of our compiler using the GrAnSim simulator. The purpose of this chapter is essentially to exercise and evaluate the parallelism in the compiled code obtained using our parallelism model. In other words, this chapter measures the effectiveness of our parallelism and strictness annotations, the behaviour of the multi-threaded, message-passing parallel code that we generate and the effectiveness and costs of our runtime design decisions.

Because the goal of a parallel implementation is to strike a balance between execution time and resource usage, by transforming a short-wide sequential execution profile into a tall-narrow parallel profile [RuWa95], we set out to measure, using our *Mizani* simulator described in Section 6.4, some of the issues affecting parallel performance. These include space usage, speed of execution (measured in machine cycles), the number of parallel function activations and the effect of different distribution and scheduling algorithms. We have experimented with two distribution algorithms: pseudo-random and round-robin (i.e., deterministic). Whenever an exportable task is encountered, while the parent task is executing on some processor i, the pseudo-random distribution algorithm (which provides a repeatable effect from one run to another) selects a processor j, ships this child task to j while execution of the parent task continues asynchronously on i. The deterministic scheme on the other hand will offload the child task to the next processor in the cycle (i.e., the processor identified by $((i + 1)\ mod\ n)$ where n is the number of processors in the machine). The astute reader would have realised that using these eager task distribution policies, overloading the machine processors with work, for some applications, is an inevitable possibility as we discussed further in the chapter. Naira does not currently possess a load-throttling mechanism (which is not required for the benchmarks presented here) but we propose implementing Ostheimer's load bounding scheme [Osth93], as further research. We have also measured performance results using both thread-based and function call-based unit of task scheduling.

In comparison to the more robust and highly tunable simulator–GrAnSim–used to parallelise the compiler in the previous chapter, Mizani like the idealised version of standard GrAnSim, GrAnSim-Light, assumes communication has zero cost. Compared with other parallel simulators [RuWa95, Desc89, Roe91] which, for the most part either deal with highly idealised parallel machines or are accurate simulations of real or proposed parallel machines [HLP95], Mizani does not make the simplifying assumption that (as in [RuWa95]) each supercombinator reduction takes an identical amount of time which can lead to a further overstatement of the degree of parallelism in a program.

The rest of this chapter is organised as follows. Section 8.2 presents the benchmark programs for which performance measurements are presented. Section 8.3 highlights the general format in which our results are presented. The next six subsections contain the experimental results for each benchmark in this format. Section 8.4 concludes the chapter with general comments on our experiences with the parallelisation of these benchmarks on Naira.

8.2 Benchmark programs

We have experimented with many small and medium-sized popular benchmark programs in exercising our compiler. For economy of space, we elect to report our experimental results on the following representative programs: *nfib*, *nqueens*, *tak*, *coins*, *matmul* and *soda*. These benchmarks are taken from previous research on parallel implementation of functional languages and each has been used in different parallel implementation projects to estimate performances.

The choice of these benchmarks is motivated by two main concerns. The first consideration is that we want a fair collection of programs capable of exercising various aspects of our compiler like function call (or stack operations) efficiency, the costs of arithmetic operations and data structures manipulation overheads. Secondly, since these benchmarks have been used to test other implementations, like [Mara91, KLB91, AuJo89b], we can use them to evaluate our compiler vis-a-vis the results of those implementations.

As is the case with many other parallel machine simulation projects, for instance [Desc89, Roe91, RuWa95], we restrict ourselves to the use of these small to medium-scale benchmark programs as an initial 'proving stage' for the performance of our implementation. Using the performances on these benchmarks as a basis, the implementation can then be extended to make it more robust so that we can analyse more complex applications as in the implementations of [THM+96, HCAA94].

We remark that each of these benchmark programs (see Sections 8.3.1 to 8.3.6) is either expressed using a divide-and-conquer algorithm (probably the best-known parallel programming paradigm [THLP98]) or has some element of divide-and-conquer when parallelised. Other parallel paradigms, like data-oriented parallelism, are also exploited in some of the benchmarks. Producer/consumer and pipeline parallel paradigms have been extensively exploited in the previous chapter where we dealt with large and complex data structures.

8.3 Analysis of the benchmarks

We now focus on the analysis of the runtime behaviour of each of our benchmark programs in turn. For each benchmark program we collect the runtime statistics of its sequential execution and of its parallel execution resulting from the best parallelised program we arrived at. We have run the experiments for each benchmark program on 2, 4, 8 and 16 processors. The processor identifier and the frame pointer (as explained in Section 5.2) are represented as a single machine word. Larger numbers of processors can be simulated by storing the processor identifier as a separate word. For the larger benchmarks the programs are parallelised in stages showing the effect of each parallelisation step. The statistics gathered are summarised in tabular and in graphical forms.

The tabular information contains the time taken for the program to run (measured in machine cycles), the total number of threads executed in this time, the average parallelism measured, the number of *ready-evaluated tasks*[1] and the total amount of heap space allocated during the execution of the program. The average parallelism and speedup entries in the table are calculated using the same relations as given in Section 7.2. Following Roe [Roe91] we measure efficiency using the following relation:

$$efficiency = \frac{speedup}{number\ of\ processors}$$

The parallelism profiles are plotted using data collected by taking the census of the heap after every 100 machine cycles. As mentioned in Section 8.1, we have experimented with two different

[1]Ready evaluated tasks are those that either hit a "black-hole" or find their graph in WHNF when they are activated. Roe in [Roe91] refers to ready-evaluated tasks as *useless tasks*. The number of these tasks indicate the amount of sharing in a program and can affect the heap residency of a program as explained in our analyses in this section.

distribution strategies and with two different units of work scheduling.

We analysed each of our test programs at two levels of granularity. At the first level, the machine assigns a small slice of its time within which each virtual processor executes exactly one thread out of the runnable threads in its context store (if any). At the second level, each processor is allowed to execute a quantum of threads within a single simulated machine cycle.

Overall, the quantum-based scheduling policy performs better than the thread-based scheduling policy. This suggests that partitioning a program into excessively fine-grained threads is less favourable than partitioning into coarser, longer executing threads. This may be the reason why current implementations based on dataflow machines [Trau91] seem to be shifting emphasis from exploiting instruction-based parallelism (as is traditional on these architectures) in favour of a multi-threaded style where a collection of dataflow instructions is treated as a sequential thread. Furthermore, our results also agree with experimental results, on stock machines, as reported by Kusakabe *et al* in [KMIA96][2].

8.3.1 Analysis of nfib

We start our experiments with nfib, arguably the most oft-quoted benchmarking program in the functional programming research community. This algorithm is used to compute the number of calls to the Fibonacci function. It uses arithmetic heavily as well as being function call intensive with a doubly recursive structure. This algorithm can be written in Haskell as shown in Figure 8.1

$$\boxed{\mathit{nfib}\ n = \mathbf{if}\ n <= 1\ \mathbf{then}\ 1\ \mathbf{else}\ 1 + \mathit{nfib}(n-1) + \mathit{nfib}(n-2)}$$

Figure 8.1: First definition of nfib.

The best parallelisation for this function is perhaps to always make the recursive calls in parallel since they do not have any data dependencies that can inhibit this. This can be achieved in Naira by annotating the two recursive calls with process as shown in the parallelised program shown in Figure 8.2. As the parent creates these tasks it proceeds to perform the additions (this is fork-and-join).

A disadvantage of this version of the parallelised program is that it does not make optimal use of processors. This is because the parent task cannot do much useful work since it has to wait for the results of the two child tasks to arrive before it performs the additions. It is more efficient to

[2]Their experiments involved coalescing light-weight messages, corresponding to fine-grained dataflow operations like parameter sending in a function call, provided non-strictness is not hindered, into larger threads to minimise the storage and message passing overhead of dataflow machines.

```
nfib n = if n <= 1 then 1 else 1 + n1 + n2
     where n1 = process(nfib (n-1))
           n2 = process(nfib (n-2))
```

Figure 8.2: Second definition of `nfib`.

spark only one child task and to entrust the parent process to do both the additions and the other recursive call to `nfib` (this is evaluate-and-die or lazy task creation). This results in the program of Figure 8.3.

```
nfib n = if n <= 1 then 1 else 1 + n1 + nfib(n-2)
     where n1 = process(nfib (n-1))
```

Figure 8.3: Third definition of `nfib`.

Although this version of the parallelised program does not give as many parallel processes as the previous one, it does increase the locality of data as well as increasing the granularity of the computations. We use this version of the parallelised program in our analysis using varying input sizes. Table 8.1 gives a summary of the statistics collected (for the program of Figure 8.3) while running this benchmark to calculate the number of calls in `nfib 20`.

Benchmark:**nfib 20**	*Number of PEs*				
	1	2	4	8	16
Threads scheduled	306732	317677	317677	317677	317677
Runtime (in machine cycles)	306732	163801	81626	42195	21247
Ready-evaluated tasks	21900	25152	27546	28042	28016
Space allocated	14268K	14644K	14607K	14600K	14600K
Average parallelism	1.0	1.9	3.9	7.5	15.0
Speedup	1.00	1.87	3.76	7.27	14.44
Efficiency	1	97%	97%	94%	93%

Table 8.1: **nfib 20**: statistics summary

This benchmark provides good regular parallelism, with all parallel tasks generated by the same function and the same fashion, leading to an even sharing of the computation among the available processors. Because of this high amount of regular parallelism, combining our deterministic task distribution scheme with quantum-based scheduling policy gives approximately the same runtime as the random distribution strategy combined with thread-based scheduling.

Table 8.1 summarises our experimental results on 1, 2, 4, 8 and 16 simulated processors. Notice from this table that the number of threads executed in the sequential run is smaller by 10945 (the number of parallel function calls generated by `process(nfib (n-1))` in the program) than the number of threads executed in the parallel version. This is because the sequential program on which the figure is based was slightly modified (to introduce the local definition in the third definition of `nfib` so that the `process` annotations can be attached).

Recall also that a thread in our setting is characterised by a sequence of C code and ended by a `return NULL` (signifying the thread's termination which is represented by a horizontal line in our compilation scheme). By observing our compilation rule for the `Let` construct (to which the **where** construct is translated), we see that each closure of the `Let`, like `nfib (n-1)` in the current program, constitutes a thread. The closure for `n1` in `nfib`, which is executed 10945 times, therefore accounts for the increased number of threads executed in the parallelised program.

This table also shows that the number of ready-evaluated tasks (those which are found to be in WHNF when their values are demanded) differs in the execution of the sequential and parallel program. The figure also varies for the same parallel program as the number of simulated processors vary. In general, the number of ready-evaluated tasks and `sendEval` messages is expected to increase in a parallel program involving the use of the `value` annotation. We explain this with the following example. Consider the function call `f (value e1)(value e2)(value e3)`. The code generated for this expression ensures that e_1, e_2 and e_3 are evaluated *before* the code for f is entered. Evaluating e_1, e_2 and e_3 involves, in analogy to the `Let`-bound closures described above, the execution of three threads, one for each expression. When f is entered, the compiler does not know that f's arguments are already in WHNF and so requests for their values are made, via `sendEval` messages, with each request returning immediately with the WHNF value, since each of e_1, e_2 and e_3 will be found to be ready-evaluated. Notice that for this example the number of threads handled, ready-evaluated tasks and `sendEval` messages will each increase by three.

The variation in the number of ready-evaluated tasks as the number of processors vary is generally accounted for by sharing and scheduling constraints. For sharing, it is obvious that since every suspension, s, is evaluated at most once, any subsequent demand for the value of s after the first will find it ready-evaluated.

For the scheduling factors affecting the number of ready-evaluated tasks, recall (from Sections 6.5 and 6.6) that our simulator ensures that within a given time-slice (machine cycle) the machine dedicates its resources to each of the virtual processors so that each of them executes a fixed number of threads. This means that the order in which threads are executed in the parallel program (and even for the same parallel program run with varying number of processors) will be different from the order in which the threads are executed in the sequential program. The number of threads executed by each processor within a time-slice also affects the number of ready-evaluated tasks.

For example, it may be the case that the processor holding the thread which should execute next (according to the sequential execution flow) is not the one to have the machine resources dedicated to it next (assuming execution under a round-robin scheduler). Thus unless the next processors in the cycle have no threads to execute, the order of execution of the threads in the parallel program will differ from that of the sequential run.

Notice that the order in which threads are executed does not affect the overall program result. This is because each thread terminates after execution and the program terminates only when all the threads in the virtual processors are executed. The only effect of the scheduler is that some threads in the parallel program will be executed earlier than they would otherwise be in the sequential program. Notice that this may be beneficial with regards to the heap residency of the program depending on the trade-offs between the space usage of its subgraphs and that of their corresponding values during execution. Peyton Jones has given a detailed discussion of these trade-offs [Peyt87].

Each of the 21891 calls to `nfib` in `nfib 20` returns with a basic value and the associated frames are therefore freed immediately the calls returned using our simple memory manager described in Section 5.3.1. The amount of 'space allocated' entries in Table 8.1 indicate the total heap space used during the execution of the program. The amount of heap allocated for this benchmark varies inversely to the number of ready-evaluated tasks suggesting that executing some threads earlier than the sequential execution order demands (as explained above) carries some space saving benefits.

The activity profiles of Figures 8.4 and 8.5 show the runtime behaviour of this program under the two scheduling strategies. As in the profiles of the previous chapter, the average parallelism is the area covered by the running (green or medium-gray) threads, normalised with respect to the total runtime. The large area (amber or light-gray) of runnable threads indicates that this program can easily use all available processors and indeed that there is excess parallelism. The red (or black) area indicates the number of blocked tasks in the program.

Notice that each of these profiles has an initial sequential part at the beginning of the computation before enough parallel tasks are created to utilise the available processors. After this the profiles show that most of the time 16 threads are running, utilising all available processors. To-

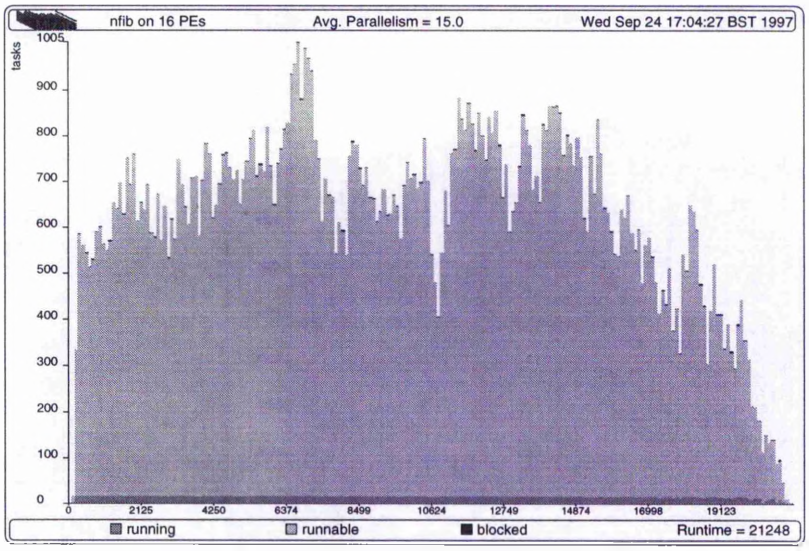

Figure 8.4: **nfib 20**: thread-based scheduling.

wards the end of the computation there is a sharp drop in the number of runnable threads causing some occasional dips in the green area.

Notice from these figures that although the average parallelism is almost the same in both cases (15.0 for the thread-based and 14.6 for the quantum-based), there is an enormous number of runnable tasks in the thread-based scheduling profile compared with that for the quantum-based profile. Given a parallel machine with a large number of processors, the thread-based scheduler can provide an ample opportunity for parallelism. Such medium-grained parallelism can be judiciously exploited if the underlying machine supports the two fundamental issues of parallel multicomputing, namely tolerance to long-latency requests and fast context-switching, as espoused by Arvind *et al.* in [ArIa87]. On machines with a few number of processors, however, the quantum-based scheduler will be preferable to localise work as much as possible and to minimise the costs of communication. The high percentage of runnable threads in these profiles is partly accounted for by our eager task distribution policy and our scheduling scheme as we now explain. The code generated for the function `nfib` consists of 16 threads: four threads[3] each for the computation of the arguments $(n-1)$ and $(n-2)$ to the recursive calls, two threads[4] each for the computation of the other three

[3]One is a parent thread which governs the argument computation (see compilation rule for call-by-need, Section 5.5.4) and the other three are child threads which compute the specified value. An expression of the form $x - y$ (which we represent as $x + (-y)$) consists of three threads.

[4]The two code segments specifying the computation of the value of a binary operator are very similar and they

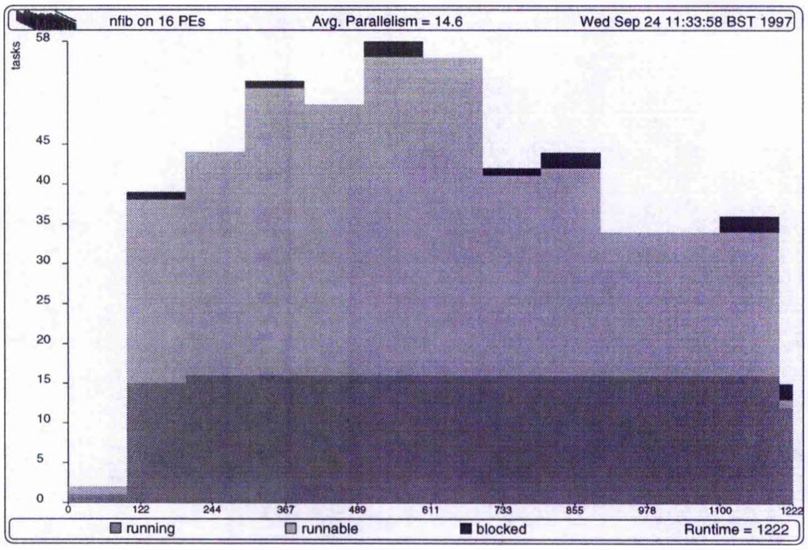

Figure 8.5: **nfib 20**: quantum-based scheduling.

binary operators, one thread that communicates the value 1 (when the conditional expression returns true) and the topmost handler thread for `nfib`.

The topmost handler thread, when executed, allocates the function frame associated with this call, stores the address to send the value of the call to and a pointer to the argument suspension and then terminates (see compilation rule for supercombinators, Section 5.5.5). In general, there are more threads queued (with runnable status in the message stores of the virtual processors of the machine) than are executed between one `nfib` call and another. For example, 10 threads are queued on the first call out of which 7 are executed before the next call and so on. The times at which more threads are executed than are queued are relatively fewer—typically when the first arm of the conditional is taken. For the quantum based scheduler on the other hand, 16 threads are forced to be executed each time and thus significantly minimise the number of runnable threads during the program's runtime. Note that although the queued threads are consumed in a LIFO order (on a thread termination in the sequential run), the parallel scheduler alters this order making the number of threads executed between one function call and another unpredictable.

The results reported above are largely supported by the results we obtained for this benchmark when run under the GrAnSim simulator [HLP95, Loid96]. For example, the average parallelism of

are generated as such to ensure the correct synchronisation and semantic behaviour regardless of the order in which they are executed.

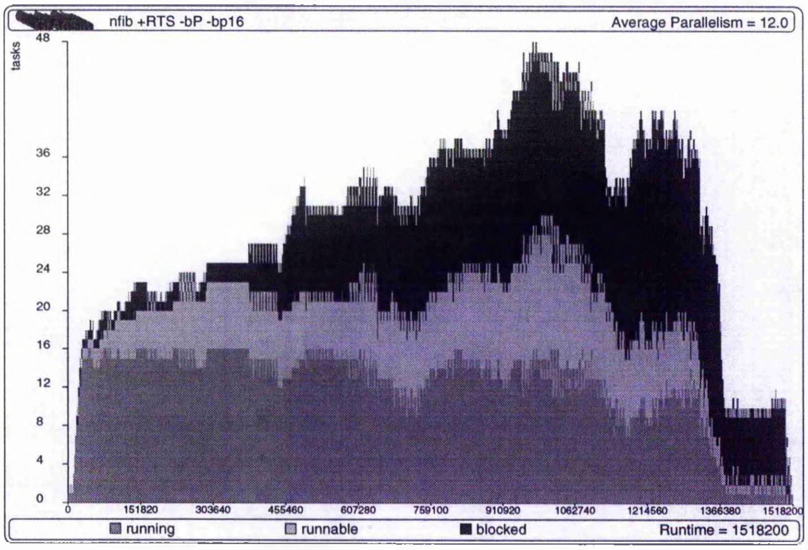

Figure 8.6: **nfib 20**: on GrAnSim-Light.

`nfib 20` on 16 processors is 12.0 as the parallelism profile of Figure 8.6 shows while the average parallelism on GrAnSim-Light with an unbounded number of processors was 22.4. The average parallelism we obtained using Naira (for `nfib 20`) was higher than that obtained using GrAnSim probably because GrAnSim performs more accurate runtime costings. Furthermore, for all our experiments with this benchmark using GrAnSim, we obtained better average parallelism (up to 1.6 higher) when simulating shared memory machines than when simulating distributed memory ones with asynchronous communication. The average parallelism figures for these architectures, however, tend to be the same as the input size increases for this benchmark. This is because more parallelism is introduced with higher input sizes and thus the likelihood of some processors staying idle (due to lack of thread migration in the distributed memory set up) is highly reduced. It is also because evaluate-and-die allows tasks to be absorbed.

