Skeleton-Based Parallelisation of Functional Programs

by

Tore Andreas Bratvold

Submitted for the Degree of

Doctor of Philosophy

at Heriot-Watt University

on Completion of Research in the

Department of Computing and Electrical Engineering

November 1994

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made, and has not been submitted for any other degree.

Tore Andreas Bratvold

Contents

List of Tables

List of Figures

Acknowledgements

This thesis could never have been written without the helpful support from a number of people, to whom I would like to pay tribute.

First and foremost I would like to thank my supervisor, Greg Michaelson, for his strong support and genuine enthusiasm for my work throughout the project. Greg has always had time to discuss my progress and problems, no matter how big or small, and his many critical questions have been instrumental in the elaboration of my ideas. I am especially grateful for his most helpful comments on my written work.

Marisa Barja has been a source of constant support and encouragement, for which I am deeply grateful. Her efforts in keeping me focused on my work have been important in allowing me to complete the thesis on schedule.

David Busvine's work on the PUFF compiler has been a good source of reference. I am indebted to David for providing a stand alone version of his SML to occam2 translator for use in my compiler.

I would also like to extend my thanks to Patrick Foulk and R.Hunter Davis for patiently reviewing the thesis, and for providing valuable feedback in the final stages of its preparation.

I am grateful to the fellow researchers I have been in contact with in Britain and abroad, the exchange of ideas with whom has greatly contributed to my own understanding of many problems.

This research has been funded by a three-year University Scholarship from Heriot-Watt University. I gratefully acknowledge the financial support received both from the University and from the Department of Computing and Electrical Engineering.

Last, but certainly not least, I would like to thank my family and friends, whose confidence in me always seemed greater than my own.

Abstract

The computational power of parallel computers could be applied with great benefit to many application areas, but the lack of adequate programming tools complicates the development, verification and portability of parallel software. Hence, parallel processing is not widely used in general purpose computing.

Parallel programming would be simplified by tools providing abstractions from details of parallelism without sacrificing performance. Skeletons are templates of parallel operations that allow programs to be expressed with a high level of abstraction while retaining parallelism in a structured form suitable for efficient compilation. Functional programming languages provide a natural framework for skeleton-based parallel programming as skeletons can be elegantly expressed as higher order functions.

This thesis investigates an implicit skeleton-based approach to the parallelisation of functional programs. It describes the design and implementation of SkelML, a skeleton-based parallelising compiler for Standard ML that exploits parallelism in certain higher order functions and in function composition. The compiler supports a profiler based on structural operational semantics, and uses performance models to predict execution times of program components. This information is utilised in the allocation of machine resources. Automatic transformation is supported for performance optimisation. The thesis considers the underlying philosophy of the approach, discusses in detail the compiler internals, and presents a number of results obtained by applying a prototype implementation of the compiler to example programs.

This work is novel in presenting a practical implementation integrating all of the above components, demonstrating that they form a promising framework for skeleton-based compilation. Hence it adds weight to the argument that skeletons are well suited as a basis for parallel programming.

To my parents

Chapter 1

Introduction

1.1 Motivation

Since their commercial appearance more than four decades ago, computer systems have been applied successfully to a wide and increasingly complex variety of tasks. This development has been matched, and partly driven by a no less staggering and continuous series of advances in computer hardware. The horizon is however still full of new challenges, all requiring a further massive increase in computing power. Such challenges include real time image processing, graphics and visualisation, multimedia information systems, simulation, and a number of other problems in scientific computation and artificial intelligence. The ever increasing demands on future computing systems highlights the importance of further research into high performance computing.

Up until a few years ago, virtually all computing took place on uniprocessor systems based on the von Neumann architecture. Although such architectures are still evolving, their performance is limited by a fundamental restriction of sequential execution of program instructions [99]. Much research effort has therefore gone into the development of novel architectures. An appealing approach is to build computers where many processors cooperate by working in parallel. Parallel computing has grown into a very active research area, reflected in a myriad of conferences, seminars, workshops, and publications. Parallel machines are commercially available from a number vendors, including giants such as IBM and Fujitsu.

Parallel computers are however still used almost exclusively for very specific scientific purposes, and by a very narrow range of users. It is widely believed that the principal reason why parallel computing has yet to see a major commercial breakthrough is associated with the lack of adequate software development methodologies and tools [78].

1.2 The Parallel Programming Problem

In the following chapter we examine two key issues which make parallel computing problematic: the development and the portability of parallel software. While the development of parallel implementations certainly involves addressing a number of complex issues, it seems a fundamental requirement for the widespread use of parallel computers that they are easy to use. Ideally a parallel computer should not be much harder to use than a traditional uniprocessor machine. The programmer should be allowed to expect ease of programming and a high degree of portability without having to suffer poor performance. It is claimed that unless parallel software can be developed and maintained with relative ease, many companies will find it hard to justify the initial investments in parallel equipment [156]. Thus, there is a need for methodologies and tools that can help managing the complexities of parallelism.

This thesis addresses this problem by investigating a high-level approach to the development of parallel software. This implies a shift in responsibilities from the programmer to the compiler. We base our approach on a structuring mechanism known as *skeletons*, proposed by Cole [54]. Skeletons offer the programmer a high level of abstraction, while at the same time providing the compilation system with parallelism in a form which can be efficiently managed and compiled. A more detailed description of the thesis objectives is found at the end of the following chapter, after the principal concepts with which we deal have been discussed in detail. Briefly, the thesis proposes a framework for skeleton-based compilation in the context of functional programming languages. We elaborate our approach by outlining the design of a compilation system based on a fixed set of skeletons. Characteristic features of the compiler are its profiling and performance modeling capabilities, and its transformation system. We present and discuss a number of results obtained by a prototype implementation.

When this work commenced in November 1991, publications devoted to skeletons were few and far between. Over the past three years, interest has increased to the extent that relevant papers regularly appear in journals and conference proceedings. Still, there is very little practical experience with the approach, and the tasks involved in building a skeleton-based compiler have not been deeply explored. The main contribution of our work is that it narrows this gap by presenting a concrete proposal which has been largely realised in a prototype implementation.

1.3 Thesis Outline

In the following chapter we present relevant issues in the areas of parallel and functional programming. Sequential and parallel implementation techniques of functional languages are reviewed and discussed.

Chapter 3 examines what we believe to be the crucial issues in using skeletons for parallel programming. We highlight important properties and survey relevant work. A classification and description of proposed skeletons is presented.

In chapter 4 we lay down a framework for skeleton-based parallelisation of functional programs. We introduce the SkelML compiler and outline its internal organisation, including a description of the source language and code generation. An overview of supported skeletons and their implementation is provided.

Chapter 5 deals with performance prediction and modeling. We review approaches to collecting complexity information, and describe an approach based on profiling through structural operational semantics. Performance formulae are given for supported skeletons.

In chapter 6 we discuss how to transform a functional program into an internal representation for manipulation in the compiler. We advocate a process network representation of nodes and arcs that can be structured into pipelines. We show how the process network can be derived from the source program, and point out implications for performance prediction.

Chapter 7 deals with how to map the process network down to the physical hardware. We outline a general algorithm, focusing on distributed memory transputer based machines. We also discuss how program transformation techniques can be used for performance improvement, and describe the transformation system of the SkelML compiler.

In chapter 8 we present experimental results of running the compiler against a set of three test applications. We outline the programming and compilation of each program, and discuss performance figures.

Finally, chapter 9 contains a summary of the achievements and limitations of our work. We draw some conclusions, and point to avenues of future research.

The appendices are organised as follows: **A** contains SML definitions of various functions referred to in the thesis, including the higher order functions used to represent skeletons, **B** contains a listing of transformation rules supported by the compiler, and **C** contains listings of SML source code for example programs.

Chapter 2

Background

The first section of this chapter examines of the problems associated with the development of parallel software. A possible solution to these problems is offered by functional programming languages. In section 2.2 we introduce the principal concepts of functional programming and give an overview of sequential implementation techniques for functional languages. Section 2.3 reviews a number of approaches to using functional languages for parallel programming. In section 2.4 we summarise our conclusions and formulate the objectives of the thesis.

2.1 Parallel Programming

For parallel computers to be used widely, it seems a fundamental requirement that they are easy to use, i.e. not much harder than are traditional uniprocessor computers today. We will examine two key points identified as being problematic: the development and the portability of parallel software. Developing and maintaining software for traditional sequential computers is already far from trivial, and as we shall see in section 2.1.1, parallel programming involves an additional number of very complex issues. Sequential software can normally be moved from one platform to another with relative ease, and with few surprises regarding performance. In section 2.1.2 we discuss the problems associated with the portability of parallel software.

2.1.1 Program Development

Parallel programming inherits most problems associated with sequential programming, but there are also a number of additional issues which are unique to the development of parallel software. Depending on the choice of programming methodology and tools these issues will have to be addressed either by the programmer, the compiler or both.

- **Identifying Useful Parallelism**

  From the problem description an algorithm must be developed which exploits potential parallelism inherent in the problem. This normally involves partitioning the program into sequential chunks of code that can run in parallel. However, the fact that two tasks can be solved in parallel does not imply that doing so will yield a better performance than executing the two sequentially. It may very well be that the cost of communicating data between processors outweighs the gain of parallel execution. Hence, there is a difference between *useful* and *non-useful* parallelism. There are several degrees of useful parallelism depending on the number of processors that could be usefully employed for solving the problem. Making successfully such distinctions requires a very detailed knowledge of the characteristics of the target hardware platform as well as of the run-time behaviour of the code. Insight into the program's behaviour can be gained through profiling, or the effect of various approaches to exploiting parallelism can be observed and explored on a try-and-fail basis in series of test runs. Thus, the successful identification of useful parallelism seems to require either extensive knowledge or systematic experimentation.

- **Resource Allocation**

  To exploit useful parallelism, the program must be mapped down to the physical resources of the target architecture. This may involve grouping tasks together and scheduling their execution on the same processor, balancing the total load of computation evenly between the processors, multiplexing logical communication channels over a single physical link or routing data through intermediate processors. It may well be that limitations in the available resources prevent the full exploitation of parallelism, in which case it would have been better if this could have been taken into account at the very early stages of developing the program. In the world of sequential programming these problems are in effect non-existent, as there is a simple one-to-one mapping between program and processor.

- **Managing Process Interaction**

  Writing a parallel program is not just a matter of writing a number of sequential threads of code. These threads will normally have to cooperate in some way, for example by explicit communication. This holds a great potential for problems, and care must be taken to avoid the pitfalls of deadlock, starvation and non-termination.

- **Coping with Non-Determinism**

  When processors run independently of each other, it is not always possible to predict the state of every processor at every given time. As a result, parallel programs may not be deterministic, i.e. two separate runs of the same program with the same input may give different results. This may give rise to very subtle errors, which may or may not show up during a given test run. Careful use of synchronisation mechanisms must be enforced to ensure that non-determinism does not impose undesired effects.

- **Ensuring Program Correctness**

  As parallel programming is more complex than sequential programming there are more things that can potentially cause errors. While verifying the correctness and proper working of sequential software is demanding enough, doing the same for parallel software is much harder, in practice often quite infeasible. For parallel programs, programming errors may manifest themselves in many ways, including loss in performance. Note that as long as there is a possibility of non-determinism, merely testing the program against a suite of test data is by no means a guarantee of correctness, since a subsequent run may give a different (and incorrect) result.

- **Debugging**

  Debugging is an important part of program development with respect to tracking down errors and tuning performance. However, with parallel computers featuring not only a few tens or a hundred processors, but many thousands, it is hard to imagine a debugging tool, let alone a programmer, potent enough to have a reasonable overview of the state of computation.

In languages where parallelism is controlled explicitly, the responsibility of handling some or all of these issues rests mainly with the programmer. This makes developing correct and efficient parallel programs a very difficult task. But development is only the very first stage of a program's life cycle. Every time the program is modified, all the above problems are manifested again. Maintaining a piece of parallel software is bound to be both difficult and costly, even if the code is relatively transparent and well-documented. The parallel equivalent of imperative "spaghetti-style" code is of course even less appealing to look after.

While taking an existing sequential language and adding explicit parallel constructs is an easy way of making parallelism available to the programmer, it introduces a myriad of complex problems. Hence we believe this approach is not a good basis for parallel program development. For especially interested enthusiasts working on very specific problems this may still be a barely adequate solution, but it cannot be the basis for the mainstream use of parallel computers. Many current approaches such as occam2 and various C and FORTRAN dialects suffer from these problems. It is also worth noting that the general trend in programming languages is towards increasingly more powerful abstractions, rather than "empowering" the programmer with the responsibility for dealing with low level and machine dependent details.

2.1.2 Portability

Programs typically have a much longer life span than the hardware on which they run. It is therefore highly desirable that the moving of software from one platform to another should pose relatively few problems. This is broadly the case in sequential programming because there are few differences in the underlying architectures. Parallel computers however, come in many different flavours with respect to processor and memory specifications and topologies, and it seems likely that this will not change in the near future. Migrating parallel programs between widely different platforms is not trivial, because the way in which parallelism is best exploited depends on the machine architecture. Even between very similar architectures portability is still hard. Consider for example upgrading a machine with slightly faster processors. This changes the balance of computational power versus speed of communication. As a consequence, what was before considered to be worthwhile doing in parallel may now in fact be done faster on a single processor. Hence, only minor changes to the hardware may imply redeveloping large parts of the original program.

2.1.3 Implicit vs Explicit Parallel Programming

We have seen that parallel programs are hard to develop and costly to maintain, and that portability between parallel computers is low. The one main cause of this is that parallel programs contain implicit or explicit assumptions about the target parallel architecture. These assumptions create strong dependencies between the source program and the target platform. Adopting a higher level approach to parallel programming, many of these dependencies could be removed with the following consequences:

- The programmer's task becomes easier because there is no longer a need for making accurate judgements and decisions about parallelism. Consequently the development and maintenance of programs become easier, and there is less scope for programmer error.
- Programs become portable, because there are no detailed low-level descriptions of parallelism that tie them to a particular platform

By taking an implicit approach to parallel programming, the task of dealing with low-level issues of parallelism is shifted from the programmer to the compiler. However, as we shall see in section 2.3.1 the automatic efficient exploitation of implicit parallelism has proved to be problematic. Human intuition, knowledge and insight seem to be essential in producing good parallel implementations. This is a classic and recurrent dilemma in computer science: abstraction versus performance.

Knowing that automatic parallelisation is hard, it is paramount that the task of the parallelising compiler is made as easy as possible to begin with. This includes choosing a programming language or -paradigm in which parallelism can be easily identified, extracted and reasoned about. Much effort has centered around extracting parallelism from imperative languages, most notably from FORTRAN programs. This appears appealing because there are already a very large number of such programs in use today which could benefit instantly from such parallelisation. However, imperative languages are typically tied to a step-by-step sequential model of evaluation. Imperative programs also generally contain side-effects, and this severely complicates the process of finding parallelism. Detailed analysis is needed to discover which parts of a program are dependent on others, both with respect to data and control flow. A general overview of such techniques is given by Banerjee *et al* [18], while Girkar and Polychronopoulos [80] and Sarkar [146] discuss techniques used in specific compilers. Such analysis introduces unnecessary complications into the process of finding parallelism, which could have been avoided by using a more "compiler-friendly" language.

The problem of imperative languages being closely tied to the von Neumann architecture with its step-by-step model of evaluation was pointed out by Backus [15], who proposed a higher level of programming based on a mathematical or functional notation. Functional programs contain easily detectable implicit parallelism [128], and the functional paradigm therefore seems well suited for parallel evaluation. In the following section (2.2) we give a brief introduction to the basics of functional programming and review techniques for exploiting parallelism in functional programs.

2.2 Functional Programming

Functional programming has attracted research attention for more than thirty years, its clean semantics making it an attractive vehicle for investigating various programming language concepts. Current active areas of research include type systems, automatic memory management, state, as well as parallel and sequential implementation techniques. In this section we give a brief overview of functional programming; its foundation, its virtues, its drawbacks and its implementation. This is done mainly for two reasons: firstly to demonstrate that there are several important reasons why writing programs in a functional language may be preferable to using a traditional imperative one. Secondly, we aim to introduce a basic notion of functional programming that will be needed in the remainder of the thesis.

2.2.1 Background

Church's lambda-calculus [49] is the formal basis of modern functional programming languages. In fact, one may think of a functional programming language as the lambda-calculus wrapped in a user friendly syntax to facilitate its semantic power [120]. It has been shown that the lambda-calculus can compute anything a Turing machine can compute [108]. Functional languages based on the lambda-calculus can thus claim to be computationally complete in this sense. We do not intend to give a complete or comprehensive introduction to lambda-calculus here, but will rather present properties and terminology of the calculus relevant to the rest of the thesis.

An expression in the pure lambda-calculus is composed solely from three syntactic objects: function abstractions, function applications and identifiers. An application is reduced (or evaluated) by replacing occurrences of the function's formal parameter with copies of its argument (β-reduction). A reducible expression is called a *redex*, and the evaluation of a lambda-expression takes place by repeatedly identifying redexes and reducing them. When there are no more redexes, the expression is in *normal form*. In *applicative order* (also referred to as *strict* or *eager*) reduction, the argument in a function application is reduced prior to doing the β-reduction. In *normal order* reduction the β-reduction is performed directly with the unevaluated argument. Regardless of which reduction order is used for evaluating a lambda-expression, two different results cannot be reached. This very important property of the lambda-calculus follows from the first Church-Rosser theorem

[48], and implies that a lambda-expression can be evaluated using any combination of reductions. This freedom allows for the reduction of redexes in any order, in principle also in parallel, without affecting the final result.

Evaluation of a lambda-expression may however not terminate. The second Church-Rosser theorem states that if a normal form exists for a lambda-expression, this normal form can be reached by normal order reduction, i.e. by always reducing the leftmost outermost redex first. From an implementational point of view, however, normal order reduction may lead to function arguments being evaluated two or more times instead of just once. Applicative order reduction avoids this, but may lead to non-termination. *Lazy* evaluation takes the best from both by delaying evaluation of arguments until their values are needed. Examples of strict functional languages include Hope [35] and FP [15], while Miranda[1] [166] and Haskell [96] are based on lazy evaluation.

A very attractive feature of the lambda-calculus is that side effects can not occur. The value of a function expression depends only on the value of its arguments, and the evaluation of an expression has no other effect than producing the result value. This implies that any expression can be replaced by its value at any time, a property known as *referential transparency*. Functional languages in which functions are the only control structure and which do not allow side-effects are *pure*. Functional languages like Standard ML [175] and Scheme [160] allow side effects and are thus *impure*. If side effects occur, the property of referential transparency is generally lost, and so is the freedom of reducing redexes in any order.

It is fully possible to write complete programs in the lambda-calculus. Modern functional programming languages tend however to favour a less rigid syntax, and to provide a number of features to facilitate software development. **Pattern matching** allows the function body to be specified on a case-to-case basis depending on the value and/or structure of its argument [22]. **Polymorphic type checking** allows the writing of generic functions that can cope with arguments of many different types. Functional languages are also often **strongly typed** with **static type checking systems**, and support implicit

[1] Miranda is a registered trademark of Research Software Ltd

type inference such that there is no need for the programmer to specify the type of every identifier [90]. **Partial application** allows a function to be applied to the first few of its arguments only, returning a new function [169]. For example `(+ 2)`, the binary function `+` applied to only one argument, can be viewed as a new function that takes one argument and adds `2` to it. Functions allowing partial application are said to be *curried*. **Higher order functions** can take other functions as arguments and return functions as result values. Functions are thus treated as first class objects alongside with, for example, booleans and integers [97]. Viewed as a tool for software engineering, functional languages have a number of favourable characteristics. A functional program consists solely of a hierarchy of functions, each of which are dedicated to computing a specific value. At a suitable level of the hierarchy, functions may very well be viewed as independent modules responsible for a well-defined part of the program's computations and with a tidy interface (input parameters and result value). Thus, details of the program can be efficiently dealt with at a low level of the function hierarchy, which gives a **high level of abstraction**. The programmer is enabled to view a program as a collection of high level units without getting hung up in computational details [90, 175]. Referential transparency and the formal mathematical basis of the lambda-calculus eases the task of reasoning about a function's or a program's behaviour, thereby facilitating **verification**, for example by induction proofs [22]. Because there are no side-effects, functions are in some sense independent of the program context. This encourages **re-use of code** in different contexts without modifications [175]. A common feature of functional language implementations is **automatic memory management** [131]. This frees the programmer from explicitly allocating and deallocating memory cells, which makes programming simpler and removes an important source of errors. Functional languages are often said to support **declarative programming** because programs can be written in a mathematical manner often corresponding closely to problem definitions.

2.2.2 Sequential Implementation Techniques

This section looks at the different techniques used for implementing functional languages on uniprocessor systems. For the sake of uniformity over functional languages, we may assume that the source program is transformed into some form of (possibly enriched) lambda-calculus. The steps of such a transformation are covered in detail by Peyton Jones [129].

Stack Based Reduction

The SECD machine, proposed by Landin [112], is an an abstract machine whose state is given by four explicit stacks [77]. Program evaluation, or reduction, takes place by examining the (sub)expression ready for evaluation, from the top of the *control* stack, and depending on it, changing the current state of the four stacks according to a state-transition function. An expression is evaluated within an explicit environment, represented in the *environment* stack, holding the values of the free variables in an expression. The *dump* stack is used for keeping previous states of the machine, to be restored on return from function invocation. The *S-stack* is used for storing and retrieving objects, e.g. results from intermediate computations. When the evaluation of an expression is completed, the result is at the top of the S-stack. The most outstanding characteristic of the SECD machine is the explicit environment holding the values of free variables. In this respect the SECD machine is similar to imperative block-structured languages.

The SECD machine can be used for both lazy [66] and eager evaluation [77], but is best suited for the latter. Cardelli's Functional Abstract Machine (FAM) [43] is a fast, optimised SECD machine for applicative order evaluation.

Graph Reduction and Combinators

In the graph reduction model of evaluation a lambda-expression is represented as a directed graph. Constants, built-in functions and variables are represented as leaves, lambda-abstractions as lambda nodes, while function application is represented as binary apply-nodes [129]. The graph is reduced by repeatedly overwriting apply-nodes with the value of its left branch (the function) applied to its right branch (the argument). An immediate advantage of the graph representation is that common subexpressions can be shared conveniently. Bound variables in a function body are instantiated with a *pointer* to the argument rather than a copy of the argument itself. When the argument is evaluated, all the pointers to it automatically benefit. This makes graph reduction particularly well suited for lazy evaluation.

The main operation of a graph reduction machine is that of performing β-reductions by creating an instance of the function body where occurrences of the bound variables are replaced with (pointers to) the arguments. It is therefore of highest importance that this particular operation can be performed efficiently, preferably such that code for β-reductions can be fully created at compile time. When the body of a lambda-abstraction contains free variables, its value depends not only on those of its arguments. The function body will be instantiated differently in different runs of the program depending on the free variables. This is an obstacle to efficient compilation.

Schönfinkel [148] showed that it is possible to abstract free variables from an expression by using a fixed set of *combinators*. Combinators are functions (or lambda-abstractions in the lambda-calculus) containing no free variables [19]. This was shown to be possible also for the lambda-calculus by Curry and Feys [57], and was used by Turner for the implementation of SASL [167]. Free variables can be extracted from a lambda-expression using a bracket abstraction algorithm as illustrated in [70]. The two combinators **S** and **K**, defined by

$$\mathbf{S}\ \mathsf{f}\ \mathsf{g}\ \mathsf{x} \rightarrow \mathsf{f}\ \mathsf{x}\ (\mathsf{g}\ \mathsf{x})$$

$$\mathbf{K}\ \mathsf{x}\ \mathsf{y} \rightarrow \mathsf{x}$$

are sufficient for expressing any lambda-expression, but in a real implementation (such as SASL) a further few may be added to improve efficiency. The main problem with SK combinators is that the execution steps are very small. The overhead of linking the steps together during execution may thus be large compared to the amount of useful computation performed.

Hughes [98] generalised the idea of SK combinators by introducing combinators especially designed for the particular functions in the program. A supercombinator is a lambda-abstraction containing no free variables, and whose body contain no lambda-abstractions which are themselves not supercombinators [129]. A supercombinator is efficiently instantiated by simultaneously replacing all occurrences of its bound variables by (pointers to) their values. Supercombinators can be derived by a process of repeatedly allocating lambda-abstractions containing no inner lambda-abstractions, and replacing them with supercombinators where the free variables are made parameters.

Supercombinators are normally much larger than SK combinators, so the "grain" of computation becomes bigger, to the benefit of efficiency. Augustsson and Johnsson's Lazy ML compiler [13] is an efficient implementation of lazy supercombinator graph reduction based on the the G-machine abstract machine [100]. A similar technique is used in the Glasgow Haskell Compiler [132].

Hudak and Goldberg's serial combinator [94] is a supercombinator which contains no concurrent substructure and which is not a part of a larger combinator with the same property. The main motivation for this development is to facilitate the execution of functional programs in parallel by assuring that the body of the combinator can be executed sequentially with no risk of parallelism being lost.

Evaluation Strategy and Performance

A key question in the implementation of functional languages, and for high level languages in general, is the efficiency of their implementations. This is also of importance to parallel implementations, since the time for computation of sequential tasks heavily affects parallel performance as well as the identification of useful parallelism. The performance of functional language implementations has for a long time been viewed as comparing unfavourably with those of imperative languages. However, the performance of functional language implementations, strict as well as lazy, has improved drastically over the past few years. In [13] results are quoted showing both the Lazy ML and SML-NJ compilers being within a small factor (2-5) of that of C and Pascal implementations on a small test suite. In [41] convincing results are presented showing the strict functional language SISAL being fully competitive with, and in some cases faster than FORTRAN for large scale scientific computing tasks on a Cray Y-MP/864. [72] quotes results for a floating point intensive benchmark program for a variety of functional language implementations showing many, both lazy and strict, to be within a small constant factor (1-3) of that of a C version of the program. The performance of functional languages is constantly improving, and at the present time, should not be a major issue discouraging their use. An exception may be for a very narrow range of applications where peak performance is the paramount concern.

The choice of lazy or strict evaluation strategies is a recurring topic of discussion in functional programming circles. Lazy languages allow certain elegant programming techniques to be used, for example infinite data structures. The evaluation strategy is however more complex than that of their strict counterparts, which is reflected in slower execution speeds [129]. On the other hand, it is debatable or unclear both to what degree the claimed benefits of lazy languages are of paramount importance, and also whether the execution time overhead of lazy evaluation is a significant inherent cost, or one which can be virtually removed by optimisation techniques. We will not pursue further discussion of this, but simply conclude that neither evaluation strategy seems to pose a major problem with respect to efficiency.

2.3 Exploiting Parallelism in Functional Programs

Almost all forms of computation contain some form of implicit parallelism either at an algorithmic level, e.g. in divide and conquer algorithms, or at lower levels, e.g. in (3+6)*(5+1) where the two additions can be performed in parallel. One main obstacle to exploiting this parallelism is that up until very recently, the implementation of an algorithm has been synonymous with its *sequential* implementation. There has been no need to consider whether parallelism present at the algorithmic level would be preserved or extractable in the implementation of the algorithm. The idea of *implicit* parallel programming is that the programming language should preserve the parallelism available at the algorithmic level and facilitate the subsequent extraction of the parallelism from the program.

One should therefore aim to use a language which supports rather than resists the extraction of parallelism, and where parallelism is not hidden, but easily detectable [10]. Thanks to the Church-Rosser theorems, functional languages seem well suited in this respect [128]. The semantics of the lambda-calculus is in nature parallel, as no particular execution order is enforced. Thus, functional programs contain implicit parallelism at all levels. The price for this is however *not* an obscure and programmer hostile language. On the contrary, as we have seen in section 2.2, functional languages possess a number of other attractive properties just as tools for general software development. For these reasons, research has been devoted to exploiting functional languages for parallel processing. In the following we will discuss approaches ranging between two extremes:

- The **purely implicit**, where an ordinary functional language with no parallel additions whatsoever is implemented on a parallel architecture. The main problem with this approach is that compilers seem to have difficulties in deciding when parallel evaluation is worthwhile, and when standard sequential evaluation is preferable.

- The **purely explicit**, where a functional language is given extra syntactical constructs through which the programmer can inform the compiler that parallel evaluation should take place. Here the burden of explicit parallel programming is put back onto the programmer, whose skill and knowledge is instrumental to the efficiency of the parallel implementation.

In section 2.3.1 we consider implicit approaches, specifically the dataflow model of computation and parallel graph reduction. Section 2.3.2 discusses explicit approaches, while section 2.3.3 discusses an alternative implicit approach based on higher order functions as parallel skeletons.

2.3.1 Implicit Approaches

The Dataflow Model of Computation

The concept behind the dataflow model of computation is simple, and differs quite drastically from that of traditional von Neumann computing. A program is represented by a directed dataflow graph [67] in which the nodes are machine instructions and the arcs are paths along which data-carrying tokens travel. When data is present on each input arc of an instruction node, the node *fires*, i.e. it computes the instruction using the data on its input arcs as arguments, and passes the result out through its output arc. As the firing of nodes is driven by data availability, data is said to be active while instructions are passive.

Since several nodes may fire simultaneously, the dataflow model gives rich opportunities for parallel evaluation. The parallelism is of the finest granularity; that of machine instructions. An important property of the model which allows the parallelism to be safely exploited is that instructions can have no other effect than computing the output from its input arguments, i.e. side effects cannot occur.

As soon as a node has computed an output value, that value cannot subsequently be modified or in any other way change. In dataflow languages this is reflected in a single-assignment property, meaning that as soon as a variable has been assigned a value, that variable cannot later be updated with a new value. This may be viewed as attaching a name to a value rather than the imperative approach of assigning a value to a memory location (represented by a name). This property is the same as found in the lambda-calculus; as soon as a value has been associated with a bound variable in a lambda-abstraction, that binding cannot subsequently change.

There are two principal approaches to implementing the dataflow model [9]. The *static dataflow* approach allows only one token per arc at any time. The *tagged-token dataflow* approach allows an in principle unbounded number of tokens on each arc, but requires each token to carry a tag indicating its role in the computation. In the tagged-token approach, a node fires when all its arcs contain tokens with corresponding tags. A major effort in the development of static dataflow machines is the work of Dennis [69], which has been highly influential also in the development of tagged-token computers. The first tagged-token machine was developed in Manchester [86], while the work of Arvind *et al* at MIT (see below) has been highly influential. We give some examples of dataflow languages and systems:

- VAL [118] is a high level functional language designed for dataflow architectures. It aims to exploit implicit parallelism, e.g. in `forall` operations on arrays. It is also possible to enforce sequential execution by using a sequential iteration statement. VAL enforces single assignment on identifiers.
- Id [71, 123] (Irvine Dataflow) is a modern functional language with higher order functions, polymorphic type checking and non-strict reduction semantics. Implicit parallelism at a fine grain level is exposed by the compiler. Id is designed for execution on a dataflow architecture, and this is reflected in its features, maybe most notably in *I-structures* which are a type of array enforcing single binding (single assignment) restrictions on any location to prevent concurrent read-write problems. Implementations of Id include the MIT tagged-token dataflow architecture [11] and the MIT/Motorola Monsoon Dataflow System [92].
- SISAL [154] (Streams and Iteration in a Single-Assignment Language) is a functional language aimed at efficiently solving large scale scientific problems. It contains imperative-style iterative constructs like `while` and `forall`, but a single assignment principle must be obeyed. Parallelism is entirely implicit, and is extracted from for example iteration over (possibly multidimensional) arrays, overlapping producer-consumer processes on streams and parallel evaluation of function arguments. In [145, 147] Sarkar describes how to detect and exploit parallelism automatically, based on characteristics of the target multiprocessor and the execution profile of the program. To avoid too many small tasks a granularity threshold is used. Tasks

with execution times less than the threshold are considered not worthwhile of further expansion. SISAL is compiled down to an intermediate dataflow graph language IF1 [153]. Implementations exist on a number of architectures, both specialised dataflow machines and general purpose parallel computers. In [41] results are presented showing the performance of parallelised SISAL being fully comparable that of parallelised FORTRAN programs using 4 and 8 processors on a Cray Y-MP/864.

The absolute performance of dataflow systems is often hard to evaluate with respect to more conventional systems. This is partly due to the highly specialised hardware of dataflow architectures, and also because of major differences in the languages and compilers used [92]. To allow parallelism to be more efficiently exploited, a trend has been to extend the dataflow model to support threads of an appropriate grain size, thus allowing hybrid dataflow and control flow evaluation [79]. While simple and sound, there are doubts whether the extensive multithreading of dataflow systems is as attractive as originally thought. Culler *et al* point out two fundamental limitations [56], namely that latency tolerance is limited in practice and that local scheduling policies are inadequate.

Parallel Graph Reduction

Most research into parallel implementation of functional programming languages has centered around parallel graph reduction techniques [128]. Note that the SECD machine presented in section 2.2.2 is a problematic choice for parallel implementation. Its explicit environment would have to be globally available, and would be expensive both to access and maintain. Supercombinator graph reduction on the other hand, seems well suited and has been adopted as the main mechanism of reduction in a large number of implementations.

The idea of parallel graph reduction is to perform simultaneous reductions on the program graph. Expressions or subexpressions ready for reduction constitute independent tasks that can be evaluated as parallel threads. Tasks are kept in a central or distributed store, from where they are distributed to the individual processors (PE's). The evaluation of a thread may lead to new tasks being created. Graph reduction is thus very dynamic in nature, as tasks are continuously created and executed. This places heavy demands on efficient mechanisms for granularity control, scheduling and resource allocation.

In the following we review a selection of parallel graph reduction machines. Note that many of these rely upon various degrees of programmer annotation to identify useful parallelism, and are thus not quite implicit.

- ALICE [62, 89] (Applicative Language Idealized Computing Engine) is a pioneer as a specialised parallel architecture for graph reduction execution of functional programs, with three prototype machines built. The program graph is derived from a functional program written in Hope [35], and is stored in a shared task pool as *packets* ready for execution. The PE's are called *agents* and evaluate the program by fetching an executable packet from the pool, rewriting it, and returning the result, which may be newly created packets, back to the pool. ALICE's hardware consists of up to 40 transputers (as agents), memory boards holding the distributed packet pool and an interconnection network. The major limitation of ALICE is that the grain size of the packets was found to be too small compared to the inherent overheads and latencies of communication [89, 103, 171].

- Flagship [103, 171] is a follow-up to ALICE with a number of changes. In Flagship's architecture, the processors have local memory with the idea that this gives faster access to local packets. Distributing the program graph in such a way that locality is preserved is however not easy, and in practice non-local access is hard to avoid. Flagship's performance failed to impress, with a 16 processor (M68020) machine struggling to keep up with sequential execution on Sun3 workstations [103]. Results for transaction processing tasks were however found to be more encouraging, indicating the architecture's suitability as a database engine.

- PRISM [32] (Parallel Reductions In Shared Memory) is based on lazy parallel reduction of Turner-style combinators from SASL programs. The machine architecture consists of 15 PE's each with 32Kb local memory and 16Kb globally shared memory. Access to shared memory is controlled by hardware locks. Very fine grain parallelism was not found to be worth exploiting, but with coarser grain and simultaneous evaluation of list elements up to four processors were usefully employed in relative terms. Conclusions drawn were that lazy evaluation and list data structures hamper parallel evaluation, and that a better approach would be eager evaluation and higher level data structures, e.g. sets.

- The $<\nu$,G$>$-machine [12, 101] is based on a modified parallel version of the G-machine [100]. Lazy ML programs are compiled down to supercombinators and translated into G-machine instructions. Parallelism is introduced by the programmer using SPARK annotations. An implementation on a Sequent Symmetry SM parallel machine showed speedups in the order of 5-11 on 15 processors for three benchmark programs when compared to a fast sequential implementation using the conventional G-machine.

- GRIP [130] (Graph Reduction In Parallel) is a special purpose architecture for parallel supercombinator graph reduction of Haskell programs. It is built from separate boards containing 4 PE's (M68020), each with 1Mb of local memory and an Intelligent Memory Unit (IMU) with 20Mb of memory. Up to 20 boards can be connected together with a fast high-bandwidth bus. One PE is assigned the task as system manager, while the others work as graph reduction machines. The IMU's jointly constitute a shared memory holding the program graph. Subgraphs ready for evaluation constitute tasks, and a program is evaluated by the PE's repeatedly grabbing tasks, reducing them, possibly resulting in new tasks being *sparked*. Various strategies have been examined to control the sparking of new tasks including programmer annotation and dynamic approaches based on the total work load on the machine [88]. GRIP has obtained absolute speedups on a variety of applications when compared to the equivalent Haskell programs running on a comparable uniprocessor system, see for example [117].

- CTDNet [85] is a machine for reducing lambda-graphs, using an eager reduction mechanism. It is implemented on CTD; a hardware architecture where PE-memory pairs are interconnected in an indirect hypercube. Z80's, and later, transputers have been used as PE's. Interprocessor communication takes place through message passing. CTDNet2 [176] is a further development aimed at real-time applications.

- The HDG (Highly Distributed Graph) Machine [113] is a lazy fully implicit approach implemented on transputers. The functional code is compiled down to instructions for an abstract machine, each of which corresponds to a macro in transputer assembler. Evaluation is based on Burn's Evaluation Transformer model of reduction [33]. The HDG machine has been implemented on a network of four fully connected transputers. Experimental results have shown speedups in the 3.5 range with respect to the parallel code running on a single processor, and a speedup of 2.2 with respect to purely sequential code. A major question is whether this approach is scalable.

- Alfalfa and Buckwheat are implementations of the functional language ALFL on commercial multiprocessors [81]. The functional language used is ALFL, a lazy language compiled down to serial combinators. A heterogeneous execution model is adopted, where graph reduction is combined with stack based sequential execution of the body of serial combinators. Alfalfa [82] is implemented on a DM Intel iPSC hypercube, where load balancing is performed by diffusion scheduling; a processor can only offload tasks to its nearest neighbours. For four benchmark programs relative speedups were in the range 2-7 on a 32 processor machine [81]. Buckwheat is implemented on an SM Encore Multimax multiprocessor, where tasks are kept in a task queue in the shared memory. Using 12 processors relative speedups ranged 7-9 for four benchmark programs [81].

- MaRS (Machine à Réduction Symbolique) [55] employs parallel graph reduction based on a fixed set of combinators. A pure functional language MaRS_Lisp is used, and parallelism is explicitly requested by the programmer. The MaRS architecture consists of five types of processors responsible for specific tasks, namely reduction, memory management, I/O, floating point operations and communication. As for

ALICE, an Achilles heel of the system is identified as costly communication between the reduction processors and the memory units.

- ZAPP [36] (Zero Assignment Parallel Processor) is a transputer based architecture for evaluation of process trees. A potentially large process tree is mapped onto and executed on a physical virtual tree of processors. A concern is to keep the process tree balanced. This is relatively easy to achieve for divide and conquer type of algorithms, for which very good results, e.g. a 39.9 speedup on a 40 processor ZAPP [128] have been reported.

- Concurrent Clean [124, 133] is a lazy higher order language designed for parallel graph reduction through term graph rewriting. Parallel evaluation is requested explicitly by the programmer through annotations in the source code. Concurrent Clean is compiled down to code for an abstract machine, the PABC machine, providing an abstraction over DM MIMD machines. Implementations exist for a 32 processor transputer machine, for heterogeneous networks of workstations, and for ZAPP. Results published in [105] show speedups in the range 1-11 on a 16 processor transputer machine for a suite of eight programs.

2.3.2 Explicit Approaches

There have been many proposals as to how functional programming languages can be extended to allow explicit programmer control over the invocation of parallelism. Many of the entries reviewed in the previous section are in this respect more explicit than implicit, since the programmer is required to indicate opportunities for fruitful parallel evaluation with various annotations. In Hudak's *para-functional programming* [95] the programmer can schedule expressions to be evaluated sequentially or in parallel, and even specify on which processor a given expression should be evaluated. Extensions have been made to ML, for example Concurrent ML [142] and paraML [17], where tasks (threads) and communication channels are explicitly handled. In Kelly's Caliban language [104] the programmer can specify network topologies of processes and arcs.

The general drawback of explicit approaches is that they place upon the programmer the burden of making decisions about how parallelism in the program should best be exploited. This requires deep knowledge of both the program as well as the characteristics of the target architecture. Since what constitutes useful parallelism depends on the hardware characteristics, unfortunate bindings arise between the program and the target platform. A subsequent migration of the program to another platform may require the whole program to be revised, as discussed in section 2.1.

2.3.3 Skeletons and Higher Order Functions

It has been observed that parallel programs written in explicitly parallel languages consist of two different kinds of code, often tightly interwoven; pieces of *task specific code* implementing the individual steps of the algorithm, and code for structuring the program into *patterns of computation and communication* for parallel execution [58]. It is typically only in the latter kind of code that there is a need for describing low-level machine dependent resource allocation and for dealing with those aspects of parallel programming known to pose the greatest problems. In the low level description of the parallel infrastructure of programs the programmer's unit of attention often ends up being that of the *individual* processor with concern for low level details. On the other hand parallelism is typically (and paradoxically!) being exploited according to a few well known parallel paradigms, each determining the operation and coworking of *groups* of processors. The number of commonly used patterns or paradigms is observed to be quite small. Pritchard *et al* [134, 135] discuss three broad principal classes:

- Processor farms, where the same operation is applied independently to a number of tasks. The tasks are distributed to independent worker processors which compute the operation on disjoint task partitions.

- Geometric, where a data structure is distributed uniformly between a number of processors, each of which is responsible for a spatial "area" or "volume" of the input. Processors may communicate with their neighbours.