8.3.2 Analysis of nqueens

In the **nqueens** benchmark, the set of all possible solutions to the classical n-queens problem is obtained. This is an example of a program which illustrates 'back-tracking' in the sense that solutions to a problem are searched for along many possible paths, returning to the last untested option to try again if the current path fails to deliver the desired result. **nqueens** uses a lot of arithmetic, comparison and list-handling operations.

The goal of this program is to place N queens on an NxN chessboard such that no two queens are placed on the same rank, file or diagonal. In a parallel system therefore, the ideal situation is to devise a suitable algorithm so that the searches for solutions can be performed in parallel. The following Haskell functions in Figure 8.7 implement the main part of this algorithm.

```
search rank file n board =
        if file > n then 0 else  -- no more solutions on this rank
        if rank > n then 1 else -- a solution has been found
          first rank file n board + search rank (file+1) n board
first rank file n board =
    firstcond rank file n board (compatible rank file (rank-1) board)
firstcond rank file n board False = 0
firstcond rank file n board True = search (rank+1) 1 n (file:board)
```

Figure 8.7: First definition of nqueens.

The program (which computes `search 1 1 8 []`) starts by placing a first queen on each file (i.e., column) along rank (i.e., row) 1 thus giving an immediate N-way parallelism in the search. Each of the N-way parallel searches then attempts to place a second queen on rank 2 in all possible non-attacking positions (in the second equation of `firstcond`). Each of these potential solutions (on row 2) is used in turn to generate further searches on rank 3 and so on, while accumulating a list of the successful file positions. If any potential solution fails to place a queen then that search is terminated at the rank reached. If rank N is reached then the list of the file indices of all N queens are used to encode a solution [Rob89]. The complete program is listed in Appendix A.2.

This program can be parallelised by ensuring that the N-way searches (each characterised by the call to the `first` function) are started in parallel. Similar to the `nfib` program, more parallelism can be created by making the recursive call to `search` into an exportable task as well. Better processor utilisation, locality and coarser computation grains result if the recursive call to `search` is absorbed into the parent process. Notice from the definition of `first` that before a queen is placed it must be checked that it is compatible (i.e., non-attacking) to those already placed. More parallelism can be generated by spawning a child task to perform the compatibility check.

Notice also that to ensure the exploitation of only conservative parallelism, the searches generated by `firstcond` should not be performed speculatively. We introduce an auxiliary function `search1` (to replace `search` inside `firstcond` when the 8 searches in the program are started in parallel) which is the same as `search` except that it does not spawn child tasks to perform the

*Benchmark:*8 queens	*Number of PEs*				
	1	2	4	8	16
Threads scheduled	3774K	3806K	3806K	3806K	3806K
Runtime (in machine cycles)	3774K	2474K	1232K	631K	319K
Ready-evaluated tasks	559644	559644	559394	559329	559537
Space allocated	179991K	181588K	181592K	181593K	181590K
Average parallelism	1.0	1.5	3.1	6.0	11.9
Speedup	1.00	1.53	3.06	5.97	11.81
Efficiency	1	77%	77%	75%	74%

Table 8.2: 8 queens: conservative parallelism figures.

```
search rank file n board =
        if file > n then 0 else   -- no more solutions on this rank
        if rank > n then 1 else -- a solution has been found
           let fs = process(first rank file n board)
           in  fs + search rank (file+1) n board
first rank file n board = firstcond rank file n board c
      where c = process(compatible rank file (rank-1) board)
firstcond rank file n board False = 0
firstcond rank file n board True = search1 (rank+1) 1 n (file:board)
                                   -- search (rank+1) 1 n (file:board) -- speculative
```

Figure 8.8: Second definition of nqueens.

speculative searches from the queens placed starting from row two[5]. This leads to the following parallel program of Figure 8.8.

To introduce speculative computations we replace `search1` by `search` in the code segment in Figure 8.8. This will make all subsequent searches generated from all potential solutions, starting from the successfully placed queens at row 2, for each of the 8 parent tasks to be performed in parallel. These speculative searches, while started in parallel, are also forced to completion.

Tables 8.2 and 8.3 summarise the results obtained for 8 queens when exploiting conservative

[5]Note that this is the stage at which speculation can start. The queens placed in the first rank (corresponding to the parent tasks *search 1 i 8 []*, $1 \leq i \leq 8$) found 4, 8, 16, 18, 18, 16, 8 and 4 solutions respectively.

Benchmark: **8 queens**	*Number of PEs*				
	1	2	4	8	16
Threads scheduled	3774K	3900K	3900K	3900K	3900K
Runtime (in machine cycles)	3774K	2027K	1090K	620K	382K
Ready-evaluated tasks	559644	581060	577540	574932	573452
Space allocated	179991K	185552K	185607K	185648K	185671K
Average parallelism	1.0	1.9	3.6	6.3	10.02
Speedup	1.00	1.86	3.46	6.09	9.87
Efficiency	1	96%	89%	79%	64%

Table 8.3: **8 queens**: speculative parallelism figures.

and speculative parallelism respectively. The results summarised in these tables show that the number of threads executed in the parallel program is higher in both cases than in the sequential program. This is because, as similarly explained in the `nfib` program of the previous section, two local definitions are introduced (inside `search` and `first`, see code above) in the conservative case and a third one is introduced to force each call of `firstcond` in the speculative case.

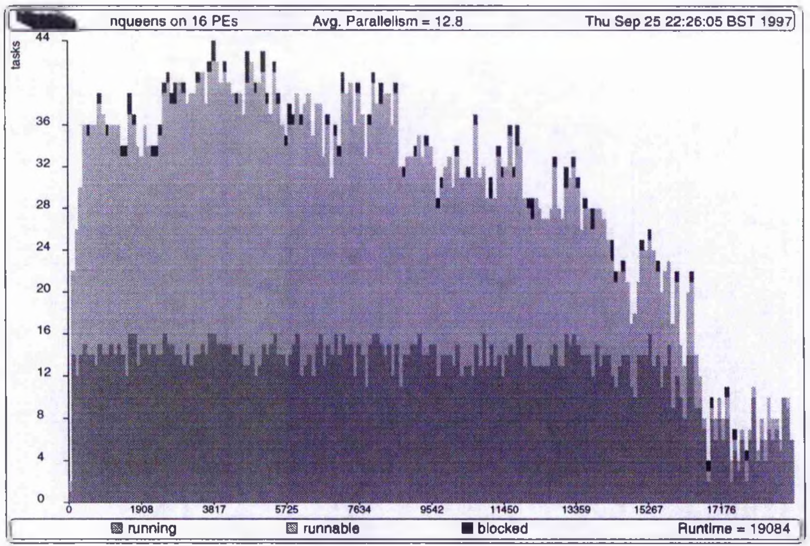

Figure 8.9: **8 queens**: conservative parallelism profile.

The number of ready-evaluated suspensions in the parallel program, in both cases, varies inversely to the amount of heap usage of the program indicating that executing some threads earlier than the sequential execution flow demands leads to slightly higher residency of this program.

Comparing the figures in Tables 8.2 and 8.3 we see that although more computations are performed in the speculative case (as expected), the abundance of parallel tasks leads to better load sharing in the speculative case for our experiments using 2, 4 and 8 processors. The amount of space used in the speculative program is higher because of the use of the `value` annotation on calls to `firstcond` for the same reasons as explained in the analysis of the results of the `nfib` benchmark.

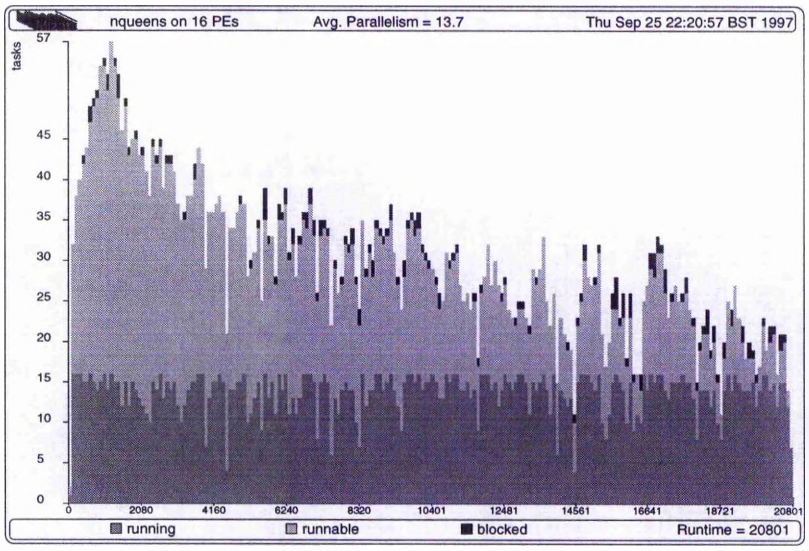

Figure 8.10: **8 queens**: speculative parallelism profile.

Figures 8.9 and 8.10 respectively show the activity profiles for the conservative and speculative versions of the program on sixteen processors using quantum-based scheduling. These profiles show an encouraging performance of Naira on this benchmark with about thirteen tasks running at any time during the program execution. The occasional dips in the running tasks area probably indicates the backtracking in the algorithm.

Figure 8.10 gives a slightly higher average parallelism than Figure 8.9, in contrast to the corresponding figures in Table 8.2 and 8.3 (obtained using thread-based scheduling). This shows the comparative effect between the thread-based scheduler and the quantum-based scheduler for this benchmark. The runtime measured for the conservative case is better than that in the speculative case.

8.3.3 Analysis of tak

Tak is the Takeuchi function which is function call intensive and highly recursive (with non-primitive recursion). It performs only a simple test, uses only simple small-integer arithmetic and does no storage allocation at all. This means that like the `nfib` function above, this benchmark essentially tests the stack operations (i.e., the efficiency of function call) in the underlying abstract machine. This function can be expressed in Haskell as shown in Figure 8.11.

```
tak x y z = if x <= y then z else
                tak (tak (x - 1) y z) (tak (y - 1) z x) (tak (z - 1) x y)
```

Figure 8.11: First definition of `tak`.

The natural parallelisation of `tak` is to ensure that the three inner calls are always made in parallel with the outer call (i.e., achieving vertical parallelism whereby the evaluation of the function and that of its arguments proceed in parallel). This can be achieved by annotating each of the argument applications with `process`, leading to the exploitation of vertical parallelism for each call to `tak` in which the value of the first argument is greater than the second. This results in the parallelised program of Figure 8.12.

```
tak x y z = if x <= y then z else
    tak (process(tak (x - 1) y z)) (process(tak (y - 1) z x)) (process(tak (z - 1) x y))
```

Figure 8.12: Second definition of `tak`.

We conducted our experiments using this version of the parallel program with varying input sizes and varying distribution and scheduling algorithms. Table 8.4 summarises the statistics for `tak 18 12 6`[6] using a deterministic distribution algorithm and a thread-based scheduler.

The experimental results in this table show that, in contrast to `nfib` and `nqueens` described earlier, fewer threads are executed in the parallel program than in the sequential program. The reason for this can be seen by referring to our compilation rule for parallel function calls (Section 5.5.4) of which

tak (*process*(*tak* (x − 1) y z)) (*process*(*tak* (y − 1) z x)) (*process*(*tak* (z − 1) x y))

is a direct instance. As can be seen from this compilation rule, there are no horizontal lines

[6]The value returned for the call `tak 18 12 6` is 7.

Benchmark:**tak 18 12 6**	*Number of PEs*				
	1	2	4	8	16
Threads scheduled	1097K	1049K	1049K	1049K	1049K
Runtime (in machine cycles)	1097K	557K	282K	144K	69K
Ready-evaluated tasks	143121	142676	13506	136302	136074
Space allocated	53744K	55502K	55621K	55601K	55605K
Average parallelism	1.0	2.0	3.9	7.7	15.1
Speedup	1.00	1.97	3.89	7.57	14.98
Efficiency	1	99%	98%	95%	94%

Table 8.4: **tak 18 12 6** statistics summary.

(indicating termination) after the compilation functions of the argument expressions. This provides the opportunity for the function call to proceed in parallel with the computation of its arguments.

In contrast to our compilation rule for lazy function calls (also given in Section 5.5.4), of which

$tak\ (tak\ (x - 1)\ y\ z)\ (tak\ (y - 1)\ z\ x)\ (tak\ (z - 1)\ x\ y)$

is an instance, the code for each argument expression is followed by termination, increasing the number of threads count by three for each such call (see also Section 8.3.1)

As in the nqueens program described in the previous section, the number of ready-evaluated tasks and the program's residency vary inversely with one another. The number of ready-evaluated tasks decreases as the number of processors is added except for the case of simulating eight processors. Similarly, the heap usage increases except in this particular case (of using eight processors).

With the creation of three parallel processes on each call to tak which has its first argument greater than its second, this program generates good regular parallelism. As in the nfib benchmark, since the parallel tasks created contain very similar amount of computations, there is no significant difference between the deterministic and random distribution of the parallel tasks and between thread-based and quantum-based scheduling strategies.

As a matter of fact, and as Figures 8.13 and 8.14 show, parallelism generated by tak, even at the quantum-based level of scheduling is too fine-grained. This is because each call to tak performs only a simple test before returning or generating similar lightweight tasks. These figures show that there are a lot of runnable tasks throughout the computation and that even though the number of runnable threads is halved by using the function-based scheduler, there are many of these tasks awaiting execution, at any moment, throughout the computation. This is because, in common

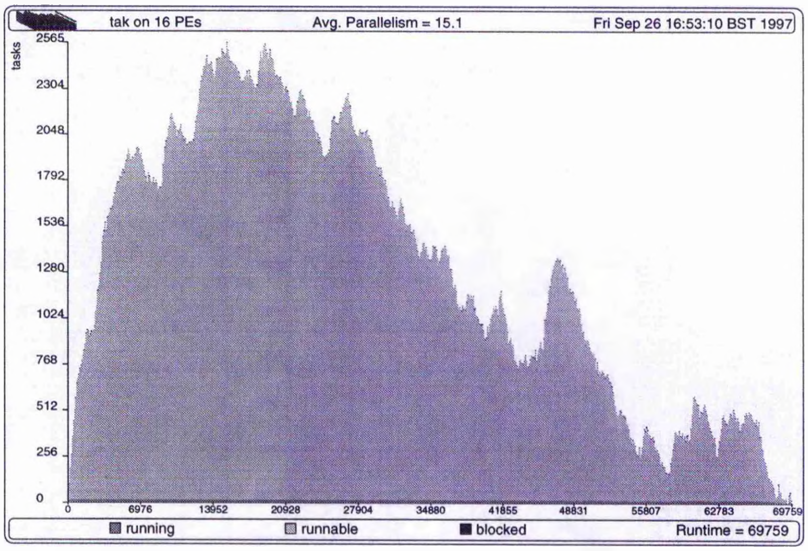

Figure 8.13: **tak 18 12 6**: thread-based scheduling.

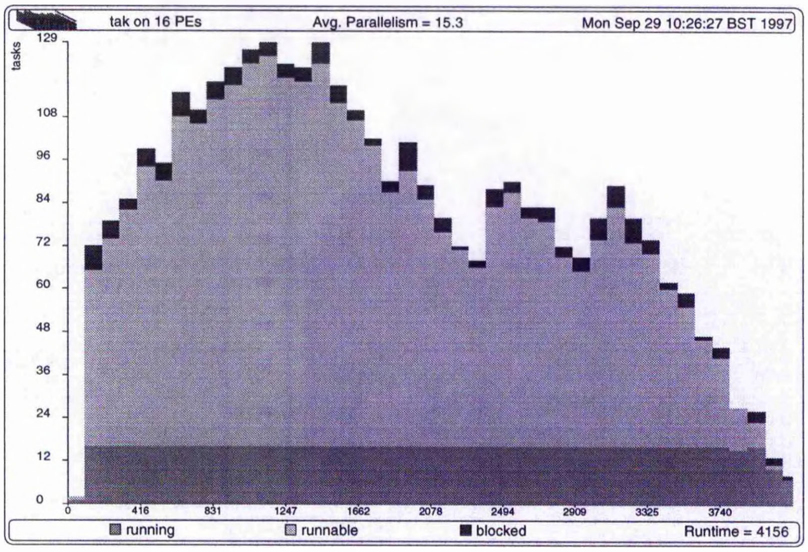

Figure 8.14: **tak 18 12 6**: function-based scheduling.

with the discussion on `nfib`, the code generated for `tak` consists of 23 threads and fewer threads are executed than are queued from one call to another. The overall performance of Naira on this benchmark is quite good.

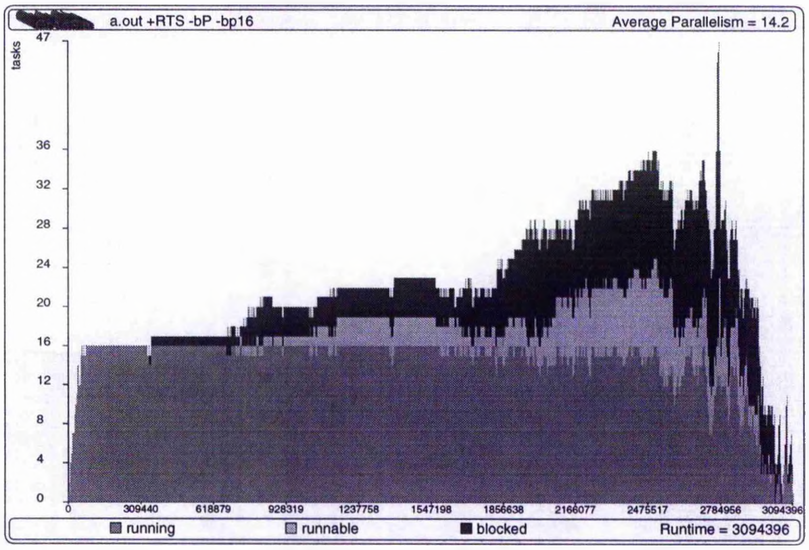

Figure 8.15: **tak 18 12 6**: on GrAnSim-Light.

The profile of Figure 8.15 is obtained for the `tak` program when run under GrAnSim-Light with sixteen processors. This records an average parallelism of 14.2 and a speedup of 12.01. The average parallelism and speedup figures on a GrAnSim-Light with an unbounded number of processors are 24.9 and 21.03 respectively. Notice that the shapes of the parallelism profiles obtained from our simulator and that from GrAnSim-Light are different probably because of the differences in some implementation issues in the two systems. For example, while we use an active distribution policy and a notification model of synchronisation, GHC uses passive task distribution with evaluate-and-die synchronisation.

8.3.4 Analysis of coins

This benchmark is of more real-life application than those of the preceding sections because it addresses a practical issue (e.g., as in daily usage on vending machines) and also for its use of typical functions (map, filter) used in everyday programming. It is taken from the Spectral subset of the Glasgow `nofib` benchmark suite. Given a collection of coin denominations, `coins`, and a certain amount of money, `amount`, for an item this benchmark computes the number of all possible ways in which the given amount can be paid.

The mutually-recursive functions `pay_num` and `aux` of Figure 8.16 define the core of this algorithm. The complete code of this problem is given in Appendix A.4.

```
pay_num :: Int → Int → [Int] → Int
pay_num _ 0    coins   = 1
pay_num _ val  []      = 0
pay_num pri val coins  = sum (map  (aux pri val bar_coins) (mynub bar_coins))
    where bar_coins  = (dropWhile (>val) coins) --value?
aux pri val coins1 c   =
    pay_num  (pri−1) (val−c)(del (dropWhile (>c) coins1) c)
```

Figure 8.16: First definition of coins.

The statistics we report for this program is the result of calculating the number of ways of paying an amount of 137 using a collection of coins as 250, 100, 25, 10, 5, 1. That is the value of the program is the result of the call (pay_num 100 137 coins).

As the above program shows, each call to pay_num starts by computing bar_coins by dropping those coins from the current coins list which are higher in value than the current amount, amount, whose different ways of payment is being found by the current pay_num call. The most expensive part of the computation is the step which computes, using the coins bar_coins, the ways of paying (amount-c), for each distinct coin c in bar_coins.

From the foregoing, it is clear that the natural parallelisation of this algorithm (at the topmost level) is to conduct the computations for the possible ways of payments, at each level, of (amount-c) in parallel with each other. This can be achieved by using parmap in place of map in pay_num. We use the following data-parallel encoding of parmap:

```
parmap f []      = []
parmap f (x:xs)  = process(f x) :value(parmap f xs)
```

It can be understood from the resulting program (in the light of Example 1.1 of Section 1.3) that since the function enclosing the parmap application, namely sum, is strict the parallel processes (of the form aux pri amt coins coin) spawned by parmap are activated in parallel with each other.

The profile of Figure 8.17 is obtained after the above step in the parallelisation of this program. This profile depicts good processor utilisation with very few runnable but unemployed tasks and very few blocked tasks throughout the computation. About nine processors were active most of the time during the execution with an average parallelism of 8.5 on 16 processors.

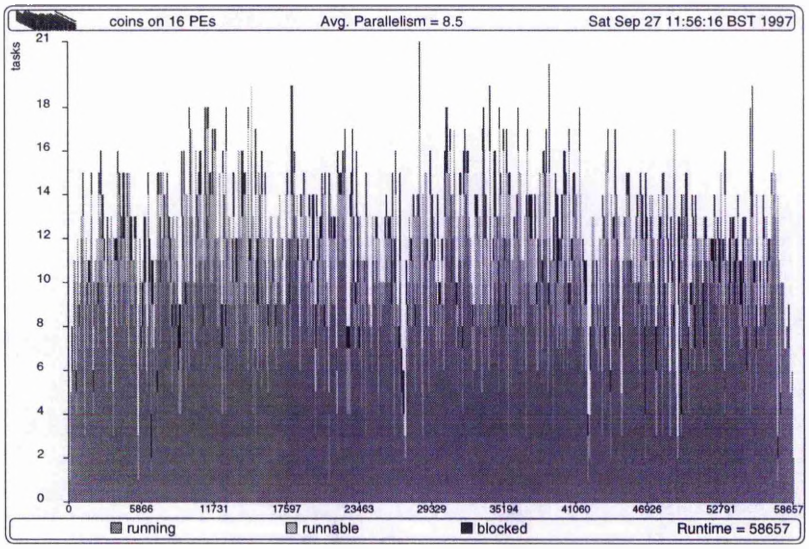

Figure 8.17: **coins 137**: parallelism profile **version 1**.

The performance of the above program can be improved by passing `bar_coins` by value while spawning a child task to compute the list argument of `parmap`. Furthermore, the tail call of `pay_num` inside `aux` can be parallelised by passing its third argument by process. Finally, we parallelise some of the auxiliary functions used by this program to obtain the parallelised program in Figure 8.18.

```
pay_num :: Int → Int → [Int] → Int
pay_num _ 0   coins    = 1
pay_num _ val []       = 0
pay_num pri val coins = sum (parmap  (aux pri val bar_coins) (process(mynub bar_coins)))
    where bar_coins  = (dropWhile (>val) coins) --value?
aux pri val coins1 c   =
     pay_num  (pri−1) (val−c)(process(del (dropWhile (>c) coins1) c))
```

Figure 8.18: Second definition of `coins`.

With this final parallelised program the activity profile of Figure 8.19 obtains. Compared with the parallelism profile of version 1 (Figure 8.17), this profile indicates better parallel behaviour: the number of blocked tasks is reduced, and the average parallelism has improved to 10.5. The heap usage and runtime are also better for the current version of the program.

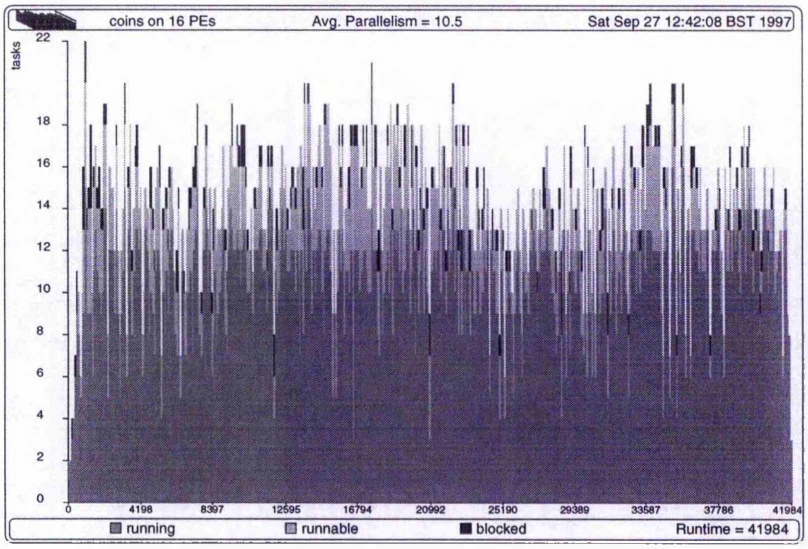

Figure 8.19: **coins 137**: parallelism profile **version 2**.

As the profiles show, there are high prospects for parallelism in this benchmark depending on the input. For the statistics we report here there are initially five parallel processes which then start other parallel computations required to find the ways of paying the amounts 37, 112, 127, 132 and 136 using varying coin denominations. There are thus 6516 parallel tasks created which constitute about 4% of the function calls in the program.

For this benchmark there is a significant difference between the thread-based and quantum-based scheduling because there are a few top-level value bindings defining the input data all of whose suspensions are executed on a particular processor. The comparative analysis of these two scheduling schemes is presented in Section 8.3.7.

Table 8.5 summarises the statistics collected after running this program. This table shows that there are more threads executed in the parallel program than in the sequential one, for the same reason as explained in earlier benchmarks. The number of ready-evaluated tasks and space allocation vary directly with each other in this program.

This program has a high space usage because the function **pay_num** has three arguments and **aux** has four arguments, one of which in both cases is a list structure. Notice that although frame accesses are local operations in our implementation (which may otherwise involve accesses

*Benchmark:*coins 137	*Number of PEs*				
	1	2	4	8	16
Threads scheduled	3024K	3039K	3039K	3039K	3039K
Runtime (in machine cycles)	177905	107814	70402	27542	41983
Ready-evaluated tasks	502091	514332	527042	510599	512680
Space allocated	143026K	143691K	143750K	143721K	143727K
Average parallelism	1.0	1.9	3.6	6.5	10.5
Speedup	1.00	1.65	2.53	3.46	4.24
Efficiency	1	83%	63%	43%	27%

Table 8.5: **coins 137** statistics summary.

to remote processors for a function argument), since all functions are lambda-lifted, we pay a space penalty (as do all implementations supporting lambda-lifting) since free variables have to be added to the arguments and threaded all the way to the place of usage. The more arguments a combinator has the more space it uses in our implementation since each message required to activate a handler thread must contain all the data required to make the call.

The performance of Naira on `coins` using quantum based scheduling is quite good. Using thread-based scheduling, however, gave less favourable results because of the irregularity of work sharing among the virtual processors. Loidl and Hammond gave a detailed study of the parallel behaviour of the `coins` program, using granularity control mechanisms, on different architectures using GrAnSim. They have also analysed its performance on a sized time system where the annotations needed to parallelise the program are derived using the sized time system [LoHa96a]. On the sized time system they considered a set of 100 different coins and a price of 55, yielding a speedup of about 13 on 32 processors.

8.3.5 Analysis of `matmul`

Matrix multiplication is an interesting application which is encountered in many real problems. Our `matmul` benchmark therefore, is aimed at testing the suitability of our compiler for performing such arithmetic-intensive operations as matrix multiplication. The experiments we performed involved multiplying a pair of 32 by 32 integer matrices.

As with the other benchmarks, we tried different implementations of the matrix multiplication algorithm so as to get a better implementation that can be parallelised more effectively. The implementation of Figure 8.20 is the best of those we tried.

```
foldint l u f g = if l == u then f l else
                     g (foldint l mid f g)(foldint (mid+1) u f g)
   where mid   = div (l+u) 2
matmul mA n    = foldint 1 n (row mA n) (++)
row mA  n i    = [(foldint 1 n (sprod mA n i) (++))]
sprod mA  n i j = [foldint 1 n (mult mA i j) (+)]
mult mA  i j k  = sel((sel mA i)) k × sel((sel mA k)) j
```

Figure 8.20: First definition of matmul.

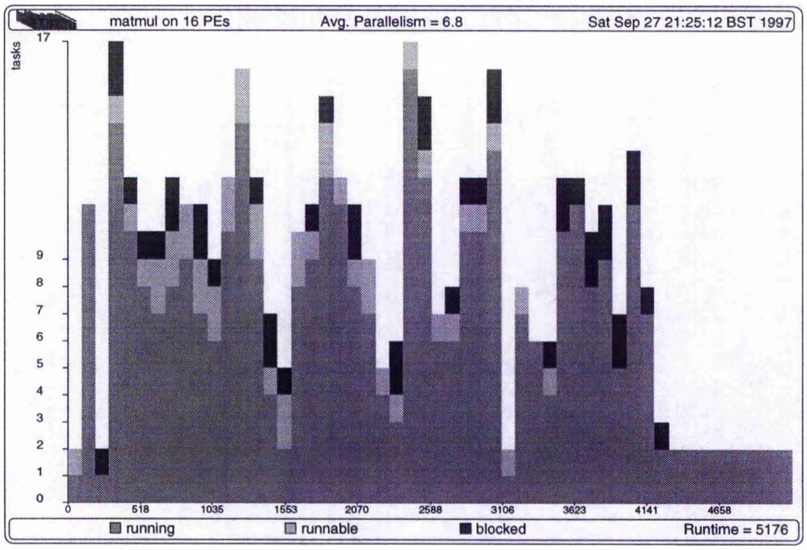

Figure 8.21: **matmul**: Parallelism profile version 1.

The matmul program can be parallelised at the topmost level by making the two recursive calls to foldint and the call to div in parallel. We do this by annotating these calls with process which ensures the exploitation of vertical parallelism since the parent process can proceed with the call to g as the two child processes compute the arguments. foldint1 in the right-hand sides of row and sprod is the sequential version of foldint used to visualise the effect of parallelising these functions in stages as described below. Figure 8.21 shows the activity profile of matmul with this parallelisation of foldint.

As the profile shows, there is no significant parallelism introduced on account of this change.

```
pAppend l1 l2 = pfoldr tailCons l2 l1
pfoldr f z []      = z
pfoldr f z (x:xs) = ans
        where rest = process(pfoldr f z xs)
              ans  = value(f x rest)
tailCons x xs = x:value xs
```

Figure 8.22: Definition of `pAppend`.

This is because the two recursive calls to `foldint` correspond to the two arguments to the (`++`) function (except in `sprod` where they are arguments to the (+) function), which is non-strict in its second argument. Furthermore, (`++`) is also not strict in the list elements.

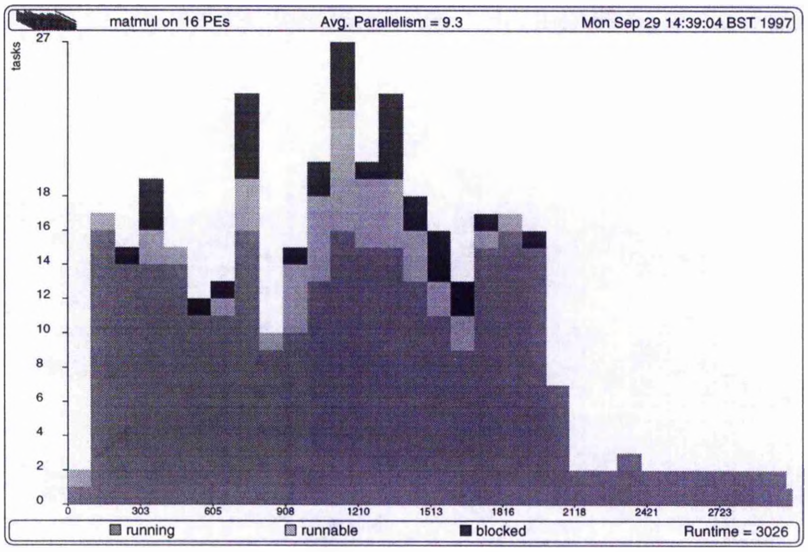

Figure 8.23: **matmul**: Parallelism profile version 2.

As a possible way round this therefore, we can use a version of (`++`) which forces its arguments so that the two arms of the divide and conquer are always initiated simultaneously. We define the function `pAppend` function in terms of a parallel function `pfoldr` as shown in Figure 8.22.

It turns out, again, that the use of `pAppend` does not help matters much; there is still not much parallelism measured. This is because of the fact that the parallelism is completely hidden inside the bodies of `row` and `sprod` which perform the meat of the computations. To expose this parallelism, we must force the evaluation of the lists in the right-hand sides of row and sprod beyond WHNF while using the parallel foldint in place of foldint1.

In order to visualise the individual effects of exposing parallelism inside the definitions of row and sprod, we first force the evaluation of the value of each call of row and without forcing the evaluation of that of sprod. This can be achieved using value annotations as follows:

row mA n i = value[value(foldint 1 *n* (*sprod mA n i*) (++))]

The activity profile of Figure 8.23 shows the effect of this change. Notice the improvement over the previous step: the average parallelism rose to 9.3, number of blocked tasks decreased and the runtime of the program almost halved.

Next, we force the result of sprod within each call of row (i.e., creating 32 parallel tasks each computing an element of a 32-element vector which forms part of the resulting 32 by 32 matrix), by using value annotations inside the definition of sprod analogously as in row above:

sprod mA n i j = value[value(foldint 1 *n* (*mult mA i j*) (+))]

More parallelism can be generated at the lowest granularity level in this algorithm by creating a further 32 tasks during the computation of a matrix element while the parent task combines their results to perform the 31 additions. It is clear that creating such tasks, each of which performs a single integer multiplication, leads to too fine-grained tasks and our experiments show that this is not beneficial.

Figure 8.24 shows the activity profile of the final parallelised program. This profile shows a modest improvement in parallel behaviour over the previous parallelisation step. Table 8.6 summarises the runtime statistics obtained for this benchmark on different machine configurations.

This table shows that the number of threads created in the parallel program is higher than in the sequential one because of the introduction of local definitions and the use of value annotations in the parallel program, as explained in the preceding benchmarks.

The number of ready-evaluated suspensions in the parallel program increases as the number of processors increases and the heap residency of the program decreases with it. This indicates than the use of our value annotation in this program, which ensures that expressions are evaluated as early as possible, is beneficial since the representation in memory of the closures computing the matrix elements is more expensive than the representation of the matrix elements themselves.

As the parallelisation process shows matrix multiplication is a good source of parallelism ranging from vector multiplication level of granularity to the lowest level of instruction level of parallelism. The activity profiles we obtained during our experiments indicate better performance when we

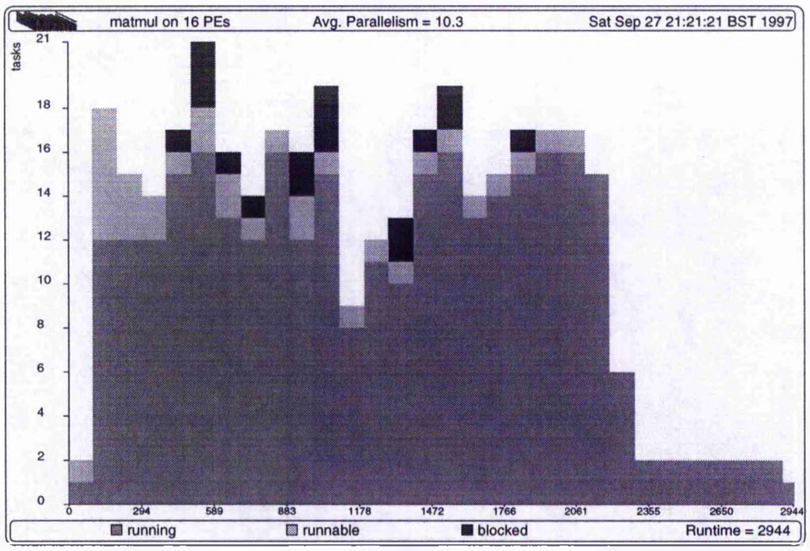

Figure 8.24: **matmul**: Parallelism profile version 3.

Benchmark:**32 by 32 matmul**	*Number of PEs*				
	1	2	4	8	16
Threads scheduled	195239	216001	216001	216001	216001
Runtime (in machine cycles)	5917	4652	3632	3138	2943
Ready-evaluated tasks	14640	20892	21512	21681	21787
Space allocated	10188K	11092K	11082K	11080K	11079K
Average parallelism	1.0	1.9	3.6	6.1	10.3
Speedup	1.00	1.27	1.63	1.89	2.01
Efficiency	1	70%	45%	26%	14%

Table 8.6: **32 by 32 matmul** statistics summary.

adopted the level of granularity in which a single task performs the computation of a single element of the resulting matrix.

The space usage of this benchmark is very small compared with the space usage of other benchmark programs. This is because all intermediate structures allocated during the computation are freed immediated the associated function returned since the computations are conservatively performed and no parts thereof are delayed or suspended.

8.3.6 Analysis of soda

Soda is a word search program taken from [RuWa95]. This benchmark consists of twelve given words which are to be searched horizontally, vertically and diagonally from a grid of letters (i.e., a list of words). The hidden words may be written both forwards and backwards. Accordingly the program derives three grids from the original which represent its columns and diagonals. Each hidden word is thus searched in eight different directions as shown in the core function of the program in Figure 8.25. Soda is a data-structure-intensive benchmark and we parallelise it in stages (as in [RuWa95]) displaying the activity profiles showing the effect of each step.

```
main reps = [AppendChan stdout (concat (map find hidden))]
  where
    find word = word ++ " " ++ concat dirs ++ "\n"
      where
         dirs = map snd (filter (any (contains word) . fst)
            [(r, "right"),(d, "down"), (dl, "downleft"), (ul, "upleft")] ++
            filter (any (contains drow) . fst)
            [(r, "left"),(d, "up"), (dl, "upright"), (ul, "downright")])
         drow   = reverse word
    r  = grid
    d  = transpose grid
    dl = diagonals grid
    ul = diagonals (reverse grid)
```

Figure 8.25: Definition of soda.

```
parmap f []     = []
parmap f (x:xs) = process(f x) :value(parmap f xs)
```

Figure 8.26: Definition of parmap.

We approach the parallelisation of soda in a data-parallel fashion in such a way that the search for the hidden words can proceed in parallel with each other. This can be achieved by replacing map in main with its parallel version parmap (as in the coins program of Section 8.3.4), defined as in Figure 8.26

This definition of **parmap** corresponds to a data-parallel algorithm in the sense that one parallel process is spawned for each list item. This modification results in a profile very similar to that obtained for sequential execution. This means that there is not much improvement in performance due to the use of **parmap**.

The reason for this lack of observable improvement is because although the searches have been started in parallel[7], there is little work done before the processes are suspended, thanks to the laziness of (++). That is, the result of (**find word**) reaches WHNF before the other computations necessary for the completion of the search are even started! We must therefore force the search for a word in all 8 directions to complete before the result of (**find word**) is returned.

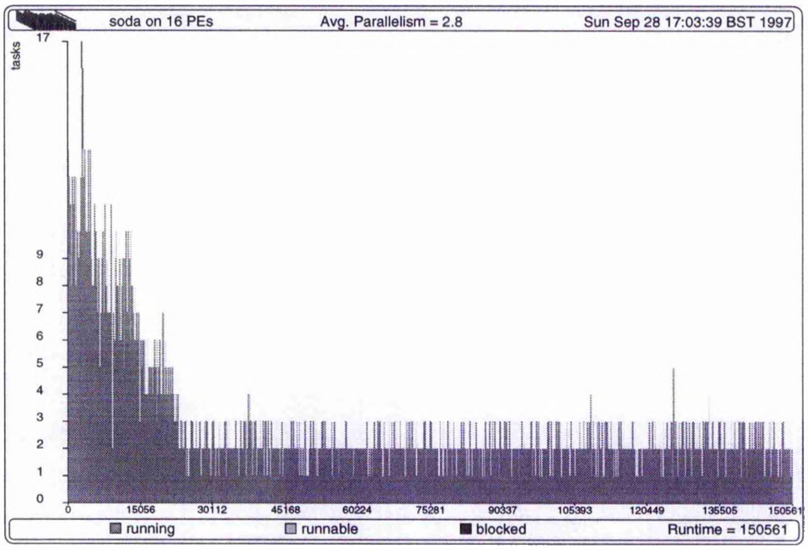

Figure 8.27: **soda**: parallelism profile version 1.

This can be achieved by forcing the (head and tail) strict evaluation of the list of directions along which the **word** in (**find word**) may have been found[8]. We make use of the function **force**, Figure 8.28, to realise this:

The function **force** ensures that the spine of the list **dirs** as well as its elements are completely evaluated. Notice that with the above definition of **force** all searches for a particular word are

[7]This is because of the tail-strictness of our **value** annotation (see Example 1.1, Section 1.3) and the strictness of the **find** function. The parallel searches are triggered immediately the enclosing function, concat, demands the head of the list created by **parmap**.