- Algorithmic, where each processor executes a small fragment of the total algorithm and data is passed between the processors. This commonly results in processors aligned in pipelines.

In the first two categories parallelism comes from the distribution of data over identical, or almost identical pieces of code (data-parallelism), while in the third parallelism is obtained by distributing the code (process- or functional parallelism). Coddington [50] reports positively on the usefulness of distinguishing these paradigms in implementing computationally intensive problems on transputer arrays. He notes that implementing parallel communication harnesses is a non-trivial exercise, and that the provision of such harnesses would greatly simplify the coding of parallel programs. Kung [111] describes nine computational models, including processor farms, geometric, pipelines, processor rings and divide and conquer. He suggests that these models may aid in classifying tools for the automatic generation of specific parallel programs, and refers to *Apply*, a tool for generation of processor farm parallelism. Hey [91], with reference to the works of Pritchard and Kung, observes that a small number of models or paradigms are "both extremely helpful and widely applicable". Hey claims that while highly cost-effective applications may be produced by utilising these paradigms, their coding is complex, and requires experienced and highly-skilled parallel programmers. He refers to Hoare and May's *superoccam* language elegantly incorporating support for farms and pipes into occam. Blelloch has argued in favour of exploiting various forms of parallelism corresponding to the manipulation of aggregate data structures, see for example [152]. Chandy and Kesselman [47] predict that future parallel programming will see a drift from low-level towards high-level notations for parallelism, to the benefit of ease of programming, verification and portability. A concrete suggestion is reusable templates allowing a more implicit handling of common parallel operations.

This all suggests that the use of patterns in parallel programming is a fundamental concept which deserves attention. In his doctoral thesis Cole addressed this issue [54], and coined the term *algorithmic skeleton*. An algorithmic skeleton is an incomplete parameterised template specification of some parallel algorithmic computation. From the programmer's perspective the skeleton is a high level semantic description of an algorithmic operation with gaps left for problem specific procedures and declarations. To use a skeleton, the programmer must supply these missing parameters. The implementation of a skeleton is however completely hidden from the programmer. This allows the same program to be implemented differently on different architectures. The implementation on a given target architecture should make use of all available optimisation tricks with respect to performance, which is fully acceptable since the implementation is transparent to the programmer. The low level code for implementing a skeleton is mostly fixed, and can be implemented in optimised form once and for all. It can then be reused *ad infinitum*, each time supporting the particular task specific code with which it has been instantiated.

Cole noted that the functionality of many skeletons can be expressed elegantly as higher order functions (HOFs) in functional programming languages. When developing a parallel program a programmer would have at his/her disposal a fixed repertoire of useful HOFs, each corresponding to a particular skeleton. By instantiating HOFs with specifications for the particular sub-tasks to be solved, parallelism can be described with minimal concern for low level details. We will take the term skeleton to mean an abstraction over some computational operation which contains implicit parallelism, and which has part of its functionality parameterised.

There are two distinct sides to skeletons: *semantics* and *behaviour* [64]. The semantics describes the declarative meaning of the skeleton, i.e. the relationship between its input and its output. The behavioural side describes how the output is computed from the input. The separation of these two fundamentally different issues is the key concept that makes programming with skeletons attractive. From the programmer's point of view it means that he/she can concentrate on the semantics of the program, while leaving the low level details of exploiting and managing parallelism entirely to the compiler. By using skeletons there will be parallelism implicit in the program which can potentially be exploited efficiently on a parallel architecture. And because the parallelism is implicit, it has the potential of being exploited in different ways on different architectures, thereby enhancing portability.

A skeleton-based compiler should be able to produce an efficient parallel implementation from the skeletons found in a program. This involves instantiating template implementations of skeletons with the programmer's task specific code, and possibly making decisions about resource management, e.g. how many processors should be allocated to each skeleton. By using accurate performance models of each individual skeleton, the compiler has a good basis from which to work out an optimal instantiation. As the program then consists of fewer entities, each of which has a well modeled behaviour, automatic techniques can be applied for mapping and scheduling. In order to be attractive, such compilers would have to perform not much worse than a skilled programmer using an explicitly parallel language. Since much of the complexity of parallel programming can be hidden and dealt with inside skeletons, the compiler's task is made considerably easier. It has been argued that by restricting the exploitation of parallelism to that of skeletons only, a compiler should be able to do a satisfactory job [127].

In the following chapter we will discuss skeleton-based compilation and the embedding of skeletons into a programming system. Here we summarise a number of significant benefits that this can bring. There is **less programming** because the programmer no longer has to specify low-level details of parallelism. Not only does this relieve the programmer of a tedious and delicate task, but it also severely reduces the risk of programming error. Programming via skeletons also gives a **high level of abstraction**. The programmer can focus on functionality rather than the detail of implementation. Massive parallelism involving complex communication patterns between a large number of processors can be controlled by simple and powerful high level language constructs. Using skeletons helps structuring the parallelism into separate functional modules. A **modular program structure** makes programs more transparent and easier to understand, and also encourages **reuse of program components**. Because skeletons separate semantics from behaviour there is no direct connection between program and architecture. Any implementation which conforms to the semantic specification of the skeleton is tolerated, which leaves the skeleton implementor free to choose whichever implementation is the best on the platform in question. When moving to another platform, the program remains the same, while the actual implementation can benefit from the implementation of the skeletons on the new machine. Thus, skeleton-based programming encourages **portability**. Many skeletons can be implemented using a varying number of processors. As the usage of computing resources should not be specified in any way in the program itself, programs have potential for **scalability**.

A high level of abstraction often carries performance penalties. With skeletons this may not be the case. In fact by pre-implementing skeletons in a highly optimised form, it may even be the case that by using skeletons, one can achieve a **better performance** than by writing the code by hand from scratch. For many skeletons one can construct **accurate performance models**. This implies that it is possible to **predict the behaviour** of parallel computation which is a large step towards automatic management of computational resources. We will return to this topic in chapter 5.

Mapping a process network down to a physical architecture is traditionally a very complicated undertaking. However, using skeletons, process graphs become more structured, so that each skeleton can be mapped down to a whole group of processors. Thus, the process of **mapping is simplified**. We will examine this further in chapter 7. Automatic resource management involves making decisions about the degree of parallelism to be exploited. This may even imply making a decision not to exploit parallelism at all and instead make a sequential implementation of a skeleton.

Because a skeleton's semantics is separated from its behaviour, the compiler is free to make such decisions as it sees fit. By performing semantics preserving **transformations between skeletons** one can change one parallel program into another. Transformations can be used to derive a more efficient implementation, using performance models as a guideline, and also to transform the program into skeletons which are known to be efficiently implementable on some particular architecture. Transformations will be dealt with further in chapter 7.

Although promising "on paper" there is still not much practical experience with this approach. A few encouraging results have however been reported, for example [29, 172, 173] exploiting implicit HOF parallelism in SML prototypes to build parallel implementations. We conclude this section noting that skeletons seem to provide a good interface between the programmer and the hardware. The programmer benefits from a high-level structured approach to parallelism, while the very same property enables compilers to make good decisions regarding resource management.

2.4 Summary and Thesis Objectives

This section summarises our findings so far, and on that background we formulate the objectives of the thesis. Parallel computer architectures seem promising for responding to current and future ever increasing demands for more processing power. Parallel programming is however a very complex undertaking, and portability of parallel code is low. These problems can be largely solved by adopting an implicit approach to parallel programming. Functional programming has much to offer as a software development tool and due to its attractive semantic properties, a lot of implicit parallelism is present in functional programs.

Efficient exploitation of this parallelism by parallel graph reduction and dataflow approaches have however been mostly unconvincing. The many specialised hardware architectures constructed have been impressive developments, but have represented no breakthrough for functional programming as a paradigm for efficient parallel software development. In fact, for many parallel graph reducers, the problem is rather to make the parallel implementation run faster than the best *sequential* implementations. Both dataflow and graph reduction approaches aim to exploit parallelism in a *uniform* manner; i.e. all kinds of parallelism are exploited in exactly the same way. Darlington [60], with respect to graph reduction approaches, argues that when parallelism is exploited at a too fine-grained level, important information about the logical decomposition of a problem is lost. As often happens when the evaluation model of a high-level programming language is not accompanied by an efficient implementation mechanism, poor performance results.

Thus, we believe that rather than designing specialised machines to fit a particular model of execution, a more promising approach is to base a parallel programming model on what has so far proved to be successful in the programming of more conventional parallel computers. The skeletons approach is motivated by this, allowing different kinds of parallelism to be exploited in different ways on different computers, with a high level of abstraction.

The overall objective of the thesis is to explore the use of skeletons for parallel implementation of functional programming languages. Our philosophy is that as much as possible of the burden of dealing with parallelism should be shifted from the programmer to the compiler. This requires an implicit approach to parallelism, where the programmer should in no way be permitted to order explicitly pieces of code to run in parallel. To be more specific, we will

- propose a framework for integrating skeletons into a programming environment with respect to the choice of programming paradigm and language, identification and exploitation of skeleton parallelism, and the allocation of responsibilities between the programmer and the compiler.
- outline how these ideas can be incorporated in the design of a skeleton-based parallelising compiler, and discuss relevant issues in all stages of the compilation process.
- discuss the role and importance of performance modeling and prediction for skeleton-based compilation, and describe mechanisms for modeling the performance of communication and computation.
- elaborate on the internal representation and handling of skeleton parallelism in a compiler
- outline a framework for mapping in skeleton-based compilers
- investigate how skeleton transformations can be used for performance enhancement

We have developed a prototype compiler based on our suggestions, and applied this to some example programs to investigate its practical behaviour.

Chapter 3

Skeletons for Parallel Programming

This chapter reviews and discusses work and ideas related to the use of skeletons for parallel processing. We begin by reviewing a selection of proposed approaches exploiting skeleton parallelism. In section 3.2 we discuss in more detail some specific issues by which skeleton-based systems can be classified, and relate these to past and ongoing work. Section 3.3 points out which characteristics seem the most promising and significant for inclusion in a parallelising compiler. In section 3.4 we review and classify a selection of proposed skeletons and their implementation. Section 3.5 concludes the chapter with a more general discussion.

3.1 Review

This section presents a review of some of the approaches that have been proposed for using skeletons in connection with parallel processing. We do not aim to give an exhaustive review or listing of all past or ongoing activity in this field. Our aim is rather to present what we see as the main achievements so far, and to give a representative account of the avenues investigated. The study of skeletons for parallel programming is a relatively young field of research, which as it will be seen, is active and diverse. This is reflected in the variety of terminologies used. For instance skeletons may be referred to as templates, paradigms, patterns, forms or parallel constructs. The main concepts are however very often the same, and in the following we will largely ignore minor differences in the way these terms are used.

3.1.1 Cole's Approach

As we have outlined in section 2.3.3 the main principles put forward by Cole, we will here briefly supplement this with some further details. Cole proposed a programming environment based on a fixed menu of skeletons expressed as specialised higher order functions. The programmer's task is to select one skeleton as the outermost function in the program, whereafter the full power of the host language may be used to write the problem specific code with which the skeleton will be instantiated. The implementation of each skeleton should be tailor made to the target architecture to extract the maximum performance. Cole suggested the following four skeletons and analysed them with respect to implementation and asymptotic performance:

- Fixed Degree Divide and Conquer (FDDC) is a version of divide and conquer in which any problem which can not be solved directly is divided into a fixed number of sub-problems.
- Iterative Combination (IC) performs a type of reduction on a set of objects by combining pairs of objects according to a "best partner" criterion.
- Cluster (C) is a generalised combination of the previous two.
- Task Queue (TQ) is a version of a processor farm where worker processors pick tasks from a task queue, process them, and place any created tasks back into the queue until the queue is empty.

Cole described how each of these skeletons could be mapped down to a 2D mesh, and gave formulae describing their time complexity. To our knowledge no implementation based directly on this proposed system exists. The ideas put forward have however been highly influential and have motivated a range of research activities into skeletons.

3.1.2 The Bird-Meertens Formalism

What has become known as the Bird-Meertens Formalism (BMF) was developed by Richard Bird and Lambert Meertens, originally with the idea of providing a notation and a calculus for deriving efficient programs from problem specifications [21]. Most work has focused on a theory of lists [23, 159], but other data types have been explored as well, see for example [121] for a theory of multidimensional arrays. A theory in BMF describes the behaviour of a data type, and provides a set of operators on that type as well as transformational laws which relate them. For the theory of lists the operators include map (written $*$), reduce (written $/$) and filter (written $\triangleleft$). For unary functions f and g, an associative infix binary function $\oplus$ and boolean predicates p and q we have

$$f * [a,b,c,\ldots,z\,] = [f\ a,\ f\ b,\ f\ c,\ \ldots,\ f\ z\,] \qquad \text{(map with } f\text{)}$$

$$\oplus\ /\ [a,b,c,\ldots,z\,] = a \oplus b \oplus c \oplus \ldots \oplus z \qquad \text{(reduce with } \oplus\text{)}$$

while

$$p \triangleleft [a,b,c,\ldots,z\,] \qquad \text{(filter with } p\text{)}$$

gives a list of those elements from $[a,b,c,\ldots,z\,]$ for which p holds. Assume $\cdot$ denotes function composition and $+\!\!+$ denotes list concatenation. Some laws are:

$$(f \cdot g) * = (f *) \cdot (g *) \qquad \text{(map distributivity)}$$

$$(p \triangleleft) \cdot (q \triangleleft) = (q \triangleleft) \cdot (p \triangleleft) \qquad \text{(commutativity of filters)}$$

$$\oplus\ /\ (x +\!\!+\ y) = (\oplus\ /\ x) \oplus (\oplus\ /\ y) \qquad \text{(reduce distributivity)}$$

By repeatedly applying these and other laws it has been shown possible for a number of problems to start with a formal, possibly naive specification, and step by step convert it into an efficient program.

BMF operators like $*$, $/$ and $\triangleleft$ contain implicit skeleton parallelism, and the equational laws can be used for transforming one parallel structure into another. Skillicorn has over the past few years strongly advocated BMF as a model for parallel computation [155, 158]. BMF is abstract enough to allow programs to exist independently of any particular architecture, and its strong theoretical underpinning provides a good framework for engineering of correct software. Skillicorn has shown that BMF is universal over four important classes of parallel architectures; SIMD, tightly-coupled MIMD, hypercuboid and constant-valence MIMD [158]. In practice this means that the cost of executing a program on any of these architectures is of the same order as the cost in the model. A cost-calculus for BMF is given in [157], where it is also illustrated how the calculus can be used to guide the application of transformational laws. Work is in progress on using BMF as a basis for parallel programming systems, see for example [6], where the functional language Adl is mapped down to BMF.

3.1.3 P^3L

P^3L is the programming language of the Pisa Parallel Processing Project (P^4) [14, 16, 58, 127], developed by the University of Pisa and the Pisa Science Center/Hewlett-Packard Laboratories. The project is aimed at exploiting massive parallelism for DM MIMD computers. P^3L has two components: a sequential part used for writing algorithmic code in an imperative language, and a set of constructs for structuring parallelism. C++ is chosen as the imperative language, while supported constructs include:

- `farm`, for performing the same operation independently on all elements of a data stream.
- `map`, which decomposes a data structure, for example a matrix, into disjunct parts and applies the same operation independently to each part.
- `pipe`, for process pipelines
- `tree`, for process trees

- `geometric`, for more general data-parallel algorithms, where there may be interactions between the computations on each partition of the input.
- `loop`, for computation involving iterative execution of a loop body until some condition is met.
- `reduce`, which reduces the elements of a data structure using an associative operator.

There is also a `sequential` construct for encapsulating sequential blocks of code. A P^3L program consists of a hierarchical composition of constructs, each of which may again contain further constructs. Using a combination of suitable constructs, possibly nested, the programmer specifies all parallelism he/she wants to exploit. The flow of data between constructs is specified by `in(...)` and `out(...)` parameter lists in construct interfaces.

P^3L is compiled down to an abstract machine P^3M. The middle-section of the compiler deals with choosing one of several implementation templates for each construct by using heuristic rules. A number of tools can be invoked for optimising the program to target a specific architecture. The compiler uses a set of libraries containing the implementation templates and analytic performance formulae (*mapping library*), transformations (*optimization library*), and actual code for implementing the process templates (*process template library*). Libraries are supported for a 2D mesh topology. The modular design of the compiler allows other architectures to be targeted by supplying further libraries.

3.1.4 Darlington *et al*'s Approach

Darlington *et al*'s work at Imperial College, London, is based on the functional language Hope+ in which skeleton-parallelism is expressed through a set of higher order functions [60, 64]. The semantics of each skeleton is given by a higher order function definition in Hope+. There is a strong emphasis on using program transformation techniques [65] between skeletons to convert a program into compositions of skeletons known to be efficient on any particular target architecture. This is claimed to guarantee portability *and* efficient execution across a range of parallel machines, from SIMD to MIMD. Each skeleton-machine pair has associated a parameterised performance model enabling performance prediction, which is particularly useful for resource allocation. Supported skeletons include:

- PIPE for linear process-parallelism
- FARM for simple data-parallelism
- DC for divide and conquer parallelism
- RaMP, reduce and map over pairs, where for a list of objects, values are computed for any pairwise interaction of two objects.
- DMPA, dynamic message passing architecture, for systems of processes where the interaction between processes is dynamic, i.e. determined using run-time data.

In [61] a prototype implementation for a transputer-based architecture is described. Skeleton performance models are discussed, and the results of compiling and running a ray tracing example is presented with encouraging results with respect to both performance and the accuracy of performance models.

3.1.5 PUL

PUL (Parallel Utilities Library) is developed at the Edinburgh Parallel Computing Centre (EPCC) to provide a portable interface to the efficient programming of parallel computers [31]. PUL is developed on top of CHIMP, (Common High-level Interface to Message-Passing) which supports send/receive of single messages and which is implemented on a wide range of parallel platforms. PUL provides a library of utilities which support frequently used parallel paradigms as skeleton modules accessed through module interfaces. There are also more general, non-specific modules which support extended message passing services and I/O. PUL modules can be used both with programs written in C and in FORTRAN. Applications can be developed using a mixture of modules in combination with CHIMP primitives for explicit communication. PUL skeleton modules include support for task farms, divide and conquer, regular and spatial domain decomposition and mesh based problems. CHIMP/PUL was developed for the programming of EPCC's industrial and commercial work, and is used in the development of a number of industrial applications as well as by a range of academic institutions.

3.1.6 PUFF

Busvine's parallelising compiler PUFF (Parallelisation Using Farmed Functions) exploits processor farm parallelism through a wholly implicit approach [39, 40]. PUFF extracts farm parallelism from linearly recursive functions in SML programs. Consider for example

```
fun f x  = if Call x
           then Base x
           else Combine (f (Update x), Work x);
```

At each level of recursion, a farmer processor generates an individual task packet, `Work x`, that can be sent out to be processed at one of many worker processors. The farmer then collects the results and reconstructs the final result (`Combine`) as the recursion folds up. Although based on a simple idea, this is in fact a very powerful approach, because linear recursion is a principal and frequently used control structure in functional programs. PUFF relies heavily on profiling information and performance modeling to decide in each individual case whether a processor farm or ordinary sequential implementation should be used. PUFF supports the full process of compilation from the parsing of the source SML program and the collection of profiling data to producing the final parallel code in occam2. The system has been implemented for a transputer based Meiko Computing Surface, and for a number of programs encouraging results have been produced.

3.1.7 FAST/Caliban

The FAST (Functional Programming for Arrays of Transputers) project was a joint collaboration between Southampton University and Imperial College aiming to produce a highly optimised compiler for a parallel lazy functional language [1]. Parallelism is expressed through annotations in a restricted version of Kelly's Caliban language [104], which allows explicit programmer control over data partitioning and placement. Caliban introduces a notation for describing process networks within a functional language, where the interconnection topology is specified in terms of nodes and arcs. Descriptions of network topologies can be defined as ordinary higher order functions. The programmer can build a repertoire or library of commonly used process structures such as pipelines and fans which can subsequently be used and reused with a special `moreover` keyword to distribute functions to process nodes. The source language of FAST is Haskell, in which Caliban is implemented as a standard Haskell constructed data type. The power of this approach is that process network descriptions are themselves part of the source language which are amenable to analysis and simplifying transformations during the compilation process. In both Kelly's original work on Caliban [104] and in the collected FAST papers [1] there are several examples of this approach being applied to solve various problems.

3.1.8 NESL

NESL is a data-parallel functional language developed by Blelloch *et al* at Carnegie Mellon University [25]. NESL has an ML-like syntax, supports polymorphic types, and uses strict evaluation. Parallelism arises from the application of data-parallel constructs to sequences;

- apply-to-each constructs map a function over a sequence, using a form of list-comprehensions, thereby allowing sub-selection of sequence elements.
- sequence functions operate on whole sequences, for example reduction/fold with an associative operator or permutations of sequence elements.

The elements of a sequence can themselves be sequences. NESL allows data-parallel constructs to be nested, i.e. data-parallel constructs can be applied at different levels of a nested structure of sequences producing nested parallelism. NESL also supports the calculation of asymptotic complexity measures of run times both for serial and parallel execution.

NESL is compiled down to a portable intermediate language VCODE [26], flattening nested parallelism in the process. VCODE is implemented on a number of parallel platforms.

3.1.9 Object-Oriented Approaches

Object-oriented languages with their strong support for encapsulation and abstraction are natural candidates for supporting skeletons. Mentat [84] offers encapsulation of both intra-object and inter-object parallelism in an extension of C++. Intra-object parallelism arises from parallelism inside a member function of a class. Inter-object parallelism arises when invocations of several member functions can execute in parallel. It is the responsibility of the Mentat compiler and runtime system to ensure that data dependencies are satisfied and to handle communication and synchronisation issues. This makes issues of parallelism mostly transparent to the programmer, but it is still the programmer's responsibility to declare through annotations which classes/member functions contain useful parallelism, and also to give the compiler hints on process placement. Implementations of Mentat exist for a range of platforms, further details and performance samples are found in [84].

ODA (Distributed Active Objects) [42] is an object based specification scheme implemented on top of occam. An ODA module consists of a description of a virtual process topology together with code for handling its internal communication. An application process implements specific sequential algorithmic code. The virtual topology description is independent of the actual physical topology. The number of processors used can be parameterised and fixed at configuration time, while a router provides a communication path between any two nodes. Essentially the ODA approach allows the implementation of global ADT's, and for their control structure to be hidden from the user code. ODA has been used to develop several algorithms on transputer systems, for example divide and conquer, backtracking and branch and bound.

3.1.10 Other Approaches

Templates in Parallel Languages

Several parallel development tools have a concept of templates. The Linda Program Builder (LPB) [3] is a high level programming environment that aids in the development of explicitly parallel Linda [44] programs. An LPB template is a partly specified program component to be supplemented with the programmer's own task specific code.

Parsec (the Parallel System for Efficient Compilation) is developed at the University of British Columbia, and is an attempt to incorporate skeletons into an environment for parallel processing [74, 75]. Each skeleton has parameters such as the number of processors, topology and granularity, and when developing a parallel program values must be found for these parameters. The system supports collection of performance information during test runs, which can subsequently be analysed and used together with analytic performance models for each skeleton to derive values for the skeleton parameters. By repeating this process the application can be tuned supervised by the programmer through a graphical interface. Supported skeletons are processor farms and divide and conquer. The underlying logical processor topology is a tree, but the physical topology may be different. The objective of this work is to develop an environment which is easy to use, and which provides scalability and reuse of code.

Other Functional Systems

In his doctoral thesis, Stoltze describes a compilation system based on the strict functional language FPS, with stack based evaluation [161]. The approach is biased towards mathematical and technical programming, and has strong support for operations on array-structures. A uniqueness type system is supported, allowing in place updates in some cases. Variations of `map` and `fold` skeletons are supported, and also a range of more specialised skeletons for matrix operations. For example, there are skeletons for rotating matrix rows and columns between processors, giving the programmer control of data distribution. The compiler targets a transputer based distributed memory architecture, for which a number of performance results are given.

Rabhi and Schwartz [136, 137] propose POPE (a "Paradigm-Oriented" parallel Programming Environment), supporting the development of a particular kind of coarse grain data-parallel algorithms with an iterative control structure. POPE consists of several transformation modules which turn the initial functional specification in Haskell into an executable parallel program. This process is partly assisted by the programmer. Using the Glasgow Haskell Compiler and PVM, a POPE prototype will produce parallel C code to be run on a network of workstations.

Schreiner describes the design and implementation of pD, a parallel variant of the small and strict functional language D [149, 150]. pD offers several ways of expressing parallelism, for example by explicit task creation. Low-level primitives may be used to build skeletons as higher order functions. The motivation behind pD is to develop parallel computer algebra applications using a high level language, but without loss of efficiency compared to using a language where parallelism is handled at a lower level. pD is used as the interface to the PACLIB computer algebra package. The PACLIB kernel is implemented on a Sequent Symmetry shared memory platform, and the pD compiler produces explicit parallel C code for this machine. Results for several applications show that the code produced by the pD compiler executes with the same performance as that of the equivalent programs written in explicitly parallel C.

3.2 Issues in Exploiting Skeleton Parallelism

3.2.1 Choosing an Embedding Programming Language

In order to be practically applicable, the concept of skeletons must in some way be embedded in a programming language. The least sophisticated alternative is to program in an explicitly parallel language using skeleton harnesses that can be specialised "by hand" for each individual problem. With some knowledge of the performance characteristics of the harnesses, performance prediction should be possible. This technique has been used successfully to implement image processing algorithms from functional prototypes exploiting the correspondence between skeleton HOFs and harnesses [170, 172]. Using an explicitly parallel functional language, e.g. Concurrent ML [142], harness code can be re-used in a similar way [46], but with a smoother instantiation by using HOFs. Other related approaches based on functional programming include Caliban (described in section 3.1.7) and the Concurrent Functional Processes notation [27].

These approaches all facilitate “do-it-yourself” skeletons, where the programmer can build a library of useful constructs which can be instantiated and reused. However, one of the main advantages of skeletons is that the compiler should be able to relieve the programmer of resource allocation and optimisation tasks. This seems hard to achieve fully unless the detailed characteristics of each skeleton is well known to the compiler, i.e. skeletons should be a native part of the programming language. Further, the ability to express parallelism in other ways than by using a skeleton invalidates the advantages of simplified mapping through a restricted interface to parallelism. PUL’s (see section 3.1.5) library-based approach suffers from this because unless the compiler knows all available skeletons in-depth and the relationships between them, certain optimisations such as transformations are harder to perform.

An important factor which makes the compiler’s life easier is the amenability to analysis of the embedding programming language. Functional languages are favourable to their imperative counterparts in this respect. In addition, skeletons can be elegantly expressed and conveniently manipulated as HOFs, and for program transformation there is already a rich body of work from which to draw experience. P^3L is functional in the sense that referential transparency is maintained at the level of each skeleton construct, but the way in which the constructs are integrated with the imperative language still introduces an impedance mismatch, e.g. the programmer must do a bit of “plumbing” to direct the flow of data between skeletons. The nativeness of HOFs in functional languages makes for a much smoother integration. A more drastic approach is to use a language which is based wholly or mostly on higher order constructs, e.g. FP. While skeleton parallelism is certainly plentiful in such a language, the style of programming enforced may make the expression of certain algorithms less than intuitive. Approaches based directly on BMF may exhibit the same problem. Nevertheless, a proposal for skeleton-based implementation of FP is described in [59].

3.2.2 Identification of Skeletons, Implicit vs Explicit

At an early stage of the compilation process, a compiler must identify the skeletons present in the source program. By identifying each skeleton by an explicit constructor or HOF, this is a trivial task. However, this has the apparent drawback of requiring the programmer to code the algorithm using a rather restricted set of constructs by which to express parallelism. Programming is already a hard task, and as a general principle algorithms should be allowed to be expressed in whichever way the programmer finds most natural. Busvine's PUFF compiler showed that farm parallelism could easily be extracted from linear recursion. Pipeline parallelism can be found in function composition [172]. We believe it should also be possible to identify general forms of divide and conquer, for example

```
fun f x = if p x then q x else Ef x;
```

where `p` and `q` are not dependent on `f`, and `Ef` contains at least two (recursive) calls to `f` [68]. `p` can be seen as a check for triviality while `q` solves a base case trivial problem. Good results with implicit forms of divide and conquer parallelism were obtained on ZAPP [36] using a virtual tree of processors. For more complex skeletons, for example various geometric domain decomposition strategies, such extraction may prove harder. We therefore believe that the use of some sort of explicit skeleton constructs is hard to avoid. The integration of such constructs should however be as smooth and seamless as possible. Adding explicit constructs to an existing language is in this context a rather crude approach. The use of HOFs in functional languages stands out as a more natural solution. HOFs are a native part of functional languages used because of their expressive power regardless of the potential for parallelism. The programmer's use of HOFs is thus just as much a description of functionality as it is an annotation for parallelism. An example of this is found in Data Parallel Haskell (DPHaskell) which exploits the implicit use of HOFs by extracting data-parallelism from map, scan and fold operations [93].

3.2.3 Skeletons - Directives or Annotations?

When a skeleton is found in a program, be this through the use of an explicit construct or some sort of analysis, the compiler can take this either as a directive to do parallel implementation or it can judge for itself whether parallel or sequential implementation is preferable. The skeleton may contain enough parallelism to keep a large number of processors usefully employed, but it may also contain no useful parallelism at all. This depends on the particular functionality the skeleton has been instantiated to support, the characteristics of the target architecture, and the input data to the skeleton. Without correctly establishing the amount of useful parallelism for each skeleton in a program, it seems very hard to sensibly allocate physical resources or to obtain an efficient program implementation.

If the use of a skeleton in a program is to be taken as a directive to produce parallel code, this severely undermines many benefits of using skeletons. The principal problem is that the final decision regarding the presence of useful parallelism is made by the programmer, according to his/her perception of the program, input data and hardware characteristics. Thus programs are no longer just semantic specifications, they contain implicit architecture dependent assumptions. On the other hand, if the compiler is to make such decisions it will need deep knowledge about the characteristics of the hardware, the program, and of the input data. A decisive question is whether these parameters can be adequately captured and efficiently used in a compiler. We subscribe to the view that hardware characteristics and the performance of skeletons can be modeled in formulae and constants, while the properties of the input data can be recorded through instrumentation of the program in profiling runs. A detailed discussion of these issues is found in chapter 5. For now we will be content with the conclusion that accurate performance modeling and -prediction are paramount to the success of automatic skeleton-based compilation.

3.2.4 Freedom of Expression of Parallelism

With respect to the programmer's freedom of expressing parallelism, there are only two options: either parallelism is expressed through skeletons only, or additional parallel primitives are supported. It may seem like a drawback that as far as parallelism is concerned, the only interface to the parallel computer is through a limited number of skeleton constructs. Hence, if the parallelism present in the original algorithm cannot be expressed by the available constructs, it cannot be exploited. Many long-term advocates of skeletons, e.g. Cole [54], Skillicorn [158] and Pelagatti [127] agree that allowing non-skeleton ways of exploiting parallelism should be banned. It is precisely the restricted and well structured exploitation of parallelism that allows accurate performance prediction and automated compilation. Thus, the apparent "gain" by increasing the programmer's freedom of expression is in fact a serious drawback which more than outweighs the apparent limitation of maintaining a skeletons-only restriction. This dilemma is in some sense analogous to allowing the freedom of using goto statements in imperative languages or allowing side effects in functional languages, which both destroy important properties; structured programming and referential transparency respectively.

3.2.5 Deciding on a Palette of Skeletons

If a fixed set of skeletons is to be supported, a criterion is needed for deciding exactly which skeletons to include. In general the set should be complete enough to represent efficiently all types of parallelism one may want to exploit. The experiences of Pritchard [135], Kung [111], Hey [91], and Danelutto *et al* [58] as reported in section 2.3.3 suggest that most common and useful parallel paradigms can be covered by in the order of ten principal skeletons. The most central and generally agreed upon are pipeline, farm, tree, and geometric decomposition. However, such a collection of skeletons would still be relatively general and might not allow the most efficient implementation of particular algorithms. Skeletons aimed at particular problem domains have been suggested, e.g. for graphics [164] and matrix multiplication [162]. PUL supports both paradigm specific and domain specific skeleton modules. Very specialised skeletons may be excellent for the tasks for which they are tailor-made, but have limited use for other purposes. Hammond [87] notes that a possible infinite number of skeletons might be needed for efficiently expressing parallelism in this way. More general skeletons have a greater potential for reuse in different contexts. It seems infeasible for a general purpose compiler to support a wide range of specialised skeletons by default. Such a compiler would be enormous, and rather inelegant. We would suggest a more dynamic design based on a fixed set of a few general purpose skeletons and then some library system for including very specialised functionality on a by-need basis. This requires a standardised format for specifying performance characteristics and implementation options for library skeletons. Transformations can be specified between library skeletons and the fixed set of general skeletons, while transformations between skeletons in different libraries may be harder to support. The problem still remains of how to select the fixed basis of general skeletons. A criterion of usefulness seems hard to define, but we believe that to include those skeletons which have proved to be frequent building blocks in parallel applications is a good place to start. BMF is claimed to possess an important property in this respect, guaranteeing a form of completeness in the set of operators for a type such that no crucial operation is missing [155].

3.3 Conclusions

In the previous section we identified and discussed a number of important characteristics of skeleton-based systems. Here we summarise what we conclude to be the most significant of these.

In general an implicit approach to parallel programming is preferable, provided efficiency is not severely compromised. With this in mind functional languages seem attractive because for the programmer there is no difference in paradigm between skeletons (as

HOFs) and other code, and there is also no conceptual difference between HOFs which contain useful parallelism and those which do not. Thus, the source program exists as an independent functional specification regardless of its content of parallelism.

Since the programmer shall not have to worry about useful and non-useful parallelism, their identification falls on the compiler. Any skeleton may or may not contain useful parallelism depending on the functionality it supports, the input data to the program, and the characteristics of the target architecture. A compiler will need strong performance modeling capabilities capturing the above factors in order to make accurate decisions about useful parallelism.

To facilitate efficient automatic resource management, parallelism must be in a highly structured form, i.e. as skeletons. No other way of expressing parallelism should be permitted. We also believe that the emphasis should be on a few general purpose skeletons rather than many specialised ones. As part of the resource allocation process it may be possible to apply transformations, either to ease mapping, or to improve performance. In any case, a skeleton-based compiler should have a transformation capability to take advantage of this.

Of all the proposals and systems we have reviewed, not many satisfy these requirements. Apart from the SkelML framework proposed in this thesis, only Darlington *et al*'s approach have indicated support for all of these features. However, not much is published about the status of this project regarding implementation details and performance. We therefore believe our work to be novel in that it is one of the very first to integrate support for performance modeling and profiling, optimising transformations and automatic resource management in a practical compiler implementation for general-purpose skeleton-based parallel programming.

3.4 A Classification of Proposed Skeletons

This section presents an overview of various skeletons proposed in the literature. Many are identical or almost identical, but it is interesting to note that there are still small but important differences. We have classified skeletons into six broad categories; pipelines, processor farms, reductions, process trees, geometric and miscellaneous. In the definition of skeleton functionality we will make use of some auxiliary functions, which can be found in appendix A.

3.4.1 Pipelines

A simple process pipeline consists of two or more composed functions, the output of one function being the input to the next and so on. Figure 3.1 shows a pipeline corresponding to the composition `f1 (f2 (f3 x))`.

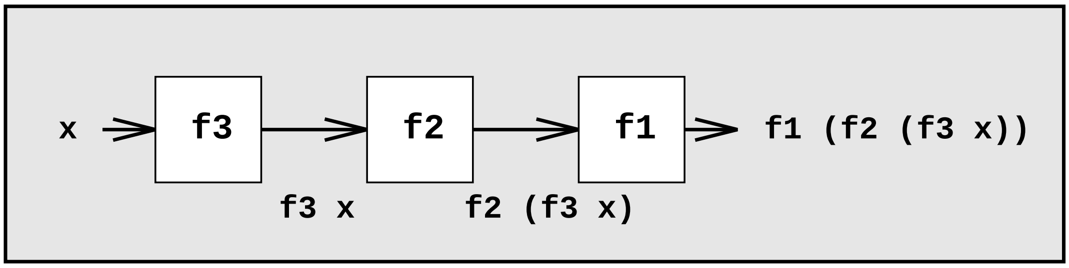


Figure 3.1: Process pipeline

Parallelism arises when the pipeline is applied to a stream of input data and the computation of the pipeline functions is distributed over several processors. The definition

```
fun pipe fs = foldr (op o) id fs;
```

where `fs` is a list of functions and `(op o)` the function composition operator is often seen, e.g. [64, 138]. This has the drawback that all functions must be of the same type $\alpha \rightarrow \alpha$ since lists are traditionally type homogeneous. Extending or re-designing the type system to allow heterogeneous lists in this special case, or introducing a new sort of sequentiation type are *ad hoc* approaches that can be used to overcome this.

3.4.2 Processor Farms

A number of skeletons have their natural implementation as a processor farm, which is a popular and powerful technique for applying the same function to each element of a stream of data. A typical implementation, as described for example in [102], is depicted in figure 3.2. Each *worker* processor in the farm computes the function independently on a disjoint partition of single elements from the data stream. The *farmer* processor distributes data elements to the workers and then collects the results.

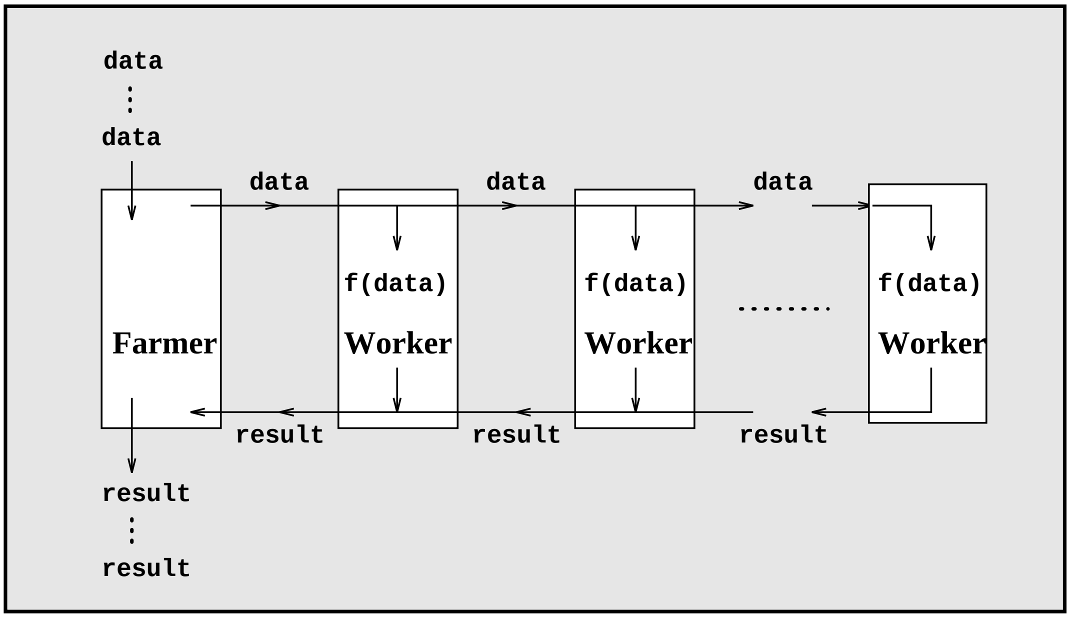


Figure 3.2: Processor farm; linear chain of workers

The corresponding HOF definition is:

```
fun map f [] = [] |
    map f (h::t) = (f h)::(map f t);
```

Other topologies than a linear chain have been studied, for example trees, which give a shorter distance to the most distant worker, but this may not necessarily be a more efficient solution [83]. Variations of `map` include `filter` and `map2`:

```
fun filter p [] = [] |
    filter p (h::t) = if p h then filter p t else h::(filter p t);

fun map2 f [] [] = [] |
    map2 f (h1::t1) (h2::t2) = (f h1 h2)::(map2 f t1 t2);
```

A more general alternative is Rabhi's distributive independent skeleton [138]:

```
fun di f n divide combine prob = combine (map f (divide n prob));
```

where `n` is the number of processes created, `divide` divides `prob` into `n` slices and `combine` combines the `n` part solutions. Depending on the resources available, an implementation might in some cases be able to decide an appropriate value for `n`.