[8]Alternatively, each search can be forced to complete before returning its one line message by using a version of (++) which forces its arguments in parallel as in the **matmul** program of the previous section.

```
force []       = []
force (x:xs)   = value x:force xs
find word      = word ++ " " ++ concat (force dirs) ++ "\n"
```

Figure 8.28: Definition of **force**.

conducted on the current processor since there is no **process** annotation. The resulting profile from this change is depicted in Figure 8.27

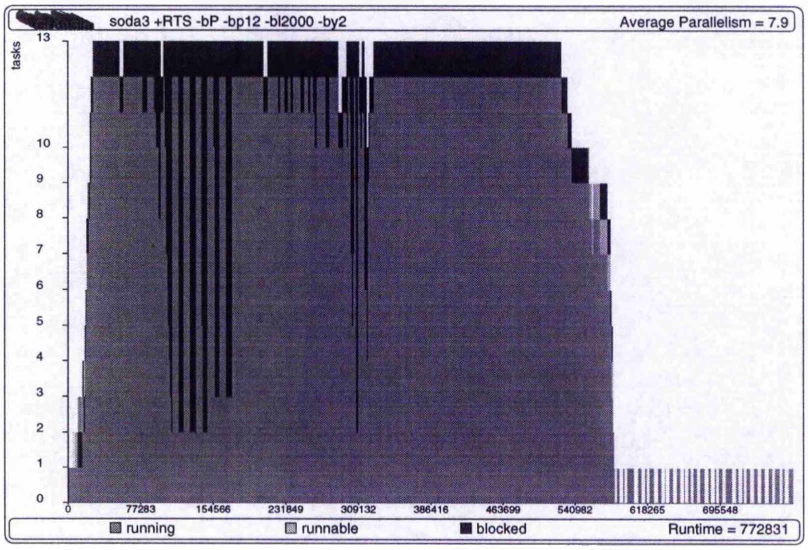

Figure 8.29: **soda**: a sample GranSim profile.

This profile (Figure 8.27) indicates a better parallel activity (with an average parallelism of 2.8) over the one obtained as a result of using only **parmap**. The twelve processes performing the searches seem to have done more work than in the previous case although as the profile shows there was a gradual downward slope settling at fewer than 3 tasks for the rest of the computation.

In contrast to our experiments on GrAnSim with this benchmark, we found that GrAnSim exhibited better parallel activity, with an average parallelism of 7.9, as shown in Figure 8.29. Although the shape of the profiles look similar, the shortest height of the GrAnSim profile constituted about 25% of the execution time. Runciman and Wakeling [RuWa95] reported an average parallelism of 11.4 at this parallelisation step on their idealised simulator.

The short-wide portion of the profile of Figure 8.27 indicates that the processes performing the

```
d  = process(transpose grid)
dl = process(diagonals grid)
ul = process(diagonals (reverse grid))
```

Figure 8.30: Annotating the grids of words.

search become blocked. To ensure that this blocking is avoided, we annotate the derived grids (see Figure 8.30) so that their evaluation proceeds in parallel with the searches.

This modification results in a slight increase in parallel activity in comparison to the previous step. Using a certain combination of annotations on GrAnSim required to spark these derived grids and to evaluate them to WHNF degraded the performance to a speedup of 6.1. As with our profile, Runciman's simulator recorded the same small increment in speedup of 0.1.

```
dirs = value(map snd (forw ++ back))
forw = process(filter (any (contains word) . fst)
                 [(r, "right"), (d, "down"), (dl, "downleft"), (ul, "upleft")])
back = process(filter (any (contains drow) . fst)
                 [(r, "left"), (d, "up"), (dl, "upright"), (ul, "downright")])
```

Figure 8.31: Improved definitions of `forw` and `back`.

More parallelism can be created by arranging that the forwards and backwards searches in `dirs` proceed in parallel. We can achieve this by making `forw` and `back` in the program segment of Figure 8.31 into parallel tasks. The profile of Figure 8.32 results from this modification. The program now displays an average parallelism of 4.0 and executes with a reduction of about 20% of its sequential execution time.

We have so far been conservative in the search for a word in a given grid. We can introduce more parallelism by employing some speculative evaluation so that the searches for a given word in a particular grid proceed in parallel. This means that the given word is searched in parallel along each line in the grid. Obviously some of these searches will lead to unwanted computations since the word may be found much earlier than the next subsequent (redundant) searches take to complete. On the other hand, the speculation may be desirable for the cases where the word is found towards the end of the grid. The speculation may be introduced by replacing the any function in the program with `parany` defined in Figure 8.34.

The effect of this change on the parallel behaviour of the program is shown in Figure 8.33. The

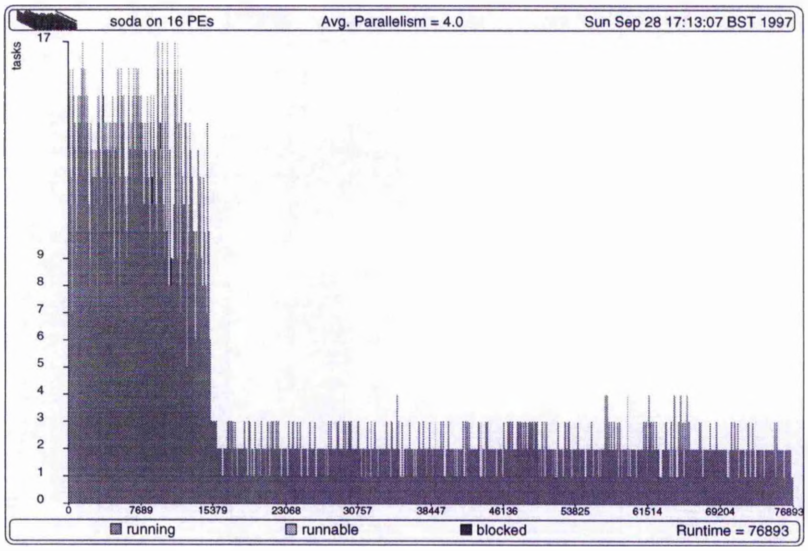

Figure 8.32: Parallelism profile for version 2.

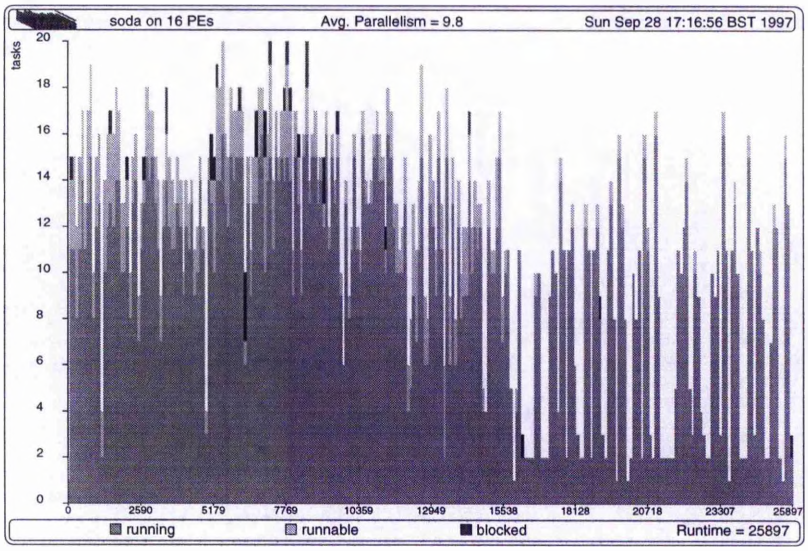

Figure 8.33: Parallelism profile for version 3.

average parallelism obtained on this version of the program on Naira and GrAnSim are 9.8 and 10 respectively.

As with the other benchmark programs, we ran this final version of the soda program on varying

```
parany f l = or (parmap f l)
forw = process(filter (parany (contains word) . fst)
              [(r, "right"), (d, "down"), (dl, "downleft"), (ul, "upleft")])
back = process(filter (parany (contains drow) . fst)
              [(r, "left"), (d, "up"), (dl, "upright"), (ul, "downright")])
```

Figure 8.34: Introducing speculation using parany.

*Benchmark:*soda	*Number of PEs*				
	1	2	4	8	16
Threads scheduled	1393K	1518K	1518K	1518K	1518K
Runtime (in machine cycles)	81949	65776	39490	30054	25896
Ready-evaluated tasks	204181	220610	221275	221617	221314
Space allocated	64483K	70580K	70570K	70565K	70570K
Average parallelism	1.0	1.8	3.5	6.2	9.8
Speedup	1.00	1.25	2.08	2.73	3.16
Efficiency	1	68%	57%	37%	22%

Table 8.7: **soda** statistics summary.

numbers of processors. The thread-based scheduler does not perform well on this benchmark for the same reasons as in the **coins** program of the preceding section. Table 8.7 summarises the runtime statistics obtained for soda.

The statistics summary in this table shows that, as in some of the benchmarks we have analysed earlier, more threads of execution are generated in the parallel program because of the modifications introduced in the parallelisation stages. It also shows that the number of ready-evaluated suspensions in the parallel program increases as the number of processors increases (except in the last column in the table). This increase is desirable in this program because it indicates some sharing as well as reducing the heap residency of the program. The measured speedup also increases as more processors are simulated.

Naira's performance on soda in comparison to the results of the other two simulators is encouraging given that no optimisations are made to the code generator. We envisage Naira to do better than this after optimising the current code generator which is somewhat naïve. In Section 9.2 we mentioned some of the optimisations we would like to consider for Naira beyond this research.

8.3.7 Thread versus quantum scheduling

As mentioned in Section 6.4 and the previous subsections we have experimented with a thread-based and a quantum-based scheduler. Using a thread-based scheduler, each simulated processor executes exactly one thread in a given machine cycle. Using the quantum scheduler, each processor executes a quantum of threads (the average number of threads per function call) in every machine cycle. The purpose of this section is to present a comparative analysis of these two scheduling schemes based on the benchmark programs analysed in Sections 8.3.4 to 8.3.6.

Recall from our discussions in Sections 5.2 and 6.3.2 on the runtime organisation of our compiler and parallel scheduling, respectively, that suspension pointers are globally accessible to the machine. Suspension objects inherit the processor identifier of their parent (i.e., the function frame/invocation in which they are part) and that the code associated with a function invocation is executed on the processor on which the frame is allocated. For the top level non-function value bindings in a module, these bindings are grouped together and allocated in a single 'module frame'. That is, they are considered as if they occur as local definitions within a single top level definition.

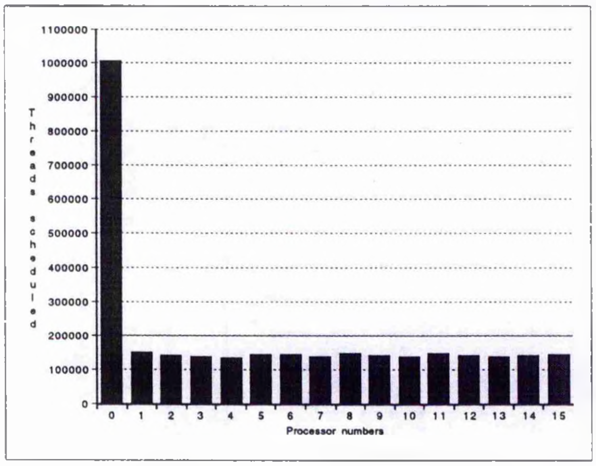

Figure 8.35: **coins 137** PE activity profile: thread-based scheduling.

When an input program contains many of these top-level non-function bindings all their constituent suspensions (except those that are part of a parallel task) share a common PID and are therefore scheduled on a particular processor. This can result in a disproportionate load sharing amongst the simulated processors of the machine as we demonstrate in this section.

To remedy this, some of these threads must be migrated to other processors. This migration is implemented by overriding the inherited PID (which determines where the suspension is evaluated) carried within a suspension with the PID of the current processor on which no runnable threads have been found during its turn to execute. A solution which minimises tests for work availability simply schedules threads round-robin rather than for a processor to retrospectively send a work request to its peers when it is its turn to execute and its runnable thread queue is empty. In our implementation of migration, a processor is eligible to supply work if it has runnable threads which are twice the quantum number or more.

Figures 8.35 and 8.36 show the comparative per-processor activity profiles using thread and quantum scheduling, respectively, for the `coins` benchmark program analysed in Section 8.3.4.

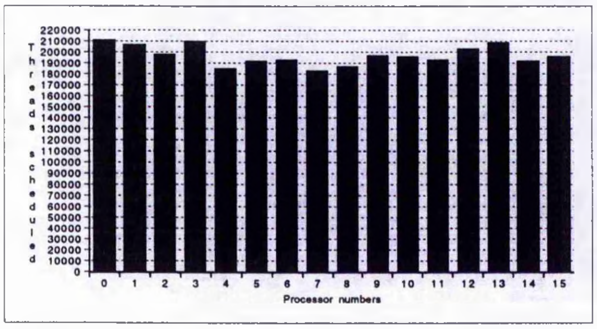

Figure 8.36: **coins 137** PE activity profile: quantum-based scheduling.

The load sharing exhibited in Figure 8.35, when the thread-based scheduler was used, is not as good as that for the case when the quantum-based scheduler was used as shown in Figure 8.36. This is due to the way threads in top-level non-function bindings within a module are scheduled as described above. Figure 8.35 indicates that all the suspensions for these bindings were scheduled on processor 0. A better load sharing can be achieved, as depicted in Figure 8.36, by ensuring that the threads are fairly scheduled and that a quantum of them are executed by each processor within a single cycle.

Next, we give similar comparative profiles based on processor activities during the runtime of the `matmul` program of Section 8.3.5. As explained in Section 8.3.5, this program multiplies a pair of 32 by 32 dense matrices represented as lists of lists of integers. The matrices data is bound at top-level as the complete program in Appendix A.6 shows.

As Figure 8.37 indicates all the suspensions for the matrices elements are evaluated on processor 0 making it execute far more threads than the other processors. Incorporating the thread migration strategy outlined above leads to a fairer load sharing equilibrium amongst the virtual processors as depicted by the profile in Figure 8.38.

Finally the profiles of Figures 8.39 and 8.40 are for the `soda` word searching program analysed

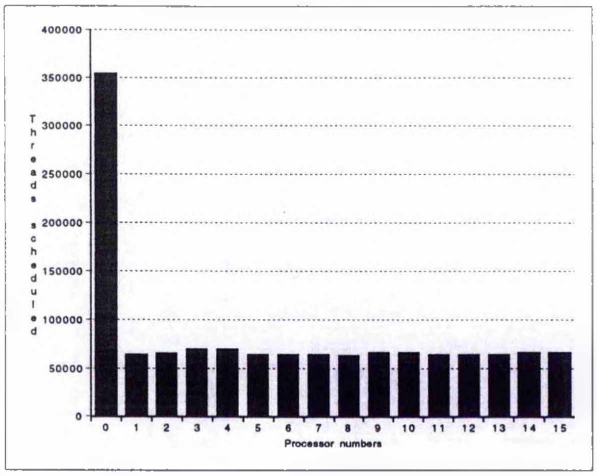

Figure 8.37: **matmul** PE activity profile: thread-based scheduling.

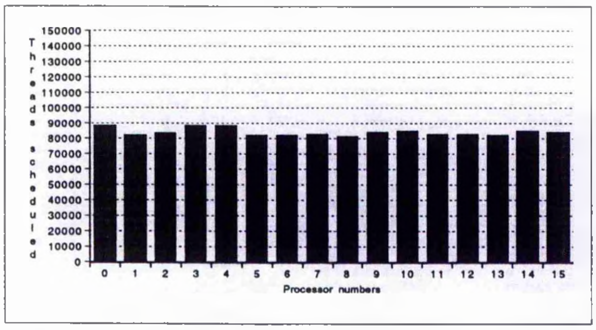

Figure 8.38: **matmul** PE activity profile: quantum-based scheduling.

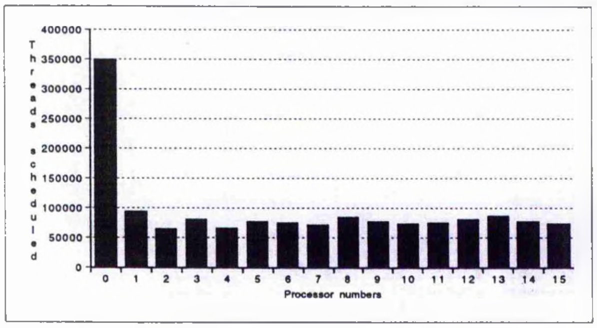

Figure 8.39: **soda** PE activity profile: thread-based scheduling.

in Section 8.3.6. The complete program (as taken from [RuWa95]) is given in Appendix A.7. The input data for this program consists of two lists of strings; one for the hidden words and the other for the rectangular grid of letters from which the hidden words will be matched.

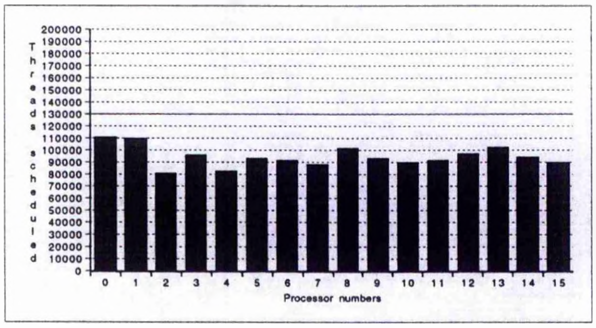

Figure 8.40: **soda** PE activity profile: quantum-based scheduling.

The top level binding which calls the functions that perform the meat of the computations in this program contains a few parallelism annotations which help to improve load sharing. The suspensions for the elements of the grid and hidden words lists are, however, executed on a particular processor as shown in Figure 8.39. As in the two preceding cases in this section, thread migration and quantum scheduling are used (see Figure 8.40) to improve the load equilibrium amongst the processors.

Our experiences from experimentation using both the GrAnSim (see Chapter 7) and Mizani simulators reveal that thread migration is essential for good performance on parallel systems. This has already been shown experimentally by [HaPe92] and theoretically by [BuRa94]. This adds to our confidence in the correctness of our experimental results. It is also clear from the analyses in this section, with insights from earlier results [LoHa96, HMP94], that large grained tasks (by way of quantum scheduling here) are tremendously important.

8.4 Summary

In this chapter we have conducted experiments and reported statistical measurements aimed at evaluating the performance of the generated multi-threaded parallel code from the Naira compiler. We achieved this by executing the compiled C code resulting from the source programs which contain our `process` and `value` annotations for introducing parallelism.

We considered a variety of popular benchmarking programs, as analysed in Sections 8.3.1 to 8.3.6, so as to exercise various aspects of the compiler, like its function call efficiency, arithmetic, data structures and closure handling operations. Our experiences with placing annotations in a bid to achieving maximum performance of programs reveal that the exercise can be extremely tedious and uncertain. This is because it is not always straightforward to determine an effective combination of annotations and the fact that adding or removing annotations may disturb the balance of others.

For the benchmarks we reported, namely `nfib`, `nqueens`, `tak`, `coins`, `matmul` and `soda`, we found that in general the effect on parallelism of using a deterministic or random distribution mechanism is insignificant. Of much more impact on parallelism is the scheduling policy we use. We used two different scheduling policies; one at thread level and the other at quantum level.

For the first three benchmarks (where the pattern of generating parallelism was regular), the average parallelism obtained using the thread-based and the quantum-based schemes is quite close. The difference in this case is in the number of runnable threads which is much higher for the cases using the thread-based scheduler than in the cases using the quantum scheduler. For the last three programs, on the other hand (where load sharing amongst the processors is uneven), the quantum-based scheduler performs much better.

The enormity of runnable threads generated by Naira, as exemplified by these benchmarks, can be usefully exploited by parallel machines with large numbers of processors and which support Arvind's twin fundamental issues of parallel computing—tolerance to unpredictable communication latencies and fast context-switching [ArIa87]. On the other hand, some form of throttling may be necessary for some programs in order to avoid the processors' context-stores from overflowing.

We found in our experiments that increasing or decreasing the quantum size (the average number of threads per function) and making each processor executes this number of threads in every machine cycle can lead to better performance in some cases. This can be viewed as some kind of *ad hoc* 'granularity analysis' mechanism used to maximise processor utilisation. It indicates the desirability of a systematic granularity analyser, along the lines of Hammond *et al* [HMP94], for effective parallel programming.

In comparison with the experimental results of other parallel implementations of functional languages, the performance of Naira is encouraging, even though our code generator and runtime system are yet to be optimised. At one level we used the idealised set-up of the highly tunable GrAnSim simulator to profile these programs and compare the results with those obtained using our simulator. By and large, the speedups obtained on the two simulators for the six programs are within small differences with each other (see Table 8.8). These similarities not withstanding the different shapes (in some cases) of the parallelism profiles obtained from the simulators which may be attributed to the different design choices in the two systems: while GHC uses passive task

Implementation	**PEs**	*Benchmark programs*							
		nfib		**queens**		**tak**		**matmul**	
		size	*speedup*	*size*	*speedup*	*size*	*speedup*	*size*	*speedup*
Naira	16	20	14.44	8	11.81	18 12 6	14.98	32	2.01
GrAnSim-Light	16	20	11.94	8	13.17	18 12 6	12.01	32	3.55
GrAnSim-Light	∞	20	18.72	8	24.25	18 12 6	21.03	32	3.47
Concurrent Clean	16	30	8.71	10	7.73	-	-	32	2.50
ZAPP	08	30	7.16	8	5.63	-	-	64	3.47
GRIP	20	20	17.00	-	-	-	-	-	-
HDG-machine	04	20	3.40	6	2.76	18 12 6	3.60	-	-
GAML	08	30	5.75	10	5.65	-	-	-	-
$< \nu, G >$ machine	15	30	8.00	10	9.50	-	-	-	-
PAM	12	20	10.63	8	9.44	-	-	10	7.70

Table 8.8: 'Comparative' results of different parallel implementations

distribution, evaluate-and-die synchronisation and unfair scheduling, Naira uses an active task distribution, notification model of synchronisation and fair scheduling.

At another level, we review the performance of some implementations[9] on the benchmarks we analysed as summarised in Table 8.8. The Concurrent Clean implementation on Transputers [Kess96], which generates efficient code and uses thresholds to control the grain size of computations, records speedups of 8.71 and 7.73 for `nfib 30` and `10 queens` respectively both on 16 processors. The implementation does not employ any load-balancing technique for these divide-and-conquer programs. They used a random task distribution mechanism over the network which somewhat balances the machine load if the number of processes is high enough.

Concurrent Clean records a speedup of 0.8 on 16 processors for multiplying 64 by 64 matrices represented by lists of lists of floating point numbers and a speedup of 2.5 for multiplying 32 by 32 matrices represented by 2-dimensional arrays of floating point numbers. The parallel speedups of the partial implementation of Concurrent Clean on ZAPP gives similar figures [Kess96].

[9] We note that these implementations, while employing graph reduction, are varied in some important ways: some are idealised simulators (Naira, GranSim-Light, PAM), while others are based on real shared memory (GAML, $< \nu, G >$-machine) and distributed memory (Concurrent Clean, ZAPP, HDG-machine) machines. The GRIP multiprocessor consists of features of both shared memory and distributed memory architectures.

The speedup figures for `nfib 30`, `8 queens` and 64 by 64 matrix multiplication on an 8-node ZAPP are 7.16, 5.62 and 3.47 respectively. The relatively slight improvement in performance on ZAPP over Concurrent Clean on `nfib` (speedup of 5.55 on 8 processors) is probably because of the special support for divide-and-conquer parallelism on ZAPP (ZAPP does not support other forms of parallelism, like stream processing, though).

Hammond and Peyton Jones [HaPe90] gave, amongst other things, a detailed analysis of the `nfib` program using various throttling strategies which are aimed to be exploited in other divide-and-conquer algorithms—probably the best-known parallel programming paradigm [THLP98]. They obtained a relative speedup of 17.0 for (a hand-tuned version of) `nfib 30` on GRIP with 20 processors and an absolute speedup of 5.34 on 6 processors.

As with our implementation, the HDG-machine based implementation (on Transputers) of Kingdon *et al* [KLB91] lacks a distributed garbage collector and therefore input sizes in their experiments were kept low. They recorded speedups on 4 processors of 3.4, 3.6 and 2.8 for `nfib 20`, `tak 18 12 6` and `6 queens` respectively. Maranget's G-machine based implementation of GAML [Mara91] reports relative speedups of 5.75 and 5.65 respectively for `nfib 30` and `10 queens` on 8 processors.

Augustsson and Johnsson's parallel implementation based on the $<\nu, G>$-machine [AuJo89b] measured speedups of 8 and 9.5 for `nfib 30` and `10 queens` respectively on a 15-node Syquent Symmetry^{TM}. The implementation by Loogen *et al* of PAM [LKID89] gave good speedups of 10.63, 9.44 and 7.70 for `nfib 20`, `8 queens` and 10 by 10 matrix multiplication on an OCCAM Transputer with 12 processors.

Table 8.8 summarises these somewhat comparative results[10] and a detailed review of each of those implementations (and more) whose measurements we quote in the table is provided in Chapter 3, our chapter on related work.

We note that the limitations of Naira as it stands now is that we are only able to consider programs whose results are of integer, string and Boolean or a list of values of these types. In particular, we have not yet experimented with floating-point problems and those based on complex numbers like the fast fourier transformation problem. In addition, the fact that our implementation does not support a garbage collector serves as an impediment to executing larger test programs.

Finally, we find it appropriate to sign the usual benchmarking caveat (especially when using idealised simulations) that the experimental results we report will be affected when the extra

[10]For the other two benchmarks not included in the table namely, coins and soda, we have already presented a comparative analysis between Naira's performance and those of [LoHa95, LoHa96a] and [RuWa95] (in Sections 8.3.4 and 8.3.6 respectively).

constraints imposed by a parallel machine, like communication and thread migration, are fully costed. Based on our experimental results using GrAnSim-Light and standard GrAnSim simulation of distributed memory architectures (in Chapter 7, see Table 7.10) these costs may affect our results by up to 30% to 35%.

Chapter 9

Conclusions

9.1 Summary of the thesis

In this thesis we have presented our research work on the design and implementation of a research compiler, Naira, for a rich, purely functional programming language. The research was aimed at, within the context of an SERC research project, improving the compile-time efficiency as well as the runtime efficiency of functional language compilers under a message-passing execution framework.

Broadly speaking, we identify three main issues of our work viz: a) the development of a pilot sequential compiler for the research b) writing a parallel code generator based on a compilation scheme specified using π-calculus and b) evaluating the overall compiler by parallelising the front and back ends and presenting experimental measurements.

The aim of this section is to briefly summarise the contents of the chapters of the thesis so as to highlight the *what* and *how* of our research.

In Chapter 1 we gave an overview of the thesis highlighting previous work from which our work directly took off while summarising our contributions. We have also introduced the two identity combinators which we used to introduce parallelism and to control the execution of our input programs. While introducing our `process` and `value` annotations, we compared them with the annotations used by other researchers, notably the par and `seq` annotations of GpH, Glasgow Parallel Haskell, which we used heavily in Chapter 7.

The subject matter of Chapter 2 was to give a brief historical overview of the persistent need of making computer programming easier. This quest for programming at a higher level of abstraction, while providing a language base with clean and simple semantics, culminated in the birth of functional languages. The chapter then explored some of the main advantages of programming in functional languages, outlines their computational foundation, their implementation techniques as well as pointing out the usual price paid by functional programmers in exchange for the many virtues of the languages that they exploit. We have also described π-calculus, the computational model that we used to specify our compilation scheme, while pointing out the necessary restrictions to the calculus that are required to guarantee the asynchronous message-passing communication of Naira.

In Chapter 3 we reviewed previous and ongoing research work on parallel implementations related to our research. The chapter started by isolating the main issues that require to be addressed carefully in a parallel implementation. The main alternative approaches to handling these issues were described while pointing out the advantages and disadvantages of each approach.

Our overview of related work covered researches based on the implicit, explicit and automatic methods of parallelisation. This is because even though the ways in which parallelism is introduced in these implementations may be different, they share other common characteristics with our implementation as summarised in Table 3.1.

Our survey revealed that Concurrent Clean has probably the most complete set of annotations since in addition to the usual user annotations (which change the evaluation order of a specific function application, which they call 'local annotations'), global annotations are generated by the compiler (in definitions of new types and in type specification of functions) which change the reduction order of all applications of a function. The language also includes annotations for graph copying and for specifying the destination of a parallel task. We also found that among the researches reviewed only the Glasgow Haskell research team seem to perform detailed analyses on issues like granularity, spark strategies and load control mechanisms.

The organisational structure and implementation of the front end of our research compiler was presented in chapter four. The chapter covers our choice of symbol table representations that are used in this part of the compiler. We then described the main features of the five major components of the compiler front end.

We have also described the optimised intermediate language produced by the compiler front end. We found it beneficial to optimise the intermediate language so that the size of the parallel code that is eventually generated is substantially reduced and the runtime execution costs also minimised with it.

The contents of chapter five include a description of the idiosyncracies of the parallel machine model that best suits our compilation system, a detailed explanation of our representation of data objects in the heap, an overview of the different kinds of messages we support and a description of our asynchronous message-passing model. We have also covered in this chapter a description of the C parser which interfaces the front and back ends of the compiler by transforming the Haskell structured intermediate code into equivalent C structures suitable for input to the code generator.

This chapter also presented in some reasonable details the organisation of our code generator and then described the code generation process for each of the main constructs of the intermediate language. As the compilation schemes are presented, the relevant correspondences between them and their π-calculus counterparts are drawn.

Finally the chapter concludes with a description of how the separately compiled modules of a program are linked, the execution of the program triggered and the program value printed (or a runtime error generated if the program is buggy).

The basic theme of chapter six was a description of our implementation of stream I/O, the implementation of comparison operations over values of user-defined algebraic data types (to compensate for our non-support for type classes) and scheduling issues. We have also included a description of our quasi-parallel simulator which takes snapshot statistics during execution, writes the numeric information into data files and then generate postscript graphical profiles from this data. The PostScript-graph generator is adapted from the GrAnSim simulator to work for our data formats.

In chapter seven we have presented the parallelisation and performance analysis of the front end of our compiler. Our parallelisation process uses the evaluation strategies of GpH, Glasgow Parallel Haskell, and adopts the top-down parallelisation methodology proposed by Trinder *et al.* [THLP98]. The compiler was also analysed on real hardware with good resulting speedups.

We used GrAnSim, a fairly accurate simulator with a rich set of runtime system options that enable the simulation of a wide variety of different architectures, in our experimental measurements in this chapter. As our compiler is targeted to run on distributed memory machines without shared memory, as mentioned in chapters one and five, we have conducted our experiments using a GrAnSim configuration specific to these architectures. We have also used other setups and reported measurements for comparison purposes and as a means of 'debugging' the sources of runtime overheads in our experiments.

Our top-down parallelisation involved parallelising the top-level pipeline of the compiler phases by defining evaluation strategies on the complex data structures (parse trees) produced by the phases. We then parallelised four of the individual phases in isolation with the others (i.e., setting the others to run sequentially). We measured an average parallelism and speedup of up to 3.7 and 3.58, respectively, on distributed memory machines with 8 processors at the top-level parallelisation stage. Speed-ups for the individual phases were about 25% each for the pattern matcher, lambda lifter and the optimiser and up to about 350% for the type checker.

Having parallelised the constituent phases in isolation, we finally activated all our parallelisation code within these phases in order to evaluate their overall effect on the parallel behaviour of the compiler. Our experiences with the parallelisation at this stage revealed that it is now harder to understand and to predict the parallel behaviour of the compiler. This is because bits of the strategic code are in different places in the compiler source and changing one strategic code may disturb the balance of others leading to dramatic change in performance for some inputs. In our overall parallelisation of the compiler we obtained speedups of up to 5.53 on eight processors.

In comparison with the results of other projects on parallelising large-scale lazy functional programs [THL+96, THLP98], our experimental results indicate very high potential for parallelism in our compiler. In their parallelisation of Traffic Accidents Case-study [THL+96], Trinder recorded an average parallelism of 3.6 on an idealised machine. In their experiments on Lolita [THLP98], a natural language processing system, they reported an average parallelism of about 3.1 on GrAnSim emulating a shared memory 4-processor machine.

Despite our good simulated performance we note that Naira is an experimental compiler rather than a state-of-the-art optimised compiler like, for example, the Glasgow Haskell Compiler which is optimised for *sequential* compilation and which may thus be less susceptible to parallelisation. Lately, our experiments with Naira on real hardware (a network of Sun workstations running under a PVM communication harness) demonstrated similar speedups as it did under the GrAnSim simulator.

In our second chapter of experimentation with Naira, Chapter 8, we considered six simple popular benchmark programs and used them to evaluate the performance of the generated parallel code. While we analysed the efficiency of the parallelised front end of the compiler in Chapter 7, we measured the performance of our parallel code generator and runtime system in Chapter 8.

During the analysis of each of our chosen benchmark programs, we experimented with two different task distribution mechanisms and two different scheduling policies. The different schemes were used in order to carry out a moderately rigorous analysis of the benchmarks and to isolate (up to annotations subtleties) the best parallelised version of a program. Results of our experiments were summarised in tabular as well as in graphical forms. The tabular statistics (which is obtained from the best parallelised program) contains more detailed information on the parallel behaviour of each benchmark on different machine configurations.

Our experimental results revealed that the choice between a deterministic and random task distribution scheme has little effect on the parallel performance of a program provided there is some reasonable parallelism in the program. The choice of scheduling policy on the other hand, was found to have much more impact on performance especially in programs with irregular granularity.

We found that the scheduling scheme in which a quantum of threads is executed on each virtual processor in a single time-slice is, in general, better than the one in which only a single thread is executed on each processor in a time-slice. Our experiments (in both Chapters 7 and 8) revealed that task migration, even on high latency distributed memory machines, is indispensable in some cases in order to get better results.

Finally, we compared our experimental results from Naira with those of other parallel implementations of functional languages. While none of these other implementations has provided performance results for all our benchmarks (in fact some implementations, like [GoHu86, HaPe90], concentrated on a single benchmark and used it to study particular issues in detail, like diffusion heuristics and throttling strategies), we found it appropriate to evaluate Naira vis-a-vis these implementations.

Result of the overall comparisons puts Naira on a positive footing given that no optimisations have yet been implemented on the code generator and the runtime system. We should point out, however, that these comparisons should be taken with all the different (and diverse) details on board. For example while our results are simulated, others were obtained from interpreted abstract machine code and some are obtained from shared memory (e.g., [AuJo89b]) and others from distributed memory (e.g., [HaPe90]) machines.

9.2 Summary of contributions

In this section we give a brief summary of the main contributions of the thesis. These are

- *Design and implementation of a parallel functional language compiler, Naira.* As mentioned in the preceding chapters, Naira processes a rich subset of the full Haskell language. This Haskell subset serves as both the implementation language for Naira as well as the specification for valid programs acceptable to the compiler. Stream I/O and a parallel runtime system for the compiler have been implemented together with the other contributions enumerated in this section.

- *Extension and implementation of many compile-time program analyses.* After lexical and syntax analyses, the research concentrated on the implementation of four main other compiler phases—pattern matching, lambda lifting, type inference and intermediate language optimisations—which were subsequently parallelised. The pattern matching compiler is based on Wadler's algorithm described in [Peyt87] which we extended to incorporate some important transformations (Section 4.2.3). After pattern matching, alternatives in case-expressions consist of variable patterns or constructor patterns with variable subpatterns only.

The lambda lifter is based on that of Johnsson [John87] which forms and solves equations to compute complete sets of free variables of functions. The main differences with our lambda lifter is that we do not lift functions-turned-combinators to the top level. This provides opportunities for local optimisations (for example inlining) of the local combinators while not increasing the number of global identifiers and the cost of their house-keeping therewith. The non-lifting of local combinators reported here is similar to what Peyton Jones does in the STG language [Peyt92] except that while STG identifies the free variables it does not pass them as extra arguments and a two-level environment—one for free variables and the other for arguments—is maintained. This two-level environment reduces the movement of values from the heap to the stack, but it is no clear whether this is a big improvement or only a marginal one [Peyt92]. In contrast to the lambda lifting algorithms of Hughes [Hugh83], Peyton Jones and Lester [PeLe91] and the lambda hoisting algorithm of Takeichi [Take88], our algorithm, in similarity with Johnsson's, does not incorporate the full-laziness optimisation.

The type inference algorithm implemented here draws ideas mainly from [Read89, ReCl91, Peyt87, FiHa88]. The basic Milner-Damas type inference algorithm usually given in the literature does not contain inference rules for general **let**-expressions and lambda abstractions involving patterns nor are rules for inferring the types for other λ-calculus embellishing constructs like `case` and conditional expressions. A complete type inference algorithm involving the inference rules for the general expression forms in the previous sentence has been implemented for Haskell and with parallelisation ideas in mind. An important aspect of the implementation is that substitutions are applied lazily and only when needed rather than applying substitutions eagerly to update types in the type environment after each inference step. This is similar to what is proposed by Read in [Read89] and Paulson in [Paul91] probably with the same motivation for minimising the cost of type inference. Notice that applying substitutions eagerly (which is avoided in this implementation) will increase parallelism (since the substitutions can be applied to the type environment at the same time the main inference computation is going on) but decrease speedup because of the extra work performed due to the eagerness.

Resulting from the aforementioned analyses is a simple optimised intermediate language suitable for efficient parallel code generation. In this language, all constructors and primitive operators, as in the STG language of Peyton Jones [Peyt92], are saturated but unlike STG **case**-expressions in this intermediate language are much more simplified (Section 4.3.4) thereby avoiding the cost of maintaining local environments for variables in the patterns of case-alternatives as described in [Peyt92]. The language FLIC (Functional Language Intermediate Code) proposed by Peyton Jones and Joy has a similar representation for case-expressions except that only the constructor names are represented by small integers and their variable subpatterns are not compiled out [PeJo90, PeLe91].

- *Extensive application of the parallel programming technology of Trinder et al [THLP98].* A wealth of experience has been built from using evaluation strategies in small-sized programs where the actual workings of the technology has been explored and in the parallelisation of the benchmark programs of Chapter 8 as well as applying the technology to parallelise Naira itself. To the author's knowledge, Naira is the second largest parallel program written in a lazy, purely functional language following the Lolita natural language processor [LoTr97]. Unlike Lolita, the design, implementation and parallelisation of Naira was not conducted under a joint collaborative research but is the entire work of a single author. Furthermore, we recorded a much higher absolute speedup for Naira than the absolute speedup obtained by researchers on Lolita [LoTr97].

- *Design and implementation of a parallel name-server.* Allied to the use of evaluation strategies to exploit parallelism is the use of a parallel name-server which creates unique names to enable otherwise data-dependent computations to proceed in parallel. A modestly efficient parallel name-server has been designed and used which maintains the simplicity and avoids the problems of overflow and inefficiency associated with other proposed name-servers by Hancock in [Peyt87] and Augustsson *et al* in [ARS94].

- *Compiling a lazy, purely functional language via π-calculus.* Naira is the first parallel compiler (for lazy functional languages) that generates parallel code using compilation rules specified using an asynchronous π-calculus. There is a high-level concurrent programming language, Pict, proposed by Pierce and Turner [PiTu97], which is based on the π-calculus and which also translates into a π-calculus core language. Thus while Pict is based entirely on the π-calculus, Naira combines the best of the two worlds of the λ-calculus and the π-calculus. The Pict compiler performs all of its static analyses of programs, optimisations, and code generation using a π-calculus core language while Naira's static analyses are based on an enriched λ-calculus and the code generation is based on the π-calculus. The Pict compiler, like Naira, compiles to C in similar continuation-passing style and, although there are no concrete performance results provided for the Pict compiler, the behaviour of its generated code is likely to be similar to that of Naira since they are based on the same code generation and runtime execution philosophy.

Dataflow languages usually compile via dataflow graphs [Nikh89,ArNi90] or dual graphs [Trau91]. Although aspects of Naira's runtime implementation (like asynchronous message-passing, function call mechanism and frame management) are similar to those in the implementation of Id (see Section 3.4 for differences), Id compiles into dataflow graphs which are directly executable machine code [ArNi90]. The role played by *tokens* (along which data values are carried between operators) in the dataflow setting is strikingly similar to the role played by channel names in π-calculus except that the latter are at a higher level of abstraction (compare the general format of tokens in Section III of [ArNi90] with the `sendMessage` in Section 5.4).

- *Design and implementation of the* **process** *and* **value** *annotations.* The design, implementation and demonstration of the use of the **process** and **value** annotations, as a vehicle for specifying parallelism and strictness in user programs, have successfully been realised in this research. Performance measurement figures reported (in Chapter 8) indicate good parallel behaviour (of the benchmarks) resulting from the use of these annotations based on an idealised simulator.

- *Extension of Ostheimer's π-calculus-based compilation scheme for a first-order functional language to cover an expressive higher-order functional language.* Ostheimer's work was first extended to cover a complete first-order language by adding compilation rules for code-generating **case**-expressions, individual modules and complete programs. It was then significantly enhanced by adding rules for higher-order functions. A working implementation of the complete rules is first provided in this thesis (Chapters 1, 2, 5, 8).

- *Generating multi-threaded parallel code based on the extended compilation scheme.* Because of the laziness of Haskell and the need to tolerate long communication latencies, programs are compiled into a large number of threads (in this implementation) which specify a compile-time instructions ordering valid for all contexts in which a function can be called (as in [CSS+91]). Furthermore, the order-independent way in which threads can be executed can lead to some space-saving efficiency when threads specifying the computations of large suspension objects with relatively smaller values are executed early (see analyses of the results in Sections 8.3.1–8.3.6). Our parallel code generator also automatically determines frame sizes for functions thereby avoiding extra analyses like the stacklessness analysis of Lester [Lest89].

- *Achieved good absolute speedups on both simulated and real hardware.* The parallel compiler is successfully assessed using the latest technology both on simulators and on real hardware.

A wall-clock speedup of 2.46, and a relative speedup of 2.73 have been measured on a network of five workstations which confirm the speedup of 3.01 predicted by the GrAnSim simulator.