3.4.3 Divide and Conquer Process Trees

Trees are most commonly used for divide and conquer type parallelism. Figure 3.3 shows a balanced binary tree indicating the flow of problems, sub-problems (`prob`), solutions and part solutions (`sol`). Each process executes the same piece of code; if the incoming

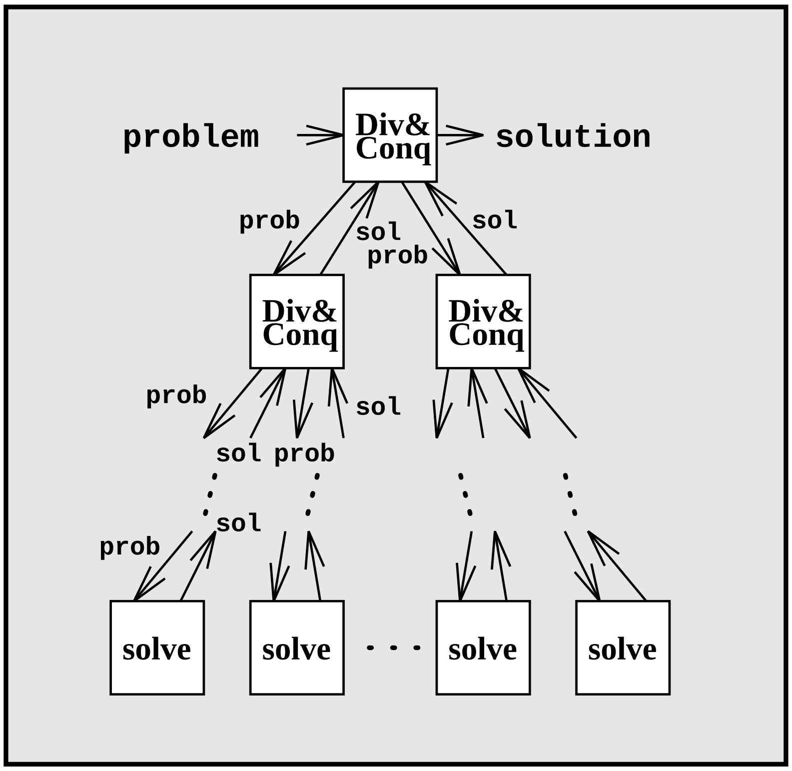


Figure 3.3: Binary divide and conquer process tree

problem is trivial it is solved locally, otherwise it is divided into sub-problems and passed down the tree. Solutions are formed by combining part solutions from lower processes. A general, i.e. not bounded by creating a fixed degree of sub-problems, definition is [54, 64]:

```
fun dc trivial solve divide combine prob =
   if trivial prob
   then solve prob
   else combine (map (dc trivial solve divide combine) (divide prob));
```

A variation is Rabhi's recursive partitioning skeleton [138] given by:

```
fun rp trivial solve divide combine prob =
   if trivial prob
   then solve prob
   else combine prob (map (rp trivial solve divide combine) (divide prob));
```

which differs from `dc` in that `combine` takes as parameter also the original problem `prob`. [138] gives examples of its use.

The tree of processes can be mapped down to physical processors in a number of ways depending on the architecture. [68] notes that this may involve problems since a divide and conquer tree is a very dynamic structure unfolding at run time. A transformation to a process pipeline is suggested, which may be a better implementation route on relatively static architectures.

3.4.4 Reduction Process Trees

Reduction operators most commonly take the form of / in BMF or variations of `fold` in functional languages,

```
fun fold f a [] = a |
    fold f a (h::t) = f h (fold f a t);
```

If the binary function `f` is associative, parallelism can be exploited as shown in figure 3.4, where `@` represents an infix version of `f`. In [116] an interesting variation is given when a function is applied to each element of the input prior to the reduction:

```
fun foldseq f g a [] = a |
    foldseq f g a (h::t) = f (g h) (foldseq f g a t);
```

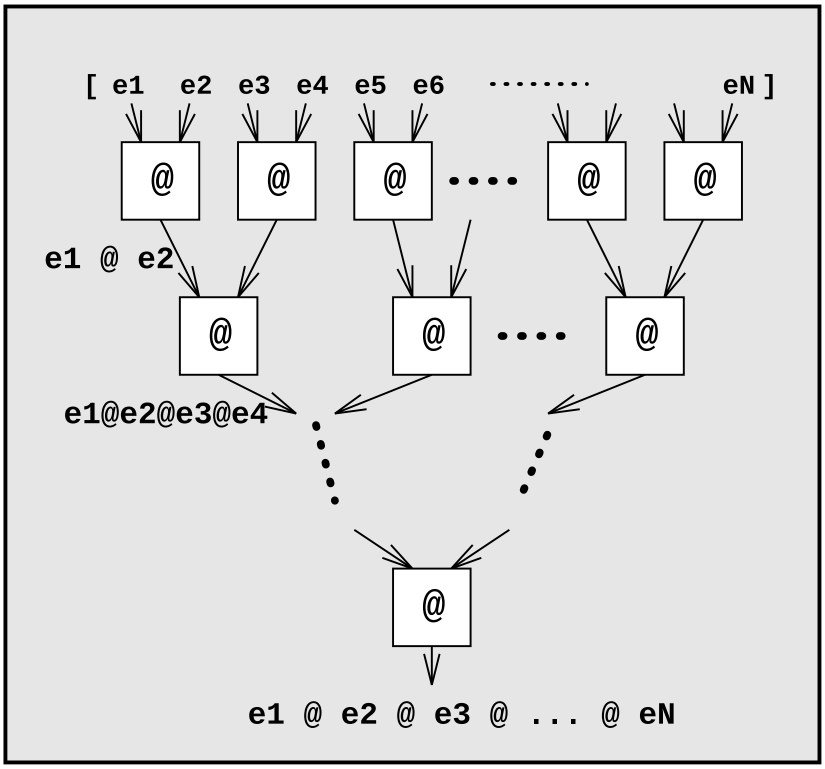


Figure 3.4: Binary reduction process tree

This is effectively equivalent to a composition of a `fold` and a `map`, i.e.

```
fun foldseq f g a = (fold f a) o (map g);
```

3.4.5 The Geometric Paradigm

The geometric paradigm captures a class of data-parallel algorithms more general than that of `map` [126]. The principle is still that each processor gets assigned a partition of the input domain on which some operation is computed, and the final result is assembled from the partial solutions from each processor. During the course of computation a processor may also need data resident on other processors, involving interprocessor communication. It is desirable to allocate partitions such that processors that need to communicate are physically close or directly connected, and such that the work load is balanced. This is relatively easiest to achieve for regular problem domains where a spatial area can be assigned directly to each processor, such that communication takes place on edges/surfaces between adjacent partitions/processors. Orlando and Vanneschi discuss the implementation of the geometric paradigm on a 2D mesh topology [126], while Pritchard considers the implementation on a grid of transputers [135].

PUL supports two variations of the geometric paradigm [31]. The Regular Domain Decomposition module deals with problems where the domain is of a regular nature, such that the work load is similar for each sub-region. The Scattered Spatial Decomposition module handles problems where the work load can vary greatly between sub-regions. A scattering process is applied, splitting each sub-region into smaller blocks, which are then allocated to a group of processes.

Chan [46] discusses the implementation of a geometric decomposition skeleton corresponding to the following SML definition:

```
fun geometric_decomposition terminate transform grid =
   if terminate grid
   then grid
   else geometric_decomposition terminate transform (transform grid);
```

The input is represented as a 2D grid of data objects, which is transformed through several iteration steps until a termination criterion is met. An implementation is described where a master process distributes objects to a number of slave processes which perform some operation before returning the results back to the master. This process is repeated over several iterations, the master checking the termination criterion at the start of each iteration. Rabhi [138] discusses a more general iterative transformation paradigm operating on a set of objects each with some local data and possibly some shared global data. At each iteration step computation involving both local and global data may be performed. Objects may also be combined at each iteration, in which case the algorithm is termed dynamic. The transformation of objects from iteration to iteration is given by a list of transformation functions, and iterations take place until a termination criterion is met.

3.4.6 Miscellaneous

Here we will briefly mention a few further proposed skeletons that do not fit directly in the previous five categories.

Variations of `scan`, defined by

```
fun scan f a [] = [a] |
    scan f a (h::t) = a::(scan f (f a h) t);
```

are frequently encountered in the literature, sometimes under the name of parallel prefix. `scan` is similar to `fold` in that its result is a combination of list elements using a binary operator, but the output of `scan` also contains intermediate results. Its BMF equivalent is named accumulate, and is described in [23]. A parallel `scan` is supported in DPHaskell [93], while Reekie considers `scan`'s in Haskell both for streams and vectors [139]. O'Donnell describes formally a parallel `scan` algorithm on a tree of processors, and proves this algorithm correct.

Darlington *et al*'s RaMP (Reduce-and-Map-over-Pairs) [64] can be defined by:

```
fun RaMP f g L =
   let fun h x = foldr1 g (map (f x) L) in
      map h L
   end;
```

RaMP computes all interactions between all pairs of objects in `L` using the binary function `f`, and then folds the results with `g` to give a combined value of interactions for each original object. [64] suggests two implementations derived by transformations to other skeletons; as a pipeline and as a processor farm.

Another skeleton proposed by Darlington *et al* is DMPA (Dynamic-Message-Passing-Architecture) [64]. It describes a set of processes, where any process can interact directly with any other via message passing. The actual connections are determined using run-time data. Values local to each process are kept in its internal state. Messages from other processes may modify the process's state and generate new messages to other processes. DMPA is closely related to Rabhi and Schwartz's skeleton for iterative transformation algorithms [136].

3.5 Discussion

In this and in the previous chapter we have outlined the principles of using skeletons for parallel processing, and we have presented a selection of relevant proposed approaches. We believe that the concept of skeletons as an abstraction mechanism is a fundamental concept in parallel computation, and various aspects of skeletons are becoming an integral part of an increasing number of parallel programming tools.

We have argued in favour of an implicit approach based on a fixed set of skeletons. This shifts the burden of making low-level decisions of parallelism from the programmer onto the compiler, but requires precise performance modeling capabilities to succeed. The strong correspondence between higher order functions and skeletons makes functional languages particularly well suited for skeleton-based programming.

While there is much good to be said about skeletons, it is at the same time clear that there are still major issues to be addressed regarding the implementation of the approach. One issue we have discussed is that of deciding which skeletons should be included in the fixed set known to the compiler. Another relates to the specification of algorithms using specific skeletons. The programmer's choice of which skeletons are used to express an algorithm may have a heavy influence on the efficiency of the parallel implementation. Thus the programmer may be forced to make early and fundamental design decisions in devising an algorithm. Although this applies to fully explicit approaches as well, it means that there are limitations to the degree of implicitness that can be claimed using skeletons. A powerful transformation system might be used to transform a program into its most efficient form. However, such a transformation may not be straightforward whether guided by the programmer or automated.

We do not address these issues further here. Our prime concern is to investigate the practical implementation of the already recognised strong-points of skeletons, which we will proceed to do in the following chapters.

Chapter 4

Overview of the SkelML Compiler

This chapter presents the framework of our approach, and gives an overview of the compiler system we propose, from now on referred to as *SkelML*. Section 4.1 outlines the main features of the approach with reference to the previous chapters. In section 4.2 we discuss the features of the embedding programming language, and give its syntax. Sections 4.3-4.5 look at the front-end, middle-section and back-end of the compiler respectively. Briefly, the front-end takes care of parsing and type checking, the middle-section takes care of profiling, identification of skeleton-parallelism, optimisation and mapping, while the back-end takes care of code production. We will here concentrate mainly on the front- and back-ends, while the middle-section will be examined in detail in chapters 5 - 7. Section 4.6 presents the set of initial skeletons supported, and outlines their implementation. Figure 4.1 shows an overview of the compilation stages with references to the respective chapters and sections.

4.1 Introducing the SkelML Compiler

The underlying philosophy of our approach is to shift as much as possible of the burden of dealing with parallelism from the programmer to the compiler. This suggests an implicit approach, where the programmer should not be permitted to order explicitly pieces of code to run in parallel. We use a functional language in which parallelism is exploited

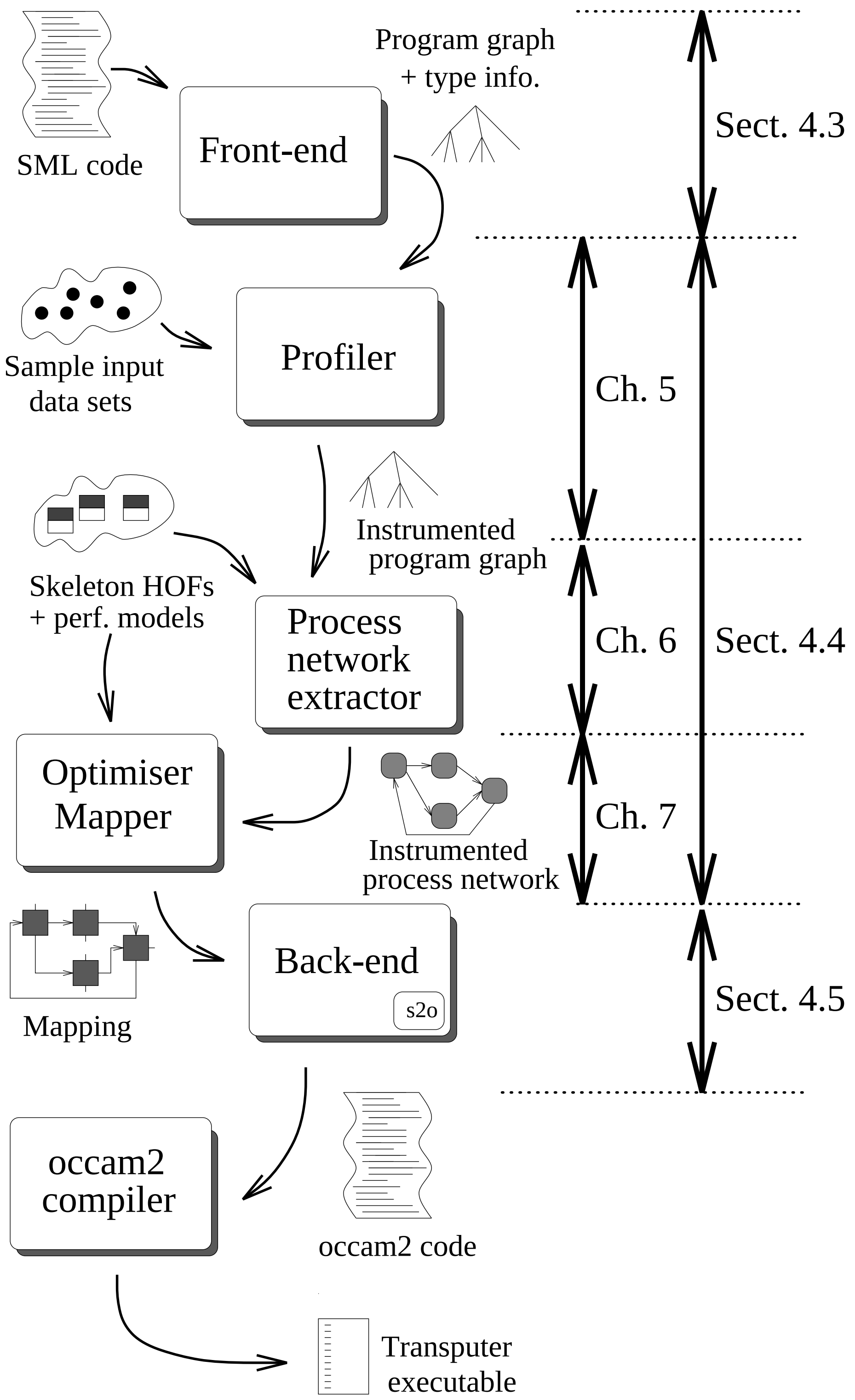


Figure 4.1: Overview of the SkelML compiler

through a fixed repertoire of skeletons. Our compiler will identify skeletons mainly by the occurrence of skeleton HOFs in the functional program, but the design will not exclude other mechanisms of extraction. A pre-processing stage that looks for and extracts skeleton-parallelism as discussed in section 3.2.2, and transforms this into known skeleton HOFs can be imagined. We will not investigate such techniques further here. We *will* however extract a form of pipeline parallelism from function composition. Ideally, the only difference between a program written for our compiler and any other functional program should be a bias towards expressing functionality through certain higher order functions, many of which are commonly used by functional programmers already.

In principle we would like the compiler to make all decisions regarding the exploitation and implementation of parallelism. This requires deep knowledge of the performance characteristics of the source program as well as of each individual skeleton. Such information will be provided through performance models and by automatic extraction of profiling data. Using this information we aim to perform automatic mapping and optimising transformations.

We illustrate our approach by the implementation of a prototype of the compiler. In the design of the compiler's front-end and middle-section there should be no strong bias towards one specific parallel architecture or another. We do however need to pick one specific platform on which to elaborate our work. We have chosen a transputer based DM MIMD architecture, specifically a Meiko Computing Surface. The SkelML compiler will produce explicitly parallel occam2 [102] code subsequently to be compiled and executed on the Computing Surface.

The proposed design should be seen as a framework for a skeleton based compiler, where the functionality of each individual component can be subject to future extensions and refinements. For example, we will include and examine a basic set of skeletons, but allow for more to be added without affecting the main structure of the compiler.

Specifying and implementing a full-scale parallelising compiler is a very large and complex task, clearly in excess of what we can hope to accomplish. Hence we will have to concentrate our efforts on what we believe to be the important concepts to illuminate. One constraint we enforce is that skeletons cannot be nested, i.e. the subcomponents of a skeleton can not themselves be skeletons. An exception is a process pipeline, which allows other skeletons as stages. Support for general nesting of skeletons is certainly worth investigating, but would have complicated mapping and performance modeling far beyond the resource limitations of this project. Another restriction is that the interprocessor communication of strings is not implemented in our prototype compiler. This should be no harder to support than the communication of user defined datatypes, which is supported. Its exclusion does however save us some time that have been invested in other parts of the compiler. Note that strings can still be manipulated internally in sequential pieces of code. Further restrictions will be mentioned later when the implicated parts of the compiler are discussed.

The process of compilation should proceed as follows, see figure 4.1. The programmer expresses the algorithm using a functional language in which skeleton HOFs are used to express functionality whenever convenient. The programmer also provides a set of test data representative of that to which the program will later be applied. The compiler parses and type checks the program (**Front-end**) and executes it with the test data as input. During this process the compiler records the program's execution profile (**Profiler**). The compiler then identifies skeleton HOFs in the program (**Process network extractor**), and using performance models and profiling information decides which parts of the program could and could not be run in parallel. At this stage it may be possible to perform certain optimising transformations (**Optimisation**) to improve the expected performance. Finally the physical resources available are allocated to the program (**Mapping**) and all occam2 code is produced accordingly (**Back-end**). The resulting code is ready to be compiled and executed on the parallel platform.

4.2 A Subset of SML

Having chosen to use a functional language, the next step is to decide on its flavour. In order to retain referential transparency and to ease the extraction of parallelism, having a pure, i.e. side-effect free, language seems preferable. We would also like a language with strict semantics. As indicated in section 2.2.2, strict languages tend to have an edge over their lazy counterparts with respect to performance. Due to the simpler model of evaluation, their run-time behaviour is also more straightforward to predict. With this background we choose to use a pure subset of Standard ML's core language [122, 175]. Standard ML is a well-known and widely used functional language with a fairly rich set of features.

However, some of these features are surplus to our requirements, and could give rise to unnecessary complexities throughout compilation. We exclude exceptions, local function definitions, abstract datatypes, type aliasing and records. Nor do we wish to support I/O, as this may introduce severe complications in conjunction with parallelism. We are still left with a fairly rich language with support for user defined datatypes, polymorphic type checking and pattern matching.

The syntax of the resulting language is given in table 4.1, while the corresponding semantics is as given in the SML definition [122]. Alternative choices are separated by | and optional phrases are enclosed in { and }. Exceptions to this are *fvalbind*, *conbind* and *match*, where | indicates the character |. All phrases are subject to type correctness. For further syntactical and semantical restrictions, precedence hierarchy and left/right associativity rules: see the definition [122].

4.3 The Front-End and Its Environment

The front-end performs three main tasks. It parses the source program, pre-processes it, and type checks it. The pre-processing takes place as described in [122], and transforms the program into a simpler and more restricted form. The result is a set of declarations

Declarations and bindings:

$prog$ ::= $dec\ ;\ \{\ prog\ \}$

dec ::= $\texttt{val}\ valbind \mid \texttt{fun}\ fvalbind \mid \texttt{datatype}\ datbind$

$valbind$ ::= $pat = exp\ \{\ \texttt{and}\ valbind\ \}$

$fvalbind$ ::= $var\ pat_{11}\ ...\ pat_{1n} = exp_1 \mid$
$var\ pat_{21}\ ...\ pat_{2n} = exp_2 \mid$
...
$var\ pat_{m1}\ ...\ pat_{mn} = exp_m\ \mathbf{(n,m \geq 1)}$
$\{\ \texttt{and}\ fvalbind\ \}$

$datbind$ ::= $tyvarseq\ tycon = conbind\ \{\ \texttt{and}\ datbind\ \}$

$conbind$ ::= $con\ \{\ \texttt{of}\ ty\ \}\ \{\ \mid conbind\ \}$

Expressions and matches:

exp ::= $scon \mid var \mid con \mid \texttt{(}exp_1\texttt{,...,}exp_n\texttt{)}\ \mathbf{(n \geq 2)} \mid$
$\texttt{[}exp_1\texttt{,...,}exp_m\texttt{]}\ \mathbf{(m \geq 0)} \mid \texttt{let val}\ valbind\ \texttt{in}\ exp\ \texttt{end} \mid \texttt{(}\ exp\ \texttt{)} \mid$
$exp_1\ exp_2 \mid exp_1\ var\ exp_2 \mid exp\ \texttt{:}\ ty \mid exp_1\ \texttt{andalso}\ exp_2 \mid$
$exp_1\ \texttt{orelse}\ exp_2 \mid \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \mid$
$\texttt{case}\ exp\ \texttt{of}\ match \mid \texttt{fn}\ match$

$match$ ::= $pat \texttt{=>} exp\ \{\ \mid match\ \}$

Patterns:

pat ::= $\_ \mid scon \mid var \mid con \mid \texttt{(}pat_1\texttt{,...,}pat_n\texttt{)}\ \mathbf{(n \geq 2)} \mid$
$\texttt{[}pat_1\texttt{,...,}pat_m\texttt{]}\ \mathbf{(m \geq 0)} \mid \texttt{(}\ pat\ \texttt{)} \mid con\ pat \mid$
$pat_1\ con\ pat_2 \mid pat\ \texttt{:}\ ty$

Type expressions:

ty ::= $tyvar \mid tyseq\ tycon \mid ty_1\ \texttt{*}\ ...\ \texttt{*}\ ty_n \mid ty_1 -> ty_2 \mid \texttt{(}\ ty\ \texttt{)}$

$tyseq$ ::= $\mid ty \mid \texttt{(}ty_1\texttt{,...,}ty_n\texttt{)}\ \mathbf{(n \geq 2)}$

$tyvarseq$ ::= $\mid tyvar \mid \texttt{(}tyvar_1\texttt{,...,}tyvar_n\texttt{)}\ \mathbf{(n \geq 2)}$

$tyvar$ ::= $\texttt{'}var \mid \texttt{''}var \mid \texttt{'}\_var \mid \texttt{''}\_var$

$tycon$ ::= an alphanumeric identifier
built-in *tycon*'s are:
`int` | `real` | `string` | `bool` | `list`

Constructors, Variables and Special constants:

con ::= an alphanumeric identifier
built-in *con*'s are:
`true` | `false` | `[]` | `::`

var ::= an alphanumeric identifier
built-in *var*'s are:
`not` | `~` | `abs` | `floor` | `real` | `sqrt` | `sin` | `cos` | `arctan` | `exp` |
`ln` | `size` | `chr` | `ord` | `explode` | `implode` | `hd` | `tl` | `/` |
`div` | `mod` | `*` | `+` | `-` | `^` | `@` | `=` | `<>` | `<` | `>` | `<=` | `>=`

$scon$ ::= an integer, real or string constant

Comments:

Comments are enclosed in `(*` and `*)`

Table 4.1: Grammar of SML subset

Expressions:			
exp	::=	$scon$ |	Special constant
		con |	Constructor
		var |	Identifier
		$(exp_1, \ldots, exp_n)(\mathbf{n} \geq \mathbf{2})$ |	Tuple
		$exp_1\ exp_2$ |	Function application
		`let val` $valbind$ `in` exp `end` |	Let-expression
		`fn` $match$	Function abstraction

Table 4.2: Simplified grammar of SML expressions

of the form[1]

```
val f = E;
```

where if `f` is a function `E` is an `fn` expression. The resulting grammar of expressions, given in table 4.2, is what will be used for referring to language constructs in the rest of the thesis.

Type checking proceeds as described in the SML definition and will not be commented on further. During type checking each single expression in the program graph is tagged with its type. This information will be needed later when dealing with interprocessor communication.

A useful pre-pass is that of higher order function removal. The process is described in [39] and consists of building specialised new functions for calls to higher order functions. This leaves the program in a form which is more amenable to various forms of analysis. We will return to this topic where it is relevant in the appropriate later chapters. Such a pre-pass has not been implemented in the SkelML prototype however, so we will assume that programs are restrictive with respect to the use of higher order functions. Specifically we will assume that all calls to skeleton HOFs are fully saturated, i.e. all arguments to a function are present at the site of invocation.

The front-end is encapsulated in a programming environment, with a similar look and feel to that of the New Jersey [8] and Edinburgh SML [43] compilers. The environment serves two purposes. On one side it is an interactive environment for developing ML programs.

[1] We use the `val` form for both `val` and `val rec` declarations.

An interpreter for ML with a mark and sweep garbage collector is supported for program execution[2]. The second purpose of the SkelML environment is to act as a command line control panel for initiating and controlling compilation. To what degree the programmer should be involved in the process of compilation is an issue we will return to in the later chapters.

The SkelML prototype is implemented in C++. The object-oriented paradigm allows the coding of the interpreter to be based directly on the structural operational semantics of the SML definition [122]. The implementation technique used is fully object-oriented, where C++ class hierarchies and methods correspond directly to the semantic object hierarchies and inference rules respectively. This technique has strong merits for compiler construction in that it provides a close and transparent mapping between semantics and implementation, see [20] for a further elaboration on this. The main benefit of taking this implementation route for our approach is that it conveniently allows profiling data to be associated with the language's semantics rather than with a particular implementation of it. As the interpreter is used for recording complexity information during the first phase of compilation, this is a significant plus with respect to portability. We will discuss this in detail when dealing with the profiling system in chapter 5.

The interpreter has few optimisations, and is thus very slow compared to for example the New Jersey SML compiler. However, our concern is that of producing efficient parallel code, while the issue of producing the code very efficiently falls outside our scope.

4.4 Middle-Section

The middle-section of the compiler is the topic of investigation in the following three chapters. Briefly,

- the **profiler**'s task is to collect complexity information from the source program when applied to test data. This information is used in conjunction with performance

[2] We mention *en passent* that this part of the compiler has been used separately as the execution engine in a persistent functional programming environment [177].

models to predict performance on the parallel architecture, and guides the compiler's decisions on resource management.

- the **process network extractor** identifies potential skeleton-parallelism, and creates an intermediate process network representation of nodes and arcs.

- the **mapper** maps the process network down to processors and communication links, preceded by process network optimisation.

4.5 Back-End: Code Generation

The task of the compiler's back-end is to generate all occam2 code for supporting parallel execution. In this section we will describe how code is generated for sequential pieces of SML, for dealing with communication, for skeleton harnesses, for processor administration, and for wiring and placement tables.

4.5.1 Sequential Code Generation

Sequential occam2 code is generated for pieces of SML code found not to contain useful parallelism and for pieces of sequential task specific code to be used for instantiating skeletons. We make use of Busvine's SML to sequential occam2 translator [38], from now on referred to as *s2o* for this purpose. s2o is implemented in SML, and is a stand-alone version of the translator originally developed as part of the PUFF compiler [39]. To produce occam2 code for an SML expression s2o must be given as input all datatype and function declarations referred to either directly or indirectly by the expression. The expression must be made into a declaration, i.e. given a name, whereafter s2o produces a set of occam2 procedures. The top level occam2 procedure corresponding to the top level SML declaration in the input is of the form

```
PROC ExpName(INT result, VAL INT arg1, arg2)
  ... occam code for expression
```

To abstract from unnecessary details in occam2 code fragments, we use `...` to indicate an irrelevant or folded (hidden) piece of code. `result` is a call-by-reference parameter to which the result value of the procedure is written, while the parameters `arg1`, `arg2` etc are call-by-value, and correspond to the arguments of the SML expression. If the SML expression is not a function there are no argument parameters.

All occam2 code produced by SkelML uses the same memory management as s2o. We will briefly outline this. On each processor there is a heap declared as a global occam2 array `[HEAPSIZE]INT Mem:`. Hence each cell on the heap is a 32-bit occam2 `INT`. This heap is used for storing intermediate results during run-time, by both s2o-produced and SkelML-produced code. The procedure `Alloc(INT memptr, VAL INT n)` is used to allocate a segment of `n` consecutive cells in `Mem` returning the starting array index in `memptr`. All data is represented by a single 32-bit occam2 `INT` either directly as a scalar value or as a pointer to the heap. If a piece of data is stored in the `INT` variable `i`, this is interpreted according to the type of the data:

- For `int`s and `real`s the value is contained directly in `i`. The occam2 `RETYPES` statement is used to refer to `real`s through `INT`s.
- For a `string` of n characters the value is stored as a consecutive series of n+1 bytes, each `INT` in `Mem` starting from `Mem[i]` storing four bytes. The end of the string is indicated by a byte of value 0.
- For a tuple of arity n, the value is stored in n consecutive cells of `Mem` starting from index `i`; one cell for each element of the tuple[3]. For example, the third element of a tuple could be accessed in `Mem[i+2]`.
- For an n-ary constructor, `Mem[i]` would contain an integer code for the constructor, while if n>0 the cells `Mem[i+1]` to `Mem[i+`n`]` contain the constructor's n arguments.

As an example, the expression `(1,[2,3],4.0)`, which is short hand for `(1,cons(2,(cons (3,nil))), 4.0)`, would be represented as shown in table 4.3. This structure would be

[3]This is a slight simplification. In fact an extra cell is used for a tuple constructor indicating the arity of the tuple. This cell has however no significance for us, and we will simply ignore it.

`Mem` index	Value	Comment
866	...	
867	1	1st component of tuple; integer value 1
868	870	2nd component of tuple; a list at `Mem` address 870
869	1082130432	3rd component of tuple; `INT` representation of 4.0
870	2	1st element of list; 2 = code for list cons
871	2	1st element (head) of list; integer value 2
872	873	`Mem` address for tail of list
873	2	Tail of list; a new list cons
874	4	Next list element; integer value 3
875	876	`Mem` address for tail of list
876	1	1 = code for list nil
877	...	

Table 4.3: Example of representation in `Mem` : `(1,[2,3],4.0)`

referred to internally by its address in `Mem`, namely 867. Note also that the list `[2,3]` could be referred to separately by its address 870.

s2o is called from within SkelML to produce sequential occam2 procedures later to be called by other code produced by SkelML. If an SML function `f1` is to be applied to some data stored in `memptr`, SkelML could produce the corresponding occam2 code by first calling s2o to produce occam2 for `f1` and then applying `f1` to the argument, i.e.

```
PROC f1(INT Out.f1,VAL INT A.1) -- this PROC produced by s2o
  ... occam2 code for the body of f1
:

INT result:
SEQ
  f1(result,memptr)
  ... rest of program, where the value in 'result' might be used
```

A few limitations of s2o should be noted. s2o deals with SML `int`s and `real`s with 32-bit precision, while other SML implementations typically use a 64-bit representation for `real`s. The SkelML interpreter can be compiled to deal with either, thus 32-bit mode can be used for confirming the correctness of the occam2 implementation, while the 64-bit mode is useful for confirming consistency with respect to other SML implementations. During compilation 32-bit mode is used.

Further, s2o has no support for garbage collection. This restricts the input data to which programs can be successfully applied. During parallel evaluation the total amount of memory available is that of all transputers added, which allows a fair amount of computation to take place. The main problem arises when producing a purely sequential version of the program, as then there is only one single heap being used during evaluation. Overall we find that the lack of garbage collection does not pose a serious problem in demonstrating our approach. Often a processor executes several independent tasks in sequence with no need to store any data between the tasks. In such cases it might be possible for SkelML to reset the heap occasionally and thus do a form of brutal garbage collection. However, no such scheme has been implemented in the prototype. Intuitively one would expect the lack of garbage collection to yield artificially good run times. However, as Busvine notes, the lack of garbage collection may in fact yield a *worse* parallel speedup [39], since in a parallel implementation the distributed heaps need collection less frequently than that of the sequential implementation against which speedup is measured.

Another point worth noting is that s2o at the topmost level only accepts first order non-polymorphic data as a function's result or arguments. For other lower level functions this restriction does not apply.

The performance of the s2o generated occam2 code obviously puts a bound on the performance of the whole system. We have made no extensive tests of s2o properties in this respect. In section 8.2 we make a brief comparison with results obtained using Concurrent Clean on similar hardware.

4.5.2 Communication

Communication takes place by sending/receiving data on an occam2 channel common to both the sending and the receiving process. SkelML generates code for dealing with four distinct modes of communication. If data is communicated between processes both located on the same processor, they operate on the same global heap, and only a single `INT` needs to be communicated. We call this type of communication mem2mem. In the other cases data is either sent from the local heap to another processor (mem2chan), received from another processor and placed into the local heap (chan2mem), or received from another processor and directly passed on to a third (chan2chan). In the latter case data is not stored locally, merely routed through to another destination.

Communication channels are declared as `CHAN OF INT`, and all interprocessor communication takes place by sending/receiving streams of `INT`s. This requires a flattening of any data structure before sending it, and a corresponding rebuilding at the other end if receiving in chan2mem mode. For any first order non-polymorphic SML type SkelML is capable of producing the corresponding occam2 piece of code for any of the four communication modes. For recursive types intermediate `Mem` pointers and state numbers are pushed and popped from internal stacks, since occam2 does not support recursion. For example to communicate data of the type

```
datatype 'T BINTREE = NIL |
                      NODE of 'T BINTREE * 'T * 'T BINTREE;
```

with `'T` instantiated to `int * int` in mode mem2chan the following code illustrates the approach, assuming the data is stored in `dataptr` and `Con.NIL` and `Con.NODE` are the occam2 constants representing the constructors `NIL` and `NODE` :

```
[STACKDEPTH]INT state, memptrs:   -- state #'s  and Mem pointers stacks
INT stackptr, mptr, mptrbase, constructor, curstate:
SEQ
  stackptr := 0                   -- initialise stack pointer
  ... push dataptr/state#1 to stacks

  WHILE stackptr > 0              -- repeat until stacks empty
    SEQ
      ... pop mptr/curstate from stacks
      CASE curstate               -- branch on current state
        1                         -- state#1 : (int*int) BINTREE
          SEQ
            constructor := Mem[mptr]
            CASE constructor      -- branch on constructor
              Con.NIL             -- constructor: NIL
                chan ! Con.NIL    -- send constructor
              Con.NODE            -- constrictor: NODE
                SEQ
                  chan ! Con.NODE -- send constructor
                  ... push mptr+2/state#2 ((int*int) tuple + right tree)
                  ... push mptr+1/state#1 (left tree)
              ELSE
                STOP               -- ERROR: Illegal constructor
```

```
2                          -- state#2 : (int*int) and right tree
  SEQ
    mptrbase := mptr       -- save mptr
    mptr := Mem[mptr]      -- mptr points at 1st int of tuple
    chan ! Mem[mptr]       -- send first int of tuple
    mptr := mptr + 1       -- update mptr
    chan ! Mem[mptr]       -- send second int of tuple
    mptr := mptrbase + 1   -- mptr points at right subtree
    ... push mptr/state#1 (right tree)
ELSE
  STOP -- Illegal state#
```

The two stacks `state` and `memptrs` store state numbers and pointers to substructures of the `BINTREE`. The variable `stackptr` points to the current cell in the stacks. Push'ing involves assigning a state number and a `Mem` pointer to the stacks followed by an increment of `stackptr`, while a pop is the inverse operation. The principle mechanism used is to first process the left subtree (state#1), then the `(int * int)` tuple followed by the right subtree (state#2). The receiver would execute similar code in mode chan2mem or chan2chan depending on whether the data were to be stored locally or passed on to another processor.

The adopted mechanism of communication is very simple, and can be optimised in a number of ways to lower communication time. Local optimisations on occam2 statements and on the use of local variables should be quite straightforward to include. For example instead of the sequence

```
SEQ
  mptrbase := mptr       -- save mptr
  mptr := Mem[mptr]      -- mptr points at 1st int of tuple
  chan ! Mem[mptr]       -- send first int of tuple
  mptr := mptr + 1       -- update mptr
  chan ! Mem[mptr]       -- send second int of tuple
  mptr := mptrbase + 1   -- mptr points at right subtree
  ... push mptr/state#1 (right tree)
```

we could have written

```
SEQ
  chan ! Mem[Mem[mptr]]   -- send first int of tuple
  chan ! Mem[Mem[mptr]+1] -- send second int of tuple
  ... push mptr+1/state#1 (right tree)
```

which might execute faster, although we have not tested this out in practice. Further it is possible in many cases to use a simple loop rather than a stack mechanism for tail recursive datatypes, and to alter the order in which a constructor's arguments are passed to make less use of push's and pop's. Also data may be clustered together and sent as a single message on a `CHAN OF ANY` rather than as a series of small one-`INT` messages. The latter optimisation requires a cheap way of packing/unpacking data at each end of the communication link, while the current scheme has the advantage of having the sender and receiver working as a two-stage pipeline on different elements of the data stream. We have experimented with some optimisations to the communication protocols, and although improved performance has been observed (see section 8.4), the overall performance gains have been relatively small. Thus, the communication mechanism of the prototype remains as outlined above.

4.5.3 Skeleton Code

For each supported skeleton implementation SkelML emits the corresponding code in the form of occam2 procedures; `PROC`s. There is one procedure for each different type of functionality required in the skeleton, e.g. for a linear chain processor farm we may have two procedures: a farmer and a worker. Each procedure is a template which is tailor made to exhibit the precise functionality desired. s2o is called to produce the complementary pieces of sequential code, and communication code is created according to the input and output types of the skeleton. A skeleton procedure may have a number of communication channels as parameters, and possibly also further information, for example an identifier to describe the processor's exact role within the skeleton. Section 4.6 contains more detailed information on the implementation of the skeletons currently supported.

4.5.4 Processor Administration

The resulting occam2 program consists of one independent top level procedure for each processor. These procedures contain definitions of further procedures to support the specific functionality required of each processor during the execution of the program. At different stages of program execution a processor may well perform different tasks. A processor may for example start out performing a sequential piece of calculation, then later act as a worker in a processor farm. The allocation of tasks to processors is handled by the mapper (see chapter 7). A processor's top level run-time behaviour is that of a server; it waits to be told which task to execute, and then calls the corresponding procedure with the proper arguments. When the task terminates, the processor waits for new tasks to be requested until the whole program terminates.

One particular task is the *terminal*, allocated to the processor in charge of I/O. The terminal is the program's topmost task responsible for collecting the program's input and for displaying the output. The terminal task sends the initial request(s) for tasks to be executed on its neighbour processes.

4.5.5 Processor Placement and Wiring Tables

The processor placement code consists of a specification of how each processor should be configured. This involves describing how communication links should be connected between processors, and also making the top level procedure call for each processor. A wiring table is a more concise description of interconnection of processors via communication links. This information is produced trivially from the result of the mapping.

4.6 Skeletons Supported by SkelML

We have implemented support for six skeletons in the SkelML prototype. They are `map`, `filter`, `fold`, `filtermap`, `mapfilter`, and `foldmap`, the latter three being various compositions of the first three. In addition a process pipeline is supported, extracted from compositions of the above six. These skeletons are chosen because of their general usefulness as building blocks both for parallel and functional programs. Although this set is sufficient both for expressing many interesting parallel algorithms, and for illustrating our approach, it would still have been desirable to support a further few skeletons. Given the limited time at our disposal this has not been possible. The compiler is however extensible with respect to the number of skeletons supported, and more could be added. In particular it could prove useful to have a divide and conquer and a geometric skeleton.

4.6.1 `map`

A linear chain implementation of `map` is supported as discussed in section 3.4.2. We will describe this implementation in some detail, as the implementations of the other skeletons are similar. The farm is controlled by a farmer processor executing an occam2 `farmer` procedure, while the worker processors execute a `worker` procedure. We describe the main functionality of the farmer and worker in terms of pseudo occam2 code, where we omit variable declarations and `SEQ/PAR` statements, and use a more lenient formatting than required in occam. Note that the actual code is slightly more complex as buffers are used in each worker to reduce latencies.