Assessment of the parallel code generated by Naira in Chapter 8, based on typical small-sized programs, gave good results in comparison with the results of other similar parallel implementations (Section 8.4). To the author's knowledge, Naira is the first parallelising compiler for a lazy purely functional language which itself runs in parallel. Furthermore, the absolute speedup obtained when measuring Naira has not been achieved for a similarly large, irregular parallel lazy functional program.

9.3 Limitations of the thesis

Having presented the major contributions of the thesis in the previous section, this section outlines the main limitations of the work.

- *Naira does not compile itself.* One of the long-term aims of Naira is to make it as a complete stand-alone system. This has not been realised within the time-frame of this research. A major contributory factor to this being the absence of a (distributed) garbage collector which is, in its own right, a challenging research topic. Consequently, Naira is not self-compiling but is, most importantly, successfully parallelised using a 'foreign' system (Chapter 7).

- *The class of admissible program values in Naira is restricted.* Although the language Naira compiles is polymorphic, valid programs in Naira are, at the moment, restricted to have the basic integer, character and Boolean values or lists and tuples of these. Clearly, more work needs to be done in this area in order to compile more real-life applications and so that a more far reaching assessment of the compiler can be given.

- *The compiler's own simulator, Mizani, is an idealised simulator.* The experimental results in Chapter 8 are to be accepted within the limitations of an idealised simulator. The guiding hypothesis in using quasi-parallel simulators of this form is to find indicators for the necessary conditions for the availability of exploitable parallelism in a parallel application. A notable limitation of Naira's own simulator is the assumption that communication has zero cost. Because of this unrealistic assumption in a message-passing system, the experimental results presented in that chapter are likely to have been exaggerated by upto 35% based on experiments with GrAnSim (Section 7.9) and with GUM (Section 7.10).

- *Load-bounding.* Complete parallel systems usually embody some form of load control mechanism to help the runtime system manage the eventual problem of 'parallelism explosion' (see

Section 3.4) where too much parallelism can generated, if not controlled, beyond machine resources. Naira consists only of a simple *ad hoc* mechanism for possible load control based on quantum scheduling. A more systematic load bounding scheme is needed when large, highly parallel applications are considered.

- *Scalability.* An important issue not (fully) addressed in this thesis is the analysis of the scalability of the parallel code generated by Naira. Realistically, scalability is a property that should be expected from large, non-trivial, regular parallel programs. Although Naira does not measure such programs, attempts have been made to demonstrate scalability using the GrAnSim simulator. This is done by combining several constituent modules of Naira into a single input module and using it to simulate distributed machines of different sizes (i.e., with different numbers of processors). This is ongoing work and we hope to report details of this and other related issues elsewhere [JuTr98].

9.4 Further work

The aim of this section is to briefly mention some optimisations and other research issues which we hope to pursue and which when implemented will greatly improve the robustness and performance of Naira. These issues span the various aspects of the compiler from language constructs in the front end to code generation and runtime system issues in the back end.

First at the language level, there are some minor constructs, like user-introduced fixity definitions, which we want to implement fully. Although these do not affect parallel behaviour, implementing them will improve the user-friendliness of our language base.

One issue which has a direct bearing on expressing parallelism and which we want to investigate further is the suitability of our `process` and `value` annotations in specifying dynamic behaviour. For small and medium-scale programs these annotations can be used to conveniently parallelise programs. For larger programs involving aggregate data values, however, the parallelisation process is not as convenient because forcing functions have to be defined over such values to specify the desired dynamic behaviour. We hope to overcome this limitation, as part of our further research plan on Naira, by extending our annotations along the lines of Trinder's evaluation strategies [THLP98] which are higher-order annotation that can be used to cleanly parallelise programs.

In our experiences with lists and binary tree data structures in Chapter 7, we realised that using sorted binary trees to maintain some of our symbol tables was computationally more expensive than using ordinary lists. This is largely due to the extra computations (usually string comparisons which are represented as lists of characters) required to maintain the sorted trees. We hope to use compact strings, as in some modern highly optimised functional languages compilers, to minimise the cost of string handling so that balanced binary trees can be used to provide faster accesses at minimal costs.

Our symbol table implementation in the code generator, as described in Chapter 5, is based on a linked-list data structure. We plan to use a more efficient implementation based on hash-tables. More importantly, we aim to modify the code generator so as to reduce the size and improve the quality of the generated parallel code.

There are a few improvements and extensions we would like to incorporate into the runtime system. Naira's simulator, Mizani, can be enriched to cost interesting runtime aspects like communication and other thread management issues, as does GrAnSim. As described in Chapter 6, the type of program values that Naira can admit is limited. A possible way to remove this limitation is to use runtime tags to distinguish different types of data values.

As mentioned in Chapters 1 and 5, suspension objects are global to the machine in our implementation and are allocated in the heap. Some form of sharing analysis which can distinguish shared suspensions from unshared ones can provide a cheap way of deforesting the heap thereby tremendously reducing the heap residency of Naira programs.

We propose to implement, in software, the probablistic load bounding mechanism described by Ostheimer [Osth93] in such a way that we can integrate it into our compilation scheme. This will provide a more systematic control over parallelism than the *ad hoc* task coalescing scheme we experimented with in Chapter 8. Finally, a distributed garbage collector is needed in Naira to provide it the required support to fully manifest its potentials.

9.5 Concluding remarks

The research presented in this thesis was on the design, implementation and evaluation of a parallel, parallelising compiler for a rich, purely functional programming language. The compiler was successfully parallelised, using state-of-the-art, fairly accurate tools with rewarding speedups of up to about 6 on 8 processors. Similar speedups were recorded for Naira when run under real machine hardware (i.e., an Ethernetted collection of Sun workstations running under Solaris 2 operating system). The compiler was also found to have high parallelising potentials based on typical, popular test programs. We hope that the directions we explored will help focus any parallelisation efforts in engineering large software.

As a post-mortem, the author notes that Software Engineering as a trade has the potentials of being a mixture of agonies and ecstasies— it is really self-satisfying to define a task and see through its execution especially when the road to the fulfillment of such a task was marred by bare hindering milestones which one can, now and in the future, refer to in order to reappraise one's efforts for achieving the accomplished task!

The research objectives set out and achieved in this thesis, so broad as they are have, to a great extent, defined and polished so much life experiences that the author lives to enjoy in this life and beyond. The English maxim 'no pain no gain' and the adage 'the proof of the pudding is in the eating', really make a lot of sense in many contexts in the frames of life!

Appendix A

Source code for the benchmarks.

```
typedef struct Exp {
  ExpTag tag;
    union {
       struct {  int val;                        } intLitExp;
       struct {  int val;                        } charLitExp;
       struct {  char *id;                       } idExp;
       struct {  struct Exp *exp;                } negExp;
       struct {  struct Exp *exp;                } notExp;
       struct {  struct Exp *exp;                } ordExp;
       struct {  struct Exp *exp;                } chrExp;
       struct {  struct Exp *left;
                 struct Exp *right;              } primPlusIntExp;
       struct {  struct Exp *left;
                 struct Exp *right;              } primEqIntExp;
       struct {  struct Exp *left;
                 struct Exp *right;              } primLeIntExp;
       struct {  struct Exp *left;
                 struct Exp *right;              } primMulIntExp;
       struct {  struct Exp *left;
                 struct Exp *right;              } primDivIntExp;
       struct {  struct Exp *left;
                 struct Exp *right;              } primRemIntExp;
       struct {  struct Exp *left;
                 struct Exp *right;              } primAndExp;
       struct {  struct Exp *left;
                 struct Exp *right;              } primOrExp;
       struct {  int tag;
                 struct MexpList *components;    } constrExp;
       struct {  struct Exp *exp;
                 int selector;                   } selectExp;
       struct {  struct Exp *testExp;
                 struct ChoiceList *choiceList;  } caseExp;
       struct {  struct Exp *condition;
                 struct Exp *consequent;
                 struct Exp *alternative;        } ifExp;
       struct {  struct FunList *funs;
                 struct Exp *body;               } defineExp;
       struct {  String name;
                 int    missing;
                 struct MexpList *args;          } closeExp;
       struct {  String name;
                 struct MexpList *args;          } callExp;
       struct {  struct Exp *fun;
                 struct MexpList *args;          } applyExp;
       struct {  struct DefList *defs;
                 struct Exp *body;               } letExp;
       struct {  struct Exp *exp;                } errorExp;
       } fields;
} Exp;
```

Figure A.1: C structure for expressions

```
nfib n = if n <= 1 then 1 else 1 + n1 + nfib(n-2)
    where n1 = process(nfib (n-1))
```

Figure A.2: Code for nfib.

```
find rank file n board =
      if file > n then 0 else if rank > n then 1 else
      let fs = process(first rank file n board)
      in fs + find rank (file+1) n board
find1 rank file n board =
       if file > n then 0 else if rank > n then 1 else
       first rank file n board + find1 rank (file+1) n board
first rank file n board = fc
      where c  = process(compatible rank file (rank-1) board)
            fc = value(firstcond rank file n board c)
firstcond rank file n board False = 0
firstcond rank file n board True  = find (rank+1) 1 n (file:board)
compatible rank file row []    = True
compatible rank file row (h:t) =
       if h == file || absVal h file == absVal row rank then False
       else compatible rank file (row-1) t
absVal a b = if a > b then a-b else b-a
val = find 1 1 8 []
```

Figure A.3: Code for nqueens.

```
tak x y z  = if x <= y then z else
             tak (process(tak (x - 1) y z))
                 (process(tak (y - 1) z x))
                 (process(tak (z - 1) x y))
val = tak 18 12 6
```

Figure A.4: Code for tak.

```
mynub []         = []
mynub (d:ds)     = d:mynub(removeDuplicates d ds)
parmap f []      = []
parmap f (x:xs)  = process(f x) :value(parmap f xs)
del [] _         = []
del (x:xs) y     = if x == y then xs else x:del xs y
pay_num :: Int → Int → [Int] → Int
pay_num _ 0   coins   = 1
pay_num _ val []      = 0
pay_num pri val coins =
          sum (parmap (aux pri val bar_coins) (process(mynub bar_coins)))
    where bar_coins = value(dropWhile (>val) coins)
aux pri val coins1 c    =
     pay_num (pri-1) (val-c)(process(del (dropWhile (>c) coins1) c))
res = pay_num 100 val coins
vals = [250, 100, 25, 10, 5, 1]
quants = [5, 8, 8, 9, 12, 17]
coinsz = (zip vals quants)
coins = concat (map (\(v,q) → [v | i ← [1..q]]) coinsz)
val = 137
```

Figure A.5: Code for coins.

```
sel (h:t) 1 = h
sel (h:t) n = sel t (n-1)
foldint l u f g = if l == u then f l else
                      g (process(foldint l mid f g))(process(foldint (mid+1) u f g))
    where mid   = process(div (l+u) 2)
foldint1 l u f g = if l == u then f l else
                      g (process(foldint1 l mid f g))(process(foldint1 (mid+1) u f g))
    where mid   = process(div (l+u) 2)
matprod mA n    = foldint 1 n (row mA n) (++)
row mA  n i     = eagerCons (process (foldint 1 n (sprod mA n i) (++))) []
sprod mA  n i j = eagerCons ((foldint1 1 n (mult mA i j) (+))) []
mult mA  i j k  = sel(process(sel mA i)) k × sel(process(sel mA k)) j
val = plength(process(matprod matA matA))
```

Figure A.6: Code for matmul.

```
matA =
   [[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32],
   [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0],
   [3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1],
   [4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2],
   [5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3],
   [6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4],
   [7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5],
   [8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6],
   [9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7],
   [10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8],
   [11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9],
   [12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10],
   [13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11],
   [14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12],
   [15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13],
   [16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14],
   [17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
   [18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],
   [19,20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17],
   [20,21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18],
   [21,22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
   [22,23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20],
   [23,24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21],
   [24,25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22],
   [25,26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23],
   [26,27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24],
   [27,28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25],
   [28,29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26],
   [29,30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27],
   [30,31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28],
   [31,32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29],
   [32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30]]
```

Figure A.7: Data for `matmul`.

```
main resps = [AppendChan stdout (process(showInt val))]
matprod mA mB = rowlistprod mA (process(transp mB))

rowlistprod [] cols = []
rowlistprod (row:rows) cols =
   --eagerCons(rowprod row cols)(rowlistprod rows cols)
   (process (rowprod row cols)) : (value(rowlistprod rows cols))

rowprod row [] = []
rowprod row (col:cols) =
     --eagerCons (dotprod row col)(rowprod row cols)
     (process(dotprod row col)) : (value(rowprod row cols))

dotprod [] []          = 0
dotprod (x:xs) (y:ys) = --x×y + dotprod xs ys
  let u = process(x×y)
      v = process(dotprod xs ys)
  in u + v

transp ([] :rows) = []
transp rows       =
   (process(headcol rows)) : (value(transp (process(tailcols rows))))

headcol [] = []
headcol ((x:xs) :rows) = eagerCons x (process(headcol rows))

tailcols [] = []
tailcols ((x:xs) :rows) = eagerCons xs (process(tailcols rows))

val = plength(process(matprod matA matA))
plength l = psum (parmap length l)

parmap f [] = []
parmap f (h:t) = (process(f h)) : (value(parmap f t))

psum l    = foldl (+) 0 l

foldl f a []      = a
foldl f a (x:xs) = let --fax = process(f a x)
                       ans = value(foldl f (process(f a x)) xs)
                   in ans
```

Figure A.8: Alternative code for `matmul`.

```
pAppend l1 l2 = pfoldr tailCons l2 l1
pfoldr f z []     = z
pfoldr f z (x:xs) = ans
    where rest = process(pfoldr f z xs)
          ans  = value(f x rest)
tailCons x xs = x:value xs

diagonals [r]    = map (: []) r
diagonals (r:rs) = zipinit r ([] : diagonals rs)
parmap f []     = []
parmap f (x:xs) = process(f x) :value(parmap f xs)
zipinit [] ys       = ys
zipinit (x:xs) (y:ys) = (x:y) :zipinit xs ys
contains xs ys = any (prefix xs) (suffixes ys)
suffixes [] = []
suffixes xs = xs:suffixes (tail xs)
prefix [] ys = True
prefix xs [] = False
prefix (x:xs) (y:ys) = x == y && prefix xs ys
transpose ([] :rows) = []
transpose rows       =
  (process(headcol rows)) : (value(transpose (process(tailcols rows))))
headcol []             = []
headcol ((x:xs) :rows) = x: (process(headcol rows))
tailcols []             = []
tailcols ((x:xs) :rows) = xs: (process(tailcols rows))
forceList []     = []
forceList (x:xs) = value x:forceList xs
parany f l = or (parmap f l)
val = concat (parmap find hidden)
  where
    find word = (word 'append' " " 'append' concat dirs 'append' "\n")
     where
         dirs = value(map snd (forw 'append' back))
         drow = (reverse word)
         forw = process(filter (parany (contains word) . fst)
              [(r, "right"),(d, "down"), (dl, "downleft"), (ul, "upleft")])
         back = process(filter (parany (contains drow) . fst)
              [(r, "left"),(d, "up"), (dl, "upright"), (ul, "downright")])
    r  = (grid)
    d  = process(init (transpose grid))
    dl = process(diagonals grid)
    ul = process(diagonals (reverse grid))
```

Figure A.9: Code for soda.

```
grid =
  [['Y', 'I', 'O', 'M', 'R', 'E', 'S', 'K', 'S', 'T'],
   ['A', 'E', 'H', 'Y', 'G', 'E', 'H', 'E', 'D', 'W'],
   ['Z', 'F', 'I', 'A', 'C', 'N', 'I', 'T', 'I', 'A'],
   ['N', 'T', 'O', 'C', 'O', 'M', 'V', 'O', 'O', 'R'],
   ['E', 'R', 'D', 'L', 'O', 'C', 'E', 'N', 'S', 'M'],
   ['Z', 'O', 'U', 'R', 'P', 'S', 'R', 'N', 'D', 'A'],
   ['O', 'Y', 'A', 'S', 'M', 'O', 'Y', 'E', 'D', 'L'],
   ['R', 'N', 'D', 'E', 'N', 'L', 'O', 'A', 'I', 'T'],
   ['F', 'I', 'W', 'I', 'N', 'T', 'E', 'R', 'R', 'C'],
   ['F', 'E', 'Z', 'E', 'E', 'R', 'F', 'T', 'F', 'I'],
   ['I', 'I', 'D', 'T', 'P', 'H', 'U', 'B', 'R', 'L'],
   ['C', 'N', 'O', 'H', 'S', 'G', 'E', 'I', 'O', 'N'],
   ['E', 'G', 'M', 'O', 'P', 'S', 'T', 'A', 'S', 'O'],
   ['T', 'G', 'F', 'F', 'C', 'I', 'S', 'H', 'T', 'H'],
   ['O', 'T', 'B', 'C', 'S', 'S', 'N', 'O', 'W', 'I']]

hidden = ["COSY", "SOFT", "WINTER", "SHIVER", "FROZEN", "SNOW",
          "WARM", "HEAT","COLD", "FREEZE", "FROST", "ICE"]
```

Figure A.10: Data for soda.

Appendix B

Acronyms in the Bibliography

1	ACM	—	Association of Computing Machinery.
2	BCS	—	British Computer Society.
3	CONPAR	—	Conference on Parallelism.
4	FPCA	—	Functional Programming and Computer Architecture.
5	GlaFP	—	Glasgow workshop on Functional Programming.
6	HPFC	—	High Performance Functional Computing.
7	IEEE	—	Institute of Electrical and Electronic Engineers.
8	IFL	—	Implementation of Functional Languages.
9	LNCS	—	Lecture Notes in Computer Science, Springer-Verlag.
10	PARLE	—	Parallel Architectures and Reduction Languages, Europe.
11	PIFL	—	Parallel Implementation of Functional Languages.
12	POPL	—	Principles of Programming Languages.
13	SIGPLAN	—	ACM Special Interest Group on Programming Languages..
14	TOPLAS	—	ACM Transactions on Programming Languages and Systems.
15	WICS	—	Workshop In Computing Science, Springer-Verlag.

Bibliography

[ANP89] Arvind, RS Nikhil, and KK Pingali, *I-Structures: Data Structures for Parallel Computing.* ACM TOPLAS, **11**(4), pp. 598-632, October 1989.

[ARS94] L Augustsson, M Rittri and D Synek, *On Generating Unique Names.* Journal of Functional Programming, **4**(1), pp.117–123, January 1994.

[ASU85] AV Aho, R Sethi, and JD Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, December 1985.

[AbSy85] S Abramsky and R Sykes, *SECD-M: A Virtual Machine for Applicative Multiprogramming.* In FPCA, Nancy France, Springer Verlag, 1985. Springer Verlag LNCS 201, pp. 81-98, September 1985.

[Abra82] S Abramsky, *SECD-M: A Virtual Machine for Applicative Multiprogramming.* Computer Systems Laboratory, Queen Mary College, University of London.

[AcHa95] P Achten and R Plasmeijer, *The Ins and Outs of Clean I/O.* In Journal of Functional Programming, **5**(1), pp. 81-110, January 1995.

[Acht91] P Achten, *Annotations for Load Distribution.* In Proc. of PIFL 1991, Southampton, Appeared as Technical Report CSTR 91-07, University of Southampton, 1991.

[Appe92] A Appel, *Compiling with Continuation.* Cambridge University Press, 1992.

[ArBr84] Arvind and JD Brock, *Resource Managers in Parallel Programming.* Journal of Parallel and Distributed Computing, **1**, pp. 5-21, 1984.

[ArIa87] Arvind and RA Iannucci, *Two Fundamental Issues in Multiprocessing.* In Proc. of 4th Int. DFVLR Seminar on Parallel Processing in Science and Engineering, Bonn, FRG, Springer Verlag LNCS 295, June 1987.

[ArNi90] Arvind, *Executing a Program on the MIT Tagged-Token Dataflow Architecture.* IEEE Transactions on Computers, **39**(3), pp. 300-318, March 1990.

[Arvi92] Arvind, *General Purpose Parallel Computing.* Lecture Slides, Open Lecture Course, Division of Computer Science, St. Andrews University, April, 1992.

[Augu91] L Augustsson, *Parallel Evaluation of Functional Programs: The G-Machine Approach.* In Proc. of PARLE '91, Springer Verlag LNCS 505, 1991.

[AuJo89a] L Augustsson and T Johnsson, *Parallel Graph Reduction with the $<\nu$,G>-Machine.* In FPCA, pp. 202-214, ACM, 1989.

[AuJo89b] L Augustsson and T Johnsson, *The Lazy-ML Compiler.* The Computer Journal (Special Issue on Lazy FP), Cambridge University Press, **21**(2), pp. 127-141, April, 1989.

[Augu85] L Augustsson, *Compiling Pattern Machine.* In the proceedings of FPCA, Nancy France, Springer Verlag LNCS 201, pp. 368-381, September 1985.

[BGvN47] AW Burks, HH Goldstine, and J von Neumann, *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument.* In John von Neumann: Collected Works, 5 pp. 34-79, New York: Macmillan, 1973.

[BHY88] A Bloss, P Hudak, and J Young, *Code Optimisations for Lazy Evaluation.* Lisp and Symbolic Computation 1, pp. 147-164.

[BNA91] PL Barth, RS Nikhil, and Arvind, *M-Structures: Extending a Parallel, Non-strict Functional Language with State.* Springer Verlag LNCS 523, pp. 538-568, August 1991.

[BWW90] J Backus, J Williams and E Wimmers, *An Introduction to the Programming Language FL.* In DA Turner (ed.), Research Topics in Functional Programming, Addison Wesley, pp. 219-247.

[Back78] JW Backus, *Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs.* Comm. ACM, 21, pp. 613-641.

[Back81] JW Backus, *The History of Fortran I, II and III.* In The History of Programming Languages, RW Wexelblat (ed.), London and New York: Academic Press.

[Bare84] HP Barendregt, *The Lambda Calculus: its Syntax and Semantics.* North Holland, 1994.

[Blel93] GE Blelloch, *NESL: A Nested Data-Parallel Language (Version 2.6).* Technical Report, CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[BrOs95] S Brock and G Ostheimer, *Process Semantics of Graph Reduction.* Research Report, CS/95/2, Department of Mathematical and computational sciences, St. Andrews University, Scotland, 1995.

[Brat94] TA Bratvold, *Skeleton-Based Parallelisation of Functional Programs.* Department of Computing and Electrical Engineering, Heriot-Watt University, Scotland, PhD Thesis, November 1994.

[BuRa94] FW Burton and VJ Rayward-Smith, *Worst Case Scheduling for Parallel Functional Programming.* Journal of Functional Programming.

[BuSl82] FW Burton and MR Sleep, *Executing Functional Programs on a Virtual Tree of Processors.* In Proc. of FPCA 1982, pp. 187-194, Portsmouth, New Hampshire, October 1981.

[Burn87a] GL Burn, *Evaluation Transformers — A Model for the Parallel Evaluation of Functional Languages (extended abstract).* In Proc. of FPCA 1987, Portland, G Kahn, ed., Springer Verlag LNCS 274, pp. 446-470.

[Burn87b] GL Burn, *Abstract Interpretation and the Evaluation of Functional Languages.* PhD thesis, Imperial College, University of London, March 1987.

[Burn90] L Burn, *Using Projection Analysis in Compiling Lazy Functional Programs* . In Proc. of the 1990 ACM Conference on Lisp and Functional Programming, pp. 227-241, Nice, France, June 1990.

[Burn91] L Burn, *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing, Pitman in association with MIT Press, 1991. March 1987.

[Burt84] FW Burton, *Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs.* ACM TOPLAS, 6(2) pp. 159-174, April 1984.

[Burt91] FW Burton, *Encapsulating Nondeterminacy in an Abstract Data Type with Determinate Semantics.* Journal of Functional Programming, 1(1), pp. 3-20, January 1991.

[Busv93] DJ Busvine, *Detecting Parallel Structures in Functional Programs.* Department of Computing and Electrical Engineering, Heriot-Watt University, Scotland, PhD Thesis, April 1993.

[BvEG+87a] HP Barendregt, MCJD van Eekelen, JRW Glauert, JR Kennaway, MJ Plasmeijer and MR Sleep, *Term Graph Rewriting.* In JW de Bakker, AJ Nijman, and PC Treleaven (eds.), PARLE: Parallel Architectures and Languages Europe, Vol I, Springer-Verlag, LNCS 259, pp. 141-158, 1987.

[BvEG+87b] HP Barendregt, MCJD van Eekelen, JRW Glauert, JR Kennaway, MJ Plasmeijer and MR Sleep, *Towards an Intermediate Language Based on Graph Rewriting.* In JW de Bakker, AJ Nijman, and PC Treleaven (eds.), PARLE: Parallel Architectures and Languages Europe, Vol I, Springer-Verlag, LNCS 259, pp. 141-158, 1987.

[BvEG+88] HP Barendregt, MCJD van Eekelen, JRW Glauert, JR Kennaway, MJ Plasmeijer and MR Sleep, *LEAN — An Intermediate Language Based on Graph Rewriting.* Parallel Computing, **9**, pp. 163-177.

[BvEL+87] T Brus, MCJD van Eekelen, M van Leer, MJ Plasmeijer and JP Barendregt, *Clean — A Language for Functional Graph Rewriting.* In Proc. of FPCA 1987, pp. 364-384, Springer-Verlag LNCS 274.

[CCC+89] A Contessa, W Cousin, C Couset, M Cubero-Castan, G Durrieu, B Lécussan, M Lemaître, and P Ng, *MaRS, a Combinator Graph Reduction Multiprocessor.* In Proc. of PARLE 1989, Springer Verlag LNCS 365, pp. 176-192, 1989

[CDL87] M Cubero-Castan, MH Durand and M Lemaître, *A Set of Combinators for Abstraction in Linear Time.* Information Processing Letters, **24**, pp. 183-188, February 1987.

[CFC58] HB Curry, R Feys, and W Craig, *Combinatory Logic.* Vol I, Studies in Logic and Foundations of Mathematics, North Holland, Amsterdam.

[CGKL94] MMT Chakravarty, Y Guo, M Köhler and HCR Lock, *Declarative Programming of Parallel Algorithms.* In Proc. of GI-Workshop: Declarative Programming and Specification, Bad Honnef, Germany, June 1994.

[CHK+92] S Cox, SY Huang, P Kelly, J Liu and F Taylor, *An Implementation of Static Functional Process Networks.* In Proc. of PARLE 1992, pp. 497-512, Springer Verlag LNCS 605.

[CHK$^+$93] S Cox, SY Huang, P Kelly, J Liu and F Taylor, *Program Transformations for Static Process Networks.* ACM SIGPLAN Notices January 1993 (Special Issue: Workshop on Languages, Compilers and Runtime Environments for Distributed Memory Machines).

[CRPC96] Center for Research on Parallel Computation, Rice University, *General Information web page*: http://www.crpc.rice.edu/CRPC/

[CSS$^+$91] DE Culler, A Sah, and KE Schauser, T von Eicken, H Wawrzynek. *Fine-grained Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine.* In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA, April 1991.

[CaHa93] M Carlsson and T Hallgren, *FUDGETS - A Graphical User Interface in a Lazy Functional Language.* In Proc. of FPCA 1993, Copenhagen, Denmark, pp. 321-330, June 1993.

[Card83] L Cardelli, *ML Under UNIX.* Polymorphism: The ML/LCF/Hope Newletter, **I**(3), December 1983.

[Card84] L Cardelli, *Compiling a Functional Language.* In Proc. of the 1984 LISP and Functional Programming Conference, Austin, Texas, (August 1984).

[Card85] L Cardelli, *On Understanding Types, Data Abstraction and Polymorphism.* Computer Surveys, **17**(4), December 1985.

[ChRo36] A Church and JB Rosser, *Some Properties of Conversion.* Transactions of the American Mathematical Society, 39, pp. 472-482.

[Chak94] MMT Chakravarty, *A Self-Scheduling, Non-blocking, Parallel Abstract Machine for Non-strict Functional Language.* In J.R.W. Glauert, editor, IFL, pp. 13.1-13.34. School of Information Systems, University of East Anglia, England, UK.

[Chur41] A Church, *The Calculi of Lambda-Conversion.* In Annals of Mathematical Studies No. 6, Princeton University Press. (Reprinted in 1963 by University Microfilms Inc., Ann Arbor, Michigan.)

[ClPeP86] C Clack, SL Peyton Jones, *The Four-Stroke Reduction Engine.* In Proc. of the 1986 ACM conference on LISP and Functional Programming, pp. 220-232, 1986.

[Cling95] WD Clinger, Electronic communications on *comp.lang.functional* newsgroup, December 1995.

[Cole89] MI Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing, Pitman, 1989.

[CuAr88] DE Culler and Arvind, *Resource Requirements of Dataflow Programs.* In Proc. of the 15th Annual Symposium on Computer Architecture, ACM, 1988.

[DaCa85] WJ Dally, P Carrick, *et al.*, *The J-Machine: A Fine-Grained Concurrent Computer.* Information Processing 89, GX Ritter (ed), Elsevier Science Publishers B.V. (North-Holland), 1989.

[DaKe82] AL Davis and RM Keller, *Data Flow Program Graphs.* IEEE Computer, pp. 26-41, February 1982.

[DaMc89] AJT Davie and DJ McNally, *CASE, A Lazy Version of an SECD Machine with a Flat Environment.* In Proc. of IEEE TENCON 1989, pp. 864-872, Bombay 1989.

[DaMo81] AJT Davie, and R Morrison, *Recursive Descent Compiling.* Ellis Horwood, 1981.

[Dama85] L Damas, *Type Assignment in Programming Languages.* PhD Thesis, University of Edinburgh (CST-33-85), April, 1985.

[Darl93] J Darlington, *Parallel Programming Using Skeleton Functions.* In Proc. of PARLE 1993, Springer Verlag LNCS 694, pp. 146-160, 1993.

[Davi79] AJT Davie, *Variable Access in Languages in Which Procedures are First Class Citizens.* St. Andrews University Department of Computational Science Report, CS/79/2, 1979.

[Davi92] AJT Davie, *An Introduction to Functional Programming Systems Using Haskell.* Cambridge Texts in Computer Science, University Press 1992.

[Desc89] JM Deschner, *Simulatin Multiprocessor Architectures for Compiled Graph-Reduction.* In Proc. of GlaFP 1989, WICS, pp. 225-237, Fraserburgh, Scotland, August 21-23, 1989. Springer-Verlag, Berlin.

[EaLa86] DL Eager, ED Lazowska, and J Zahorjan, *Adaptive Load Sharing in Homogeneous Distributed systems.* IEEE Transactions on Software Engineering, **SE-12**(5), pp. 662-675, 1986.

[Ekan91] K Ekanadham, *A perspective on Id.* In Parallel Functional Languages and Compilers, Frontier Series. ACM Press, New York 1991.

[FaWr87] J Fairbairn and S Wray. *TIM - A Simple Lazy Abstract Machine to Execute Supercombinators.* In Proc. of FPCA 1987, Portland, G Kahn, ed., Springer Verlag LNCS 274, pp. 34-45.

[FiHa88] AJ Field and PG Harrison, *Functional Programming.* Addison Wesley, 1988.

[FlNi96] C Flanagan and RS Nikhil, *pHluid: The Design of a Parallel Functional Language Implementation on Workstations.* In ICFP '96 — Intl. Conf. on Functional Programming, pp. 169-179, Philadelphia, PA, May 24-26, 1996, ACM Press.

[Flyn72] MJ Flynn, *Some Computer Organisations and their Effectiveness.* IEEE Transactions on Computers, **21**, pp. 948-960, September 1972.

[GHW90] H Glaser, P Hartel, and J Wild *A Pragmatic Approach to the Analysis and Compilation of Lazy Functional Languages.* Technical Report CSTR 90-10, Department of Electronics and Computer Science, University of Southampton.

[GMS93] R Goldsmith, DL McBurney and MR Sleep, *Parallel Execution of Concurrent Clean on ZAPP.* In Term Graph Rewriting, MR Sleep *et al* (eds.), John Wiley.

[GlHa90] H Glaser, P Hartel and J Wild *A Pragmatic Approach to the Analysis and Compilation of Lazy Functional Languages.* In Proc. of PIFL 1990, pp. 203-221.

[GoHu86] BF Goldberg and P Hudak, *Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor.* Int. Workshop on Graph Reduction, Santa Fe, New Mexico, pp. 94-113, Springer Verlag LNCS 279, September 1986.

[Gold88a] BF Goldberg, *Buckwheat: Graph Reduction on a Shared-Memory Multiprocessor.* In ACM Conference on LISP and Functional Programming, 1988.

[Gold88b] BF Goldberg, *Multiprocessor Execution of Functional Programs.* Int. Journal of Parallel Programming, **17**(5), pp. 425-473, 1988.

[Gord88] MJC Gordon, *Programming Language Theory and its Implementation.* Prentice Hall International, 1988.

[Gord93] AD Gordon, *Functional Programming and Input/Output.* PhD Thesis, University of Cambridge Computer Laboratory, Technical Report No. 285.

[Grah69] RL Graham, *Bounds on Multiprocessing Timing Anomalies.* SIAM Journal of Applied Mathematics, **17**(2), pp. 416-429, 1969.

[Guo94] Y Guo, *Definitional Constrant Programming.* PhD thesis, Department of Computing, Imperial College London, 1994.

[HCAA94] J Hicks, D Chiou BS Ang and Arvind, *Performance Studies of Id on the Monsoon Dataflow System.* Computation Structures Group Memo 345-3, MIT Laboratory for Computer Science, Cambridge, Massachusetts, USA, August 5, 1994. Also in Journal of Parallel and Distributed Computing, **18**(3), pp. 273-300, 1993.

[HGW89] P Hartel, H Glaser, and J Wild, *Compilation of Functional Languages using Flow Graph Analysis.* Technical Report CSTR 89-4, Department of Electronics and Computer Science, University of Southampton.

[HGW91] P Hartel, H Glaser, and J Wild, *On the Benefits of Different Analyses in the Compilation of a Lazy Functional Language.* In Proc. of the 3rd Int. Workshop on the Implementation of Functional Languages on Parallel Architectures, Southampton, June 1991.

[HGW94] P Hartel, H Glaser, and J Wild *Compilation of Functional Languages using Flow Graph Analysis.* Software–Practice and Experience, **24**(2), pp. 127-173, February 1994.

[HHLT97] K Hammond, CV Hall, H-W Loidl and PW Trinder *Parallel Cost Center Profiling.* In Proc. of GlaFP 1997, Ullapool, Scotland, September 14-17, 1997.

[HKL91] G Hogen, A Kindler, and R Loogen *Automatic Parallelisation of Lazy Functional Programs.* Aachener Informatik Bericht 91-20, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Germany, 1991

[HLP95] K Hammond, H-W Lodil and A Partridge *Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell.* In Proc. of HPFC 1995, pp. 208-221, Denver, CO, April 10-12, 1995.

[HMP94] K Hammond, JS Mattson Jr. and SL Peyton Jones, *Automatic Spark Strategies and Granularity for a Parallel Functional Language Reducer.* In CONPAR 1994, Springer Verlag LNCS, September 1994.

[HMT88] R Harper, R Milner and M Tofte, *The Design of Standard ML Version 2.* Technical Report ECS-LFCS-88-62, Laboratory for the Foundation of Computer Science, Department of Computer Science, Edinbugh University, August 1988.

[HPW92] P Hudak, SL Peyton Jones, and P Wadler (eds.), *Report on the Programming Language Haskell Version 1.2.* ACM SIGPLAN Notices 27(5), May 1992.

[HaLa92] PH Hartel and KG Langendoen, *Benchmarking Implementations of Lazy Functional Languages.* Technical Report CS-92-20, Department of Computer Systems, Faculty of Mathematics and Computer Science, University of Amsterdam.

[HaPe90] K Hammond, SL Peyton Jones, *Some Early Experiments on the GRIP Parallel Reducer.* In Proc. of the 2nd Intl. Workshop on Implementation of Functional Languages on Parallel Architectures, Plasmeijer MJ (ed.), University of Nijmegen, pp. 51-71, June 1990.

[HaPe92] K Hammond, SL Peyton Jones, *Profiling Scheduling Strategies on the GRIP Multiprocessor.* In Proc. of the 1992 Workshop on Parallel Implementation of Functional Programming Languages, Aachen, 1992.

[Hals85] R Halstead, *Multilisp: A Language for Concurrent Symbolic Computation.* ACM TOPLAS, 7(4), pp. 501-538, October 1985.

[Hamm88] K Hammond, *Compiling Functional Languages for Parallel Machines.* PhD thesis, School of Information Systems, University of East Anglia, Norwich, England, UK, 1988.

[Hamm90] K Hammond, *Efficient Type Inference Using Monads.* In Draft Proc. of GlaFP 1990, Ullapool, Scotland, August 1990.

[Hamm94] K Hammond, *Parallel Functional Programming: An Introduction.* In PASCO 1994, First International Symposium on Parallel Symbolic Computation, World Scientific Publishing Company, September 94.

[Hans75] BP Hansen, *The Programming Language Concurrent Pascal.* IEEE, Transaction on Software Engineering, 1, pp. 199-207.

[Hart94] PH Hartel *et al. Pseudoknot: A Float-Intensive Benchmark for Functional Compilers.* In Proc. of IFL 1994, pp. 13.1-13.34. School of Information Systems, University of East Anglia, Norwich, UK.

[Hend80] P Henderson, *Functional Programming: Application and Implementation.* Prentice Hall International, 1980.

[Hick93] JE Hicks, *Experiences with Compiler-Directed Storage Reclamation.* In Proc. of FPCA 1993, pp. 95-105, June 1993.

[Hill92] JMD Hill, *Data Parallel Haskell: Mixing Old and New Glue.* Technical Report 611, Department of Computer Science, Queen Mary and Westfield College, University of London, December 1992.

[Hill94] JMD Hill, *Data-Parallel Lazy Functional Programming.* PhD thesis, Department of Computer Science, Queen Mary and Westfield College, University of London, September 1994.

[Hofm94] R Hofman, *Scheduling and Grain Size Control.* PhD thesis, Wiskunde en Informatica, Universiteit van Amsterdam, May, 1994.

[Huda84] P Hudak, *ALFL Reference Manual and Programmer's Guide.* Technical Report YALEU/DCS/TR-322, Yale University, Department of Computer Science, August 1984.

[Huda86] P Hudak, *Collecting Interpretations of Expressions.* Research Report 497, Yale University, Department of Computer Science, 1986. (Submitted to the Third Workshop on the Mathematical Foundations of Programming Language Semantics.

[Huda89] P Hudak, *Conception, Evolution, and Applications of Functional Programming Languages.* ACM Computing Surveys, **21**(3), September, 1989.

[Huda91] P Hudak, *Para-functional Programming in Haskell.* In Parallel Functional Languages and Compilers, BK Szymanski (ed.), Addison-Wesley, 1995.

[HuGo85a] P Hudak and B Goldberg, *Serial Combinators: 'Optimal' Grains of Parallelism.* In FPCA 1985, pp. 382-399, September 1985.

[HuGo85b] P Hudak and B Goldberg, *Efficient Distributed Evaluation of Functional Programs Using Serial Combinators.* In IEEE Transactions on Computers C-34(10), October 1985.

[HuOp80] G Huet and DC Oppen, *Equations and Rewrite Rules: A Survey.* In Formal Languages: Perspectives and Open Problems, RV Book (ed.), Academic Press, London.

[Hugh82] RJM Hughes, *Supercombinators - A New Implementation Method for Applicative Languages.* In Proc. of the 1982 ACM Symposium of Lisp and Functional Programming, pp. 1-10, Pittsburgh, 1982.

[Hugh84] RJM Hughes, *The Design and Implementation of Programming Languages.* Technical Monograph PRG-40, Oxford University, July 1983, pub. 1984.

[HuSu89] P Hudak and RS Sundaresh, *On the Expressiveness of Purely Function I/O Systems.* Technical Report YALEU/DCS/RR665, Yale University, Department of Computer Science, December 1988.

[Hutt92] G Hutton, *Higher Order Functions for Parsing.* Journal of Functional Programming, **2**(3), pp. 323–343, July 1992

[HuWa90] P Hudak and P Wadler (eds.), *Report on the Programming Language Haskell, a Non-strict Purely Functional Language (Version 1.0).* Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[Hwan82] K Hwang *et al*, *A UNIX-based Local Computer with Load Balancing.* Computer Journal, **15**(6), pp. 50–56, 1982

[Iann88] RA Iannucci, *Toward a Dataflow/von Neumann Hybrid Architecture.* In Proc. of the 15th Int. Symposium on Computer Architecture, pp. 131-140, 1988.

[JoHu93] MP Jones and P Hudak, *Implicit and Explicit Parallel Programming in Haskell.* Research Report YALEU/DCS/RR-982, Department of Computer Science, Yale University, 1993.

[John87] T Johnsson, *Compiling Lazy Functional Languages.* Ph.D. Thesis, Department of Computer Science, Chalmers University of Technology, G oteborg, Sweden, 1987.

[Jone84] SB Jones, *A Range of Operating Systems Written in a Functional Style.* Report TR.16, University of Sterling, Dept. of Computer Science, September 1984.

[Jone94] MP Jones, *The Implementation of the Gofer Functional Programming System.* Technical report YALEU/DCS/RR-1030, Department of Computer Science, Yale University, New Haven, Connecticut, USA, May 1994

[JuTr98] S Junaidu and PW Trinder, *Measuring Naira on GUM.* Under Preparation.

[KHM89] DA Kranz, RH Halstead, and E Mohr *Mul-T: A High Performance Parallel Lisp.* In ACM SIGPLAN '89 Conference on Programming Languages Design and Implementation, ACM Press, pp. 180-90.

[KMIA96] S Kusakabe, T Morimoto, K Inenaga and M Amamiya *Towards Practical Implementation of a Dataflow-based Functional Language on Stock Machines.* In Proc. of the 8th International Workshop on the Implementation of Functional Languages, Bonn/Germany, pp. 243-260, September, 1996.