The farmer consists of two inner procedures running concurrently. The `FarmerInputMux` passes on input data to the first worker. Ordering of list elements is preserved by tagging each element with a unique integer index indicating its position in the list, i.e. 1,2,3,... etc. The `FarmerOutputMux` receives results from the worker and by inspecting the index tags ensures that the ordering is preserved as the result list is passed out from the farm.

```
PROC farmer(CHAN OF INT inchan, from.workers, outchan, to.workers,
            VAL INT no.workers)
  PROC FarmerInputMux(CHAN OF INT inchan, to.workers, request)
  -- Forwards incoming data to workers on request
    index, running := 1,TRUE  -- initialise, first index = 1
    WHILE running
      request ? token         -- request for more data?
      inchan ? constructor    -- List cons or nil
        IF constructor = Con.NIL THEN running := FALSE -- end of input
        ELSE                  -- forward list element to workers
          to.workers ! index  -- tag to preserve ordering
          index := index + 1  -- update index
          <read 1 list element from inchan to to.workers mode chan2chan>
    to.workers ! END.TOKEN    -- end of data token
    <closedown>
  :
  PROC FarmerOutputMux(CHAN OF INT outchan, from.workers, request, no.workers)
  -- Receives results from workers. Issues requests
    nextindex,running := 1,TRUE  -- next index to expect
    <send no.workers tokens down request channel> -- make workers busy
```

```
    WHILE running
      from.workers ? index       -- receive result from workers
      IF index = END.TOKEN THEN running := FALSE -- all data processed
      ELSE
      IF index <> nextindex THEN -- store result temporarily
        <read 1 list element from from.workers into temporary buffer>
      ELSE <read 1 list element from from.workers to outchan>
      IF <any elements in buffer ready to be sent out>
      THEN <send to outchan>
      <update nextindex>         -- next index to expect
      request ! REQ.TOKEN        -- issue request for more work
    <closedown, send list nil constructor to outchan>
  :
  }}}
  CHAN OF INT request:
  SEQ
    ... Initialisation
    PAR
      FarmerInputMux(inchan,to.workers,request)
      FarmerOutputMux(outchan,from.workers,request,no.workers)
:
```

The worker consists of three sub-procedures running concurrently. The `WorkerInputMux` handles incoming data either for processing locally or for forwarding to downstream workers. The `WorkerOutputMux` forwards results towards the farmer. The `Process` procedure processes list elements locally using a function `f` produced by s2o. The corresponding pseudo code is:

```
PROTOCOL INTINT IS INT;INT:              -- Send index+data as 1 message
PROC worker(CHAN OF INT from.farmer, from.workers, to.farmer,
                        to.workers, VAL INT wno, no.workers, arcno)
  ... s2o generated code for function - PROC f(INT Out,VAL INT A.1)
  PROC WorkerInputMux(CHAN OF INT from.farmer, to.workers, from.process,
                      CHAN OF INTINT to.process)
    process.idle, running := TRUE, TRUE  -- Process is initially idle
    WHILE running
      PRI ALT                            -- wait for event
        from.process ? token             -- Process wants work
          process.idle := TRUE           -- set flag to idle
        from.farmer ? index              -- fresh incoming data
          IF index = END.TOKEN THEN running := FALSE -- end of data
          ELSE                           -- process list element
            IF process.idle THEN         -- store data for local processing
              <read 1 element from from.farmer to Mem index mptr>
              to.process ! index ; mptr  -- send index and data to Process
              process.idle := FALSE      -- Process now busy
            ELSE                         -- forward data to other workers
              to.workers ! index         -- first send index, then data
              <send 1 element from from.farmer to to.workers mode chan2chan>
    <closedown, send END.TOKEN to Process and to.workers>
  :
  }}}
  PROC Process(CHAN OF INTINT from.input.mux, to.output.mux,
               CHAN OF INT request)
```

```
      running := TRUE
      WHILE running                       -- process until end of data
        from.input.mux ? index; in        -- receive data from InputMux
        IF index = END.TOKEN THEN running := FALSE -- end of data
        ELSE                              -- process data
          f(out,in)                       -- applying SML function
          to.output.mux ! index; out      -- send index+result to OutputMux
          request ! REQ.TOKEN             -- request more work
    <closedown>
  :
  PROC WorkerOutputMux(CHAN OF INT to.farmer, from.workers,
                       CHAN OF INTINT from.process)
    running := TRUE
    WHILE running
      ALT                                 -- receive from Process or workers
        from.process ? index; mptr        -- locally processed data
          IF index = END.TOKEN THEN running:=FALSE -- end of data
          ELSE                            -- forward result towards farmer
              <send index and data in Mem[mptr] to to.farmer mode mem2chan>
          from.workers ? index            -- result from other workers
            IF index = END.TOKEN THEN running := FALSE -- end of data
            ELSE                          -- forward result towards farmer
              to.farmer ! index           -- first send index tag, then data
              <send 1 list element from from.workers to to.farmer mode mem2chan>
      <closedown, send END.TOKEN to to.farmer>
      to.farmer ! END.TOKEN
  :
  CHAN OF INTINT imux2proc, proc2omux:   -- internal channels
  CHAN OF INT request:                    -- request for work channel
  SEQ                                     -- code for worker
    <initialise>
    PRI PAR                               -- mux'es on high PRI is fastest
      PAR
        WorkerInputMux(from.farmer, to.workers, request, imux2proc)
        WorkerOutputMux(to.farmer, from.workers, proc2omux)
      Process(imux2proc, proc2omux, request)
:
```

4.6.2 filter

The implementation of `filter` is very similar to that of `map`. Each `Process` procedure in the workers now executes a boolean function and returns a negated index number when the function evaluates to false. The application takes place as

```
f(out,in)                        -- apply SML function
IF out = Con.false               -- f evaluated to false
THEN to.output.mux ! -index; 0  -- (0 indicates a dummy value)
ELSE to.output.mux ! index; in  -- simply return the input element: 'in'
```

The `FarmerOutputMux` procedure is modified correspondingly to discard negative indexes. Thus only elements for which `f` evaluates to `true` are retained. It is however still important to keep track of the index values to ensure that ordering of list elements is preserved.

4.6.3 `mapfilter` and `filtermap`

These skeletons correspond to the composition of a `map` and a `filter` and vice versa, e.g. if we have

```
fun f x = ...; (* a function of type 'a -> 'b *)
fun p x = ...; (* a function of type 'a -> bool *)
fun top InputList = map f (filter p InputList);
```

it is possible to implement the two as one farm rather than as two. These compositions can be implemented as a `filter` farm with a minor adjustment in how the SML functions are applied in the `Process` procedure of the worker. Assume `f` is the function to be mapped while `p` is a boolean function. The application in the `mapfilter Process` procedure is:

```
p(boolean,in)                -- applying the boolean SML function p
IF boolean = Con.true THEN   -- p evaluated to true
  f(out,in)                  -- applying SML function f
  to.output.mux ! index; out -- send out the result of f
ELSE                         -- p evaluated to false
  to.output.mux ! -index; 0  -- 0 indicates a dummy value
```

while for `filtermap` we have

```
f(out,in)                    -- applying the SML function f
p(boolean,out)               -- applying the boolean SML function p
IF boolean = Con.true THEN   -- p evaluated to true
  to.output.mux ! index; out -- send out the result of f
ELSE                         -- p evaluated to false
  to.output.mux ! -index; 0  -- 0 indicates a dummy value
```

Chapter 5 discusses performance issues resulting from this implementation approach, while section 7.3 deals with transformations.

4.6.4 `fold`

A fold with an associative binary function argument can be implemented as a process tree as described in section 3.4.4. We support a specialised and restricted case of this skeleton in which the argument function is also commutative. This gives freedom in implementation since combinations/reductions can be performed in any order, and there is no longer a need for each list element to carry an index tag. The implementation is based on a chain of processors, each with a similar internal structure to that of a processor farm worker. Each processor in the chain first moves one list element into a variable `accum`, and then combines successive elements with `accum` to produce an updated `accum`. Assuming a binary associative and commutative argument function `f` produced by s2o, the pseudo code for the `Process` procedure is:

```
PROC Process(CHAN OF INTINT from.input.mux, from.right, to.left,
             CHAN OF INT request)
  request ! REQ.TOKEN                 -- request work from InputMux
  from.input.mux ? token; accum       -- receive data from InputMux
  IF token = END.TOKEN THEN accumEMPTY,running := TRUE, FALSE -- end of data
  ELSE                                -- now have 1 list element in accum
    request ! REQ.TOKEN               -- request more work
    accumEMPTY, running := FALSE, TRUE -- initialise flags
  WHILE running                       -- continue until end of data
    from.input.mux ? token; in        -- receive data from InputMux
    IF token = END.TOKEN THEN running := FALSE -- end of data
    ELSE                              -- process data
      f(out,accum,in)                 -- combining accum with new data
      accum := out                    -- keeping result in accum
      request ! REQ.TOKEN             -- request more work
<closedown>
:
```

For `fold` there is no `OutputMux` process. `Process` has a channel `to.left` for sending partial results back, and a channel `from.right` for receiving partial results from its downstream neighbour. When the end of the list is reached, all processors hold a value in `accum`. In the closedown phase each processor sends its `accum` back towards the first worker in the line applying the combining function `f` along the way to produce only one single piece of data to be sent out from the skeleton. The combinations performed during closedown are done in parallel as a virtual binary process tree. Figure 4.2 illustrates the process for a chain of 14 processors, where the arrows indicate an `accum` being communicated, and a "C" indicates that a combination of `accum` values using the argument function `f` takes place. In this example there are four sequential combination steps. For p processors there will be $\lceil log_2\, p \rceil$ such steps, all involving communication followed by combination. During the closedown phase each worker behaves according to its placement in the chain, i.e. the

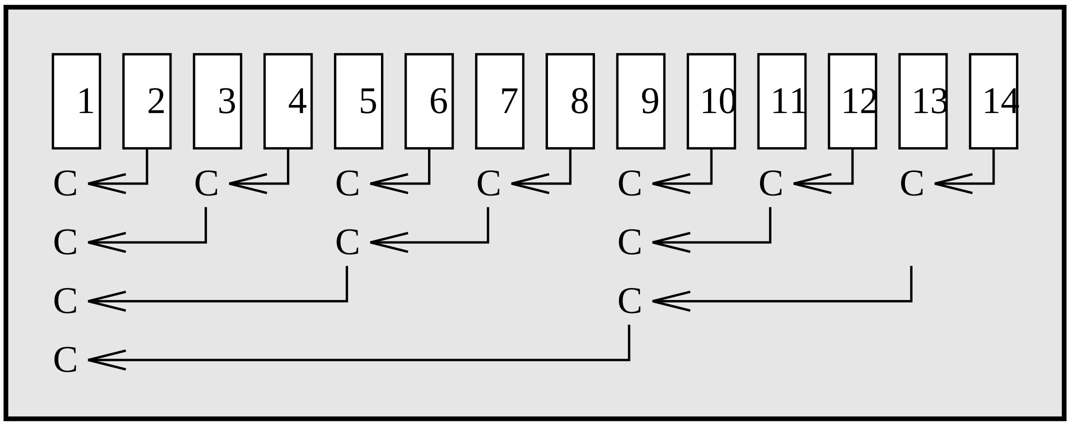


Figure 4.2: Closedown of `fold` skeleton

occam2 code is such that each worker knows when to perform combinations and when to pass an `accum` on to the next worker in line.

4.6.5 `foldmap`

`foldmap` is a composition of `fold` and `map` analogous to the combinations of `filter` and `map` discussed previously. The implementation uses a `fold` chain of processors with a slight modification to the `Process` procedure: all data arriving from the input mux is processed by the mapped function before combining with `accum`. Assume `f` is the binary argument function to `fold` and `g` is the unary argument function to `map`:

```
g(tmp,in)                  -- apply the function to be mapped, g
f(out,accum,tmp)           -- combine result with current accum
```

4.6.6 Pipelines

Pipeline parallelism is exploited when two processes, a *producer* and a *consumer* are running in parallel. Such parallelism is present in some compositions of skeleton HOFs, consider for example

```
fun f1 x = ...;
fun f2 x = ...;
fun top L = map f1 (map f2 L);
```

where the two `map`'s can be implemented as processor farms. The `map f2 L` farm processes the elements of `L`, and passes the resulting list elements one-by-one to the other farm where they are further processed. As both farms are computing simultaneously, pipeline parallelism is exploited.

4.6.7 Sequential Implementation of Skeletons

When parallelism in a skeleton is found not worthy of exploiting, sequential evaluation is employed. One straightforward approach to achieving this is to pass the original SML expression of the skeleton, e.g. `map f L`, to s2o and use the resulting occam2 code. However, sequential occam2 code produced by s2o requires all data to be present in the heap before processing commences, and none of the result can be safely accessed before processing has terminated. This has the potential drawback of blocking pipelines as described above. We therefore support our own sequential implementations of skeletons preserving the producer/consumer properties. The implementation of these is a simplification of the corresponding parallel farm harnesses discussed earlier, and will not be commented further on. For example the sequential implementation of a `map` with argument function `f` is in principle:

```
WHILE <not end of list>
  SEQ
    in := <read list cons and one list element from input channel>
    f(out,in)    -- apply SML function f
    <pass list cons and 'out' to output channel>
```

4.7 Summary

In this chapter we have explained the philosophy of the SkelML compiler and given an overview of its structure and implementation. A pure subset of SML is the source language of the compiler, while occam2 is the target language. Particular focus has been directed towards issues of code generation and integration with the s2o translator. The occam2 implementation of supported skeletons has been described.

Chapter 5

Performance Modeling

This chapter deals with the modeling and prediction of performance for skeleton-based compilation. The use of complexity information in the compilation process is a central issue in our work and we address the collection, the representation and the processing of such data. Section 5.1 introduces the principal concepts in collecting complexity information, and reviews a set of approaches suggested in the literature. In section 5.2 we present a framework for profiling for the SkelML compiler both with respect to time and the sizes of data objects. Section 5.3 describes performance models for each of the skeletons supported by SkelML and looks at the accuracy of predictions, and finally, section 5.4 summarises the chapter.

5.1 Introduction

In order to support successfully automatic parallelisation it is paramount to be able to reason about the relative costs of executing different parts of a program, both sequentially as well as in parallel. Otherwise it would be impossible to discriminate efficiently between useful and non-useful parallelism or to make good decisions regarding resource allocation. In SkelML all decisions about process placement and resource allocation are made at compile time. Thus, performance information about the program must be known at that time. Since the communication to computation ratio of a parallel program is generally critical for its performance, determining accurately the time taken for the sequential execution of a piece of code and for interprocessor transfer of arbitrary data objects are essential. The success with which one can automatically generate and efficiently exploit parallelism from a functional program largely hinges on the success with which these costs can be established.

In SkelML there are three types of performance information that are needed. A set of formulae and constants are used to model and predict the performance of skeletons. The execution times of pieces of sequential code are needed where no useful parallelism is found, and also as arguments to skeleton performance models. Finally, information about the sizes of data objects are needed in order to estimate communication times.

For skeletons we discuss the derivation of performance models in section 5.3. In determining data object sizes and sequential execution times the principal problem is: given a function

```
fun f x = ...;
```

how long does it take to evaluate the function applied to some argument, and what are the sizes of the function's argument and result? The function might be invoked several times during the execution of a program, in which case statistical properties are of interest. The minimum and maximum values are of interest for studying extreme behaviour, while the average value and the variance give information of typical behaviour. Here we will focus on average values. We consider three approaches to acquiring the needed information: static complexity analysis (section 5.1.1), programmer annotation (section 5.1.2) and profiling (section 5.1.3).

The accuracy of performance predictions is an important issue. If the predicted performance does not correspond well to the observed behaviour, the whole basis for automatic resource allocation is undermined. We will not make any efforts to quantify the required level of accuracy here, but will return to the issue later in the chapter. 100% accuracy is however hard to achieve. All of our proposed performance models can be further refined, to yield even more precise estimates. Our aim is to capture the most important factors determining the performance, and then base our models upon those.

Predicted performance figures are necessarily only meaningful for a specific target platform. To be more precise, predictions are tied to using a particular compiler for that platform (in our case an occam2 compiler), and even to using specific compilation flags for example with respect to run-time array bounds checking and optimisations. An important goal with respect to architecture independence is to separate the very specific characteristics of the target platform from the way in which performance estimates are produced. Re-targeting the parallelising compiler to another platform should thus preferably be a matter of changing only the parameters that describe the new platform rather than changing the whole process of producing performance predictions.

5.1.1 Static Program Complexity Analysis

The goal of a program complexity analysis is to say something about the time and space required for executing a given piece of code. Here we will focus on time complexity. Fundamental work in this area was done by Knuth, who applied mathematical techniques for program analysis [109, 110]. Time information is normally broken into four numbers; <*min, max, mean, variance*>, which stands for minimal execution time, maximal execution time, average execution time and variance respectively.

A separation is made between macroanalysis and microanalysis [51]. *Macroanalysis* aims to express execution time in terms of the number of times a dominant part of the function is executed in relation to a property of its argument. ACE [119] is a system for automatic macroanalysis of programs written in Backus' FP [15]. A non-recursive time-complexity function is derived from the original program using FP's powerful algebra of programs. The derived function gives an asymptotic measure of time-complexity with respect to some structural property of the input size. The result of a *microanalysis* is a measure of the execution time of a function in terms of the time taken to execute *each* elementary operation in the function. Cohen [51] discusses methods for constructing such a time formula from the original function. Assuming that the times needed for executing primitive language constructs such as arithmetic operations and function calls are known for the particular target architecture, a time formula succeeds in predicting exact execution times for unique and complete paths of execution through a function. Metric [174] is a system for mechanical microanalysis of Lisp programs. Recursion is analysed with respect to change in variables from one call to the next, and a set of difference equations is then solved to obtain the time formula.

Ideally we would like to determine the time needed to execute a function solely by static analysis. Thus, a compiler could make all of its decisions based on the source code alone, as is the case for sequential languages. There are however some fundamental problems associated with static complexity analysis, which limits its application for our purpose. First, the execution time of a program is not a decidable property, which can be seen by a direct analogy to the halting problem. Second, the techniques involved in solving difference equations are complex, and in general a solution cannot always be found symbolically [4, 51]. Third, and most important, for branching points in the program, information on the distribution of input data is generally needed in order to derive the average complexity. Note that branching points in functional programs are often disguised as alternative argument patterns in function definitions or in `fn` expressions. Consider for example the factorial function:

```
fun fac 0 = 1 |
    fac n = n * fac (n-1);
```

We are interested in the time (e.g. in microseconds) for executing `fac`. Knowing that this time is linearly dependent on the function's argument is of little use to us as long as no indication is given as to what a typical argument value might be. In most real programs there will be many branching points leading to several alternative paths of execution being present. The choice of path for a given execution of the function is completely determined by the outcome of the tests of the conditional statements, which themselves generally depend on the arguments to the function. Alas, static analysis alone is not sufficiently powerful for deriving the execution time information for general programs.

5.1.2 Programmer Annotation

Programmer annotations are information given by the programmer regarding the complexity of a program or its data. It seems unlikely that a programmer can give even reasonably accurate estimates of program execution times, as this requires detailed knowledge about execution costs of low level primitives on the target architecture. The massive amount of information that is required at several levels of the program renders this approach impractical. Further, such information would be tied to the performance characteristics of one specific architecture, thus portability would be hard. Note that any explicit scheduling of tasks for parallel execution, as for example in CML [142] or paraML [17], is an indirect annotation claiming effectively that execution time characteristics of a particular piece of code makes it suitable for parallel evaluation on a particular platform.

In the context of parallel evaluation of functional languages Hudak's para-functional programming [95] supports annotations that can be used to force evaluation of expressions on specific processors. Peyton Jones [128] notes that while programmer annotations can certainly be of help to a compiler, their use can just as well degrade as improve performance depending on the annotator's competence. A softer form of annotation regards the input data to the program. If the programmer can give good estimates of sizes of input data objects, this might be used with microanalysis to give realistic predictions. For applications dealing with complex data, this may still prove infeasible.

5.1.3 Profiling

A third approach is that of profiling or benchmarking, where run-time information is recorded during program execution for subsequent analysis. This is attractive because large amounts of information can be collected, analysed and made available to the compiler with minimal programmer effort. Profiling is traditionally used as an aid to improving software efficiency by identifying those parts of a program which consume relatively large proportions of time or space.

In the context of automatic parallelisation, the aim is to gather the same information as with the static approach, but on an empirical basis. The underlying assumption is that a program's time complexity is similar for different realistic sets of input data. If this is not the case, it is hard to make one-size-fits-all decisions regarding parallelism. This is however a general problem in devising parallel software, and not a weakness of profiling in particular. A related topic is how to choose a suitably representative set of profiling data. This may involve different considerations depending on the problem domain, but in general the profiling data should reflect the real data in all important aspects affecting performance. We will not pursue further discussion on this, but simply assume that the information derived from profiling runs is relevant and representative.

There are two principal approaches to collecting profiling information. In *sampling based* approaches an interrupt signal is issued at regular intervals, and depending on the current position of the program counter a time counter is incremented. By having many different time counters, execution time can be attributed to different parts of the program. Appel *et al* [7] describe a profiler for Standard ML based on keeping a time counter for each function in the program. A global pointer points to the counter of the function being evaluated at any time ensuring correct allocation of time costs. Sansom and Peyton Jones [144] propose a profiling technique for non-strict functional languages based on attributing both time and space information to *cost centres*. Cost centres can be set by the programmer at any expression in the program, and account for precisely those costs involved in evaluating the expression. Time information is recorded by incrementing a counter in the cost centre currently active at regular intervals, while similarly, another counter is incremented for every heap allocation.

Counter based approaches rely on frequency counters, for example at branching points, which are incremented depending on the path of execution. In IBM's PTRAN compiler [146], a parallelising compiler for FORTRAN programs, frequency information is collected about conditional branching and loop iterations. Together with the costs of primitive operations on the target architecture this information is used to calculate the average and variance of execution times. A scaling facility is supported allowing relatively small profiling runs to be used for predicting execution times for real size data. The profiling information is used for identifying useful parallelism while partitioning the program. A similar approach is taken by Busvine [39] in the PUFF compiler, where time and data size profiling information is collected at various points in the program. This is then used in the parallelisation process for estimating computation and communication times which form the basis for decisions regarding exploitation of parallelism. Runciman and Wakeling [143] describe a tool for profiling parallel graph reduction evaluation of functional programs. The motivation is that programmer annotations seem a necessary evil for efficient parallel graph reduction, but since efficient placement of annotations is very difficult, a profiling tool would be a valuable aid. Profiling information is recorded about the number of tasks running, being runnable or being blocked, and the trace is represented in tidy graphical diagrams.

5.2 A Framework for Profiling

From the discussion in the previous section we conclude that profiling seems the most convenient way of collecting complexity information. Hence, in the SkelML compiler all complexity information regarding both data sizes and time is collected from profiling. The following subsections explain in detail how this is handled. First we will make some general remarks. Profiling data is attached to the internal representation of the source program for later retrieval. The SML language construct critical for determining execution costs is branch points where alternative paths of execution may be selected. In our subset of SML (see table 4.2), there is only one place in which branching can occur, in pattern matching of the form:

```
fn p1 => e1 |
   p2 => e2 |
   ...
   pN => eN
```

where an expression on the right hand side is selected depending on which pattern is successfully matched. But there is also another important event associated with the evaluation of an `fn` expression: the binding of a value to an identifier. For this reason `fn` expressions are a natural place to collect information also about the sizes of data objects. Hence we record profiling information at every choice of patterns `p => e` depending on which patterns match successfully with values during execution. The information stored is a triple: an integer frequency counter representing the number of times that branch was selected, together with time and data size information. Every time the branch is taken during execution, the frequency counter is incremented and the time and data size information are updated.

Uncritically collecting profiling information at lexical points in the program may however result in unreliable performance predictions. Consider the declarations:

```
fun f x = ...;
fun g x = ... f ...;          (* Contains call to f *)
fun h x = ... g ...;          (* Contains call to g *)
fun top x = ... g ... h ...; (* Contains call to g and h *)
```

There are two principal issues to consider:

1. A function may be called from several sites in the program with widely different arguments. The execution times of the function may vary correspondingly. `g` can be called from both `h` and `top`, which could possibly have a great impact on its execution time. Hence profiling information about a function should be related to its call sites.

2. Even if a function is called several times from the same site, the route of execution to that place may have been different, which again may cause differences in execution times. In the example `f` can be called via the sequences `top-g-f` or `top-h-g-f`. Hence, profiling information should also be related to the sequence of enclosing function calls.

The source of the problem is that the execution information for the function is averaged over all the times it has been called, while when retrieving the profiling data we are typically interested in the average value of a function when called under some specific circumstances, e.g. from a particular call site. It is easy to construct pathological cases in which it is meaningless to talk about the average execution time of a function without also taking into account how that function was reached. Busvine addresses this problem by textually duplicating and renaming functions called from more than one call site [39]. Thus, when retrieving profiling information about a function, only information relevant to the context in which the function was called is used. We accomplish a similar distinguishing by tagging profiling information. A tag reflects the call site of the function and the preceding call sequence as outlined above. When retrieving profiling information only information associated with the relevant tag is selected. In the following we will ignore this implementation detail, and assume that only the relevant profiling information is considered.

Much of the collected profiling information will never be used, simply because some parts of the program are not very interesting with respect to parallelism. A better approach would therefore appear to be to collect profiling information only at those places where there is a potential for parallel evaluation. We are however not very concerned with the costs associated with profiling in terms of time and speed. A more important issue than time and space at compilation time is flexibility in extraction of parallelism. By first making profiling information available there is a potential for using this information when identifying parallelism. Although currently we identify parallelism only through the use of certain HOFs, we would not like to exclude the possibility that a future compiler may discover at least some skeleton parallelism through for example analysis of recursion patterns. By having profiling information available at all relevant places in the program, a compiler would be free to examine and evaluate all such possibilities in turn.

5.2.1 Time Profiling and occam2 Execution Costs

We consider two issues in time profiling: how profile times are calculated during the profiling process, and how time profiling data is associated with the program graph. The profiling times of the SML code should reflect as accurately as possible the execution times of the equivalent occam2 code as produced by s2o when compiled and running on the target architecture. This requires very detailed knowledge of the relationships between how SkelML interprets the SML code, the occam2 code produced by s2o, and the costs of executing the compiled occam2 code on the target architecture. A desirable property of the profiling system is architecture independence, such that porting SkelML to target a different platform using a different native compiler or even using a different sequential translation tool than s2o should involve no more changes than strictly necessary. On this background it is desirable to separate the process by which profiling information is collected from the information reflecting the characteristics of s2o and the target platform.

As mentioned in section 4.3, we have chosen an implementation technique where profiling costs are related to the semantics of the language rather than to the particular model of evaluation. SkelML's SML interpreter is implemented directly from the structured operational semantics of the Standard ML definition [122] as is the ML Kit [24]. While the ML Kit is implemented in SML, SkelML is implemented in C++ using the object-oriented paradigm. There is a direct correspondence between our hierarchy of C++ classes and methods and the SML definition's semantic objects and inference rules respectively. The inference rules in the SML definition all describe the evaluation of some syntactic phrase of the language, breaking the evaluation process down into small operational steps, each mirrored by a corresponding action in the interpreter. Rules are generally of the form

$$\frac{P_1 \; P_2 \; \ldots \; P_n}{C}$$

where the conclusion C holds if the premises P_1, P_2, ..., P_n hold. C may for example state that the value of an expression is v, and the premises may state certain properties about the evaluation of the expression's substructures. The evaluation of a rule takes place by evaluating the premises (which may cause a hierarchy of other rules to be invoked), after which it can be confirmed that the conclusion holds. For illustration, consider rule number 127,

$$\frac{E, v \;\vdash\; pat \;\Rightarrow\; VE \qquad E + VE \;\vdash\; exp \;\Rightarrow\; v'}{E, v \;\vdash\; pat \; \texttt{=>} \; exp \;\Rightarrow\; v'}$$

which describes the matching and evaluation of pat => exp, known as an $mrule$, against a value v. Briefly, the inference rule states that if the pattern pat matches the value v, and the expression exp evaluates to v', then the result of the evaluation of the $mrule$ is v'. E and VE represent environments in which expressions are evaluated. In the implementation of the SkelML interpreter/profiler there is a class hierarchy for patterns (with subclasses for e.g. constant patterns, tuple patterns etc), a class hierarchy for expressions (with respective subclasses), a class for $mrule$'s and so on for other semantic objects. Inference rules are represented as member functions (methods) attached to these classes. The inference rule above is represented by a method `GetValue` attached to the class for $mrule$, and is invocated whenever an $mrule$ needs to be evaluated. The evaluation of this method consists of invoking a method for evaluating the pattern pat, followed by invoking a method for evaluating the expression exp, the evaluation of which may again cause further methods to be invoked. Thus, as a program evaluates, the interpreter has full knowledge of which inference rules are invoked.

A simple step counting profiling system could be obtained by counting invoked inference rules, and scaling the result by a suitable time constant to obtain a measure of the equivalent occam2 execution time. The inference rules are however very different in nature, and a more subtle approach is needed to achieve a better correlation with actual execution times. A significant improvement is to have an individual weight associated with each inference rule. The weights are derived from empirical measures of execution time costs of s2o-produced code on the target platform. There are also weights for each basic operation such as arithmetic and boolean operators. Some inference rules do however not mirror sequential implementation concerns adequately, leading to a further differentiation having to be made. Consider rule 117:

$$\frac{E \vdash exp \Rightarrow (match, E', VE) \qquad E \vdash atexp \Rightarrow v \qquad E' + \mathrm{Rec}VE, v \vdash match \Rightarrow v'}{E \vdash exp\ atexp \Rightarrow v'}$$

dealing with the application of a function expression exp to an argument expression $atexp$ when exp evaluates to a closure $(match, E', VE)$. The $atexp$ evaluates to a value v, which is then matched against the $match$, i.e. an `fn` expression, to produce the result value v' of the application. The problem with this rule is that there is no differentiation between a top level invocation of a function and a recursive call. These two kinds of calls would often be implemented differently, e.g. tail recursive calls could be implemented by a loop. In code produced by s2o, an occam2 procedure is called for the top level invocation of a function, while an internal stack is used for recursive calls. Having just one single weight for this inference rule would thus not be accurate. Similarly for a basic operation like list concatenation, the execution time is not constant, but depends on the length of the first list, which has to be copied. In such cases some extra information is used together with the weights in cost formulae to derive the estimated execution time.

For every inference rule invoked during profiling, the interpreter consults a table to find the weight/cost of the rule. For rules where there is no single weight, like rule 117, some extra disambiguating information is used to compute the correct cost, possibly using a formula. The cost for the rule is added to an internal global clock. The clock is updated every time an inference rule is evaluated and therefore shows correct virtual time at any moment during evaluation. For attaching profile time information to the program graph, the interpreter requests the current time from the clock, i.e. the time for evaluating an expression `e`, is the current time after evaluating `e` less the current time before. As the interpreter evaluates the program graph component by component, profiling information is continuously recorded. A schematic view of the profiling approach is given in figure 5.1.

By capturing the characteristics of the occam2 code produced by s2o and of the target platform in the table of costs and formulae, we obtain a separation of architecture dependent constants and profiling mechanism. Re-targeting the profiling system should involve few other changes than updating the costs table to reflect the characteristics of the new platform. Relating profiling information directly to the semantics of the language rather than to the model of evaluation is of fundamental importance in this respect. A very appealing possibility is to represent profiling information as a vector of applied inference rules stored permanently together with the program. This would require more administration during the profiling process, as the internal clock and all internal time information would have to be replaced by a vector representation. This is however an implementation detail. Profiling information could then be fully orthogonal to the target architecture,

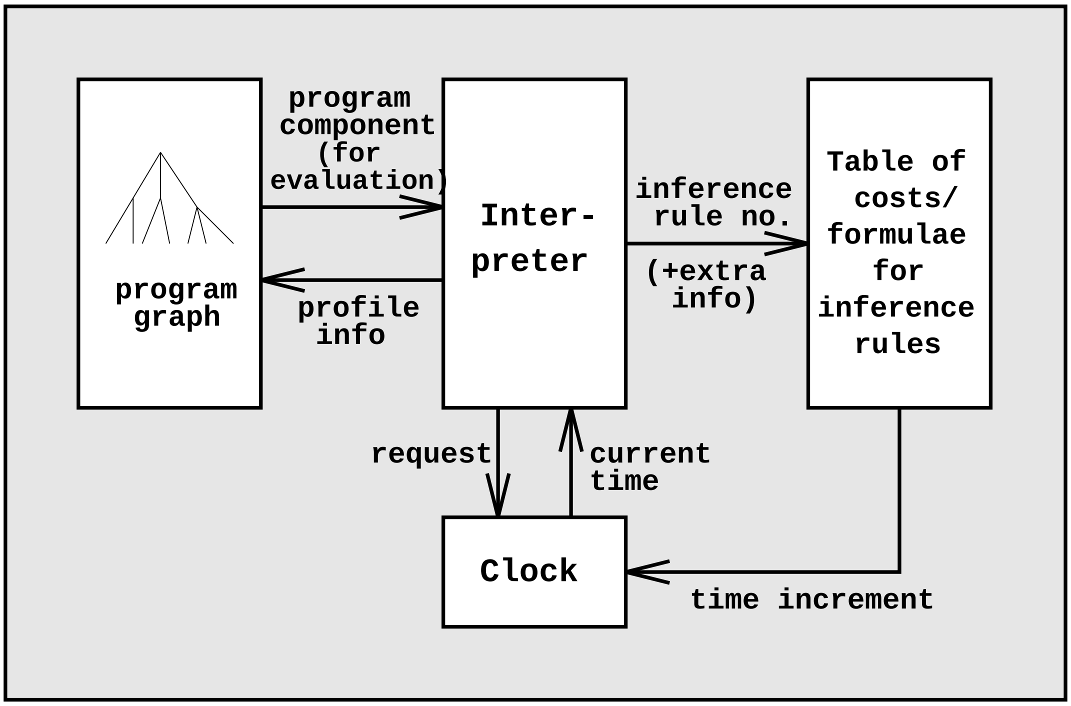


Figure 5.1: Profiling via inference rules

allowing programs to be ported without re-profiling. All the new compiler would have to do would be to examine the stored profiling vector and calculate all new profiling times using its own cost tables.

We now turn to the storage and retrieval of profiling data in the program graph. Each time profiling packet contains information necessary for retrieving the desired profiling information. The time information represents the time spent on executing the expression `e` in a pattern matching branch `p => e`. For the sake of simplicity, we will here only deal with average times, and assume that the recorded information is the accumulated total time spent in that particular branch. The average time can be obtained by dividing the total accumulated time for all executions of the branch by the number of times the branch has been taken. Additional information such as minimum time, maximum time and variance can also be recorded. Sarkar describes the computation of variance in the context of the PTRAN compiler [146], and points out that this information can be used for automatic load balancing purposes. In SkelML, profile times are calculated from the global clock which is updated as the execution of the program proceeds. The average time $\bar{t}$ of the expression

```
fn p1 => e1 |
   p2 => e2 |
   ...
   pN => eN
```

is given by

$$\bar{t} = \frac{\sum_{i=1}^{N}(f_i * \bar{t_i})}{\sum_{i=1}^{N} f_i} + Cost_{pm}$$

where f_i and $\bar{t_i}$ are the frequency count and the average execution time respectively for branch i, and $Cost_{pm}$ represents the cost of pattern matching. If the i'th branch were selected, $Cost_{pm}$ should reflect the cost of a successful match with pattern `pi` plus the cost of i-1 failed matches for the preceding i-1 patterns. No other expressions have branching points, and the execution times are calculated from the weight of the relevant inference rules.

For example the cost of evaluating

```
(4 + a, f 5)
```

would be given by the cost of evaluating a 2-tuple, the cost of evaluating two constants (`4` and `5`), the cost of looking up the value of two identifiers (`a` and `f`), the cost of an integer addition, and the cost of applying the function `f` to an argument, based on the time profiling information for `f`. Figure 5.2 illustrates the process, and indicates which inference rule costs are relevant.

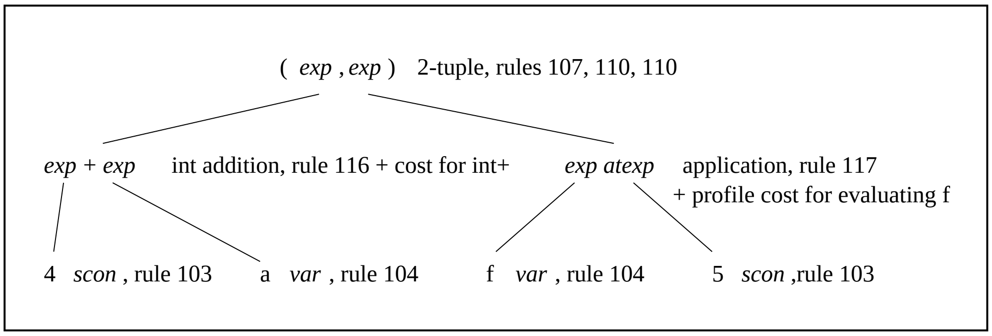


Figure 5.2: Cost of evaluating `(4 + a, f 5)`

Program	Argument(s)	Estimated	Measured	% Error
`sumn`	1000	17429	16310	6.4
`primes`	40	50527	48332	4.5
`sort`	100	52744	46883	12.5
`euler`	30 50	112345	116036	3.2
`raytrace`	det 4 sc 2	175906	174002	1.1
`bestmatch`	10 objs	70370	72320	2.8
`symarea`	20 objs	77593	86848	11.9

Table 5.1: Accuracy of sequential performance predictions

The cost of evaluating the function `f` is based on the profiling time information of the topmost `fn` expression in `f`'s definition. If `f` had been bound by an encapsulating `fn` expression, there would have been no profiling information stored. A higher order function removal process as suggested in section 4.3 would solve this problem. We are however mainly interested in profiling information not about first order expressions, but about functions. For functions there is always a top level `fn` expression holding correct profiling information about the full evaluation cost of its body, even if the body contains applications with locally bound higher order arguments.

Table 5.1 shows the accuracy of our predictions compared to the actual times measured for the occam2 code produced by s2o when applied to identical data[1], all times in microseconds. The error percentages represent how much the measured times differ from the predictions. The predictions are mostly within a margin of 10% error, in some cases even within 5%. The relatively high errors of the `sort` and `symarea` programs indicate that there are aspects of the cost tables which could benefit from from further study and tuning. One should note that the accuracy of these predictions affect not only the estimates of run times for purely sequential pieces of code, but also affect the accuracy of performance predictions of skeletons, where sequential execution times figure as parameters to the performance models. The skeleton performance models also induce some margin of error themselves, and it should be noted that the combination of several inaccurate pieces of information could magnify errors. Overall we do however find the level of accuracy obtained for sequential evaluation to be within satisfactory bounds.

[1]The SML code for these and other programs is found in appendix C

5.2.2 Data Profiling and Communication Costs

The collection of profiling information about the sizes of data objects is used for calculating estimated communication costs. For primitive and non-recursive types like `int`s and `real`s communication times can be found from type information alone. For recursive types like lists more information is needed. SkelML relies on a general framework for collecting such information, which forms the basis for all calculations of profiling times. Each data object is broken down into its primitive parts, represented in SkelML as a six-tuple of frequency counters for each component. The components recorded are the number of:

special constants (`int`s, `real`s) and `bool`s

zero-ary constructors (excluding `bool`s)

n-ary constructors (n $\geq$ 1)

accumulated arity of n-ary constructors

tuples

accumulated arity of tuples

This representation covers all first order SML types: primitives, constructors and tuples, allowing the size of any data object to be accurately recorded. The expression `(4.2,true,[2,5])` would yield the profiling tuple (4,1,2,4,1,3) for the above components respectively, meaning 4 special constants (`4.2`, `true`, `2`, and `5`), 1 zero ary constructor (list nil, `[]`), 2 constructors of accumulated arity 4 (2 list cons, `::` each of arity 2), and 1 tuple of arity 3). Similarly, `(0.6,false,[])` would yield (2,1,0,0,1,3). Accumulated information about data objects is stored by adding corresponding components of the tuples. For the two expressions above we would get (6,2,2,4,2,6), with an average of (3,1,1,2,1,3). Hence, a typical data object would in this case consist of three constants, one zero-ary constructor, one constructor of arity two, and one tuple of arity three.

One restriction mentioned in chapter 4 is that there is no support for the communication of `string`s, although `string`s can be manipulated internally in code produced by s2o. The inclusion of `string`s in the collection of profiling data can be achieved by adding two extra counters for the number of `string`s and their accumulated length.

The average size of a data object forms the basis for calculating communication times. The average expected communication time is found by combining the costs for each of the components of the data object with the costs empirically measured for communication on the target architecture. The costs are represented as a set of constants, with different costs depending on the mode of communication (see section 4.5.2). By the cost of communication we do *not* mean just the time needed to transfer a certain number of bytes between processors. The time relevant to us is the total time taken to execute the piece of occam2 code that deals with the communication. This includes any access or allocations to the internal `Mem` heap as well as all time spent manipulating internal stacks when communicating recursive types. In a similar manner to the handling of profiling time information, the architecture dependent costs of communication are captured by a set of constants kept separate from the profiling mechanism. Porting the compiler would involve revising the constants describing the target architecture characteristics. For architectures where communication times are hard to predict, a constant can be replaced by a more complex model expressed by a formula. The mechanism of storing profiling data as a basis for calculating communication times should not be affected.

Where data size information needs to be stored depends largely on how the program is partitioned, and where there is a potential for communication. For each branch `p => e` in `fn` expressions two profile tuples are stored; one representing the value bound to `p` and the other the result of evaluating `e`. Information is also stored for the result of each element in tuples and for the argument in function applications.

5.3 Skeleton Performance Modeling

This section deals with the performance modeling of skeletons. The models are expressed as formulae derived partly analytically and partly through practical experimentation. Our models deal with processor farms and processor chains corresponding to the `map/filter` and `fold` skeletons respectively, as well as process pipelines.

Quite a few publications have discussed or outlined performance models for various skeletons. The largest attention has been paid to processor farms, studied by among others Busvine [39], Danelutto *et al* [58], Hey [91], Pritchard [135] and Tregidgo and Downton [165]. In [127] Pelagatti presents a thorough discussion of models for processor farms, pipelines and data-parallelism. Feldcamp *et al* [74] give models for processor farms and divide and conquer, both implemented as trees, while Darlington *et al* [63] outline models for processor farms, divide and conquer and pipelines. The underlying principle of all of these models is to break down the skeleton's performance into its principal components with respect to all communication and computation times involved, and then combine these in formulae describing the sought performance characteristics. Our approach is the same. Depending on the degree to which all details of the skeletons' behaviour under various scenarios are modeled, models can be quite complex. Having a more pragmatic than theoretical approach to performance modeling, we devote little effort to pursuing great detail in the models as long as at least typical-case behaviour is well reflected. Our emphasis is thus on asymptotic correctness, but some attention is also paid to special cases, for example small problem instances where various overheads are accounted for.

We will make use of the following symbols:

T_C = completion time

$t_{\mathtt{f}}$ = execution time for a function `f`

N = list length

P = number of processors

L = latency

where we define the latency of a skeleton as the time from when computing commences to when the first result appears.

5.3.1 Processor Farm

We begin by modeling the performance of `map`, and then extend this model to reflect the behaviour of `filter`, `mapfilter` and `filtermap`. The performance models for `fold` and `foldmap` dealt with in section 5.3.2 are based on similar principles. The completion time can be split into three components: the startup time T_{start}, the balanced load run time T_{run}, and the closing down time T_{close}. The completion time is given by $T_C = T_{start} + T_{run} + T_{close}$. We assume initially that the ideal or optimal performance is obtained when all worker processors are busy at all times. We will use P_w to denote the number of workers in the skeleton, i.e. $P_w = P - 1$. We assume that workers are numbered in increasing order 1 to P_w with worker number 1 being the one directly connected to the farmer processor, and that `f` is `map`'s argument function, i.e. the function to be computed on each worker. During run time, a worker in the farm, say worker n is busy performing any of five tasks with time costs as follows:

Time $t_{\mathtt{f}}$:

computing the mapped function `f` on some data element.

Time t_{ep} (ep = emit-pass):

receiving a list element from worker no $n - 1$ (or the farmer if $n = 1$) and directly passing it on to the next worker in line.

Time t_{ek} (ek = emit-keep):

receiving a list element from worker no $n - 1$ (or the farmer if $n = 1$) and keeping it in local memory.

Time t_{cp} (cp = collect-pass):

receiving a result list element from worker no $n + 1$ and passing it on towards the farmer.

Time t_{ck} (ck = collect-keep):

sending the result of a local computation from local memory towards the farmer.

The value of $t_{\mathtt{f}}$ is retrieved from profiling information for `f` while the values for t_{ep}, t_{ek}, t_{cp}, and t_{ck} which reflect communication overhead are calculated from the sizes of `f`'s input and output values as recorded during profiling. We define t_p (pass time) and t_k (keep time) as:

$$t_p = t_{ep} + t_{cp}$$

$$t_k = t_{ek} + t_{ck}$$

The average number of list elements to process for each worker is N/P_w. However, the processors closest to the farmer will spend a larger proportion of their time distributing data to and collecting results from other workers than will workers further down the line. Our studies of the processor farm have confirmed that the number of list elements processed by a worker increases the farther away from the farmer the worker is located. A similar observation is noted in [74]. We assume that the differences in work load is evenly distributed, so that worker no 1 deals with $n - d$ list elements, while the last worker in line deals with $n + d$ elements, where n is N/P_w, and d is some constant. Figure 5.3 illustrates how the load of communication versus useful computation is distributed over the farm workers. Assuming that all workers are busy at all times the total computation

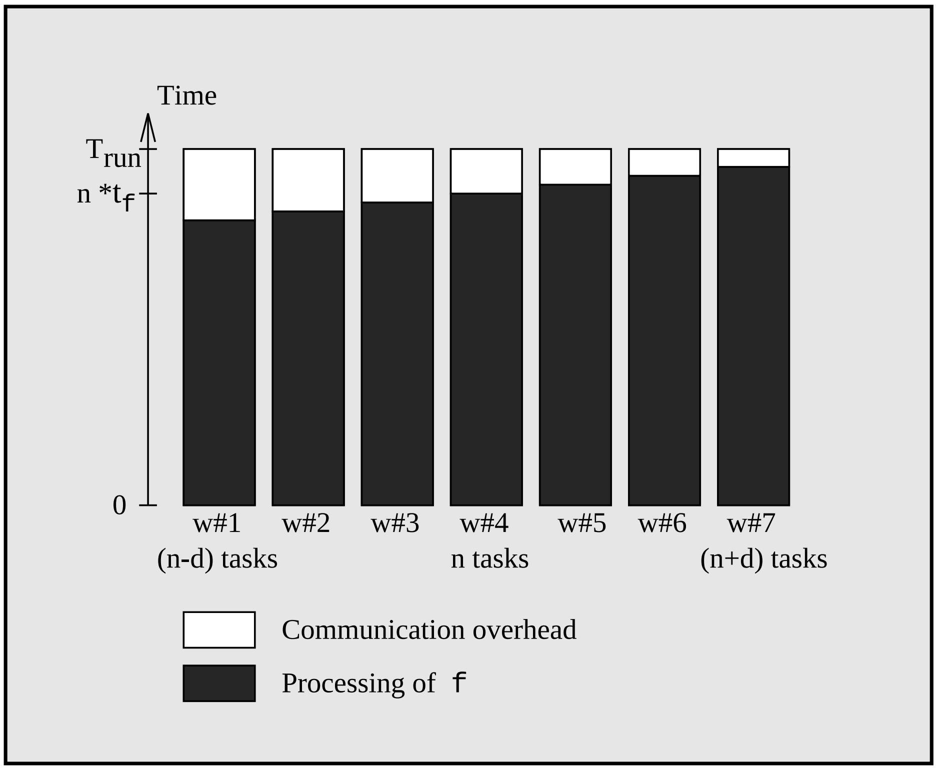


Figure 5.3: Work load on farm workers

times for the first and the last worker are:

$$T_{run1} = (n-d) * (t_{\mathtt{f}} + t_k) + (N - (n-d)) * t_p$$

$$T_{runP_w} = (n+d) * (t_{\mathtt{f}} + t_k)$$

Assuming all workers are busy at all times, then $T_{run1} = T_{runP_w}$ and d is given by:

$$d = \frac{t_p * (N-n)}{2 * t_{\mathtt{f}} + 2 * t_k - t_p}$$

Rounding d to the nearest integer we obtain T_{run1} and T_{runP_w}, and take T_{run} to be the maximum of the two, i.e.

$$T_{run} = max(T_{run1}, T_{runP_w})$$

At startup time and closing time there will be a transition period in which not all workers are busy. We compensate for this by adding some overhead constant, i.e.

$$T_{start} + T_{close} = P_w * Farm_{startclose\_add} + Farm_{startclose\_base}$$

where $Farm_{startclose\_base}$ is a fixed cost, while $Farm_{startclose\_add}$ is an additional cost incurred for every worker in the farm. Strictly speaking the overhead also depends on the size of the list elements, but we have found the current (and simpler) model to work well enough for our purpose.

The farm typically performs better the more workers it supports. There is however a point at which adding more workers does not increase performance. This is because the total time for communicating a list element from the farmer to the outermost worker becomes so large that in the meantime the element could have been processed by a worker closer to the farmer. The optimal number of workers in the farm is given by:

$$P_w = min(t_{\mathtt{f}}/(K * t_p), N/2)$$

N puts a limitation on the number of workers with respect to short input lists. As each worker has an internal buffer, a worker will process at least two list elements, thus there will be work for at most $N/2$ workers. With K set to 1.0 we obtain the maximum number of workers that can be usefully employed. This is to be viewed as an absolute cut-off point, where adding more workers will worsen rather than improve performance. We refer to this number of workers as w_{max}. However, our experiments have shown that all of these workers are not busy at all times during T_{run}, and that a K value of 1.5 should be used to represent the point when extra added workers spend some of their time being idle. We refer to this number of workers as w_{opt}, because it is in some sense an optimal balance of resources vs performance. Nevertheless the adding of workers beyond w_{opt} does result in a gain in performance. For a number of workers up to w_{opt} with a K value of 1.5, the given formulae are valid. For additional workers up until w_{max} we have experimented with a different formula based on a drop in efficiency.

Assume we want to find T_C for P_w number of workers, where $w_{opt} < P_w \leq w_{max}$. Assume further that T_{Copt} is the expected completion time when using w_{opt} workers, and that the efficiency of the farm with respect to its number of workers is given by

$$eff_{opt} = ((N * t_{\mathtt{f}})/T_{Copt})/w_{opt}$$

Each worker then processes with an average efficiency of eff_{opt}. Every added worker would be expected to work with a less degree of efficiency, down to worker $w_{max} + 1$, which would have an efficiency of 0. This assumed linear loss in efficiency forms the basis for our computation of T_C:

$$eff_{accum1} = eff_{opt} * w_{opt}$$

$$eff_{accum2} = (eff_{opt} + \frac{eff_{opt} * (w_{max} - P_w)}{w_{max} - w_{opt}}) * 0.5 * (P_w - w_{opt})$$

$$T_C = \frac{N * t_{\mathtt{f}}}{eff_{accum1} + eff_{accum2}}$$

eff_{accum1} represents the accumulated efficiency of workers 1 to w_{opt}. eff_{accum2} represents the accumulated efficiency of workers $(w_{opt} + 1)$ to P_w. Regarding the calculation of eff_{accum2}, since the assumed drop in efficiency is linear, the average efficiency of the $(P_w - w_{opt})$ workers numbered $(w_{opt} + 1)$ to P_w, is the arithmetic mean of eff_{opt} and the efficiency of worker P_w (given by fraction). The factor of 0.5 represents a division by 2 to produce this mean. $(eff_{accum1} + eff_{accum2})$ represents the expected speedup of the skeleton, from which T_C is calculated. The SkelML prototype uses a variant of these formulae.

A note should be made about the value of $t_{\mathtt{f}}$. A processor farm is relatively robust with respect to variations in the computation time for individual data elements. Hence we use the average value of $t_{\mathtt{f}}$ as the basis for our performance models. At some point however, it might prove hard to get a balanced load if the variations get too large, especially with short input lists. We have not taken any measures in our implementation to cater for this. It is fully possible, however, to extract variance information during profiling and use this data in the performance models. A variance larger than a certain upper limit would mean that a balanced load could not be achieved on a processor farm, and that for example fewer workers than normal should be used, that an alternative farm implementation should be used, or in the extreme case, that a sequential implementation might be the best solution.

By the latency of a farm we mean the time taken from the first of the input list is fed into the farm until the first result element appears. We approximate this by

$$L = T_C/N$$

The behaviour of `filter` is influenced by a constant $f_{\%}$, $0 \leq f_{\%} \leq 1$, which denotes the percentage of times the argument function evaluates to `true`. $f_{\%}$ can be found for example by taking the ratio between the length of the list returned from `filter` and the length of the argument list to `filter`. $f_{\%}$ affects the average communication times when sending data from the workers back to the farmer since when the argument function evaluates to `false` only an index token is returned to the farmer. The corresponding values of t_{cp} and t_{ck} are multiplied by a factor of $f_{\%}$ to reflect the reduced communication activity, which affects the values of t_p and t_k accordingly.

In calculating the latency of a `filter` farm we must take into account that the output list may be of a different length than that of the input list. We approximate this by

$$L = T_C/(N * f_{\%})$$

Apart from this, the performance model for `filter` is as for `map`.

The models for `filtermap` and `mapfilter` are based on that for `filter`. We use $t_{\mathtt{f}}$ to denote the total amount of work for each data element, which must reflect the fact that both argument functions are to be applied for `filtermap`, while for `mapfilter` the mapped function is applied only when the boolean function evaluates to `true`. Consider the cost of the boolean argument function to be $t_{\mathtt{p1}}$ and for the mapping argument function to be $t_{\mathtt{f1}}$. The total value to be used for $t_{\mathtt{f}}$ is then given by:

For `filtermap`: $t_{\mathtt{f}} = t_{\mathtt{f1}} + t_{\mathtt{p1}}$

For `mapfilter`: $t_{\mathtt{f}} = t_{\mathtt{p1}} + f_{\%} * t_{\mathtt{f1}}$

5.3.2 `fold` Processor Chain

For `fold` there is no distinction between farmer and worker, thus we regard all processors as workers : $P_w = P$. As for the farms, `fold`'s performance model is split into three components; T_{start}, T_{run}, and T_{close} with completion time $T_C = T_{start} + T_{run} + T_{close}$. T_{run} is calculated similarly to the equivalent phase in the farm, with the difference that for `fold` there is no sending back of results until the very final stages of computation. Thus, instead of t_p and t_k we use t_{ep} and t_{ek} respectively such that $t_p = t_{ep}$ and $t_k = t_{ek}$. The final step of computation is combining the data on all processors in the pattern of a binary tree. This is done in $\lceil log_2 P \rceil$ steps, all involving communication and computation. Assuming $S = \lceil log_2 P \rceil$, the total time for T_{close} is:

$$T_{close} = S * t_{\mathtt{f}} + (2^S - 1) * t_{cp}.$$

Note that the communication time increases by a factor of two for each combination step. In the implementation a different and less accurate formula is used, but the above would have been trivial to incorporate.

The extra cost associated with starting up the computation is calculated as

$$T_{start} = P * Fold_{start\_add} + Fold_{start\_base}$$

$Fold_{start\_base}$ is a fixed cost, while $Fold_{start\_add}$ is an additional cost incurred for every processor in the skeleton.

No result of the computation can be made available before the whole computation is completed, hence

$$L = T_C$$

5.3.3 Pipeline

We differentiate between pipelines in which all stages are skeletons, and those which have sequential components. Assume we have a pipeline of s stages, each of which is a skeleton. The total latency of the pipeline is given by

$$L = \sum_1^s (L_i + c_i)$$

where L_i is the latency of stage i, and c_i is the cost of communicating one data element from stage i to stage $i+1$, assuming $c_s = 0$. The communication costs must be added explicitly, as latency only caters for time spent internally in the skeleton. The completion time is given by

$$T_C = (max_{i=1}^s T_{Ci}) + L - L_{max}$$

where T_{Ci} is the completion time of stage i, and L_{max} is the latency of the stage for which T_{Ci} is the highest. L_{max} is subtracted to avoid it being counted twice, as it is already part of $max_{i=1}^s T_{Ci}$. There may be a different number of elements being passed between the stages, for example because some are `filter`ed away. We assume that all stages are busy producing and consuming data during the whole part of their running time such that the volume of data is of no importance.

For a pipeline of sequential stages, evaluation is purely sequential, and we have

$$T_C = \sum_{i=1}^s (T_{Ci} + c_i)$$

again assuming that c_i is the cost of communicating one data element from stage i to stage $i+1$, assuming $c_s = 0$. Further, latency is given by:

$$L = T_C$$

5.3.4 Accuracy

The accuracy of the performance models is a critical factor in our approach to compilation. Chapter 8 illustrates the quality of SkelML's predictions for many combination of skeletons and example programs. Here we will give a small example based on a processor farm and a toy program:

```
fun sumn 0 = 0 |
    sumn n = n + sumn (n-1);
fun top L = map sumn L;
```

We profile this program with an input list length of 1000 where each list element is the integer value 20. SkelML's performance model for a `map` farm produces the following performance prediction:

```
*** Performance model for map: *********
Type in: int
Type out: int
Input length: 1000
Tf: 356.959 Tk: 60.3 Tp: 36
Optimal number of workers: 7 (max 10)
Sequential time: 356959 microseconds
 No W ParTime   Speedup   Efficiency
    1  405109   0.881142  0.881142
    2  218242   1.63561   0.817806
    3  151903   2.34992   0.783305
    4  118733   3.00639   0.751598
    5   98526.5 3.62297   0.724595
    6   85563.7 4.17185   0.695308
*   7   76032.2 4.69484   0.670691
    8   69420.7 5.14196   0.642746
    9   66876.5 5.33758   0.593065
>  10   67863.5 5.25996   0.525996
*:optimal no w, >:max no w
****************************************
```

The values for $t_{\mathtt{f}}$, t_k, and t_p are calculated, together with the predicted parallel time, speedup and efficiency for up to 10 workers. "Sequential time" represents the total work

Parameter	Prediction 1	Prediction 2	Actual
d	36	38	32/38
load w1	164	162	168
load w2	182	181	180
load w3	200	200	197
load w4	218	219	217
load w5	236	238	238
T_C	98527	94385	103808

Table 5.2: Accuracy of farm model, 5 workers

load; $t_{\mathtt{f}}$ * (input length), forming the basis for speedup calculation. Efficiency is calculated with respect to the number of workers in the farm. 7 workers is indicated as best (with a K value of 1.5), while a K value of 1.0 suggests that at most 10 workers should be applied. Table 5.2 shows the correspondence between the model and the measured results when using 5 workers. We compare the accuracy of the model with respect to the value of d, the number of tasks processed by each worker, and the completion time. The "Prediction 1" column is based on a value of $t_{\mathtt{f}}$ of 357 microseconds as found after profiling. The value for $t_{\mathtt{f}}$ measured for the occam2 code is in fact 336 microseconds. The figures in column "Prediction 2" is based on this value. The model gives fair predictions for the workloads of each worker, while the measured value for T_C being 10.0% above the predicted value. Note that with the profiled value of $t_{\mathtt{f}}$, the model gives a "better" prediction with a total error of 5.4% because some of the model's error is compensated by the inaccuracy of $t_{\mathtt{f}}$. Figure 5.4 shows the accuracy of the model with 1-10 workers in the farm. As can be seen, the predicted times generally correspond well to the times actually measured.

5.4 Summary

This chapter has presented the mechanisms used for modeling and predicting performance in the SkelML compiler. After a review of techniques for establishing complexity information, we concluded that profiling offered the best prospects. We have outlined our approach to profiling both of execution time and object data sizes. Our philosophy has been to separate the collection of profiling data from the characteristics of the target

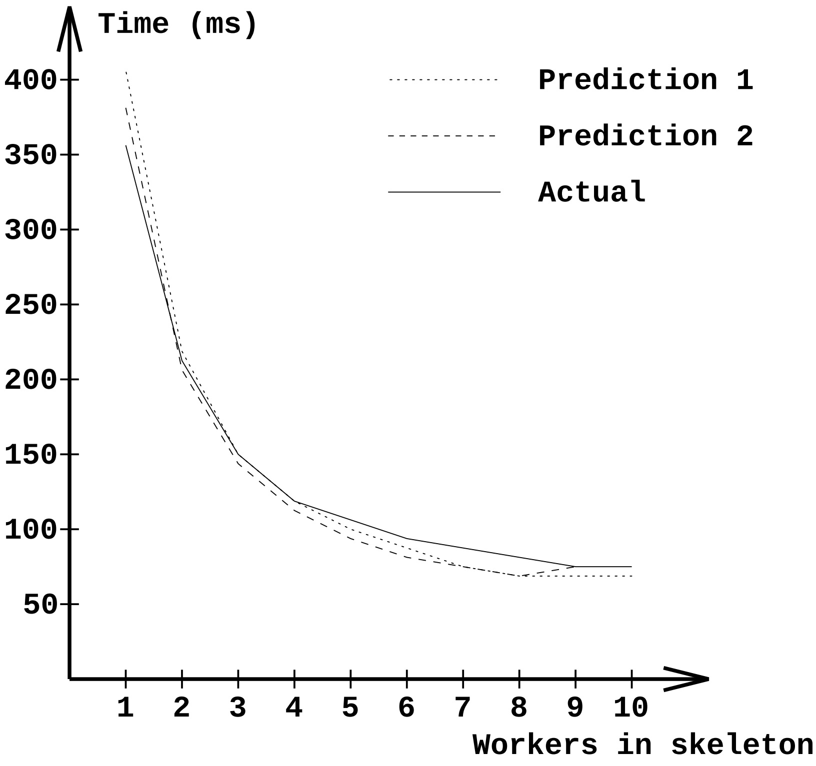


Figure 5.4: Accuracy of farm model, time vs no workers

architecture. Therefore we have based the implementation of the profiler directly on the semantics of SML, and have used separate tables of constants relating profiling data to the specific architecture. We have outlined performance models for the supported skeletons. The accuracy of the implemented modeling techniques indicates an error rate of around 10% which we consider to be within satisfactory bounds.

Chapter 6

A Skeleton Process Network

This chapter introduces an intermediate network representation used internally in the parallelising compiler. The representation takes the form of a process network in which skeleton parallelism is exposed. Section 6.1 presents the principal concerns in devising an intermediate representation and reviews some representations used in other parallelising compilers. In section 6.2 we describe the process network chosen for the SkelML compiler, and we illustrate its use. Section 6.3 describes how a process network can be derived from a source program, while section 6.4 discusses how the performance models introduced in the previous chapter can be used in the context of the process network. In section 6.5 we examine some structural properties of the network. Finally, section 6.6 concludes the chapter with a summary.

6.1 Introduction

A good intermediate program representation plays an important role in easing a compiler's tasks, and therefore has a central place in compiler development [5]. By providing suitable abstractions a good intermediate representation exposes the central characteristics of the program, and hides irrelevant details of the programming language and the intended target architecture. Thus, it can in many cases be used with various front- and back-ends to support both different languages and platforms. Our primary concern is to develop a representation which eases the compiler's handling of parallelism with respect to mapping, optimisation, performance prediction and code generation.

In [26] VCODE, the data-parallel intermediate language of NESL is described. VCODE is defined in terms of a stack machine with a heap, where a vector is the unit of storage. The instruction set is small, with most instructions operating on whole vectors at a time. No notion of processors is supported, virtual nor physical, leaving issues of low level resource allocation to the mapping stage of the compiler for the particular target architecture. VCODE is independent of both front- and back- ends, and can be used with a number of data-parallel languages as well as with a wide class of parallel machines. Although VCODE is a fairly abstract notation with support for skeleton operators, its representation in form of instructions is still a bit too concrete for our approach.

In several skeleton-based approaches, a more abstract representation is suggested. Cole [54] implies a hierarchical process structure, where one skeleton is selected as the topmost. At lower levels of such a hierarchy there may be further (nested) skeletons or pieces of sequential code. Again this representation makes no commitments regarding either the target architecture or the sequential programming language. P^3L programs are represented in a similar way, with a tree structure describing the nesting of skeleton constructs as a hierarchical composition [14]. The relationship between constructs as specified by their respective input and output parameter lists determines how the components of the tree are plugged together. During compilation the construct tree expands to a logical process structure by considering process networks reflecting the internal structure of each constructor. The compiling tools perform optimisations and transformations on this graph during mapping and load balancing. Reekie [139] considers skeleton-based programming for digital signal processing and proposes a process network representation with stream-based communication. Skeleton HOFs are represented as processes, and transformations can be applied to alter the network structure. A graphical notation is described which can be used to draw and display networks.

6.2 A Process Network Representation

6.2.1 Principles and Properties

In choosing an intermediate representation to be used in the SkelML compiler it seems natural that a program's high level skeleton structure should be clearly reflected, and that all potential skeleton parallelism in the program should be fully exposed. The fundamental conceptual units we need to represent are skeletons and sequential code, as well as the relationships between them. We have chosen a process network representation of nodes and arcs where skeletons are represented as *skeleton nodes*, while pieces of code for sequential execution are represented as *sequential nodes*. A unique *terminal node* serves as the starting- and end-point of the network. Arcs are directed, and represent the flow of data objects between nodes. Process networks are a popular means of representing parallelism in compilers. For an example, see Sarkar's "GR" [145], a graphical representation of nodes and directed arcs designed to support automatic partitioning and scheduling of parallel programs. In this section we will describe our process network representation and its properties in some detail, while section 6.3 describes a schema $\mathcal{N}$ that can be used for deriving a process network from an SML program.

The process network provides a convenient level of abstraction in that it accurately represents the functional properties of the program, while at the same time making no assumptions as to its implementation. A key property enabling this is the abstraction provided by skeleton nodes. A skeleton node indicates that there is a potential for parallelism, but makes no further commitments as to how this parallelism should be implemented, i.e. how many processors should be allocated, how processors should be interconnected, or how and when processors should communicate. This gives a compiler freedom to assign network entities to physical resources in any way which gives the best performance on a given machine. Note that the process network's entities of computation and communication correspond directly to our models for performance prediction, so that performance information is readily available during mapping. Performance information is also instrumental in the application of optimising transformations. We investigate such transformation at an inter skeleton level (see section 7.3), which matches well the entities of the process network.

Because arcs and nodes are entities closely related to communication links and processors respectively, the chosen representation is an abstraction convenient for subsequent mapping. The network itself is more general than most architectures in that no restrictions are imposed on the interconnection of nodes and arcs. Restrictions specific to the particular target architecture will typically lead to network entities being grouped together during mapping. The network representation may have a bias towards DM MIMD architectures in that its entities mirror independent processing units communicating through an external medium. It should however be possible to use this representation with other architectures. Arcs can be implemented as shared variables on shared memory machines, while data-parallelism in skeleton nodes should map quite smoothly down to SIMD architectures.

6.2.2 Arcs and Nodes

Arcs are directed carriers of data between nodes, and have a type which indicates the kind of data they can carry. This information is used by the compiler when producing communication code. Arcs are also associated with profiling information used to make communication time predictions. Only first order data objects, i.e. no functions, can be communicated between processors. Each arc corresponds to a particular binding between an identifier and a value in the program, where the value is kept by one node while the identifier is being referred to on the other. An arc can be used only for the the specific purpose of transmitting this value from one node to another. Thus, any other relations between the pair of nodes would be represented by other arcs.

Our nodes are similar to nodes in a dataflow program graph [67] in that they can be evaluated only when data is present on their input arcs, and pass out the result of the evaluation on output arcs. The evaluation of a skeleton node or a sequential node depends only on the values provided by their respective input arcs. Nodes can have an arbitrary number of input and output arcs. An exception is the terminal node which has only one input arc: the final result of the program. The output arcs of the terminal node correspond to the (input) arguments to the program.

A sequential node encapsulates an SML expression. The output arcs of a sequential node all transmit the result of evaluating the expression, and the input arcs provide values needed for its evaluation. An expression is evaluated in the environment of static function and value bindings in scope of its encapsulating declaration. The other values needed are any explicit arguments if the expression is a function, as well as any identifiers occurring free in the expression. For a function of the form `fn a => E`, the value of its argument is needed, i.e. the value to be matched with the pattern `a`, as well as any identifiers occurring free in the expression. Consider an expression `E` in which the identifier `x` occurs free. The value of `x` is then needed to evaluate `E`. SkelML handles this by adding a lambda abstraction, i.e. transforming `E` to `fn x => E`. The function expressions that SkelML passes to s2o are always such that they can be evaluated after being applied to the values of the node's input arcs.

A skeleton node represents a HOF applied to its arguments:

```
hof arg1 arg2 ... argN
```

A skeleton node can be seen as a template where each argument is represented by a slot. For a higher order argument the slot is filled with the SML expression representing the function. The expression is represented and handled in a similar way as expressions in sequential nodes. This implies that any identifier occurring free in the function body must be provided by an input arc to the skeleton node. Exceptions are explicit arguments to the function, which are dealt with internally in the implementation of the skeleton. Had nested skeleton parallelism been supported, argument slots could have been filled with node structures representing further skeleton parallelism. For a first order argument the slot may be filled by a piece of code representing a constant, or with an input arc meaning that the data is provided from another node.

6.2.3 *CallNodeExp*'s

Skeleton constructs may occur inside otherwise sequential expressions, for example

```
fn [] => [0] |
   L  => map f L;
```

In this case the subexpression `map f L` would be represented by a skeleton node. Whether the skeleton construct will be executed is not known before matching the function's pattern against an argument at run time. A mechanism is needed where skeleton nodes can be called from within a sequential expression depending on the course of execution. Thus, we introduce a new expression called $CallNodeExp$ which represents a call to another node or nodes. A $CallNodeExp$ takes the form of a new, fresh identifier applied to the values of all identifiers needed to evaluate the other nodes. The above example would be transformed to

```
fn [] => [0] |
   L  => CallNode1 L;
```

where `L` is the variable needed to evaluate `map f L` and `CallNode1` is a unique name. SkelML produces an occam procedure named `CallNode1` which sends the value of its argument, in this case `L`, to the skeleton node and then collects and returns its result. The evaluation of `CallNode1 L` thus yields the same result as `map f L`, but allows parallel execution of the skeleton. The sequential code produced by s2o would call the `CallNode1` procedure produced by SkelML just as for any other ML function.

For each $CallNodeExp$ in a sequential node there is a set of output arcs corresponding to the sending out of values, and an input arc for the return of the result.

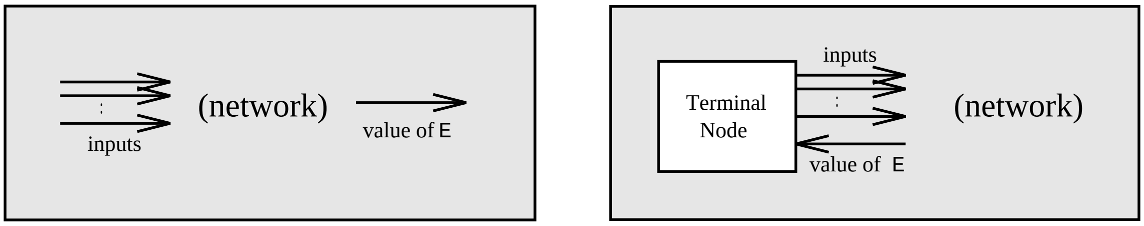
6.2.4 Process Networks Construction

For any expression there is a corresponding process network. The construction of a process network for a toplevel program expression consists of recursively assembling the process networks of all its subexpressions. Skeleton nodes are created for every skeleton HOF encountered, while sequential nodes are created when there is a need for encapsulating a piece of code. Each sequential node should encapsulate as much code as possible to keep the complexity of the network low.

For an expression `E` a general process network is depicted in figure 6.1a. The inputs are values needed to evaluate `E`, while the structure of the network depends on the structure of `E`. The topmost function definition in the program is generally of the form

```
val top = E;
```

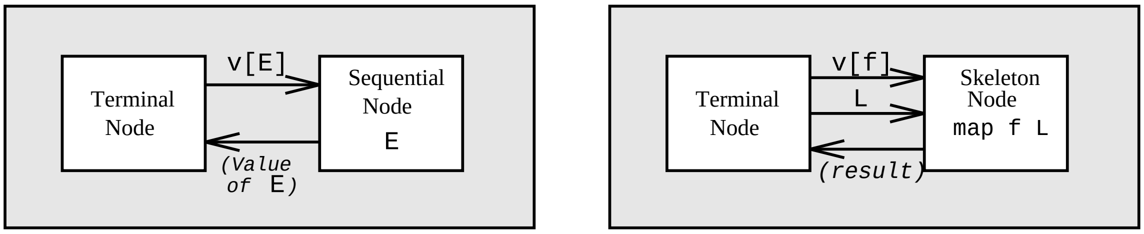
where the expression `E` is an `fn` expression. The building of a process network for the whole program consists of building a network for `E`, and then connecting the output and all input arcs to a terminal node, see figure 6.1b. The terminal node's output arcs correspond to arguments to `top`, while its input arc carries the final result of evaluating the program. Given an input program, SkelML identifies the topmost function definition, and prompts for user disambiguation if there exist several topmost definitions. Figure 6.2



6.1a: Network fragment 6.1b: Toplevel network

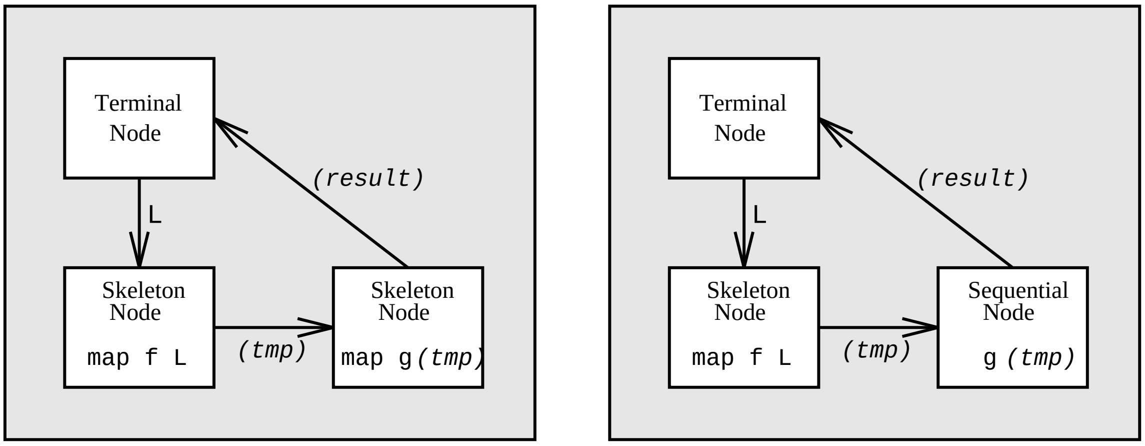
Figure 6.1: Networks for an expression `E`

shows simple process networks where `E` contains no parallelism resulting in a sequential node, and where `E` is a single `map` skeleton. In both cases the result of the evaluation is returned back to the terminal node. The notation `v[E]` is used informally to mean those values needed for evaluating the expression `E`. For simplicity in the network diagrams we will in the following assume that there are no such dependencies unless explicitly stated. Figure 6.3 illustrates networks resulting from function compositions. Note that for the network in figure 6.3a the two `map` skeletons can execute in a pipeline. In 6.3b no pipeline parallelism can be exploited because the evaluation of the sequential expression `g` can not start until the evaluation of its argument has completed. In both compositions an arc



6.2a: Sequential expression `E` 6.2b: Skeleton expression `map f L`

Figure 6.2: Networks for a sequential and a single skeleton expression



6.3a: Skeleton/skeleton composition `map g (map f L)` 6.3b: Skeleton/expression composition `g (map f L)`

Figure 6.3: Networks for expression compositions

is created (and marked *(tmp)*) for transport of the intermediate result of computation between the nodes.

The general principles for the building of process networks is that skeleton HOFs are always represented as skeleton nodes, while a composition of nodes is the network equivalent of function composition.

The evaluation of a $CallNodeExp$ in a sequential node causes data to be transmitted to other nodes for processing whereafter the result is returned. Sub networks resulting from $CallNodeExp$'s are however completely independent of other parts of the network of which the sequential node is a part. In that sense the process network structure is strictly hierarchical as illustrated by the following example. Consider the program:

```
val c = ...;      (* some constant *)
fun f1 a b = ...; (* f1 associative and commutative *)
fun f2 x = ...;
fun f3 x = ...;
fun f4 [] = [] |
    f4 L = [fold f1 c L];
fun top L 0 = [] |
    top L 1 = filter f2 L |
    top L 2 = f4 (map f3 L);
```

Here three skeleton HOFs (`fold`, `filter`, and `map`) appear in conditional branches. When creating the corresponding process network, SkelML would use $CallNodeExp$'s resulting in the following internal representation:

```
val c = ...;      (* some constant *)
fun f1 a b = ...; (* f1 associative and commutative *)
fun f2 x = ...;
fun f3 x = ...;
fun f4 [] = [] |
    f4 L = [CallNode3 L];
fun top L 0 = [] |
    top L 1 = CallNode1 L |
    top L 2 = CallNode2 L;
(*
   Semantically:
   fun CallNode1 L = filter f2 L;
   fun CallNode2 L = f4 (map f3 L);
   fun CallNode3 L = fold f1 c L;
*)
```

The `CallNode` identifiers represent the results of evaluating the $CallNodeExp$'s. The corresponding network is depicted in figure 6.4. The toplevel network consists of only

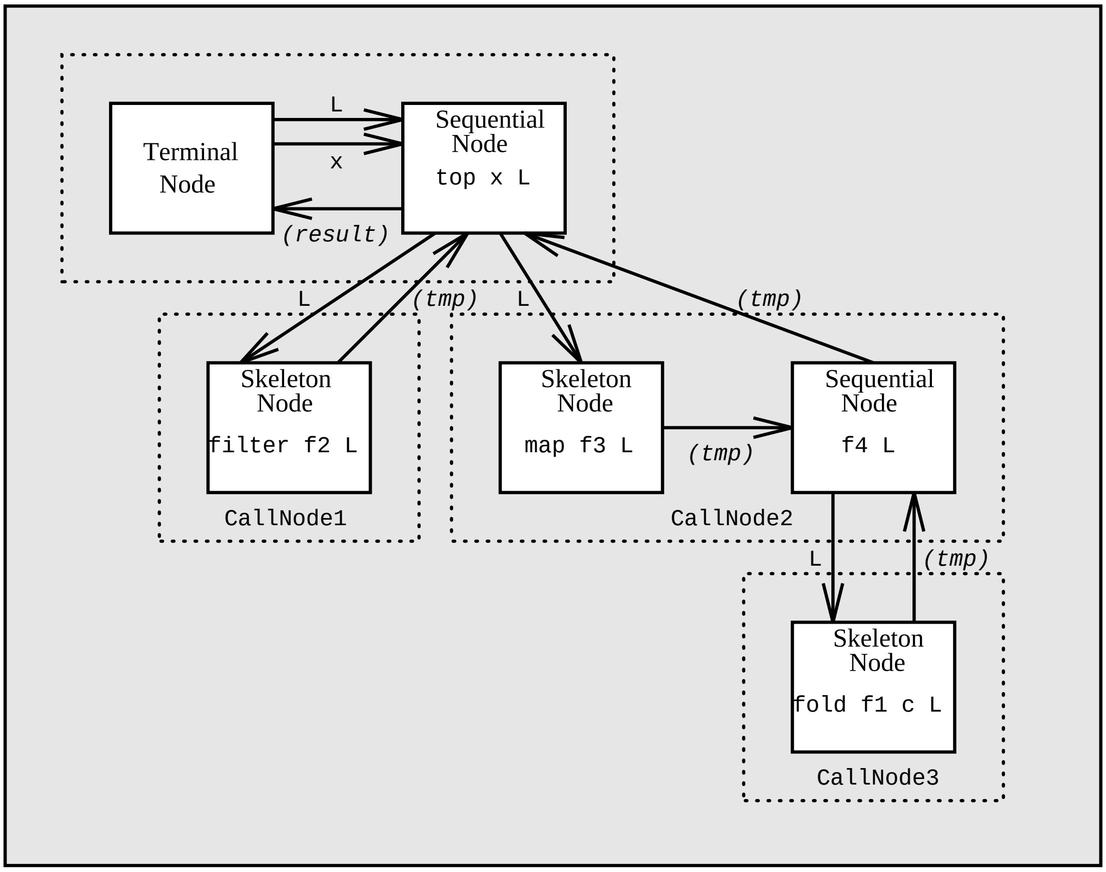


Figure 6.4: Hierarchy of sub-networks

two nodes: the terminal node, and the sequential `top` node. The two $CallNodeExp$'s in `top` result in two new sub-networks corresponding to `CallNode1` and `CallNode2`. In the `CallNode2` network there is yet another $CallNodeExp$ in `f4`, resulting in a fourth sub-network corresponding to `CallNode3`. When producing code SkelML creates occam2 procedures called `CallNode1`, `CallNode2` and `CallNode3` which make the calls to the respective sub-networks.

This representation may at first seem complex, but it is in fact very tidy and convenient for our use. Each sub-network is an individual functional unit completely captured by the $CallNodeExp$ in the calling sequential node. Generalising this further, we may view the terminal node as representing a top level $CallNodeExp$ corresponding to the toplevel function definition in the program. There is no principal difference in the way input and output arcs are handled for $CallNodeExp$'s and for the terminal node.

6.3 Deriving Process Networks

This section describes how a process network can be derived from an SML program. We have chosen a rather informal approach concentrating on highlighting the general process rather than striving for full completeness in details. Instead we will use a semi-formal notation to aid what might otherwise have been hard to express in plain text alone.

We will describe the extraction of the process network by a scheme $\mathcal{N}$. $\mathcal{N}$ takes two arguments: an SML expression and a list of identifiers. The result of applying $\mathcal{N}$ is a 4-tuple: an expression or node, a list of identifiers, a set of nodes and a set of arcs. Thus, the phrase

$$\mathcal{N}\ [\![exp]\!]\ \mathbf{V}$$

where exp is an SML expression and $\mathbf{V}$ is a list of identifiers, results in a tuple $<\mathbf{E},\mathbf{I},\mathbf{N},\mathbf{A}>$, where $\mathbf{E}$ is either an expression, tagged `EXP`, or a node, tagged `NODE`, and $\mathbf{I}$ is a list of identifiers used or needed in $\mathbf{E}$. $\mathbf{N}$ and $\mathbf{A}$ are the resulting network of nodes and arcs respectively.

$\mathcal{N}$ is applied to the top level expression, and then recursively to any subexpressions. As the recursion progresses, the list $\mathbf{V}$ is built up by adding patterns in enclosing `fn` expressions. These represent identifiers that might be needed in inner expressions[1]. As the recursion unwinds the list $\mathbf{I}$ is built up from those identifiers in $\mathbf{V}$ actually needed in the various subexpressions. When creating a node for an expression, the list $\mathbf{I}$ of the identifiers needed in the expression are turned into input arcs for the node.

We use SML's notation of lists where `[]` denotes the empty list. We further assume:

- an infix binary function $\in$: e $\in$ L holds if the element e is a member of the list L
- an infix binary function $\cup$: L1 $\cup$ L2 returns the list L1 with the list L2 appended, retaining only unique entries

[1]For clarity of description we will here assume that all patterns are single identifiers. Composed patterns, e.g. tuples, are handled in a similar, but slightly more complex way.

- an infix binary function $\setminus$: L $\setminus$ e returns the list L with the element e removed

- for brevity the same three functions will be used for sets, where the curly brackets { and } are used for textual representation.

- a list $\mathcal{H}$ of names of skeleton HOFs, $\mathcal{H}$ = `["map", "filter", "fold", "mapfilter", "filtermap", "foldmap"]`.

- a list $\mathcal{B}$ of built in functions, $\mathcal{B}$ = `["+","-","*","/","div","mod","@",...]`.

- a list $\mathcal{E}$ representing the names of user defined values at any time, and a function $\mathcal{E}_f$ that given an identifier returns the SML expression of its defining body. For simplicity we will assume that $\mathcal{E}$ always contains the bindings in scope at any given point in the program.

- $CallNodeExp$'s are introduced by the construct $CallNodeExp(exp, \mathbf{n}, \mathbf{I})$ corresponding to the application of a new fresh function name, for example `CallNode4`, to identifiers in **I**. The result of the original expression exp is delivered by the node **n**.