[KKSdV90] JR Kennaway, JW Klop, MR Sleep, and FJ de Vries *Transfinite Reductions in Orthogonal Term Rewriting Systems.* Technical Report CS-R9041, Centre for Mathematics and Computer Science, Kruislaan 413, Amsterdam, 1990.

[KKSdV93] JR Kennaway, JW Klop, MR Sleep, and FJ de Vries *An Introduction to Term Graph Rewriting.* In Term Graph Rewriting, MR Sleep *et al* (eds.), John Wiley.

[KLB91] H Kingdon, DR Lester, GL Burn, *The HDG-Machine: A Highly Distributed Graph Reducer for a Transputer Network.* The Computer Journal, **34**(4), 1991.

[KaTa92] K Kaneko and M Takeichi, *Relationship Between Lambda Hoisting and Fully Lazy Lambda Lifting.* Journal of Information Processing, **15**(4), 1992.

[Kell89] P Kelly, *Functional Programming for Loosely-Coupled Multiprocessors.* Pitman/MIT Press, 1989.

[Kenn90] R Kennaway, *The Specificity Rule for Lazy Pattern Matching in Ambiguous Term Rewriting Systems.* In Conference of European Symposium on Programming, 1990.

[Kess91] M Kesseler, *Implementing the ABC Machine on Transputers.* In Proc. of the 3rd Int. Workshop on the Implementation of Functional Languages on Parallel Architectures, Southampton, pp. 147-192, June 1991.

[Kess93] M Kesseler, *Efficient Routing Using Class Climbing.* In R Grebe, J Hektor, SC Hilton, ME Jane and PH Welch (eds.), Transputer Applications and Systems, World Transputer Congress 1993, Aachen, **2**, pp. 830-846, 1993.

[Kess96] M Kesseler, *The Implementation of Functional Languages on Parallel Machines with Distributed Memory.* PhD thesis, University of Nijmegen, 1996.

[Klop92] JW Klop, *Term Rewriting Systems.* In Handbook of Logic in Computer Science, S Abrahamsky, D Gabbay and T Maibaum, (eds.), **2**, Oxford University Press.

[Klug94] W Kluge, *A User's Guide for the Reduction System π-RED.* Internal Report 9419, University of Kiel, 1994.

[KuWa90] H Kuchen and A Wagener, *Comparison of Dynamic Load Balancing Strategies.* Technical Report AIB 90-05, RWTH Aachen, 1990.

[KvEN+90] PWM Koopman, MCJD van Eekelen, EGJMH Nöcker, MJ Plasmeijer and JEW Smetsers, *The ABC-Machine: A Sequential Stack-Based Abstract Machine for Graph Rewriting.* Technical Report 90-22, University of Nijmegen.

[LCD+86] M Lemaître, M Cubero-Castan MH Durand, G Durrieu, and B Lécussan *Mechanisms for Efficient Multiprocessor Combinator Reduction.* In Proc. of the 1986 ACM Conference on Lisp and Functional Programming, pp. 113-121, Cambridge, Massachusetts, August 1986.

[LKID89] JR Loogen, H Kuchen, K Indermark, and W Damm, *Distributed Implementation of Parallel Graph Reduction.* In Proc. of PARLE 1989, Springer Verlag LNCS 365, pp. 136-157.

[Land64] P Landin, *The Mechanical Evaluation of Expressions.* BCS Computing Journal, 6(4), pp. 308-320, January 1964.

[Land65] P Landin, *A Correspondence Between Algol 60 and Church's Lambda Calculus Notation.* Comm. of ACM, **21**(11), pp. 931-933, 1965.

[LaHa92] K Langendoen and PH Hartel, *FCG: A Code Generator for Lazy Functional Languages.* Compiler Construction 1992, Paderborn, Germany.

[Lavi91] A Laville, *Comparison of Priority Rules in Pattern Matching and Term Rewriting.* Journal of Symbolic Computation, **11**, pp. 321-347, 1991.

[LeBu89] DR Lester and GL Burn, *An Executable Specification of the HDG-Machine.* In Workshop on Massive Parallelism: Hardware, Programming and Applications, Amalfi, Italy, October 1989.

[Lest89] DR Lester, *Stacklessness: Compiling Recursion for a Distributed Architecture.* In Poc. FPCA '89, pp. 116-128, London, UK, September 1989.

[LiKe87] FCH Lin and RM Keller, *The Gradient Model Load Balancing Method.* IEEE Transactions on Software Engineering, **SE-13**(1), pp. 33-38, 1987.

[LoHa95] HW Loidl and K Hammond, *On the Granularity of Divide-and-Conquer Parallelism.* In Proc. of GlaFP 1995, WICS, Ullapool, Scotland, July 8-10, 1995. Springer-Verlag.

[LoHa96a] HW Loidl and K Hammond, *A Sized Time System for a Parallel Functional Language.* In Proc. of GlaFP 1996, Ullapool, Scotland, July 8-10.

[LoHa96] HW Loidl and K Hammond, *Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer.* In Proc. of IFL 1996, LNCS 1268, Springer-Verlag, Bad Godesberg, Germany, September 1996.

[LoTr97] HW Loidl and PW Trinder, *Engineering Large Parallel Functional Programs.* In Proc. of IFL 1996, University of St Andrews, Scotland, September 10-12, 1997. Draft Proceedings.

[Lock92] HCR Lock, *The Implementation of Functional Logic Programming Languages.* PhD thesis, Technical University of Berlin, 1992.

[Loid92] HW Loidl, *Compiling Functional Programs to Dataflow Code.* In Workshop on Parallel Computers and Programming Languages, February 1992.

[Loid96] H-W Loidl *GranSim User's Guide, Version 0.03.* Department of Computing Science, University of Glasgow, July 1996.

[Loog87] R Loogen, *Design of a Parallel Programmable Graph Reduction Machine with Distributed Memory.* Aachener Informatik-Berichte Nr. 87-11.

[Loog88] R Loogen, *Parallele Implementierung Funktionaler Programmiersprachen.* Dissertation (in German), RWTH Aachen, 1988.

[Loud93] KC Louden, *Programming Languages: Principles and Practice.* PWS-KENT Publishing Company, 1993.

[LyRi78] RE Lynch, JR Rice, *Computers: Their Impact and Use.* Holt, Rinehart and Winston, 1978.

[MCC96] Maui High Performance Computing Center, *Introduction to Parallel Programming.* Web page: http://www.mhpcc.edu/training/tutorials/Tutorials.html

[Mara91] L Maranget, *GAML: a Parallel Implementation of Lazy ML.* Springer Verlag LNCS 523, pp. 102-123.

[Mara92] L Maranget, *La Stratégie Paresseuse.* PhD thesis, University of Paris VIII, 1992.

[Mara92] L Maranget, *Compiling Lazy Pattern Matching.* In Conference of Lisp and Functional Programming 1992.

[Mara94] L Maranget, *Two Techniques for Compiling Lazy Pattern Matching.* Research Report 2385, INRIA Rocquencourt, October 1994.

[Matt93] JS Mattson Jt., *An Effective Speculative Evaluation Technique for Parallel Supercombinator Graph Reduction.* PhD thesis, University of California, San Diego, 1993.

[Maye88] HG Mayer, *Programming Languages.* Macmillan Publishing Company, 1988.

[McBSl87] DL McBurney and MR Sleep, *Transputer Based Experiments with the ZAPP Architecture.* In JWde Bakker, AJ Nijman, and PC Treleaven (eds.), PARLE: Parallel Architectures and Languages Europe, Vol **I**, Springer-Verlag, LNCS 259, pp. 242-256, 1987.

[McBSl90] DL McBurney and MR Sleep, *Concurrent Clean on ZAPP.* In Proc. of the 2nd Intl. Workshop on Implementation of Functional Languages on Parallel Architectures, Plasmeijer MJ (ed.), University of Nijmegen, pp. 73-113, June 1990.

[Miln78] R Milner, *A Theory of Type Polymorphism in Programming.* Journal of Computer and System Science, 17, 348-75, December, 1978.

[Miln80] R Milner, *A Calculus of Communicating Systems.* Springer Verlag LNCS 92, New York, 1980.

[Miln92] R Milner, *Functions as Processes.* Mathematical Structures of Computer Secience, **2**, pp. 119-141, 1992.

[Miln93] R Milner, *Elements of Interaction.* In Communications of the ACM, pp. 78-89, January 1993.

[Miln96] R Milner, *Some Applications of π-calculus.* Seminar presentation, Department of Computer Science, Sterling University, November, 1996.

[MoTo92] G Morrisett, and A Tolmach, *A Portable Multiprocessing Interface for Standard ML of New Jersey.* Technical Report CMU-CS-92-155, Carnegie Mellon University, Department of Computer Science, June 1992.

[NPA86] RS Nikhil, K Pingali, and Arvind, *Id Nouveau.* Computation Structures Group Memo 265, MIT Laboratory for Computer Science, Cambridge, Massachusetts, USA, July 1986.

[NPS91] EGJMH Nöcker, MJ Plasmeijer and JEW Smetsers, *The Parallel ABC-Machine.* In Proc. of the 3rd Int. Workshop on the Implementation of Functional Languages on Parallel Architectures, Southampton, pp. 383-407, June 1991.

[NSvEP91] EGJMH Nöcker, JEW Smetsers, MCJD van Eekelen and MJ Palsmeijer, *Concurrent Clean.* In Proc. PARLE '91, Springer-Verlag LNCS 506, pp. 202-219.

[NXG85] LM Ni, C-W Xu and TB Generau, *A Distributed Drafting Algorithm for Load Balancing.* IEEE Transactions on Software Engineering, **SE-11**(1), pp. 1163-1161, 1985.

[NiAr89] RS Nikhil and Arvind, *Can Dataflow Subsume von Neumann Computing.* In Proc. of the 16th Annual Int. Symp. on Comp. Arch., pp. 262-272. IEEE, June, 1989.

[Nikh89] RS Nikhil, *The Parallel Programming Language Id and its Compilation for Parallel Machines.* In Proc. of the Workshop on Massive Parallelism: Hardware, Programming and Applications, Amalfi, Italy, October 1989, Academic Press.

[Nöck93] E Nöcker *Strictness Analysis using Abstract Reduction.* In Proc. of FPCA 1993, Copenhagen, Denmark, pp. 255-265, June 1993.

[O'Don93] JT O'Donnell, *Bidirectional Fold and Scan.* In Proc. of GlaFP 1993, Springer Verlag WICS, 1993.

[Osth91] G Ostheimer, *Parallel Functional Computation on STAR:DUST.* In Proc. of PIFL 1991, Southampton, Appeared as Technical Report CSTR 91-07, University of Southampton, 1991, pp. 393-408.

[OsDa93] G Ostheimer and A Davie, *π-calculus Characterisations of Some Practical λ-calculus Reduction Strategies.* Research Report, CS/93/14, Department of Mathematical and computational sciences, St. Andrews University, Scotland, 1993.

[Osth93] G Ostheimer, *Parallel Functional Programming for Message Passing Multiprocessors.* PhD Thesis, Department of Mathematical and computational sciences, St. Andrews University, Scotland, 1993.

[PCSH87] SL Peyton Jones, C Clack, J Salkild and M Hardie, *GRIP—A High-Performance Architecture for Parallel Graph Reduction.* In Proc. of FPCA 1987, pp. 98-112, Springer Verlag LNCS 274, 1987.

[PHH$^+$93] SL Peyton Jones, C Hall, K Hammond, W Partain, and P Wadler, *The Glasgow Haskell Compiler: A Technical Overview.* In Joint Framework for Information Technology Technical Conference, Keele, 1993.

[PSU95] Pennsylvania State University, *Introduction to Message Passing Parallel Programming with MPI (Message Passing Interface).* Web page: http://www.psu.edu/

[PaDa92] N Paterson, AJT Davie, *An Initial Specification of Data Structures and Functions in the St. Andrews Haskell Compiler.* Research report CS/92/8, Department of Mathematical and computational sciences, St. Andrews University, Scotland.

[Paul91] LC Paulson, *ML for the Working Programmer.* Oxford University Press, 1991.

[PeHa$^+$97] JC Peterson, K Hammond (eds.), *et al.*, *Haskell 1.4 — A Non-Strict, Purely Functional Language.* April 1997.

[PeJo90] SL Peyton Jones, MS Joy, *FLIC—a Functional Language Intermediate Code.* Research Report UW-DCS-RR-148, Department of Computer Science, University of Warwick, 1990.

[PeLe91] SL Peyton Jones, D Lester, *Implementing Functional Languages: A Tutorial.* Prentice Hall International, 1991.

[PeWa93] SL Peyton Jones, P Wadler, *Imperative Functional Programming.* Prentice Hall International, 1991.

[Perr87] RH Perrot, *Parallel Programming.* Addison Wesley, 1987.

[Perr88] N Perry, *Functional I/O - A Solution.* Department of Computing, Imperial College, University of London.

[Peyt87] SL Peyton Jones, *The Implementation of Functional Programming Languages.* Prentice Hall International, 1987.

[Peyt87] SL Peyton Jones, *GRIP: A Parallel Graph Reduction Machine.* In Proc. of FPCA 1987, Portland, Oregon. In Proc. of FPCA 1987, Portland Oregon, Springer-Verlag LNCS 274, September 1987.

[Peyt89] SL Peyton Jones, *Parallel Implementation of Functional Programming Languages.* The Computer Journal (A special issue), **32**(2), April 1989, pp. 175-186.

[Peyt92] SL Peyton Jones, *Implementing Functional Languages on Stock Hardware: the Spineless Tagless G-machine.* Journal of Functional Programming, 2(2), pp. 127-202, 1992.

[Peyt94] SL Peyton Jones, *The Glorious Glasgow Haskell Compilation System, Version 0.23, User's Guide.* The *AQUA Team*, Department of Computing Science, University of Glasgow, G12 8QQ, UK, December 21, 1994.

[PiTu97] BC Pierce and DN Turner, *Pict: A Programming Language Based on the Pi-Calculus.* Indiana University, CSCI Technical Report Number 476, March 19, 1997.

[PlEe93] MJ Plasmeijer and MCJD van Eekelen, *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[PuSu90] L Puel and A Suárez, *Compiling Pattern Matching by Term Decomposition.* In Conf. on Lisp and Functional Programming, 1990.

[ReCl90] S Reeves and M Clarke, *Logic for Computer Science.* Addison Wesley, 1991.

[Read89] C Reade, *Elements of Functional Programming.* Addison Wesley, 1989.

[Rob89] IB Robertson, *Hope$^+$ on Flagship.* In Proc. of GlaFP 1989, Glasgow, K Davis and J Hughes (eds.), pp. 296-307.

[Roe91] P Roe, *Parallel Programming using Functional Languages.* PhD Thesis, Department of Computing Science, University of Glasgow, February 1991.

[RuSa87] CA Ruggerio and J Sergeant, *Control of Parallelism in the Manchester Dataflow Machine.* In Proc. of FPCA 1987, Portland, G Kahn, ed., Springer Verlag LNCS 274, pp. 1-15.

[RuWa95] C Runciman and D Wakeling, *Profiling Parallel Functional Computations (Without Parallel Machines).* In Parallel Functional Languages and Compilers, BK Szymanski (ed.), Addison-Wesley, 1995.

[SKL88] MR Swanson, RR Kessler, and G Lindstrom, *An Implementation of Portable Standard Lisp on the BBN Butterfly.* ACM Symposium on Lisp and Functional Programming, Utah, pp. 132-142, July 1988.

[SNcGP91] JEW Smetsers, EGJHM Nöcker, JHG van Groningen and MJ Plasmeijer, *Generating Efficient Code for Lazy Functional Languages.* In Proc. of FPCA '91, Springer-Verlag LNCS 523, pp. 592-617.

[SRR92] RC Sekar, R Ramesh and IV Ramakrishnan, *Adaptive Pattern Matching.* In Conf. International Colloquium on Automata Languages and Programming, 1992.

[Samm69] JE Sammet, *Programming Languages: History and Fundamentals.* Prentice Hall International, 1969.

[SaPe95] PM Sansom and SL Peyton Jones, *Time and Space Profiling for Non-strict Higher Order Functional Languages.* In Proc. of ACM Symposium on POPL 1995, San Francisco, California, January 1995.

[Sara92] V Saraswat, *Concurrent Constraint Programming: A Brief Survey.* Technical Report, Xerox PARC, Palo Alto, August 1992.

[Sark91] V Sarkar, *PTRAN—The IBM Parallel Translation System.* In Parallel Functional Languages and Compilers, BK Szymanski (ed.), ACM Press, 1995.

[Sant95] A Santos, *Compiling by Transformation in Non-Strict Functional Languages.* PhD Thesis, Department of Computing Science, University of Glasgow, July 1995.

[ScGr91] W Schulte and W Grieskamp, *Generating Efficient Portable Code for a Strict Applicative Language.* In J Darlington and R Dietrich (Eds.), Declarative Programming, pp. 239-252, Sasbachwalden, West Germany, Springer Verlag, November 1991.

[Sch096] S-B Scholz, *On Programming Scientific Applications in SAC - a Functional Language Extended by a Subsystem for High-Level Array Operations.* In Proc. of IFL 1996, LNCS 1268, Springer-Verlag, Bad Godesberg, Germany, September 1996, pp. 85-104.

[Schr93] W Schreiner, *Parallel Functional Programming—An Annotated Bibliography.* Technical Report 93-24, RISC-Linz, Johannes Kepler University, Linz, Austria, May 18, 1993.

[Sedg90] R Sedgewick, *Algorithms in C.* Addison Wesley, 1990.

[Sked91] S Skedzielewski, *Sisal.* In Parallel Functional Languages and Compiler, Frontier Series, ACM Press, New York 1991.

[StHi86] GL Steele Jr., WD Hillis: *Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing.* In ACM Conf. on Lisp and Functional programming, 1986, pp. 279-297.

[Szym91] BK Szymanski, *Parallel Functional Languages and Compilers.* In BK Szymanski (ed.), p. 400, ACM Press, 1995.

[TAL91] D Tarditi, A Acharya and P Lee *No Assembly Required: Compiling Standard ML to C.* School of Computer Science, Carnegie Mellon University.

[THL+96] PW Trinder, H-W Loidl, SL Peyton Jones, and J Wu, *A Case Study of Data-intensive Programs in Parallel Haskell.* In Proc. GlaFP 1996, Ullapool, Scotland, July 8-10.

[THM+96] PW Trinder, K Hammond, JS Mattson Tr., AS Partridge and SL Peyton Jones, *GUM: A Portable Parallel Implementation of Haskell.* In PLDI '96 — Programming Languages Design and Implementation, pp. 78-88, Philadelphia, PA, May 1996.

[THLP98] PW Trinder, K Hammond, HW Loidl and SL Peyton Jones *Algorithm + Strategy = Parallelism.* To appear in JFP—Journal of Functional Programming, 1998.

[Take88] M Takeichi, *Lambda Hoisting: A Transformation Technique for Fully Lazy Evaluation of Functional Programs.* New Generation Computing, 5, pp. 377-391, OHMSHA, LTD. and Springer-Verlag.

[Thom90] S Thompson, *Interactive Functional Programming: A Method and a Formal Semantics.* In DA Turner (ed.), Research Topics in Functional Programming, Addison Wesley, pp. 249-285.

[Trau91] KR Traub, *Multi-threaded Code Generation for Dataflow Architectures from Non-Strict Programs.* Springer Verlag LNCS 523, pp. 73-101.

[Turn79b] DA Turner, *A New Implementation Technique for Applicative Languages.* Software Practice and Experience, 9, 1979, pp. 31-40.

[Turn90] DA Turner, *An Approach to Functional Operating Systems* In DA Turner (ed.), Research Topics in Functional Programming, Addison Wesley, pp. 199-217.

[VrHa92] WG Vree and PH Hartel, *Communication Lifting: Fixed Point Computation for Parallelism.* Technical Report CS-92-07, Department of Computer Systems, University of Amsterdam, December 1992.

[Vran90] JLM Vrancken, *Reflections of Parallel Functional Languages.* In Proc. of the 2nd Intl. Workshop on Implementation of Functional Languages on Parallel Architectures, Plasmeijer MJ (ed.), University of Nijmegen, pp. 9-49, June 1990.

[WaBa90] C Walinsky and D Banerjee, *A Functional Programming Language Compiler for Massively Parallel Computers.* In Proc. of the 1990 ACM Conference on Lisp and Functional Programming, pp. 131-138.

[WaHu87] P Wadler and RJM Hughes, *Projections for Strictness Analysis.* In Proc. of FPCA 1987, pp. 385-407, Springer-Verlag LNCS 274, September 1987.

[Webe89] K Wëber, *dLisp - A Local Area Network-Distributed Lisp System.* Department of Mathematical Sciences, Kent State University, February 1989.

[WiRe49] MV Wilkes and W Renwick, *The EDSAC: An Electronic Calculating Machine.* J. Sci. Instrum., 26, pp. 385-391.

[Wiks87] A Wikstöm, *Functional Programming using Standard ML.* Prentice Hall International, 1987.