- Skeleton nodes are created by **new SkeletonNode**(v), where the identifier $v \in \mathcal{H}$ indicates which skeleton the new node represents. A function **IsSkeletonNode(n)** returns true if **n** is a skeleton node, and false otherwise. Argument slots in skeleton nodes are filled one by one by the function **AddArg(**<`NODE` $\mathbf{n}$,$\mathbf{I}_1$,$\mathbf{N}_1$,$\mathbf{A}_1$>, <$\mathbf{E}_2$,$\mathbf{I}_2$,$\mathbf{N}_2$,$\mathbf{A}_2$>**)**, where the first tuple contains information about the skeleton node **n**, and second tuple about its argument expression. **AddArg** updates the skeleton node by storing the argument in an argument slot, and returns a new set of updated bindings <`NODE` $\mathbf{n}'$,$\mathbf{I}'$,$\mathbf{N}'$,$\mathbf{A}'$> where $\mathbf{n}'$ is the updated skeleton node, and $\mathbf{I}'$, $\mathbf{N}'$ and $\mathbf{A}'$ are updated accordingly.

- A sequential node is created by **new SequentialNode(**<`EXP` $[\![e]\!]$,**I**,**N**,**A**>**)** resulting in <`NODE` $\mathbf{n}'$,$\mathbf{I}'$,$\mathbf{N}'$,$\mathbf{A}'$> where **n**' is the new sequential node corresponding to the expression e, with $\mathbf{I}'$, $\mathbf{N}'$ and $\mathbf{A}'$ updated accordingly.

For special constants and constructors there is no parallelism, thus:

$\mathcal{N}$ $[\![scon]\!]$ **V**
return <`EXP` $[\![scon]\!]$,[],$\emptyset$,$\emptyset$>

$\mathcal{N}$ $[\![con]\!]$ **V**
return <`EXP` $[\![con]\!]$,[],$\emptyset$,$\emptyset$>

For tuples, the $\mathcal{N}$ schema is recursively applied to each element. If a node is returned, then the *CallNodeExp* mechanism is used, otherwise the expression is returned. The identifiers needed in the whole tuple is the union of the identifiers of the elements.

$\mathcal{N}$ $[\![(e_1, e_2, ..., e_n)]\!]$ **V**
for i := 1 to n do
<$\mathbf{E}_i$,$\mathbf{I}_i$,$\mathbf{N}_i$,$\mathbf{A}_i$> := $\mathcal{N}$ $[\![e_i]\!]$ **V**
case
$\mathbf{E}_i$ = (`EXP` $[\![e]\!]$) : $e_i^{'}$:= e
$\mathbf{E}_i$ = (`NODE` **n**) : $e_i^{'}$:= $CallNodeExp(e_i, \mathbf{n}, \mathbf{I}_i)$
I := $\bigcup_{i=1}^{n}$ $\mathbf{I}_i$
N := $\bigcup_{i=1}^{n}$ $\mathbf{N}_i$
A := $\bigcup_{i=1}^{n}$ $\mathbf{A}_i$
return <`EXP` $[\![(e_1^{'}, e_2^{'}, ..., e_n^{'})]\!]$,**I**,**N**,**A**>

`let` expressions are handled by a transformation to `fn` expressions:

$\mathcal{N}$ $[\![$`let val` p_1=e_1 `and` p_2=e_2 `and` ... `and` p_n=e_n `in` E `end`$]\!]$ **V**
return $\mathcal{N}$ $[\![$(`fn` p_1=> ((`fn` p_2=> (...=> ((`fn` p_n=>E)e_n)...))e_2))$e_1$$]\!]$ **V**

Identifiers are handled according to the values they denote:

$\mathcal{N}$ $[\![v]\!]$ **V**
case
$v \in$ **V** : **return** <`EXP` $[\![v]\!]$,`[`v`]`,$\emptyset$,$\emptyset$>
$v \in \mathcal{B}$: **return** <`EXP` $[\![v]\!]$,`[]`,$\emptyset$,$\emptyset$>
$v \in \mathcal{H}$:
n := **new SkeletonNode(**v**)**
return <`NODE` **n**,[],{**n**},$\emptyset$>
$v \in \mathcal{E}$:
<**E**,**I**,**N**,**A**> := $\mathcal{N}$ $[\![\mathcal{E}_f(v)]\!]$ []
case
E = (`NODE` **n**) : **return** <**E**,**I**,**N**,**A**>
E = (`EXP` $[\![e]\!]$) :
if $e = \mathcal{E}_f(v)$ **then return** <`EXP` $[\![v]\!]$,**I**,**N**,**A**>
else
v' := <a new unused identifier>
$\mathcal{E}(v')$:= e
return <(`EXP` $[\![v']\!]$),**I**,$\emptyset$,$\emptyset$>

If v is already in $\mathbf{V}$ it is a locally bound identifier and there is no scope for further parallelism. Similarly, if v is a built-in function, again there is no parallelism to be exploited. If v is a skeleton HOF, then a new skeleton node is created. Skeleton nodes are created with empty argument slots which are filled in as the skeleton is applied to its arguments. If v is a user defined expression, $\mathcal{N}$ is recursively applied to its definition in $\mathcal{E}$. If this results in a different expression, it is labeled with a new identifier and entered into $\mathcal{E}$. This works even if v is a function containing inner parallelism through $CallNodeExp$'s. If v is recursive only parallelism present at the topmost level will be exploited.

`fn` expressions:

$\mathcal{N}$ $[\![$`fn` p_1=>e_1 | p_2=>e_2 | ... | p_n=>$e_n]\!]$ **V**
    if n = 1 then
        <E,I,N,A> := $\mathcal{N}$ $[\![e_1]\!]$ **(V** $\cup$ `[`p_1`]`**)**
        case
            E = (`EXP` $[\![e]\!]$) **: return** <`EXP` $[\![$`fn` p_1=>$e]\!]$,**I,N,A**>
            E = (`NODE` **n**) **: return <E,I,N,A>**
    else (n $\geq$ 2)
        for i := 1 to n do
            <$\mathbf{E}_i$,$\mathbf{I}_i$,$\mathbf{N}_i$,$\mathbf{A}_i$> **:=** $\mathcal{N}$ $[\![e_i]\!]$ **(V** $\cup$ `[`p_i`]`**)**
            case
                $\mathbf{E}_i$ **=** (`EXP` $[\![e]\!]$) **:** $e_i^{'} := e$
                $\mathbf{E}_i$ **=** (`NODE` **n**) **:** $e_i^{'} := CallNodeExp(e, \mathbf{n}, \mathbf{I}_i)$
        p **:=** <a new unused identifier>
        I := `[`p`]` $\cup$ $(\cup_{i=1}^{n}\ (\mathbf{I}_i \setminus p_i))$
        N := $\bigcup_{i=1}^{n}\ \mathbf{N}_i$
        A := $\bigcup_{i=1}^{n}\ \mathbf{A}_i$
        return < `EXP` $[\![$`fn` p_1=>$e_1^{'}$ | p_2=>$e_2^{'}$ | ... | p_n=>$e_n^{'}]\!]$, **I,N,A** >

The principle of dealing with `fn` expressions is quite similar to that of dealing with tuples. $\mathcal{N}$ is applied recursively to each alternative pattern/branch, p_i `=>` e_i. If a node is returned it is replaced by a $CallNodeExp$ expression, otherwise the returned expression replaces the original expression for the branch.

Function application is handled according to the result of applying $\mathcal{N}$ to both expressions:

$\mathcal{N}$ $[\![e_1\ e_2]\!]$ **V**
    <$\mathbf{E}_1$,$\mathbf{I}_1$,$\mathbf{N}_1$,$\mathbf{A}_1$> **:=** $\mathcal{N}$ $[\![e_1]\!]$ **V**
    <$\mathbf{E}_2$,$\mathbf{I}_2$,$\mathbf{N}_2$,$\mathbf{A}_2$> **:=** $\mathcal{N}$ $[\![e_2]\!]$ **V**
    if $\mathbf{E}_1$ **=** (`NODE` **n**) **and IsSkeletonNode(n)**
        then return AddArg(<`NODE` **n**,$\mathbf{I}_1$,$\mathbf{N}_1$,$\mathbf{A}_1$>,<$\mathbf{E}_2$,$\mathbf{I}_2$,$\mathbf{N}_2$,$\mathbf{A}_2$>**)**
    else

case

$\mathbf{E}_1 = (\texttt{NODE}\ \mathbf{n}_1)$ **and** $\mathbf{E}_2 = (\texttt{NODE}\ \mathbf{n}_2)$ **:**

<connect $\mathbf{n}_1$ and $\mathbf{n}_2$ with a new arc **a** from $\mathbf{n}_2$ to $\mathbf{n}_1$>

return $<\texttt{NODE}\ \mathbf{n}_1,\mathbf{I}_1 \cup \mathbf{I}_2,\mathbf{N}_1\cup \mathbf{N}_2,\mathbf{A}_1\cup \mathbf{A}_2 \cup \mathbf{a}>$

$\mathbf{E}_1 = (\texttt{NODE}\ \mathbf{n}_1)$ **and** $\mathbf{E}_2 = (\texttt{EXP}\ [\![e_2']\!])$**:**

$<\texttt{NODE}\ \mathbf{n}_2,\mathbf{I},\mathbf{N},\mathbf{A}> :=$ **new SequentialNode**$(<\texttt{EXP}\ [\![e_2']\!],\mathbf{I}_2,\mathbf{N}_2,\mathbf{A}_2>)$

<connect $\mathbf{n}_1$ and $\mathbf{n}_2$ with new arc **a** from $\mathbf{n}_2$ to $\mathbf{n}_1$>

return $<\texttt{NODE}\ \mathbf{n}_1,\mathbf{I}_1 \cup \mathbf{I},\mathbf{N}_1 \cup \mathbf{N}_2 \cup \mathbf{n}_2,\mathbf{A}_1\cup \mathbf{A}_2 \cup \mathbf{a}>$

$\mathbf{E}_1 = (\texttt{EXP}\ [\![e_1']\!])$ **and** $\mathbf{E}_2 = (\texttt{NODE}\ \mathbf{n}_2)$ **:**

$e_2' := CallNodeExp(e_2,\mathbf{n}_2,\mathbf{I}_2)$

return $<\texttt{EXP}\ [\![e_1'\ e_2']\!]),\mathbf{I}_1 \cup \mathbf{I}_2,\mathbf{N}_1\cup \mathbf{N}_2,\mathbf{A}_1\cup \mathbf{A}_2>$

$\mathbf{E}_1 = (\texttt{EXP}\ [\![e_1']\!])$ **and** $\mathbf{E}_2 = (\texttt{EXP}\ [\![e_2']\!])$ **:**

return $<\texttt{EXP}\ [\![e_1'\ e_2']\!],\mathbf{I}_1 \cup \mathbf{I}_2,\mathbf{N}_1\cup \mathbf{N}_2,\mathbf{A}_1\cup \mathbf{A}_2>$

If e_1 results in a skeleton node, its argument slot is filled with the result of applying $\mathcal{N}$ to e_2. Otherwise the choice of action depends on whether a node or an expression is returned for each of the two expressions e_1 and e_2. If both are nodes, they are connected, one providing input data to the other. If e_2 results in an expression, then it is converted to a sequential node, and the two nodes are connected as above. An expression applied to a node is handled by the $CallNodeExp$ mechanism, while the application of one expression to the other remains an expression. The handling of arcs and needed identifiers is slightly more complex. The returned list $\mathbf{I}_1 \cup \mathbf{I}_2$ should be taken to be the union of $\mathbf{I}_1$ and $\mathbf{I}_2$ as described previously, with two exceptions

1. If e_1 is a function defined by an `fn` expression an element corresponding to the pattern of that `fn` expression should be removed from $\mathbf{I}_1$.
2. If the same element occurs in both $\mathbf{I}_1$ and in $\mathbf{I}_2$ they should both be included in $\mathbf{I}_1 \cup \mathbf{I}_2$ unless they both come from `EXP` expressions.

Throughout the $\mathcal{N}$ scheme the restriction applies that no arc between nodes can carry a higher order argument. Since the inter processor communication of strings is not supported in the prototype implementation, no arc should carry a string either. A related problem is that only one skeleton node is created per occurrence of a skeleton HOF in a program. For the following expression

```
(fn hof => (hof L1, hof L2, hof L3)) (map f)
```

the algorithm would fail to produce an adequate network because the partially applied `map` HOF is passed as a higher order argument to the lambda abstraction. A suitable higher order function removal pre pass as described in section 4.3 could have been used to avoid this problem by transforming the expression into the equivalent of

```
(map f L1, map f L2, map f L3)
```

from which $\mathcal{N}$ would produce three different skeleton nodes as desired.

6.4 Implications for Performance Prediction

Since the entities in the process network are virtually the same as those discussed in the previous chapter for performance prediction, the estimation of execution times for network entities is fairly straightforward.

- For skeleton nodes, the performance is given by skeleton performance formulae.
- For sequential nodes the profiling times for sequential expressions can be used. An exception is made for sequential nodes containing $CallNodeExp$'s, which will be dealt with below.
- For arcs the communication time depends on the mode of communication (see section 4.5.2), and is calculated from profiling information of object data sizes.
- For pipelines, the pipeline performance model can be used.

The performance estimates in process networks will be further discussed in the following chapter.

The only extra problem incurred by the process network is the introduction of $CallNodeExp$'s. Assume a sequential node with topmost expression a function `f`

```
fun f x = ... E1 ...;
```

where f's body contains an expression E1. The profiled time for f which we will call $t_{\mathtt{f}}$ is based on E1 being evaluated sequentially as part of f's body. We assume the time to evaluate E1 is $t_{\mathtt{E1}}$. If now it turns out that E1 contains skeleton parallelism, it will be replaced by a $CallNodeExp$, e.g.:

```
fun f x = ... (CallNode21 x) ...;
```

The time taken for the evaluation of the $CallNodeExp$ is given by the relevant performance models, for example for skeleton nodes, and we will denote this time by $t_{\mathtt{CNE}}$. It may very well be that $t_{\mathtt{CNE}} \neq t_{\mathtt{E1}}$, in fact we would normally expect $t_{\mathtt{CNE}}$ to be less than $t_{\mathtt{E1}}$ if E1 contains useful parallelism. Thus, the "old" value for $t_{\mathtt{f}}$ is no longer valid. However, this is relatively straightforward to adjust.

The method used for finding the expected execution time for a sequential node is to subtract a value t_{diff} from expression's total sequential profiled time. t_{diff} represents the expected gain in time by evaluating E1 in parallel rather than sequentially. If E1 is always executed whenever f is, t_{diff} is given as the difference between $t_{\mathtt{E1}}$ and $t_{\mathtt{CNE}}$. Thus, in the above case we could have found an adjusted time $t_{\mathtt{f}}^{'}$ as:

$$t_{diff} = t_{\mathtt{E1}} - t_{\mathtt{CNE}}$$

$$t_{\mathtt{f}}^{'} = t_{\mathtt{f}} - t_{diff}$$

E1 may however have been nested inside E with several layers of encapsulating fn expressions, for example:

```
fun f x 0 = ... |
    f x 1 = ... E1 ... |
    f x 2 = ...;
```

Here the time to be subtracted from $t_{\mathtt{f}}$ should reflect the probability of the (... E1 ...) branch being chosen.

In general, for each encapsulating fn expression starting with the innermost, t_{diff} is multiplied by a factor corresponding to the probability of its branch being selected. Branching frequencies are available from profiling information. If there are several $CallNodeExp$'s inside `f`, the t_{diff} to be subtracted at the topmost level should thus represent the total expected gain from evaluating the skeleton expressions in parallel. This value can be found by following the process outlined above for all $CallNodeExp$'s inside `E`, combining the total impact of $CallNodeExp$'s at each enclosed `fn` expression.

6.5 The Structure of Process Networks

So far we have described networks of nodes and arcs, and seen how they can be derived. The examples we have given have all been for relatively simple program structures. Process networks may however, at least in pathological cases, become quite complex, and we would like to enforce structure to ease the process of mapping. We will focus on two structures: linear pipelines and identification of nodes that can evaluate in parallel. Since sub-networks as introduced by $CallNodeExp$'s are fully local to the particular sequential node they can be regarded as individual networks in their own right, subject to the same structure enforcement as the top level network. In the following we will ignore arcs related to $CallNodeExp$'s.

We will refer to nodes in a pipeline as stages, and define a linear pipeline to be a sequence of interconnected nodes where there is an arc from one node to the next. All stages except the first have exactly one input arc. All stages except the last have exactly one output arc. A special kind of pipeline is that which consists of skeleton nodes only, and where each stage but the last produces data while executing, and where each stage but the first consumes data while executing. We will refer to such pipelines as skeleton pipelines. The only form of pipeline parallelism we exploit is in skeleton pipelines. Other pipelines are in nature sequential, i.e. one stage must complete execution before the next can commence. If such a pipeline is fed with a stream of incoming data, for example if it occurs in the body of some list processing function, parallelism can indeed be exploited. We do not however investigate this type of parallelism.

In identifying pipelines, we are interested primarily in skeleton pipelines. Thus we first identify skeleton pipelines, and then other pipelines of the remaining nodes. For the special case of single nodes not being part of a pipeline we introduce a one-stage pipeline. At the end of this process, all nodes should belong in exactly one pipeline. We represent pipelines as a set of lists complementary to the process network itself. In such lists pipelines can be represented by the following constructors:

`SKELPIPE` [node$_1$,node$_2$,...,node$_n$] ($n \geq 2$)
`PIPE` [node$_1$,node$_2$,...,node$_n$] ($n \geq 1$)

A straightforward representation of nodes in this context is to assign to each a unique identification number, and use integers in the lists above. Thus, we can view the process network as a network of pipelines rather than as a network of nodes. The use of structuring constructors is best illustrated using a binary skeleton function, and we borrow the `map2` skeleton as described in section 3.4.2.

```
(*
   Assuming map2 skeleton defined by:
   fun map2 f [] [] = [] |
       map2 f (h1::t1) (h2::t2) = (f h1 h2)::(map2 f t1 t2);
*)
fun f1 x = ...;
fun f2 x = ...;
fun f3 x y = ...;
fun f4 x = ...;
fun f5 x = ...;
fun top L = f5 (map f4 (map2 f3 (f1 L) (f2 L)));
```

The corresponding process network is depicted in figure 6.5, where each node is marked with a unique integer identifier. Intermediate results are labeled *(t1)-(t4)*. The pipelines would be represented as:

`PIPE` [1], `PIPE` [2], `PIPE` [3], `SKELPIPE` [4,5], `PIPE` [6]

The second structuring mechanism regards identification of network components that can run in parallel.

`SEQ` [p$_1$,p$_2$,...,p$_n$]

denotes that the components p$_1$,...,p$_n$ cannot execute in parallel, i.e. there is no way in which p$_j$ can commence execution before p$_i$ has completed if $i < j$.

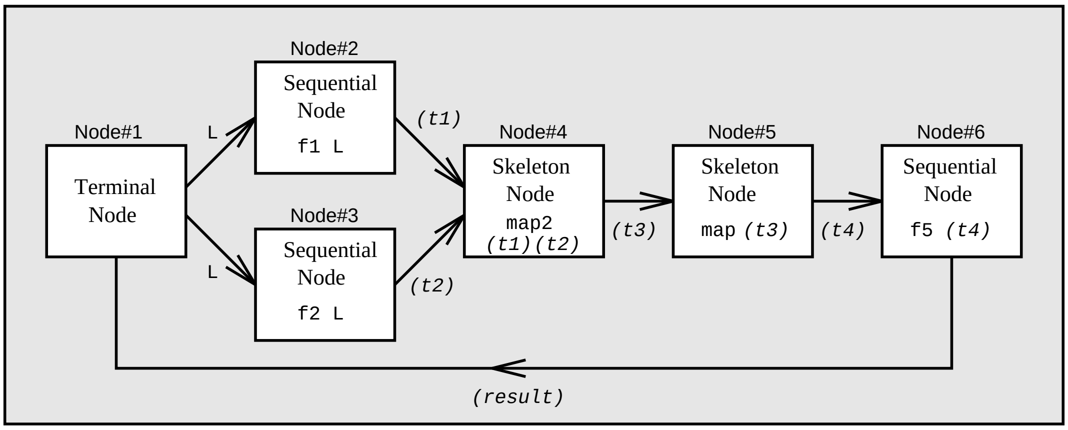


Figure 6.5: Network with node identifiers

`PAR` [p_1,p_2,...,p_n]

denotes that the components p_1,...,p_n can execute in parallel, i.e. there are no dependencies requiring synchronisation between the components.

Using these constructors the network depicted in figure 6.5 can be complemented with the following notation:

`SEQ` [`PIPE` [1], `PAR` [`PIPE` [2], `PIPE` [3]], `SKELPIPE` [4,5], `PIPE` [6], `PIPE` [1]]

Note that the terminal node figures at both the beginning and end of the outermost `SEQ` constructor representing the start and end points of the network. Similarly, if a sequential node N contains a $CallNodeExp$ resulting in a sub-network net, the net network would in principle be described as `SEQ` [N,net,N], indicating net's start and end points.

Incorporating the notation of pipelines just outlined, structural properties can be described by the constructs summarised in table 6.1. This results in the following hierarchical and nested structure of process networks:

- The top level structure of a process network is given by a nesting of `PAR` and `SEQ` constructors.

Network structure:			
net	::=	SKELPIPE [node_1,node_2,...,node_n] ($n \geq 2$) |	Skeleton pipeline
		PIPE [node_1,node_2,...,node_n] ($n \geq 1$) |	Other pipeline
		SEQ [net_1,net_2,...,net_n] ($n \geq 1$) |	Sequential components
		PAR [net_1,net_2,...,net_n] ($n \geq 2$)	Parallel components

Table 6.1: Network structuring constructors

- The unit of PAR and SEQ constructors is a pipeline. We distinguish two types of pipelines: skeleton pipelines and others.
- A pipeline consists of one or more interconnected nodes.
- There are two types of nodes: skeleton nodes which contain no further substructures, and sequential nodes which contain an SML expression.
- SML expressions may contain $CallNodeExp$'s which involve calls to independent process networks.

The process network is in many respects similar to dataflow graphs [67]. Nodes represent functional entities that execute or are *triggered* when data is present on incoming arcs, i.e. a data-driven model of execution. The result of a node's evaluation is sent to other nodes for further processing. The network demands no specific order of evaluation; any node whose arguments are present is free to execute. This similarity is not at all surprising since functional languages and dataflow languages [2] are closely related. However, while nodes in dataflow graphs may represent instruction-level functionality our nodes encapsulate rather larger functional units. Our pipelines are also different in that nodes can work on distinct parts of the same data structure rather than on distinct structures. Further, our sequential nodes allow the triggering of other nodes through $CallNodeExp$'s, which is not commonly seen in the dataflow paradigm.

6.6 Summary

This chapter has been devoted to an intermediate program representation to be used in the compiler. We have proposed a process network representation capturing the concepts of skeletons and sequential code as well as their functional relationship. The process network provides an abstract and general program representation with few assumptions about the target architecture. The abstraction provided by skeleton nodes, allows implementation issues to be postponed to a later stage of the compilation process. We have argued that the representation is convenient for mapping down to DM MIMD architectures, and that it is also good with respect to transformation and performance modeling. We will deal further with these issues in the following chapter.

We introduced a mechanism where sub-networks could be called from a sequential node, through $CallNodeExp$'s. The process of building process networks has been explained and described through a scheme $\mathcal{N}$. The main principle of building networks is that skeleton HOFs are represented by skeleton nodes, while function application involving nodes can be represented as a composition of nodes.

We also outlined how the introduction of $CallNodeExp$'s affects performance prediction, and we introduced two structuring mechanisms: pipelines and `PAR/SEQ` constructors to provide information complementary to the process network.

Chapter 7

Mapping

This chapter deals with the mapping of the process network described in the previous chapter down to physical processors and communication links. In section 7.1 we point out some principal concepts and problems associated with mapping. In section 7.2 we outline the implications to skeleton-based systems, and look at mapping in the SkelML compiler. Section 7.3 discusses skeleton transformations as an aid to improving performance, while section 7.4 concludes the chapter with a summary.

7.1 Introduction

By mapping we mean the process of assigning and scheduling an abstract representation of processes onto physical resources, determining where and when each individual process executes. The mapping problem is to find as efficient as possible an assignment with respect to some measure, e.g. completion time or resource utilisation. This problem has been studied extensively over the past two decades. In [45] Casavant and Kuhl give a taxonomy for classifying scheduling according to a hierarchy of characteristics such as static/dynamic, optimal/sub-optimal and approximate/heuristic. Another overview is given by Norman and Thanisch [125] where a number of mapping approaches are classified and considered with respect to a DM MIMD architecture. In a general representation of the mapping problem, for example the assignment of a weighted graph of modules to processors, the task of finding an optimal solution is known to be NP-complete [76]. This holds true even for several apparently simple special cases, for example two processor scheduling with jobs of one or two time units [168]. For automatic mapping in parallelising compilers it is generally desirable that the algorithm used is of a not too high a time complexity. Otherwise only small problem instances can be handled within reasonable time bounds.

One approach is to be content with an algorithm that finds a sub-optimal, but still acceptable, solution, and which is computationally tractable. Such solutions can be found by various approximation strategies or by heuristics exploiting *a priori* knowledge about the problem [45]. Bokhari [28] describes a heuristic to permute the process graph in order to map communicating nodes onto directly connected processors. Lo [114] notes that high communication costs between tasks suggest that they should execute on the same processor. To enhance exploitation of parallelism she proposes interference costs to indicate that tasks should be scheduled on different processors, and reports good results with this approach in simulations. In [107], Kim and Yi suggests a two pass algorithm with a low time complexity and good experimental results. First, tasks are mapped to processors provisionally in a forward pass by considering independently each node together with all its predecessors. Since the predecessor groups are not mutually disjoint, mapping conflicts may occur, which are resolved in a second backwards pass making permanent assignment to processors.

The other main line of approach is to impose restrictions on the topologies of either the program or the physical resources so as to make the mapping computationally tractable. Ma *et al* [115] propose a task allocation model minimising interprocessor communication cost while balancing the utilisation of each processor. A branch-and-bound technique is used to limit the solution search space, but as the problem is still in NP, a number of constraints are introduced to reduce the time complexity. For example tasks may have preference for or be excluded from executing on specific processors. Shen and Tsai [151] propose an approach for optimal task assignment based on a similar minimax criterion: minimising communication while balancing the processor load. A constraint is that there are little or no precedence relationships between the tasks. Various system constraints can also be included. Pelagatti [127] notes that for almost all mapping models proposed in the literature, the problem of finding an optimal mapping is NP-complete. However, for process graphs with a regular structure, for example chains and trees, the complexities are dramatically better. Process networks implementing skeletons are generally highly regular, and by allowing parallelism to be expressed as skeletons only it is argued that the mapping problem can be solved satisfactorily. By having skeleton nodes as a central concept in the process network, we hope to exploit this together with allocation heuristics to reduce the complexity of our mapping algorithm.

7.2 Mapping in the SkelML Compiler

Our mapping algorithm aims to assign nodes in a process network to physical resources according to a cost criterion. There are two conflicting requirements to take into consideration: obtaining a good completion time, and using a minimal amount of physical resources. Our main focus is on efficient exploitation of useful parallelism, and we will thus use completion time as the cost criterion. If the number of processors available is much less than the degree of useful parallelism present in the program, some form of down-scaling must take place. Skeletons are flexible in this respect, because the number of processors allocated can be set by the compiler in accordance with available resources. Generally we assume that ample computing resources are available, such that further down-scaling, involving for example sequentialisation of skeleton pipelines, should not be necessary. A fully-fledged implementation of the compiler should be able to address such issues by carefully weighing the exploitation of parallelism up against economical resource utilisation. This however, we consider to be outside the scope of the thesis, and we accept that the mapping process fails in such circumstances.

As explained in section 6.5, the process network can be viewed as a hierarchy of components. At the topmost level the process network is structured with `PAR/SEQ` constructs and pipelines. Pipelines consist of skeleton nodes or sequential nodes, the latter of which may refer to further process networks through $CallNodeExp$ expressions. Mapping takes place in a hierarchical fashion, where each component in the process network takes responsibility for its own resource allocation. We outline the mapping process through a set of schemes, $\mathcal{M}$. The algorithm described has its limitations, to be discussed later, and should be viewed as a sketch rather than as a definite solution to the mapping problem. In the SkelML prototype compiler a simplified version of this algorithm is implemented, also to be discussed later in the chapter.

7.2.1 Modeling and Allocating Machine Resources

Our model of the physical architecture is based on that of our target platform: a transputer network. Each processor has a fixed number of bidirectional communication links which can be statically connected to a link at any other processor. During mapping nodes and arcs are assigned to processors and communication links respectively. A processor can have assigned to it any number of nodes provided the arcs of each node can be assigned to communication links consistently with earlier assignments. A processor can also be assigned a number of internal arcs, i.e. arcs which connect nodes allocated to the same processor. Nodes are assigned to processors one by one, where the best suited processor is picked according to a best fit criterion. For each node to be assigned, an integer penalty score is calculated for all available processors indicating the load in resource utilisation incurred by assigning the node to the processor. The node is eventually allocated to the processor with the best score. An increasingly positive score indicates a progressively worse assignment, with the best possible score being 1. The score value 0 is reserved for the special purpose of indicating that the node cannot be assigned to the processor, for example because the node's arcs are incompatible with the connections of the processor's links.

By altering and weighing the factors affecting the score, we have attempted to retain some degree of flexibility in the assignment of nodes to allow economical use of resources. We have experimented with the following penalties:

- Providing no parallelism is lost, assigning communicating nodes to the same processor saves communication time costs and uses fewer communication links. Hence there is a penalty for assigning arcs to physical links rather than leaving them as internal arcs.

- Committing unused links severely decreases flexibility, and therefore carries a penalty. It is preferable to use links already committed as long as arcs assigned to the link do not communicate in the same direction at the same time.

- When all links of a processor are connected, it is only possible to assign further nodes to that processor if they have communication needs exactly matching the existing configuration of the links. Thus, for each processor there is a penalty for each connected link.

In the mapping schemes we represent processors as an array **P** where each element is a set of nodes and a set of arc/link assignments corresponding to nodes and arcs respectively that are assigned to a processor. When mapping is completed the array **P** holds all the information necessary to produce occam2 code. We also use a set $\mathcal{P}$ representing those processors that are available at any given time. This can be conveniently represented as a set of integers (processor identifiers). We will normally refer to **P** and $\mathcal{P}$ as a tuple $<\mathbf{P},\mathcal{P}>$. When assigning a node to $<\mathbf{P},\mathcal{P}>$, the assignment can only be to processors in $\mathcal{P}$, and it must be consistent with previous processor assignments as represented in **P**.

For allocation of resources we assume a scheme

$$\textbf{AssignProc n } <\mathbf{P},\mathcal{P}> \textbf{ t}$$

which given a node **n** and a processor tuple $<\mathbf{P},\mathcal{P}>$ assigns the node to the best fitting processor(s) in $\mathcal{P}$ and returns a triple $<\mathbf{P}',\mathcal{P}',\mathbf{t}'>$ of a new updated processor array **P**' and a set $\mathcal{P}' \subset \mathcal{P}$ of those processors not affected by the assignment. **t** is a time bound which when having the special value of **MINIMAL** signifies that the fastest possible implementation given the available processing resources is sought. Otherwise the time bound is an upper time limit signifying that an implementation is sought which uses as few processors as possible, while still being at least as fast as the bound. The same representation of processors and time is also used in the set of mapping schemes, $\mathcal{M}$.

AssignProc can be applied to two kinds of nodes: skeleton nodes and sequential nodes. For skeleton nodes the relevant performance formulae are used to determine the optimal number of processors with respect to **t** and the number of processors available. The node is then expanded to a corresponding number of sub-nodes to be assigned to processors. For sequential nodes, the node is allocated to one processor, while allocation of further processors to cater for $CallNodeExp$'s in the node's expression is handled separately by a call to $\mathcal{M}$. For sequential nodes the **t** parameter is not used, and a time of 0 is always returned.

We briefly sketch the functionality of **AssignProc**:

```
AssignProc n <P,𝒫> t
    if <n is a sequential node> then
        for <all processors p_i in 𝒫> do
            pen_i := <penalty of assigning n to p_i>
        <assign n to the processor with the lowest pen_i, (pen_i > 0) >
        t' := 0
    else <n is a skeleton node>
        <use skeleton performance model to find optimal
            number of processors given 𝒫 and t>
        t' := <expected completion time according to model>
        <expand n to optimal number of sub-nodes>
        for <all sub-nodes sn_i of n> do
            <assign sn_i to the processor with the lowest penalty (as above)>
    <P',𝒫'> := <<P,𝒫> updated to reflect assignments>
    return <P',𝒫',t'>
```

In some cases no assignment is possible, for example because existing node and arc assignments on the processors in $\mathcal{P}$ are incompatible with the arcs of **n**. There are several ways of overcoming this. Routing messages through other processors is one possibility. Another is to backtrack and restrict the exploitation of parallelism thus reducing the need for communicating nodes to be located on different processors. Both these options involve complexities which in a fully-fledged compiler would have to be addressed satisfactorily. We consider that elaborating further on this is outside the scope of our work. Firstly such problems arise partly because of lack of resources, requiring a highly economic resource utilisation, while we explicitly assume that ample resources are available. Secondly these issues involve problems related more to mapping *per se* than to the more general concerns of skeleton-based compilation, and thus falls on the side of the main topic of the thesis. Hence, we will have to accept that in some cases the mapping algorithm fails because of resource allocations problems. This should however not prevent us from applying the compiler to a wide and varied range of problems, nor from illustrating the merits of skeleton-based compilation.

7.2.2 Load Balancing and Distribution of Processors

In the following section we describe the mapping process through a number of schemes. One re-occurring problem in these schemes deals with the disjoint partitioning and allocation of processors between program components. In essence we may have a set of tasks T_1,...,T_n to run in parallel and a set of $\mathcal{P}$ processors available. We want to allocate a set of processors $\mathcal{P}_i$ to each task T_i such that no processor is allocated to more than one task, i.e. $\mathcal{P}_1 \cap ... \cap \mathcal{P}_n = \emptyset$, and only processors in $\mathcal{P}$ are used, i.e. $\mathcal{P}_1 \cup ... \cup \mathcal{P}_n \subseteq \mathcal{P}$, so that the completion time is minimised and preferably without using more processors than necessary. This problem occurs in `PAR` constructs in process networks, in skeleton pipelines, in tuples $(exp_1, ..., exp_n)$ and in nested function application $(...((exp\ exp_1)\ exp_2)\ ...\ exp_n)$ where two or more of the expressions exp_i are $CallNodeExp$'s. Pelagatti discusses a similar problem of balancing the load of a process pipeline where each stage may itself be a parallel module [127]. Resources can be allocated accordingly by

1. Finding the optimal completion times t_{C1},...,t_{Cn} for the n tasks.

2. Identifying the slowest of these, i.e. $t_{max} = max\ t_{Ci}$. The slowest task puts a bound on the lowest total completion time for the set of tasks.

3. Optimising resource usage by for each task T_i allocating the minimal number of processors such that $t_{Ci}\ \leq\ t_{max}$.

It is here assumed that there is no shortage of processors available. In practice we will have to deal with having fewer processors at disposal than there is parallelism available. While we do not consider the general problem of accommodating a large number of processes on a much smaller number of processors, the number of processors used in each skeleton can be varied. For example if the performance model of a farm indicates that 40 processors could be usefully employed, we could choose to use only 10 or 25 if that was all we had at our disposal. One solution is to first apply steps 1-3 as above and then iteratively reduce the number of processors used in the implementation:

4. For each task T_i compute t'_{Ci} using 1 fewer processors than required for achieving time t_{Ci}.

5. For the task for which using one less processor incurs the smallest penalty to the total completion time, i.e. where (t'_{Ci} - t_{max}) is the least, remove one processor and update t_{Ci} to t'_{Ci}.

6. Find the new t_{max} as in step 2, and repeat from step 4 until the total number of processors used is less or equal to that available.

This still fails to take into account the fact that processors are not quite identical. That is, *physically* we assume they are, but each processor in $\mathcal{P}$ may already have assigned to it a number of nodes which puts bounds on how well it can be used with the current tasks. Note that the source of this problem is that there is a shortage of processors available. Had we had more processors there would have been less pressure on each to accommodate several nodes. As already stated we will generally assume that ample computing resources are available, and we do not intend to elaborate further on the issue here. We will simply assume that the $\mathcal{P}$ processors can be split into n subsets and will accept our mapping algorithm to fail if such a partitioning cannot be done.

However, even the computation of any of the above t_C's cannot be done without some knowledge of processing resources available. The calculation of t_C for a task is dependent on the calculation of the t_C's at lower levels for each of its sub-components, which again is dependent on available resources. Thus, we end up with a relatively simple, but costly algorithm. We will return to the issue of time complexity in section 7.2.4.

We capture the balancing of a number of parallel tasks by a scheme **Balance** sketched below.

$$\mathbf{Balance}\ [\mathbf{T}_1,\mathbf{T}_2,\ldots,\mathbf{T}_n]\ <\mathbf{P},\mathcal{P}>\ \mathbf{t}$$

takes a list of tasks, a representation of processor assignments and available processors, and a time bound. **Balance** returns a triple $<\mathbf{P'},\mathcal{P}',\mathbf{t'}>$ where **P'** is the updated processor assignments and $\mathcal{P}' \subset \mathcal{P}$ contains those processors not affected by the assignment, while **t'** is the time for completion of all tasks. **Balance** follows the steps 1-6 as outlined previously. The variables **P'** and $\mathcal{P}$**'** are initially set equal to **P** and $\mathcal{P}$ and are then repeatedly updated to hold intermediate results.

```
Balance [T_1,T_2,...,T_n] <P,𝒫> t
    <P',𝒫'> := <P,𝒫 >
    for i := 1 to n do
        <P',𝒫',t_Ci > := 𝓜 T_i <P',𝒫'> t
    t_max := max_{i=1}^n t_Ci
    if t ≠ MINIMAL then
        <P',𝒫'> := <P,𝒫 >
        for i := 1 to n do
            <P',𝒫',t_Ci > := 𝓜 T_i <P',𝒫'> t_max
    while <too few processors available>
        <P',𝒫'> := <P,𝒫>
        for i := 1 to n do
            <set 𝒫' to be one fewer processors
                than when obtaining time t_Ci>
            <P",𝒫",t'_Ci > := 𝓜 T_i <P',𝒫'> MINIMAL
            diff_i := t'_Ci - t_max
            <update <P',𝒫'> w.r.t. <P",𝒫">>
        <for the task T_i with the smallest diff_i, set t_Ci := t'_Ci>
        t_max := max_{i=1}^n t_Ci
    <P',𝒫',t'> := <value of best allocation found>
    return <P',𝒫',t'>
```

The **while** loop removes processors from those stages where the effect on the overall completion time is the least. This takes place by attempting to remove processors one by one from each stage in turn. The update of $<\mathbf{P'},\mathcal{P}'>$ in the inner **for** loop should be such that for each stage only the effect of removing a processor from that stage is captured. Eventually the assignment is returned which satisfies the number of processors available.

7.2.3 Mapping Schemes

The starting point for the mapping is a set of processors, $\mathcal{P}$, and the topmost process network. Assignment of nodes and arcs to each processor is captured in an array $\mathbf{P}$, which initially has no nodes nor arcs assigned. Before starting the general mapping, the terminal node is assigned to a specific processor which is earmarked to be connected to the host machine. Thus, one link of this processor is used for toplevel I/O, and can not be assigned any arcs in the process network.

A central issue in the mapping algorithm is the handling of nodes. Sequential nodes are handled as ordinary sequential processes with the exception of $CallNodeExp$s, see later in this section. The mapping of skeleton nodes is handled in **AssignProc** by expanding each to a network of one or more sub-nodes representing the processors of the skeleton and their internal interconnection. Skeleton performance models are used to decide how many processors each skeleton should expand to, or if sequential execution is preferable. This depends on the available resources and the performance of the network of which the skeleton is a part. It is therefore not possible to treat each skeleton independently and allocate processors only from local information. Thus we cannot treat the mapping problem as dealing with nodes each of which has a known execution cost. Skeleton execution costs will have to be calculated from within the mapping algorithm itself, which complicates the mapping process.

When calculating expected executions times for the network, communication times are included. These depend on the mode of communication, i.e. inter- or intra- processor, and are calculated according to the assignments of nodes to processors.

Mapping is described by a number of schemes, denoted $\mathcal{M}$ with a subscript indicating the class of network component on which it operates. Each component of the process network has its own mapping scheme:

- For process networks, schemes $\mathcal{M}_{\texttt{PAR}}$ for PAR constructs, and $\mathcal{M}_{\texttt{SEQ}}$ for SEQ constructs.
- For pipelines, schemes $\mathcal{M}_{skelp}$ for skeleton pipelines, and $\mathcal{M}_{seqp}$ for other pipelines.
- For nodes, schemes $\mathcal{M}_{skeln}$ for skeleton nodes, and $\mathcal{M}_{seqn}$ for sequential nodes.
- For SML expressions, a scheme $\mathcal{M}_e$

We will often omit the subscript, as the choice of scheme is always unambiguous, and assume this is being handled by some pattern matching mechanism.

Throughout the description of $\mathcal{M}$ our emphasis is on concepts rather than detailed accuracy. In many places there are comments indicating points where alternative choices are available, and where simplifications are made. The notation used throughout is semi-formal with many hidden details commented in square brackets: $< >$, not to be confused with the brackets used for denoting tuples and triples of processors and time bound.

Each scheme is of the form:

$\mathcal{M}$ **X** $<$**P**,$\mathcal{P}>$ **t**

where **X** is any of the above components, $<$**P**,$\mathcal{P}>$ is a tuple of previous processor assignments and available processors, and **t** is a time bound. The result is a triple $<$**P'**,$\mathcal{P}$**'**,**t'**$>$ where **P'** is the updated processor assignments and $\mathcal{P}$' contains those processors not affected by the assignment, while **t'** represents the expected time for the execution of **X**.

PAR constructs are handled by a direct call to **Balance**:

$\mathcal{M}_{\texttt{PAR}}$ PAR $[net_1,\ldots,net_n]$ $<$**P**,$\mathcal{P}>$ **t**
    return Balance $[net_1,\ldots,net_n]$ $<$**P**,$\mathcal{P}>$ **t**

SEQ constructs:

$\mathcal{M}_{\texttt{SEQ}}$ SEQ $[net_1,\ldots,net_n]$ $<$**P**,$\mathcal{P}>$ **t**
    P' := P

for i := 1 to n do
    $<$**P'**,$\mathcal{P}_i$,**t**$_i >$:= $\mathcal{M}$ net_i $<$**P'**,$\mathcal{P}>$ MINIMAL
t' := $\mathbf{t}_n + \sum_{i=1}^{n-1}$ ($\mathbf{t}_i$ + <communication
            cost between net_i and net_{i+1}>)
if t = MINIMAL then return $<$**P'**,$\bigcap_{i=1}^{n} \mathcal{P}_i$,**t'**$>$
else
    while t' $<$ t
        for i := 1 to n do
            <compute the effect on **t'** of using one less processor in net_i>
        <remove one processor from net_i where the least extra
            time cost is incurred>
    $<$**P'**,$\mathcal{P}$**'**,**t'**$>$:= <best values when **t'** $\leq$ **t**>
    return $<$**P'**,$\mathcal{P}$**'**,**t'**$>$

The scheme first obtains the **MINIMAL** times for all stages. If **t** is a lower time bound, i.e. **t** is not **MINIMAL** then processors are removed one by one from the *net*s where this incurs the least extra time cost, until an allocation is found with **t'** just within the bound of **t**. If the `SEQ` construct is of the form `SEQ` [N,net,N], as discussed in section 6.5, the N component indicates the start and end node of a sub-network. In such cases N would have been dealt with at a higher level of $\mathcal{M}$. Thus the scheme should make no calls to $\mathcal{M}$ for N at this point, and the time for computing N should not be included in the total time for the `SEQ` construct. The communication times to and from N relate to the evaluation of the `SEQ` construct and *should* therefore be included as described.

As outlined in section 6.2.4 we distinguish two types of pipelines, sequential pipelines and skeleton pipelines. Assume a pipeline has n stages of nodes $node_1$, ..., $node_n$. For skeleton pipelines we have:

$\mathcal{M}_{skelp}$ `SKELPIPE` [$node_1$,...,$node_n$] $<$**P**,$\mathcal{P}>$ **t**
    if t = MINIMAL then
        $<$**P'**,$\mathcal{P}$**'**,**t'**$>$:= **Balance** [$node_1$,...,$node_n$] $<$**P**,$\mathcal{P}>$ **t**
        $\mathbf{t}_L$:= <latency of pipe>
        return $<$**P'**,$\mathcal{P}$**'**,**t'**+$\mathbf{t}_L>$
    else
        $\mathbf{t}_L$:= <latency of pipe>
        $<$**P'**,$\mathcal{P}$**'**,**t'**$>$:= **Balance** [$node_1$,...,$node_n$] $<$**P**,$\mathcal{P}>$ **t**-$\mathbf{t}_L$
        return $<$**P'**,$\mathcal{P}$**'**,**t'**+$\mathbf{t}_L>$

Here **Balance** is used for balancing the load between the stages running in parallel. The latency is added to the total time. If **t** is not **MINIMAL** then the balancing takes place with a lower time bound (less the latency). In the scheme we have assigned a value to $\mathbf{t}_L$ before doing the balancing, although the exact latency is not known until after the balancing has taken place. For this purpose we would need a version of **Balance**, say **BalancePipe** which takes latency into account. We have however disregarded this in the scheme and abused the notation slightly.

$\mathcal{M}_{seqp}$ `PIPE` $[node_1,...,node_n]$ $<\mathbf{P},\mathcal{P}>$ **t**
  P' := P
  for i := 1 to n do
    $<\mathbf{P'},\mathcal{P}_i,\mathbf{t}_i> := \mathcal{M}\ node_i$ $<\mathbf{P'},\mathcal{P}>$ **MINIMAL**
  $\mathbf{t'} := \mathbf{t}_n + \sum_{i=1}^{n-1} (\mathbf{t}_i +$ <communication
      cost between $node_i$ and $node_{i+1}$>)
  if t = MINIMAL then return $<\mathbf{P'},\bigcap_{i=1}^{n} \mathcal{P}_i,\mathbf{t'}>$
  else
    <...proceed as for $\mathcal{M}_{\texttt{SEQ}}$...>
    $<\mathbf{P'},\mathcal{P'},\mathbf{t'}> :=$ <best values when $\mathbf{t'} \leq \mathbf{t}$>
    return $<\mathbf{P'},\mathcal{P'},\mathbf{t'}>$

This scheme is virtually identical to $\mathcal{M}_{\texttt{SEQ}}$, and needs no further comment.

For skeleton nodes, the **AssignProc** scheme is called to handle the assignments in accordance with $<$P,$\mathcal{P}>$ and **t**, using the appropriate performance models.

$\mathcal{M}_{skeln}$ **n** $<\mathbf{P},\mathcal{P}>$ **t**
  return AssignProc n $<\mathbf{P},\mathcal{P}>$ **t**

For sequential nodes:

$\mathcal{M}_{seqn}$ **n** $<\mathbf{P},\mathcal{P}>$ **t**
  $<\mathbf{P'},\mathcal{P'},\mathbf{t'}> :=$ **AssignProc n** $<\mathbf{P},\mathcal{P}>$ **t**
  $exp :=$ <the expression of the sequential node **n**>
  $<\mathbf{P'},\mathcal{P'},\mathbf{t'}> := \mathcal{M}_e [\![exp]\!]$ $<\mathbf{P'},\mathcal{P}>$ **MINIMAL**
  $\mathbf{t}_e :=$ <sequential profiling time of exp>
  $\mathbf{t}_{inv} :=$ <cost of invoking exp from **n**>
  return $<\mathbf{P'},\mathcal{P'},\mathbf{t}_e - \mathbf{t'} + \mathbf{t}_{inv}>$

The time returned from applying $\mathcal{M}_e$ to the SML expression represents a correction of the profiled time resulting from parallel evaluation of $CallNodeExp$'s.

The set of schemes $\mathcal{M}_e$ deals with evaluation of SML expressions in sequential nodes. The time **t** passed to the scheme is always **MINIMAL**. As explained in section 6.4, the cost of evaluating $CallNodeExp$'s may well be different form the profiled time. The time returned from $\mathcal{M}_e$ represents this difference. Thus, for some expressions, for example constants ($scon$'s) the returned time is 0 because there is no difference with respect to the profiled time.

$\mathcal{M}_e$ $[\![scon]\!]$ **<P,$\mathcal{P}$> t**
    return <P,$\mathcal{P}$,0>

$\mathcal{M}_e$ $[\![con]\!]$ **<P,$\mathcal{P}$> t**
    return <P,$\mathcal{P}$,0>

$\mathcal{M}_e$ $[\![v]\!]$ **<P,$\mathcal{P}$> t**
    case
        $v \in \mathcal{B}$ **: return <P,$\mathcal{P}$,0>**
        <recursive invocation of v> **: return <P,$\mathcal{P}$,0>**
        <v locally bound> **: return <P,$\mathcal{P}$,0>**
        $v \in \mathcal{E}$ **: return** $\mathcal{M}_e$ $[\![\mathcal{E}_f(v)]\!]$ **<P,$\mathcal{P}$> t**

$\mathcal{M}_e$ $[\![$`fn` p_1=>e_1 | p_2=>e_2 | ... | p_n=>$e_n]\!]$ **<P,$\mathcal{P}$> t**
    for i := 1 to n do
        <P',$\mathcal{P}_i$,t$_i$> := $\mathcal{M}_e$ $[\![e_i]\!]$ **<P',$\mathcal{P}$> t**
        f$_i$:= <number of times branch **i** has been selected>
    return <P',$\bigcap_{i=1}^{n} \mathcal{P}_i$**,(**$\sum_{i=1}^{n}$ **(t$_i$ * f$_i$)) /** $\sum_{i=1}^{n}$ **f$_i$>**

The times for each branch are weighed with the relative frequency $\mathbf{f}_i$ of that branch being taken, as outlined in section 6.4.

$\mathcal{M}_e$ $[\![(e_1, e_2, ..., e_n)]\!]$ **<P,$\mathcal{P}$> t**
    L1 := <list of those $e_1, ..., e_n$ that are $CallNodeExp$'s>
    L2 := <list of those $e_1, ..., e_n$ that are not $CallNodeExp$'s>
    <P',$\mathcal{P}$'> : = <P,$\mathcal{P}$>
    for <each element **l2$_i$ in L2**> **do**
        <P',$\mathcal{P}_i$,t$_i$> := $\mathcal{M}_e$ **l2$_i$ <P',$\mathcal{P}$> MINIMAL**
    $\mathbf{t}_{seq}$ **:=** $\sum_{i=1}^{length\mathbf{L2}}$ $\mathbf{t}_i$
    <P',$\mathcal{P}$',t$_{par}$ > := Balance L1 <P',$\bigcap_{i=1}^{length\mathbf{L2}}$ $\mathcal{P}_i$ **> MINIMAL**
    return <P',$\mathcal{P}$',t$_{seq}$+t$_{par}$>

The treatment of tuples offers several alternatives for mapping. One option is to calculate the communication vs computation times for each expression e_1,...,e_n, and evaluate those tasks in parallel which give a gain in performance. This is attractive because every expression e_i may contain $CallNodeExp$'s. We have sketched a more moderate approach where only those expressions with a top level $CallNodeExp$ are considered for external evaluation. The evaluation of a tuple should take place by executing the sequential part locally while at the same time having all sub-networks of $CallNodeExp$'s working in parallel. The tuple expressions are split into two groups, one for those being $CallNodeExp$'s and another group for the other expressions. The scheme is somewhat inaccurate with respect to the communication costs for the $CallNodeExp$ invocations. The reason is that the code for communication to and from the sequential node will execute concurrently, and not in parallel as assumed in the **Balance** scheme. **Balance** is called with time bound **MINIMAL**. An alternative, and possibly more economic approach is to use a time equivalent to that of evaluating the expressions in **L2**. As for $\mathcal{M}_{\mathtt{PAR}}$ we would need a specialised version of **Balance** that would take into account not the time returned from its calls to $\mathcal{M}_e$, but rather the profiled time *less* the time returned from $\mathcal{M}_e$. In the prototype implementation an even more moderate approach is taken, where all tuple elements are evaluated in sequence. This still exploits parallelism in each $CallNodeExp$, but does not take advantage of the ability to evaluate several tuple elements simultaneously.

$\mathcal{M}_e$ $[\![e_1\ e_2]\!]$ <**P**,$\mathcal{P}$> **t**
  if $[\![e_1]\!] = [\![(...((e_1^{'}\ e_2^{'})\ e_3^{'})...\ e_n^{'})]\!]$ **then**
    L1 := <list of those $e_1^{'}, ..., e_n^{'}, e_2$ that are $CallNodeExp$'s>
    L2 := <list of those $e_1^{'}, ..., e_n^{'}, e_2$ that are not $CallNodeExp$'s>
  else
    L1 := []
    L2 := [e_1,e_2]
  <**P'**,$\mathcal{P}$'> : = <**P**,$\mathcal{P}$>
  for <each element **l2**$_i$ **in L2**> **do**
    <**P'**,$\mathcal{P}_i$,**t**$_i$> := $\mathcal{M}_e$ **l2**$_i$ <**P'**,$\mathcal{P}$> **MINIMAL**
  $\mathbf{t}_{seq} := \sum_{i=1}^{length\mathbf{L2}} \mathbf{t}_i$
  <**P'**,$\mathcal{P}$'**,t**$_{par}$ > := **Balance L1** <**P'**,$\bigcap_{i=1}^{length\mathbf{L2}} \mathcal{P}_i$ > **MINIMAL**
  return <**P'**,$\mathcal{P}$'**,t**$_{seq}$+**t**$_{par}$>

This scheme is virtually equivalent to the previous one in the handling of $CallNodeExp$'s. If applications are nested then the arguments may be candidates for parallel evaluation, in which case similar considerations as for tuples apply. Again, the prototype implementation is based on evaluating each expression $e_1^{'}, ..., e_n^{'}, e_2$ in sequence.

$\mathcal{M}_e\ [\![\texttt{let val } p_1{=}e_1 \texttt{ and } p_2{=}e_2 \texttt{ and } \ldots \texttt{ and } p_n{=}e_n \texttt{ in } E \texttt{ end}]\!]\ <\mathbf{P},\mathcal{P}>\ \mathbf{t}$

return $\mathcal{M}\ [\![(\texttt{fn } p_1 \texttt{=>} ((\texttt{fn } p_2 \texttt{=>} (\ldots \texttt{=>} ((\texttt{fn } p_n \texttt{=>} E)e_n)\ldots))e_2))e_1]\!]$

$<\mathbf{P},\mathcal{P}>\ \mathbf{t}$

The scheme for `let` expressions is handled in the same way as the semantically equivalent applications of `fn` expressions.

$\mathcal{M}_e\ [\![CallNodeExp(e, \mathbf{n}, \mathbf{I})]\!]\ <\mathbf{P},\mathcal{P}>\ \mathbf{t}$

N := <process sub-network that **n** is a part of>

$\mathbf{t}_e$:= <profile time of original expression e>

$<\mathbf{P'},\mathcal{P}',\mathbf{t}_{net}> := \mathcal{M}\ \mathbf{N}\ <\mathbf{P},\mathcal{P}>$ **MINIMAL**

$\mathbf{t}_{comm}$:= <time for sending and receiving data to/from **N**>

return $<\mathbf{P'},\mathcal{P}',\mathbf{t}_e - (\mathbf{t}_{net} + \mathbf{t}_{comm})>$

7.2.4 Discussion

As already indicated, $\mathcal{M}$ is a costly algorithm. The reason for this is that for some components, $\mathcal{M}$ calls itself more than once for the same sub-component. **Balance**, $\mathcal{M}_{\texttt{SEQ}}$, and $\mathcal{M}_{skelp}$ are all of this type, and we will refer to them as non-linear. If non-linear calls are nested to a depth d, and each call makes c further calls for each of its sub-components, the operations at the lowest level are performed c^d times. Thus, the time complexity of $\mathcal{M}$ is exponential in the nesting depth of non-linear calls. In general this is not very satisfactory. We will make a few points to make the picture more nuanced:

- The purpose of non-linear calls is that of resource optimisation. The first call establishes the initial best performance, while subsequent calls adjust resources to balance the load economically. If unlimited resources, or even resources for exploiting all useful parallelism maximally were available, we could have afforded a less economic implementation and there would have been no need for non-linear calls. In that case the complexity of $\mathcal{M}$ would have been linearly proportional to the size of the program, since each sub-component would only have been traversed once.

- The complexity of $\mathcal{M}$ is exponential in the nesting of non-linear calls. But nesting may not necessarily be a good measure of problem size. In fact, for the experimental results studied in the following chapter, there is no nesting. The structuring and abstraction mechanisms adopted in skeleton nodes and pipelines contribute to keeping the actual complexities of the mapping down. For many typical parallel applications the skeletal or parallel structure is relatively simple. Thus, although the worst case complexity of $\mathcal{M}$ is exponential, this may in fact not be a significant practical problem for many applications.

We conclude from this that for some purposes $\mathcal{M}$'s complexity may still be satisfactory. However, we have not intended $\mathcal{M}$ to be the ultimate solution to the mapping problem, but rather as an illustrating sketch.

Our mapping algorithm is biased towards architectures allowing a flexible configuration of communication links. The part of the algorithm dealing directly with the precise hardware interconnection topology is **AssignProc**, which assigns nodes and arcs to processors and communication links. The most critical task performed by **AssignProc** is that of allocating skeleton nodes, where a pattern of many sub-nodes is simultaneously assigned. This approach works well with the flexible interconnection facilities of the transputers, but may have to be replaced by other mechanisms to deal satisfactorily with other architectures, for example MIMD hypercubes or SIMD machines. However, by making suitable modifications to **AssignProc** in how skeleton nodes are mapped down to processors, we believe that the current hierarchical structure of the mapping can be retained. The essence of obtaining high performance over different platforms is to have tailor made mappings for each skeleton on each individual architecture so as to exploit the strengths of each architecture. In the P^3L compiler this is addressed by *mapping templates* [14]. A mapping template is a directed graph of processing nodes that describes the relationship between a skeleton implementation and the abstract machine of the target architecture. In this context, the equivalent of mapping templates in SkelML would be the **AssignProc** scheme and the networks of skeleton sub-nodes.

The implementation of $\mathcal{M}$ used in the prototype implementation of the compiler varies in a few points from what we have described here. First there we do not use the `PAR/SEQ` notation for pipelines. We have instead used a two-pass scheme for a network of pipes, a version of which is used in [145]. In the forward pass the earliest completion time for each pipe is noted, while in the backward pass the latest starting times are calculated. This allows for some amount of balancing, where non-critical paths are allowed to use resources economically. Another and more severe limitation is that internal arcs are not supported. This implies that connected nodes are allocated to different processors, which in some cases increases communication overheads and results in a less economical use of resources. As mentioned in the schemes, the implementation of tuples and application does not fully take advantage of all parallelism that might be available.

A few weaknesses of $\mathcal{M}$ itself should also be mentioned. $\mathcal{M}$'s mapping cannot claim to find the optimal assignment. This is partly because the assignment of nodes and arcs to physical resources is a sequential process, such that node A is assigned before node B, while a better mapping might have been obtained by first assigning B. Another weakness that we have mentioned earlier is the exploitation of parallelism in tuples. Here there is a potential for parallel evaluation which we have not fully exploited. Modifying the algorithm to cope with this should however be straightforward.

7.3 Optimisation and Transformation

7.3.1 Introduction

By optimisation we mean the transformation of a program into a semantically equivalent form, but with a more favourable behaviour. A program's execution time and space consumption are obvious targets in this respect. Program transformation in the context of functional programming languages has for a long time been subject to research attention. In a survey Busvine [37] identifies a wide range of approaches spanning from low-level local optimisations to program synthesis, and describes techniques of memoisation, recursion optimisation and fusion of function composition.

A major contribution is Burstall and Darlington's rule based fold/unfold methodology [34], proposed in 1977. Programs are expressed as a set of first order recursion equations.

Six simple rules are given describing how new equations can be derived from the existing. Briefly, the rules allow introduction of new equations, instantiation of equations, replacing expressions by their equals as declared by the equations (folding/unfolding), and introducing abstractions. A number of examples are given demonstrating how the rules can be used for transforming various programs.

While seemingly appealing, such program transformation is far from problem free. There are generally many applicable transformations to choose from at any given time, and it may be hard to decide which would eventually lead to an improved program. While fold/unfold transformations are very general in that most transformations can be expressed through them, each transformation step is relatively small. Feather [73] describes an approach where series of small transformation steps can be applied depending on the programmer's choice of some higher level strategies. Still, the responsibility of directing the transformation process rests mainly with the programmer. As noted by Darlington *et al* in later works [65], the lengthy sequences of transformation steps make fold/unfold transformations hard to fully automate.

Another approach is that of algebraic transformation, described by Field and Harrison [77] in the context of FP [15]. Transformations are expressed as a set of axiomatic equations defining equality of expressions. Any sub-expression in a program being an instance of the right hand side of such an equation can safely be replaced by a corresponding instance of the left hand side, and vice versa. Algebraic transformations are typically of a coarser grain than fold/unfold transformation encouraging transformation at the level of relationships between functions.

Some work on skeleton transformations are reported in the literature. Reekie [140, 141] gives examples of transformations on skeleton process networks derived from Haskell programs. Skillicorn and Cai [157] develop a cost calculus for BMF and outlines how this can be used for deciding whether or not a transformation rule should be applied, and in which direction. The middle-section of the P^3L compiler is capable of invoking rules for transforming process trees to improve performance [14]. Rules can be invoked if the program structure matches the structure of the transformation, and if a bandwidth related predicate holds. Darlington *et al* [64] stress the importance of transformation at all levels of the development process. At the topmost level for transforming a program into initial skeleton forms, then for transformation between skeleton forms, and finally to fine-tune the program for a particular target architecture. A decision tree approach is proposed, where the programmer can select which transformations to apply.

7.3.2 Transformation in the SkelML Compiler

In the SkelML compiler we support a simple form of optimising transformations in linear pipelines of skeleton nodes. A transformation operates on the pipeline by replacing one or more of its stages with semantically equivalent counterparts. This replacement is considered to be favourable if the transformed pipeline has an expected faster completion time than the original. Completion times are calculated using the performance models described in chapter 5.

Transformations are represented as equational rules of the form:

$$\texttt{trans}\ \{\mathit{boundvars}\}\ \mathit{exp}_1\ \texttt{==}\ \mathit{exp}_2$$

where the expressions exp_1 and exp_2 must be semantically equivalent, while *boundvars* is a list of identifiers occurring free in the two expressions. Any occurrence of exp_1 in the program can be replaced by exp_2 and vice versa providing the values of the free identifiers are identical. An example transformation rule is:

```
trans {f,g,l} map f (map g l) == map (fn x=>f (g x)) l;
```

stating that for any identifiers `f`, `g` and `l`, the composition of `map`s with `f` and `g` over `l` is equivalent to a single `map` with a composition of `f` and `g` over `l`. This is illustrated in figure 7.1, where we use the `o` operator to denote function composition[1]. SkelML considers

[1]The reason why `o` is not used in other places is that it is not part of our SML subset.

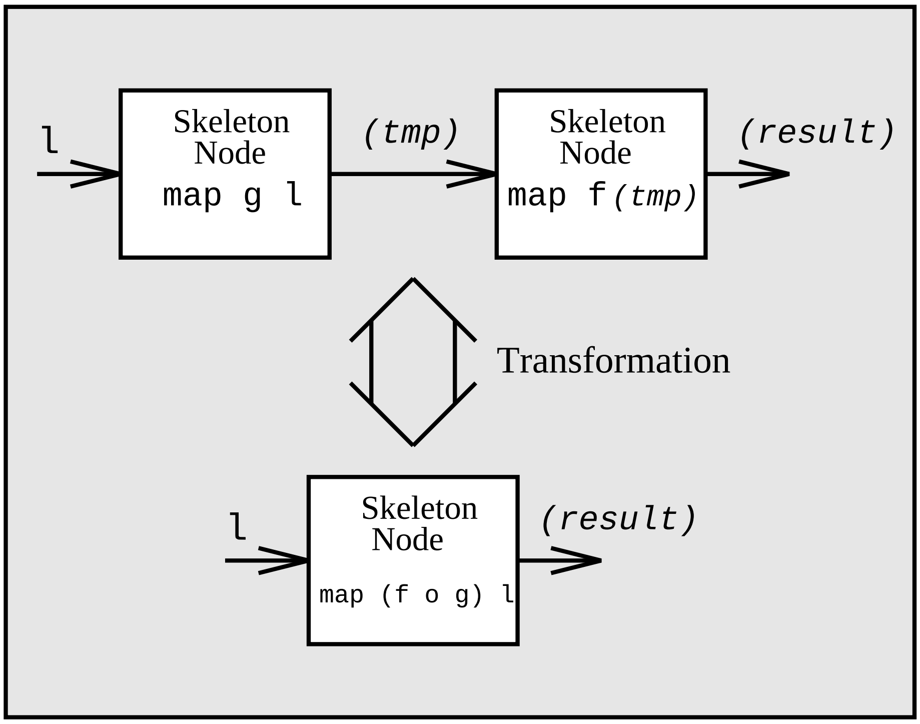


Figure 7.1: Pipeline transformation: `(map f) o (map g) == map (f o g)`

applying transformations only in skeleton pipelines. Thus, transformations not involving skeleton-HOFs have no effect. The above transformation could be used to transform a pipeline having two consecutive `map` skeletons into a pipeline where the two `map`s were replaced by one. Alternatively, a pipeline containing one `map` node with a functional argument of the form `(fn x=>f (g x))` for any expressions `f` and `g` could be replaced by two `map` nodes conversely. The transformation rules currently supported by the SkelML compiler are the following six, labelled T1-T6:

```
(* T1 *)
trans {f,g,l} map f (map g l) == map (fn x=>f (g x)) l;

(* T2 *)
trans {f,p,l} map f (filter p l) == mapfilter f p l;

(* T3 *)
trans {f,p,l} filter p (map f l) == filtermap p f l;

(* T4 *)
trans {f,g,a,l} fold f a (map g l) == foldmap f a g l;

(* T5 *)
trans {p1,p2,l} filter p2 (filter p1 l) ==
                filter (fn x => (p1 x) andalso (p2 x)) l;

(* T6 *)
trans {p1,p2,l} filter p1 (filter p2 l) == filter p2 (filter p1 l);
```

Note that the last rule, T6, corresponding to the Bird-Meertens law of commutativity of filters mentioned in section 3.1.2, is only applicable if both `p1` and `p2` are non-partial. These rules are chosen because they fit well with our set of supported skeletons.

Given that our emphasis is on automating the compilation process, it would be preferable if transformations could be handled by the compiler. For any pipeline, a number of transformations may be applicable, each of which may in turn make further transformations possible and so on. Discarding cyclic transformations the search space of semantically equivalent pipelines takes the form of a tree with each node being a pipeline, and each arc being a transformation rule. The size of the tree depends on the size of the pipeline and on the number of transformation rules. By using performance models, a compiler can evaluate the predicted completion time of a pipeline, and thus be capable of solving the search problem of finding the pipeline implementation with the lowest cost.

For any skeleton pipeline in the process network SkelML considers if transformations are applicable. The search algorithm implemented takes a pipeline P and a list of prior transformations T (initially empty) as arguments, and returns its optimal completion time Tc together with a list of transformations T' leading to the corresponding optimal implementation. The algorithm is:

1. Set Tc to the completion time of P. Set T' to T.

2. Get a list of all transformation rules applicable to this pipeline, discarding cyclic transformations. If no transformations applicable then go to step 4, otherwise for each transformation rule t in the list do step 3:

3. Apply the rule t to P producing a new transformed pipeline PP. Add t to T producing a new list of transformations TT. Call the search algorithm recursively with (PP,TT) as arguments resulting in (TTc,TT'). If TTc < Tc then set Tc to TTc and T' to TT'. Repeat for other transformation rules applicable to P.

4. Return (Tc,T').

This algorithm performs a depth-first left-to-right exhaustive search of the tree of transformation possibilities. In step 2 the list of all applicable transformations is found by checking the supported transformation rules T1-T6 in turn. Note that each rule may be applicable in either direction, LHS to RHS and vice versa. In the lists of transformations; T, T', TT, and TT' we assume that the transformation is stored together with some information on where it is to be applied, e.g. which stages in the pipeline are involved. This avoids ambiguity where a transformation rule might be applicable in more than one way in the same pipeline.

We have incorporated transformation as a first step in the mapping process, considering all pipelines in the process network before $\mathcal{M}$ is applied. The evaluation of a pipeline's completion time in step 1 is based upon the performance model of a skeleton pipeline assuming there are resources available to allow the best possible performance of each of its components. This may not necessarily be representative of the resources allocated to the pipeline when implemented, and so a better approach may be to apply transformations in a pipeline when the resources available are better known, most conveniently possibly in the $\mathcal{M}_{skelp}$ scheme. As this adds further to the mapping process, we have chosen to keep transformations and mapping apart in the prototype implementation.

7.3.3 Search Strategies

In a transformation system working on general program structures and with a large number of rules applicable, an exhaustive search for the optimal implementation may prove infeasible, as the total search space of possible implementations can be huge. In the current version of SkelML, where transformations are applied in pipelines only and where a limited number of transformation rules and skeletons are supported, an exhaustive search seems more feasible providing pipelines are not very long.

The advantage of doing an exhaustive search is that it is guaranteed that the best possible solution is found. Trying to find the best solution without an exhaustive search is hard, because the path of transformation steps from the original program to the optimal implementation may not be obvious. In particular, a greedy approach of picking the most profitable transformation at every step may lead to the finding of a local minimum. However, there is still no guarantee that a local minimum is the best solution overall. Several possibly unprofitable intermediate transformations may have to be applied on the way to reaching the goal. It would therefore seem attractive to have some strategies guiding and restricting the compiler's exploration of the transformation tree. For instance some form of heuristic rules can be explored. A heuristic rule may reflect the wisdom that under certain circumstances certain transformation rules generally tend to lead to a good solution. For example it may turn out to be good rule of thumb to group smaller units into bigger, thereby saving communication overhead. Developing such heuristics is no easy undertaking, and we have made no efforts in elaborating this further.

It is our firm belief that the programmer should be involved as little as possible in the compilation process. However, many problems involving knowledge and intuition which are hard to quantify are dealt with better by humans than by machines. It might be worth considering if that applies here. A user directed transformation system, or a system in which the user can use a transformation tool for finding local minima or performing exhaustive searches on fractions of the transformation tree might be the most practical approach. In our model of compilation, transformations are used as an optimisation mechanism, but do not play a major role in program development. In BMF however, transformations are central, and can be applied at different levels in the program. The programmer can typically start with a an obviously correct version of the program and transform it into an efficient one by using BMF's set of equational rules. If this is to be guided mainly by the programmer, the task of programming becomes a matter of choosing clever transformations as well as expressing the problem in the first place. We believe it requires considerable programmer expertise to perform such transformations skillfully, and we suggest the development of heuristics as a potentially rewarding avenue of research.

7.4 Summary

This chapter has been devoted to mapping and optimising transformations. We have concentrated on the problems involved in mapping an internal process network representation down to physical resources in the context of the SkelML compiler. The assignment of resources is captured by a scheme **AssignProc** that updates an array of processor/link assignments. The specific problem of balancing the parallel execution of possibly parallel sub-tasks has been discussed, and a scheme **Balance** outlined. A set of mapping schemes, $\mathcal{M}$ have been proposed and discussed. $\mathcal{M}$'s time complexity is exponential in the nesting of non-linear calls to other schemes in $\mathcal{M}$. Despite this we speculate that in practice $\mathcal{M}$ may exhibit an acceptable time behaviour for many problems. A few other deficiencies of $\mathcal{M}$ and an account of its implementation in the current SkelML prototype have been given. We have also discussed transformations in the context of skeleton-based compilation, and have outlined the approach taken in the SkelML compiler. Transformation rules for pipelines are expressed explicitly, and an exhaustive search mechanism is implemented that automatically finds the optimal pipeline implementation based on performance modeling information.

Chapter 8

Experimental Results

In the preceding chapters we have outlined our approach with respect to the individual components of the SkelML compiler. In this chapter we focus on the application of the compiler to three example problems. In section 8.1 we discuss a feature of the SkelML prototype implementation relevant to the compilation of these examples. Sections 8.2 - 8.4 present the three examples programs: a ray tracer, a best match problem, and a program for computing the area of symmetric objects. For each example we describe the problem and its SML implementation, we outline the compilation process and we discuss performance results. Section 8.5 summarises the chapter.

8.1 Scaling in the SkelML Prototype

In the previous chapters we have pointed out a number of aspects in which the SkelML prototype implementation deviates from the more general compiler framework discussed in the thesis. These differences are mainly simplifications or shortcuts motivated by the need for keeping the complexity of the implementation within reasonable bounds. Here we describe scaling of profiling information, which is used for compiling the three examples in this chapter.

For convenience it may sometimes be possible to use a relatively small set of profiling data and then scale the derived information up to reflect the volume of the real data. Thus, the results of small profiling runs can be used for predicting performance of larger runs. Scaling of profiling information is not uncommon in parallelising compilers. In PUFF, the programmer can manually adjust the average recursion depth of any recursive function [40], while in PTRAN a user supplied scale factor is multiplied to all loop frequencies [146]. The mechanism implemented in the SkelML prototype allows the list length of a parameter to a skeleton to be multiplied by a constant integer factor. We may for example profile a `map` with a list length of 20, and use a scaling factor of 100 if the real data has a list length of 2000. For each skeleton node in the process graph, SkelML prompts the programmer for a scaling factor, which is taken into account when predicting the skeleton's performance. Running large profiling runs is not always possible with the SkelML prototype due to problems with its memory handling, and scaling provides an efficient solution to this.

However the propagation of scaling information to other parts of the program is problematic. Consider for example

```
fun f x = ...;
fun g x = ...;
fun top L = g (map f L);
```

If the input to the `map` skeleton is scaled up by a factor of C, its output list will also be C times longer. In general, it is not possible to say how this will affect the time for computing `g` or the size of `g`'s result. Thus the compiler features no such automatic propagation of scaling information. To obtain communication and computation time estimates representative of the profiling information we have at hand, we have used SkelML's performance predictions as a basis, and where appropriate manually added an extra time cost incurred by the scaling up of the input data. For example if we know that `g` consumes time linear to its argument length, and that the `map` skeleton is scaled by a factor of C, then we can assume that `g` will take C times longer to execute than the estimate produced by the profiling data. Similarly for communication times, if the size of `g`'s output would increase as a result of the scaling, then the estimated communication costs must be scaled up accordingly. In general this would require quite extensive programmer knowledge about the time complexities of the various parts of the program. We must stress however that this approach to scaling is an ad hoc mechanism motivated by the immaturity of the compiler implementation. All estimated performance figures are directly based on the compiler's own predictions, and any error in these figures would be scaled up accordingly. This should ensure that costs for sequential pieces of code and for communication is in accordance with the performance models and with the scaling factor for the skeletons. If a more robust version of the compiler were available, we could have made profiling runs with full data sets, and thus have avoided this problem, but even in that case, a scaling mechanism could have been desired for cases where the volume of the real data sets were very big.

8.2 Ray Tracing

8.2.1 Problem Description

Our first example is a ray tracer based on an example in [104], and implemented in SML by David Busvine. The program has the following top level structure[1]:

```
datatype Impact = IMPACT of real * int | NOIMPACT;
  (* IMPACT(dist,objno) indicates impact at distance dist with object objno,
     NOIMPACT indicates no impact *)

fun FirstImpact objects ray = ... ;
  (* Finds first impact, or NOIMPACT, between ray and objects *)

fun FindImpacts rays objects = map (FirstImpact objects) rays;
  (* Finds all first impacts between objects and rays *)

fun GenerateRays (Detail,X,Y,Z) = ... ;
  (* Generates Detail * Detail rays from viewpoint (X,Y,Z) towards origin *)

fun top (Detail,XView,YView,ZView) Scene =
              FindImpacts (GenerateRays (Detail,XView,YView,ZView)) Scene;
  (* Toplevel function *)
```

The top level function is `top`. Rays are traced from the viewpoint given by the `XView, YView,ZView` coordinates towards a `Scene` represented by a list of polygons. The number

[1] A more detailed program description is given in appendix C.5

of rays is controlled by the integer parameter `Detail`, causing `Detail * Detail` rays to be generated. Each ray is tested for impact against the scene objects, and the result of the program is a list of impact information for each ray. If a ray strikes one or more objects then the impact closest to the viewpoint is returned, represented by the constructor `IMPACT`. Otherwise, the constructor `NOIMPACT` is returned. The `IMPACT` constructor carries information about the distance to the object, and the object's identifier which is an integer value. In a more sophisticated version of the program `IMPACT` could have contained other data, for example colour and intensity information.

8.2.2 Compilation

Feeding the program to SkelML, it identifies the `map` in `FindImpacts` as a potential source of farm parallelism, and produces a process network as shown in figure 8.1. We intend to

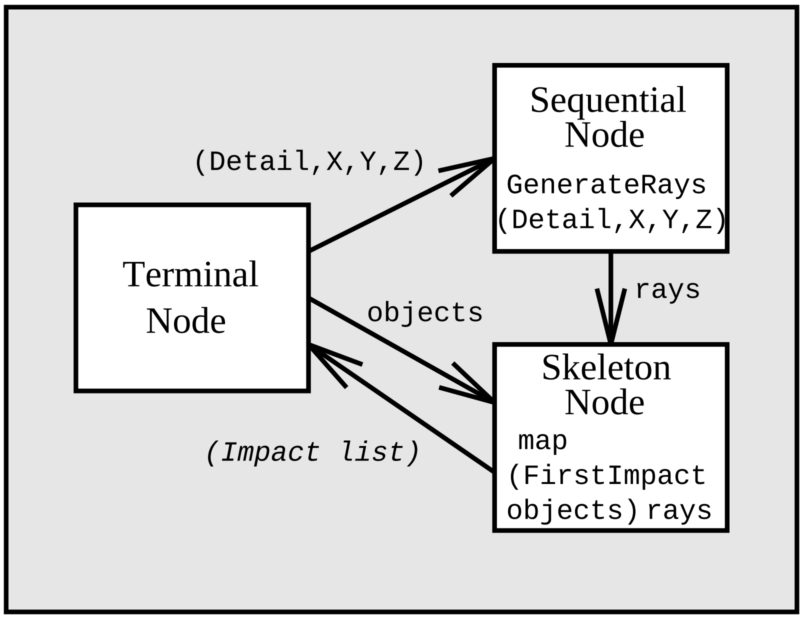


Figure 8.1: Ray tracer: process network

run this program on two different scenes: scene 1 contains six quadrilaterals, while scene 2 contains eight hexagons, three quadrilaterals and one octagon. For each scene two values of `Detail` are tried: 12, giving 144 rays, and 25, giving 625 rays. For profiling we feed SkelML with a detail of 4 (16 rays), and scale the skeleton input by a factor of 9 (144 rays) and 39 (624 rays, an approximation for 625 rays) respectively. We illustrate the use of profiling information with the predicted performance for the `map` farm for `Detail` 12 on scene 1.

```
*** Performance model for map: *********
Type in : ((real * real * real) * (real * real * real))
Type out: Impact
Input length: 144
Tf: 3338.64 Tk: 118.667 Tp: 71.0625
Optimal number of workers: 31 (max 47)
Sequential time: 480764 microseconds
  No W   ParTime  Speedup  Efficiency
     1   494425   0.97237  0.97237
     2   254043   1.89245  0.946227
     3   172773   2.78264  0.927546
     4   128752   3.73404  0.933511
   (...)
    12   47481.7 10.1252   0.843771
    13   47481.7 10.1252   0.778865
    14   44095.5 10.9028   0.778772
   (...)
    30   23778   20.2189   0.673962
*   31   23778   20.2189   0.652222
    32   23080   20.8303   0.650948
   (...)
    40   19935.2 24.1163   0.602907
    41   19775.1 24.3116   0.592965
   (...)
****************************************
```

For any number of workers greater than 1, the parallel implementation is faster than the sequential, hence the parallelism in `map` is worth exploiting as a processor farm. The corresponding mapping as produced by SkelML is shown in figure 8.2. Note that in the

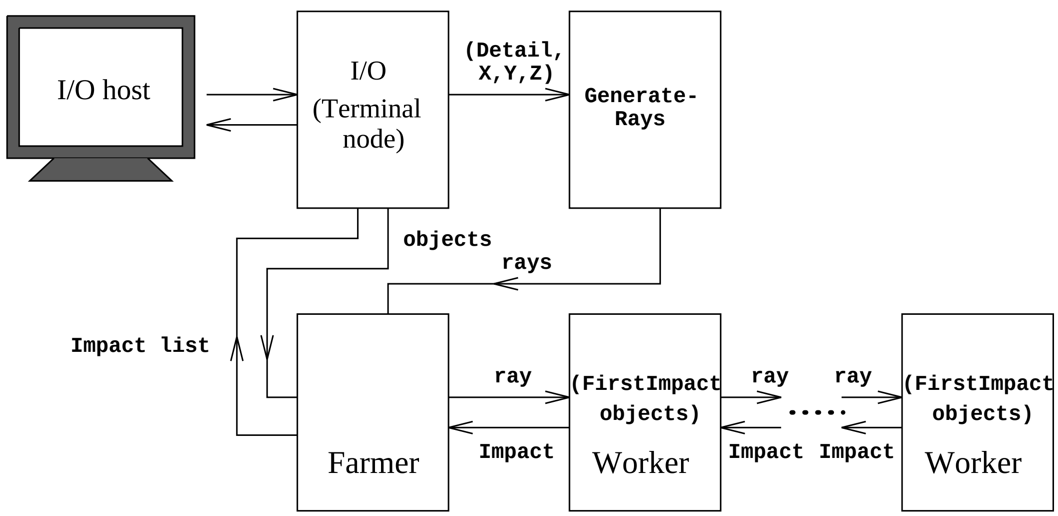


Figure 8.2: Ray tracer: processor mapping

Processors	Scene 1, Detail 12				Scene 1, Detail 25			
Workers/Total	Est.	Meas.	% Err	Sp.up	Est.	Meas.	%Err	Sp.up
(seq)	-	632	-	1.0	-	2751	-	1.0
1/4	532	616	15.8	1.0	2303	2635	14.4	1.0
2/5	292	340	16.4	1.9	1251	1431	14.4	1.9
3/6	211	247	17.1	2.6	896	1031	15.1	2.7
4/7	167	202	21.0	3.1	720	830	15.3	3.3
5/8	143	175	22.3	3.6	611	710	16.2	3.9
6/9	126	157	24.6	4.0	540	629	16.5	4.4
8/11	106	134	26.4	4.7	452	529	17.0	5.2
10/13	96	120	25.0	5.3	401	470	17.2	5.9
12/15	85	112	31.8	5.6	364	430	18.1	6.4
16/19	75	100	33.3	6.3	317	380	19.9	7.2
20/23	72	94	30.1	6.7	293	351	19.8	7.8

Table 8.1: Ray tracer: times (in milliseconds) and speedup for scene 1

Processors	Scene 2, Detail 12				Scene 2, Detail 25			
Workers/Total	Est.	Meas.	% Err	Sp.up	Est.	Meas.	%Err	Sp.up
(seq)	-	1947	-	1.0	-	8528	-	1.0
1/4	1602	1845	15.2	1.1	6940	8034	15.8	1.1
2/5	828	955	15.3	2.0	3573	4134	15.7	2.1
3/6	569	662	16.3	2.9	2450	2832	15.6	3.0
4/7	439	514	17.1	3.8	1888	2185	15.7	3.9
5/8	364	424	16.5	4.6	1553	1793	15.5	4.8
6/9	309	364	17.8	5.4	1316	1535	16.6	5.6
8/11	245	292	19.2	6.7	1035	1208	16.7	7.1
10/13	212	253	19.3	7.7	873	1015	16.3	8.4
12/15	180	226	25.6	8.6	754	884	17.2	9.6
16/19	147	207	40.8	9.4	614	723	17.8	11.8
20/23	137	203	48.2	9.6	538	626	16.4	13.6

Table 8.2: Ray tracer: times (in milliseconds) and speedup for scene 2

function to be computed in each farm worker, (`FirstImpact objects`), `objects` is free, and must thus be broadcasted to all the workers prior to farming out the rays. In the occam2 code produced by SkelML this is automatically handled.

8.2.3 Results

The program is compiled and run for scene 1 and scene 2, each with `Detail` 12 and 25, and using from 1 to 20 farm workers. The results including estimated and measured run times and speedup are summarised in table 8.1 and 8.2 for scene 1 and scene 2 respectively. The number of processors is given in terms of the number of workers in the farm. The total number of processors used is the number of workers plus three: the farmer processor, the processor evaluating `GenerateRays`, and the processor running the terminal node. Using s2o directly on the source program a sequential occam2 version of the program can be created. This program running sequentially on the terminal node processor is used for reference when calculating speedups.

The absolute performance of the program as given in the "Measured" columns indicate speedups in the range of 6.7-13.6 when using 20 worker processors. It should be noted that even when using only one worker processor the parallel version of the program is still marginally faster than the sequential. While we cannot pinpoint an exact reason for this, we suggest that since a total of four processors is actually used, the reduced load on the memory of each processor together with a fast skeletal implementation of `map` account for the difference.

In [30] we presented results running the same application on an earlier version of the compiler. There have been a number of modifications and improvements to SkelML since then, and the current implementation produces code that for this example runs around 5-15% faster. There are also minor differences ($< 5\%$) in the timings for the sequential version of the program, which we attribute to modifications in the memory management on the terminal node processor.

The estimated performance figures are based on the collected profiling information as outlined in chapter 5. Due to the scaling up of the input data to the farm performance model, we have manually scaled up the profiled times for `GenerateRays` and for inter processor communication correspondingly. The error between estimated times and measured times are calculated in relation to the estimates. With the exception of scene 1, detail 12, the error is mostly less than 20%. The errors tend to increase when the number of processors increases, although the time difference in absolute terms decreases. Figure 8.3 illustrates the differences graphically, all times rounded to the nearest 100 ms. From this we see that although there are differences between estimated and measured times,

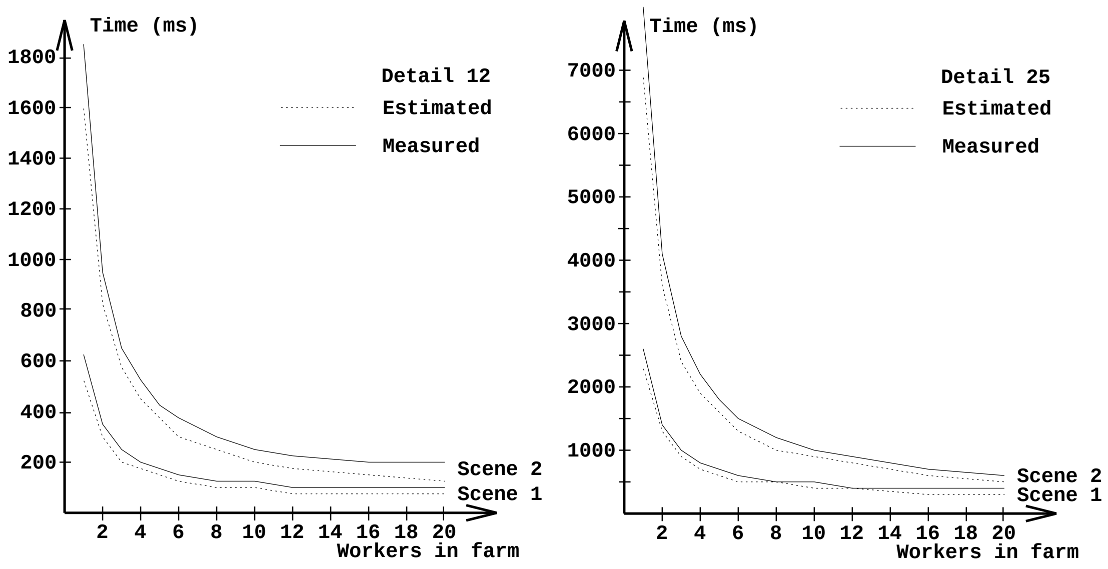


Figure 8.3: Ray tracer: estimated vs measured times

the graphical profiles of the estimates correspond closely to that of the measured times.

8.2.4 Discussion

Compared to the same application compiled with PUFF [40] our results are of the same order. This should come as no surprise since, although the two compilers are completely independent developments, they both use s2o for producing sequential code. A more interesting comparison would be against a radically different compiler. In [106] Kesseler quotes a result for the same application in explicitly parallel Concurrent Clean and running on similar hardware. There are a number of differences between the Concurrent Clean compiler and SkelML which make a direct comparison difficult. Concurrent Clean, in contrast to SkelML, supports garbage collection, and there is a 64-bit representation of floating point numbers. Our 32-bit representation enforces a coarser check on whether or not a ray strikes an object, which may again affect the amount of computation. Concurrent Clean is a significantly more mature and sophisticated development, featuring state of the art optimisation techniques and producing transputer assembler code.

Concurrent Clean evaluates scene 2, detail 300 using 16 processors in 33.0 seconds. Using 16 worker processors, SkelML runs out of heap space at around detail 90. Table 8.3 shows

Scene 2, 16 workers, time per ray (SkelML)			
Detail	No. rays	Time (ms)	Time per ray (μs)
25	625	723	1157
30	900	1036	1151
40	1600	1829	1143
60	3600	4102	1139
80	6400	7280	1138

Table 8.3: Ray tracer: time per ray for scene 2

the run times for the code produced by SkelML for detail 25-80. The average time taken to process a ray seems to be relatively constant, and assuming this scales further, we would expect a total time of around 102-104 seconds at detail 300. This comparison is highly speculative, and provides no strong basis for passing a well funded judgement on SkelML's performance. We believe however that these results indicate that the SkelML implementation performs worse than Concurrent Clean, but possibly not by more than a small factor.

As far as the accuracy of the performance predictions is concerned, the ray tracer example demonstrates that the worst case error can be high, almost 50% for 20 workers, scene 1, detail 1. An error of 20% or below is more common. It is encouraging to note that the errors mostly follow a consistent pattern, which is particularly clear in scene 2, detail 25 with all errors within the 15.5%-17.8% range. The estimates are also consistently lower than the measured results. This leads us to conclude that the performance models are fundamentally sound, but that some factors or constants are over- or under-estimated. Given the consistency of the errors it seems plausible that a properly adjusted performance model could yield estimates with considerably lower errors.

8.3 A Best Match Problem

8.3.1 Problem Description

In our second example, we consider a fixed set of known objects, *candidates*, and a set of other objects, *problems*. The aim is to identify the candidate object, which according to some matching criterion, best matches the problem objects. This is a general problem description which has instances in many areas. For example, the candidate objects may be some form of basic building blocks to be matched against more complex composite problem objects. With a suitable matching function the program could be used to find the building block most commonly occurring as a component in the composed objects.

In our example both the candidate and problem objects are 2D polygons represented as lists of coordinates. A candidate and a problem object match if one can be obtained from the other after suitable scaling and rotation. We want to identify the candidate object which matches the most problem objects, and we want to know how many problem objects it matches. The problem objects could for example be a stream of processed images from a camera, while the candidate objects might be a set of model descriptions.

In [52] and [53] Cole derives highly parallel programs from problem specifications in the context of BMF and list-homomorphisms. The method used is to devise a problem encoding such that the resulting program contains a list-homomorphism function with implicit parallelism. A similar approach is applicable for our problem, although the derivation of the highly parallel program is relatively straightforward.

Assume $+\!\!+$ is the infix list concatenation operator, and $\cdot$ is the infix function composition operator. A function h defined on finite lists is a list-homomorphism if there exists an associative operator $\oplus$ with identity element e such that

$$h\ [\,] = e$$

and

$$h\ (x +\!\!+\ y) = h\ x \oplus h\ y \tag{8.1}$$

for lists x and y. The relevance of this to parallel programming is seen from the Homomorphism Lemma [23] which states that a function h is a homomorphism with respect to $+\!\!+$ if and only if

$$h = (\oplus\ /)\ \cdot\ (f\ *) \tag{8.2}$$

for some associative operator $\oplus$ and function f, where $(\oplus\ /)$ means "fold with $\oplus$" and $(f\,*)$ means "map with f". Since the higher order functions `map` and `fold` both contain large amounts of implicit parallelism[2], any list-homomorphism has a potentially highly parallel implementation.

The top level function of our program, named `bestmatch`, should be defined as

```
fun bestmatch candidates problems = ...;
```

and have the type

<list of candidate objects> → *<list of problem objects>* →

(*<candidate object>* * *<number of successful matches>*)

`bestmatch` is not a list-homomorphism because no binary function $\oplus$ can be devised such that (8.1) holds with h as `(bestmatch candidates)`. This follows from the fact that the result of `bestmatch` contains information only about the best matching object, which is insufficient because the best matching object of two concatenated lists may not necessarily be the best matching of either. Our task is thus to find a way of expressing the problem using a `map` and a `fold` with suitable argument functions to produce a potentially highly parallel program.

We split the solving of the problem into two phases: first to generate matching information about each candidate object, and then to retrieve the candidate with the highest frequency of successful matches. Given an intermediate representation of successful matches for each candidate, a function `findmax` can be defined to retrieve the object with the highest frequency of matches. The generation of the matching information can be expressed as a composition of a `map` and a `fold` by first mapping a matching function with the candidate objects over the problem objects, and then fold the results to give the accumulated number of matches for each candidate object.

For recording accumulated matching information we need an intermediate representation of successful matches for the candidate objects. A straightforward representation of frequencies is a list of integers, where each integer holds the number of successful matches

[2] This is true for `fold` only if its argument function is associative. The current SkelML implementation also requires commutativity.

for the corresponding candidate object. With candidate objects

```
[c1,c2,c3,c4,c5]
```

the frequency list

```
[12,6,0,43,8]
```

would mean 12 successful matches with `c1`, 6 with `c2`, none with `c3` etc.

Assume a boolean function `match` which takes two objects as arguments and returns `true` if they match. This can easily be used with a calling function `matchtop` to return a frequency list for matches between the candidates and one problem object. `matchtop` partially applied to the candidates list will be the argument function to `map`. The argument function to `fold` is an associative and commutative function `combfreq` which combines two frequency lists into one.

With respect to (8.2) we have now succeeded in finding suitable functions for $\oplus$ and f, `combfreq` and `matchtop` respectively, and the program can be specified as follows[3]:

```
fun match objC objP = ... ;
   (* true if objP matches objC, rotated and/or scaled *)

fun matchtop [] objP = [] |
    matchtop (objC::T) objP =
       if (match objC objP) then [1] else 0::(matchtop T objP);
   (* Returns frequency list [0,,,1] (1 in pos i : obj#i matches) *)

fun combfreq [] L = L |
    combfreq L [] = L |
    combfreq (h1::t1) ((h2:int)::t2) = (h1+h2)::(combfreq t1 t2);
   (* zips two frequency lists with + *)

fun findmax candidates freqlist = ... ;
   (* picks best candidate from a list of frequencies *)

fun bestmatch candidates problems =
   findmax candidates (fold combfreq []
                         (map (matchtop candidates) problems));
   (* toplevel function *)
```

[3]A more detailed program description can be found in appendix C.6

8.3.2 Compilation

Feeding the source program to SkelML it is parsed and type checked. Next we execute the program applied to sample test data with the SkelML interpreter in profiling mode. We use as input a list of 9 candidate objects and a list of 10 problem objects. This results

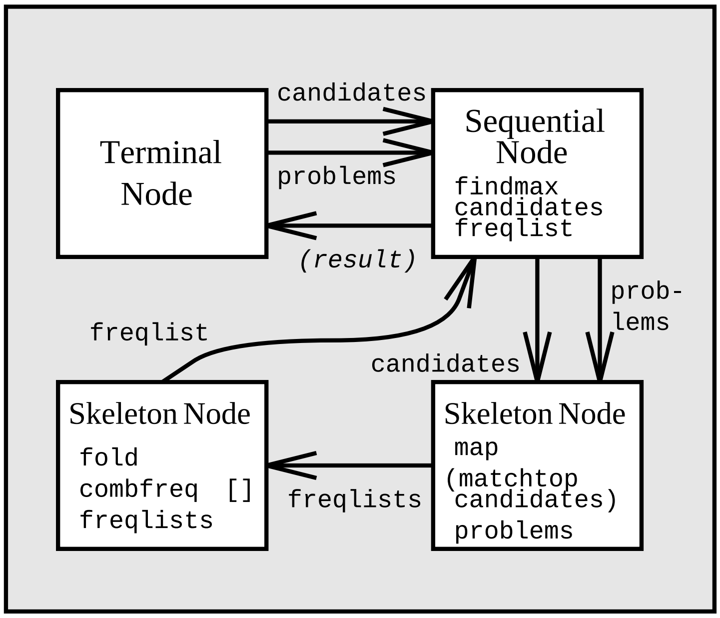


Figure 8.4: Best match: initial process network

in the program being instrumented with sizes of data structures and costs of estimated sequential occam2 execution times. The compiler then analyses the program, identifies the topmost function `bestmatch` and generates a process network as depicted in figure 8.4. The HOFs `map` and `fold` are represented as skeleton nodes. The compiler also registers that these two HOFs can execute in parallel in a pipeline. We give SkelML a scaling factor of 50 for both skeletons, i.e. the performance models assume that the input list of problem objects is 50 times longer. SkelML calculates the following performance estimates for the two skeleton nodes:

```
*** Performance model for map: *********
Type in : (real * real) list
Type out: int list
Input length: 500
Tf: 6678.5 Tk: 522.32 Tp: 361
Optimal number of workers: 12 (max 18)
Sequential time: 3.33925e+06 microseconds
  No W    ParTime   Speedup  Efficiency
     1  3.56008e+06 0.937971 0.937971
     2  1.84942e+06 1.80557  0.902785
     3  1.26803e+06 2.63341  0.877804
```

```
   (...)
    11   419894   7.95261   0.722965
*   12   392534   8.5069   0.708908
    13   366366   9.11453   0.701118
   (...)
    17   321330    10.392    0.611292
>   18   321217    10.3956   0.577534
*:optimal no w, >:max no w
****************************************

*** Performance model for fold: ********
Type in : int list
Type out: int list
Input length: 500
Tf: 268.366 Tek: 475.4 Tep: 315
Optimal number of workers: 1 (max 1)
Sequential time: 134183 microseconds
  No W   ParTime  Speedup  Efficiency
*>   1   371883   0.36082  0.36082
*:optimal no w, >:max no w
****************************************
```

Note that `fold` on its own is found not to contain useful parallelism. A 1-processor skeletal implementation may still be preferable to a sequential implementation as a pipeline component, the performance of the pipeline as a whole considered. At this point however, no attention is being paid to the usefulness of the parallelism exposed. The compiler notes that a transformation is applicable, from the composition of the two HOFs to another HOF: `foldmap`. The relevant transformation rule, listed as T4 in appendix B is:

```
trans {f,g,a,l} fold f a (map g l) ==
                foldmap f a g l;
```

`foldmap` is a version of `fold` in which an additional argument function is mapped over all elements of the input list prior to folding. The corresponding parallel implementation is generally preferable to that of the combined `map` and `fold` because there is less inter processor communication involved. The `foldmap` skeleton can be seen as an optimised implementation of list-homomorphisms. A performance model of `foldmap` is instantiated with information extracted from the other two performance models:

```
*** Performance model for foldmap: ****
Type in : (real * real) list
Type out: int list
Input length: 500
Tf: 6946.87 Tek: 307.9 Tep: 205.5
Optimal number of workers: 23 (max 34)
```

```
Sequential time: 3.47343e+06 microseconds
  No W  ParTime     Speedup  Efficiency
     1  3.62738e+06 0.957559 0.957559
     2  1.87896e+06 1.84859  0.924297
     3  1.29855e+06 2.67485  0.891615
   (...)
    22   304578    11.4041   0.518367
*   23   297324    11.6823   0.507928
    24   290069    11.9745   0.498938
   (...)
    32   253795    13.686    0.427687
    33   261025    13.3069   0.403239
>   34   253770    13.6873   0.402568
*:optimal no w, >:max no w
***************************************
```

SkelML sees that the best completion time for `foldmap` is better than that of the `map-fold` pipeline, and decides to apply the transformation. The resulting process network is depicted in figure 8.5. In the following section we present results showing the execution

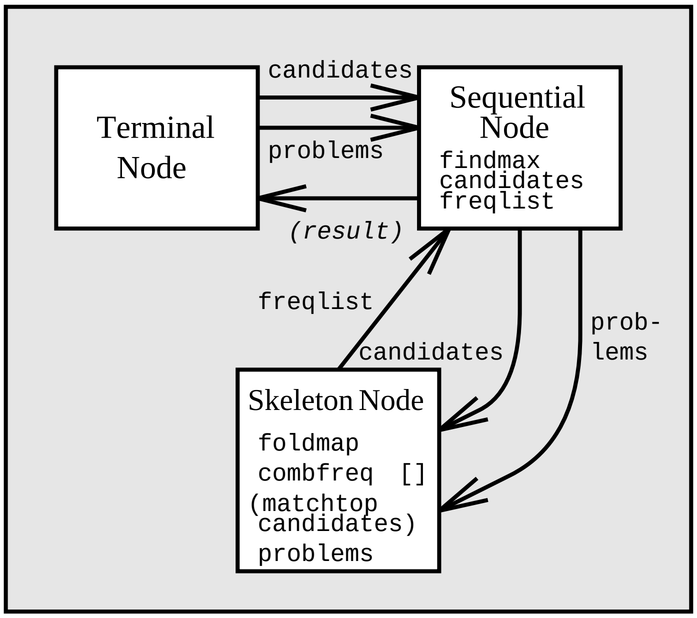


Figure 8.5: Best match: transformed process network

times of the program when compiled to execute on a range of processors. The mapping is carried out in accordance with the number of processors available, and the resulting processor topology is depicted in figure 8.6. Finally SkelML produces occam2 code for all processors, calling s2o for sequential translation where applicable. Wiring and process placement tables are also generated.

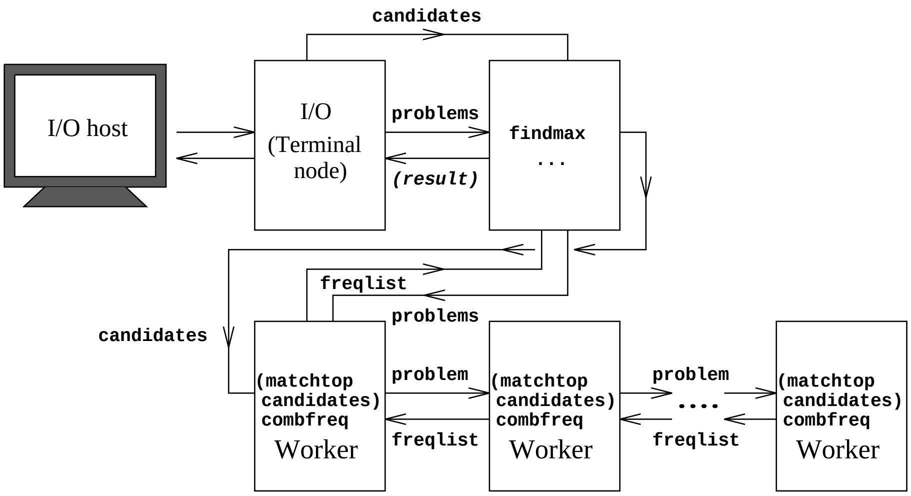


Figure 8.6: Best match: processor mapping

8.3.3 Results

The resulting occam2 program was configured using from 2 to 22 worker processors in the `foldmap` skeleton, and executed with three different data sets. All data sets have the same candidate objects, but have a different number of problem objects; 200, 500 and 1000. 500 objects should correspond to the scaled profiling data, while the other input sizes were tried to illustrate how the program behaves when input lengths deviate from profiling data. Table 8.4 shows the results for 500 objects. The number of processors is given in terms of the number of processors in the `foldmap` skeleton. The total number of processors is always 2 higher, as one processor computes `findmax`, while another runs the terminal node. The sequential run time is produced by applying s2o directly to the SML source code, and running the resulting occam2 code on the terminal node processor. A speedup of 8.9 is obtained using a total of 24 processors, although the last two worker processors make only a marginal contribution towards performance improvement. The estimated times are calculated from the profiling data and the performance estimates for `foldmap` showed previously. Since the list of problem objects is 50 times larger for the real data than for the profiling data, SkelML's estimated communication time for the problem objects should be scaled up by a factor of 50. The estimated figures in table 8.4 are the

Processors	Input length: 500 objects			
foldmap/Total	Est.	Meas.	% Err	Speedup
(seq)	-	3601	-	1.0
2/4	2032	2026	0.3	1.8
4/6	1161	1121	3.4	3.2
6/8	878	826	5.9	4.4
8/10	726	670	7.7	5.4
10/12	653	588	10.0	6.1
12/14	588	528	10.2	6.8
14/16	544	489	10.1	7.4
16/18	515	459	10.9	7.9
18/20	494	434	12.1	8.3
20/22	479	411	14.2	8.8
22/24	457	403	11.8	8.9

Table 8.4: Best match: times (in milliseconds), 500 objects

result of SkelML's prediction's with the extra communication cost manually calculated and added. As can be seen, the error of the performance estimates increases with the number of processors, but is typically around 10%. Figure 8.7 illustrates graphically how the estimated and measured times correspond. The overall profiles of the time graphs are

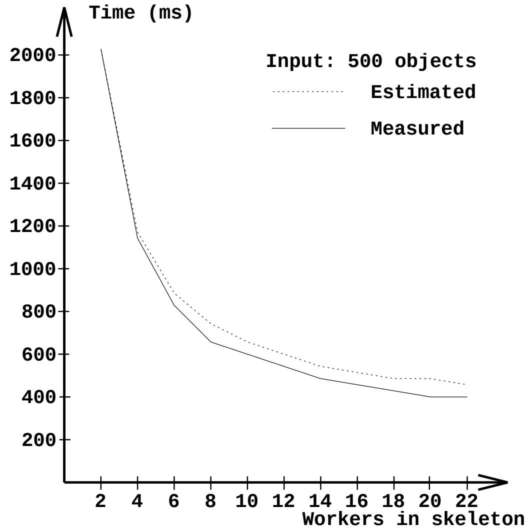


Figure 8.7: Best match: estimated vs measured times, 500 objects

as for the ray tracer very similar, and clearly reflect the increasing error as the number of processors grows.

Table 8.5 summarises the results of applying the same occam2 program to input data of lengths 200 and 1000. By this we would like to see how a change in input length affects the performance of the program. A graphical representation of the results with respect

Processors	Input: 200 objects		Input: 1000 objects	
foldmap/Total	Time	Sp.up	Time	Sp.up
(seq)	1438	1.0	7211	1.0
2/4	816	1.8	4036	1.8
4/6	460	3.1	2220	3.2
6/8	341	4.2	1616	4.5
8/10	288	5.0	1320	5.5
10/12	250	5.7	1136	6.4
12/14	234	6.1	1023	7.0
14/16	218	6.6	934	7.7
16/18	209	6.9	874	8.3
18/20	200	7.2	828	8.7
20/22	190	7.5	789	9.1
22/24	198	7.3	760	9.5

Table 8.5: Best match: times (in milliseconds), 200 and 1000 objects

to speedup is presented in figure 8.8. The speedup figures of the model are calculated

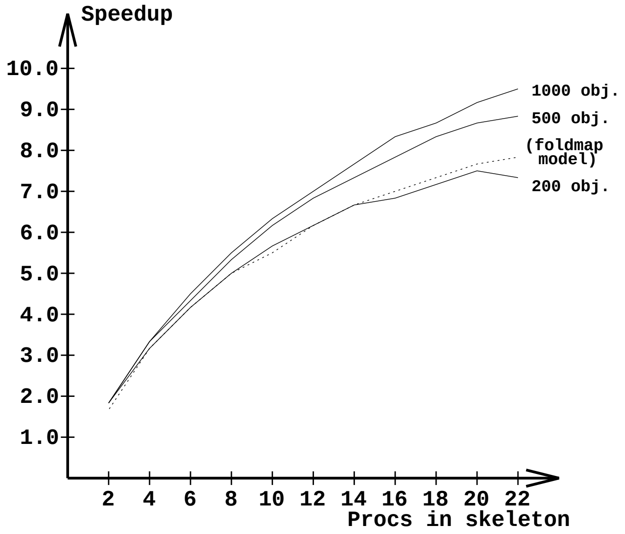


Figure 8.8: Best match: speedup graphs for model and 200 - 1000 objects

with respect to the *measured* run time for the sequential program, input length 500. For the data set of 1000 problem objects we observe a slightly better relative performance than when using 500 objects. As the number of input problem objects increases, the workload may be split more evenly among the skeleton processors, and also the relative cost associated with the startup and shutdown of the skeleton decreases in significance. A similar explanation can be made for the relatively lower performance using 200 objects. This considered, the program seems to cope satisfactorily with the change in input data lengths.

8.3.4 Discussion

Exploiting list-homomorphism parallelism in BMF-style was a convenient way of expressing the SML program. We mapped a matching function with the candidate objects over the problem objects, and then folded the results to give the accumulated number of matches for each candidate object. We could however just as well have chosen to map a matching function with the problem objects over the list of candidate objects, which would have resulted in a different parallel implementation. We are here entering a very difficult area of fundamental interest in the use of skeletons for parallel processing. If very early decisions about how to express a program in terms of skeletons have a crucial impact on performance, this means that it will we much harder than we had anticipated to free the programmer from making decisions based on useful parallelism and implementation technicalities.

The accuracy of the performance predictions were better for this program than for the ray tracer. However, while for the latter the model's estimates were consistently lower than what was measured, here they were consistently higher. As the compiler's decision to apply transformations is based solely on the performance estimates, it seems paramount that they are as accurate as possible. However, when deciding if a transformation rule is to be applied what is important is the *relative* performance of the two implementations. Thus if the model estimates both implementations with a similar error, transformation decisions are still correct.

8.4 Area of Symmetric Objects

8.4.1 Problem Description

In our third and final example we consider another example of processing 2D polygons, this time with emphasis on the transformation system. Given a set of 2D polygons we would like to know the area of those polygons that are symmetric. By a symmetric polygon we mean that its sides are of the same length and that all angles between adjacent sides are the same. The specification of this problem is similar to that of many other algorithms: first a set of objects is processed and narrowed down according to some discriminating criterion, and then a property is calculated for each remaining object. The SML implementation is based on a program written by Greg Michaelson. The structure of the program is[4]:

```
fun lengths obj = ...;
   (* computes the lengths of all sides *)
fun samesides objsides = ...;
   (* true if all sides are of the same length *)
fun samecosines objsides = ...;
   (* true if all cosines are the same *)
fun area objsides = ...;
   (* returns area of object *)
fun symarea objlist =  map area
                      (filter samesides
                      (filter samecosines
                      (map lengths objlist)));
   (* equiv. to  (map area) o (filter samesides) o
                 (filter samecosines) o (map lengths) *)
```

A polygon object is initially represented as a list (`(real * real) list`) of coordinates. For `samesides`, `samecosines` and `area`, the object representation also contains an additional tuple of the length of each side and its square, i.e. an object is represented as a `(real * real * (real * real)) list`.

8.4.2 Compilation

SkelML identifies the four skeleton HOFs, and produces a process network as depicted in figure 8.9. Since the four skeleton nodes can execute as a pipeline, transformations can be considered. As it turns out, there are 14 different implementations of the pipeline to choose from, depending on the transformation rules applied. Table 8.6 summarises the 14 variations labeled P1-P14. The structures of the pipelines are shown, and the transformation steps from P1 to each implementation is given. The transformations are referred to by their names T1-T6 as listed in appendix B. For brevity in the SML expressions we use `o` to represent the function composition operator, and the function names are abbreviated as follows:

[4] A more detailed program description is found in appendix C.7

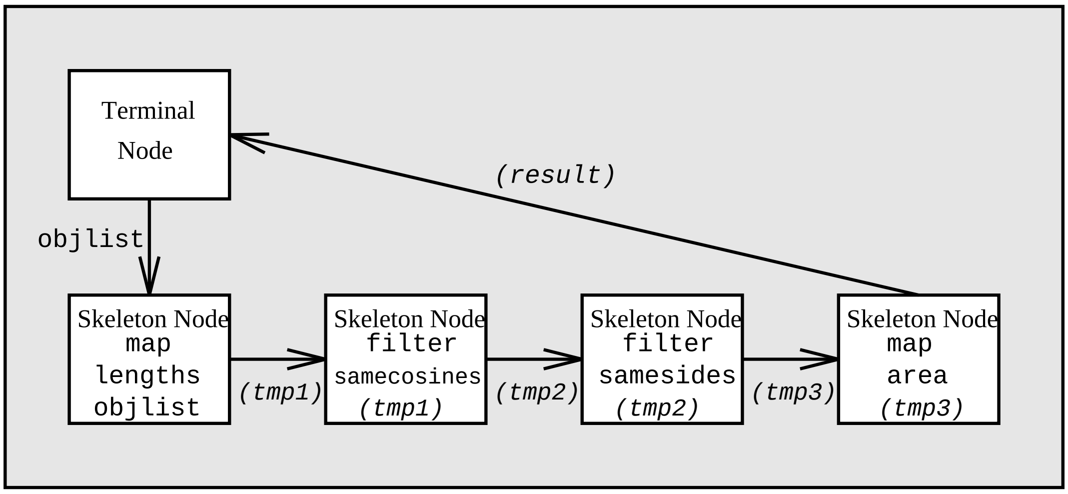


Figure 8.9: Symarea: initial process network

```
l   = lengths
ss  = samesides
sc  = samecosines
a   = area
m   = map
f   = filter
fm  = filtermap
mf  = mapfilter
p&q = fn x => p x andalso q x
```

In deciding which transformations to apply, and which implementation to choose, SkelML uses the performance models of each skeleton to calculate the expected best completion time for each alternative pipeline. For profiling we use an input list of 20 12-sided polygons and a scaling factor of 10. Figure 8.10 shows the transformation tree searched by the compiler. For brevity we use a shorthand notation where `f`, `m`, `fm` and `mf` represent skeletons, and `.` is the function composition operator. Transformations are represented with their internal number `T1-T6` and a brief description. Each implementation alternative is supplied with the estimated completion time (in milliseconds) for the implementation of the corresponding pipeline as predicted by the compiler. Note that the implementations P1-P7 are the same as P8-P14 respectively except that the two filters are swapped by first applying transformation T6. From this information SkelML identifies P7 as the best implementation, and restructures the process network as depicted in figure 8.11, while figure 8.12 shows the corresponding mapping using a total of 9 processors. A further

Possible pipeline implementations		
Label	SML expression	Transf. from P1
P1	`(m a) o (f ss) o (f sc) o (m l)`	
P2	`(mf a ss) o (f sc) o (m l)`	T2
P3	`(mf a ss) o (fm sc l)`	T2-T3 or T3-T2
P4	`(m a) o (f ss) o (fm sc l)`	T3
P5	`(m a) o (f sc&ss) o (m l)`	T5
P6	`(mf a sc&ss) o (m l)`	T5-T2
P7	`(m a) o (fm sc&ss l)`	T5-T3
P8	`(m a) o (f sc) o (f ss) o (m l)`	T6
P9	`(mf a sc) o (f ss) o (m l)`	T6-T2
P10	`(mf a sc) o (fm ss l)`	T6-T2-T3 or T6-T3-T2
P11	`(m a) o (f sc) o (fm ss l)`	T6-T3
P12	`(m a) o (f ss&sc) o (m l)`	T6-T5
P13	`(mf a ss&sc) o (m l)`	T6-T5-T2
P14	`(m a) o (fm ss&sc l)`	T6-T5-T3

Table 8.6: Symarea: possible pipeline implementations

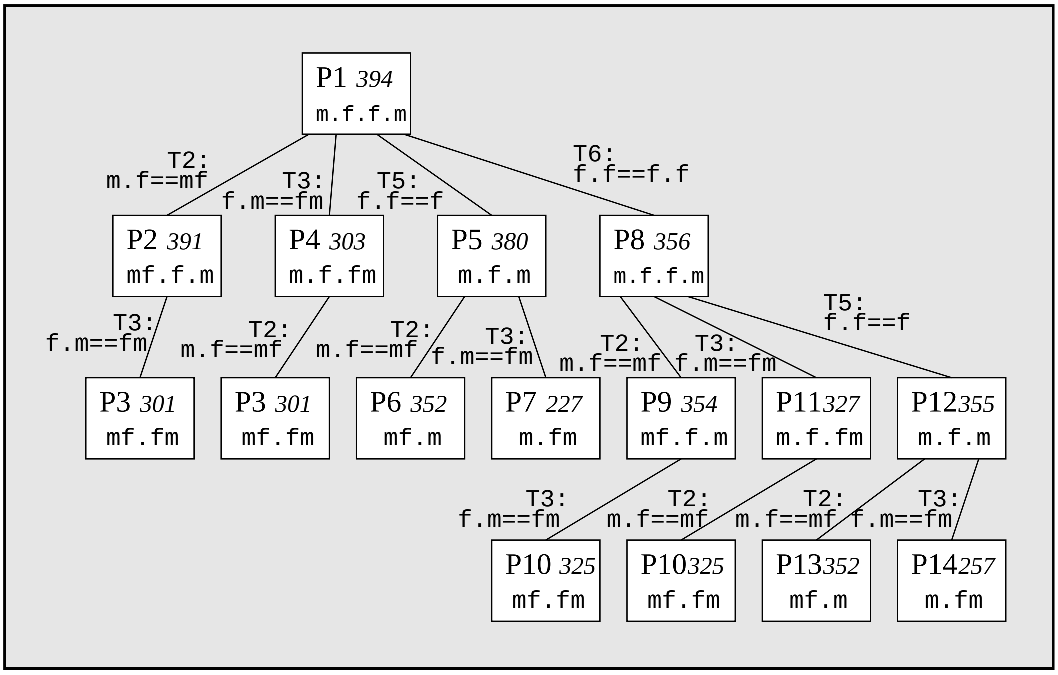


Figure 8.10: Symarea: tree of transformations

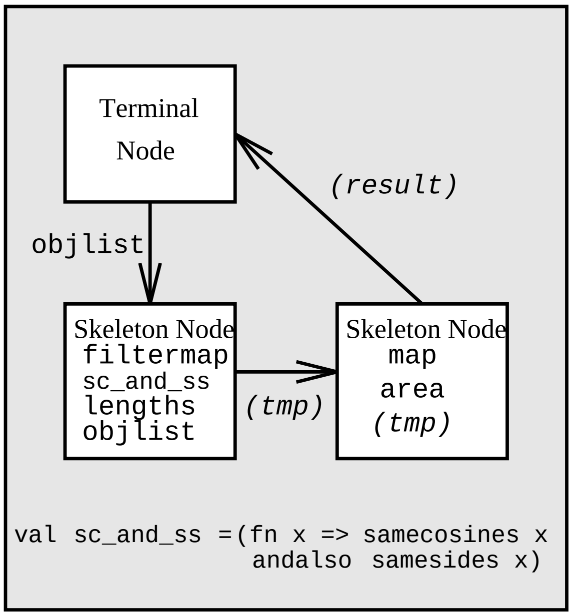


Figure 8.11: Symarea: transformed process network, pipeline P7

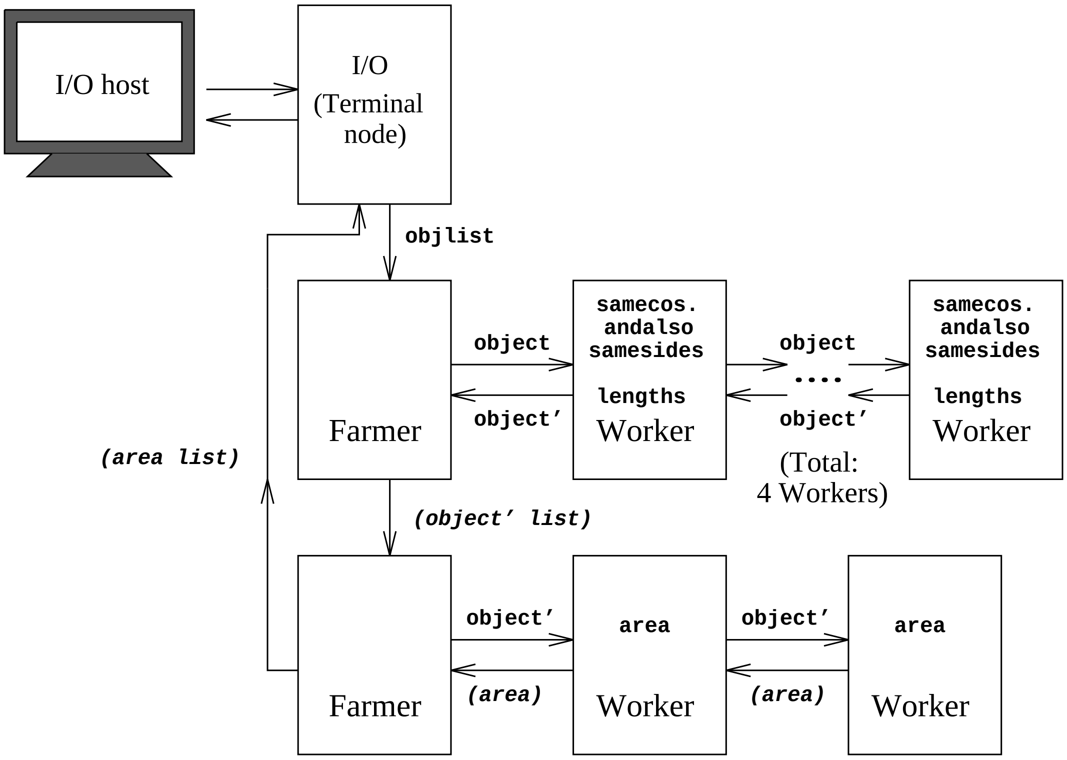


Figure 8.12: Symarea: processor mapping using pipeline P7

Performance					
Prog	No Procs	Estimated	Measured	% Error	Speedup
seq (20 objs)	1	77594	86848	11.9	1.0
seq (200 objs)	1	-	865408	-	1.0
P1	7	394386	426496	8.1	2.0
P3	8	301463	277696	7.9	3.1
P7	9	227542	215040	5.5	4.0
P12	6	355616	387712	9.0	2.2

Table 8.7: Symarea: times (in microseconds)

discussion of the accuracy and correctness of the performance estimates is found in the discussion section following the presentation of the results.

8.4.3 Results

In addition to P7, three other pipeline alternatives were implemented: P1, P3 and P12. A sequential version of the program was also produced. Table 8.7 summarises the execution times of these implementations, with 200 objects being used as input to the parallel programs. The number of processors used includes all farmers, workers and the terminal node. As can be seen the P1 implementation is about twice as fast as the sequential, while the P7 implementation offers yet another doubling of execution speed. The errors seem stable and relatively small, but a closer inspection reveals that for P1, P12 and the sequential implementation the estimations are lower than the measured times, while for P3 and P7 they are in fact higher.

8.4.4 Discussion

The success of the transformation system in finding the fastest implementation for a pipeline depends to a great extent on the quality of the performance predictions. Studying the estimated times shown in figure 8.10, the application of transformation rule T6, swapping the two `filter`'s, produces seemingly puzzling effects. Intuitively, we would expect this transformation to be either always beneficial or never beneficial, but the

Comparing P2 and P9 stage by stage								
	`mapfilter`		`filter`		`map`		Total	
Pipe	Time	Procs	Time	Procs	Time	Procs	Time	Procs
P2	300	1	387	1	350	3	391	5
P9	205	3	219	1	350	3	354	7

Table 8.8: Symarea: P2 vs P9 stage by stage

Comparing P3 and P10 stage by stage						
	`mapfilter`		`filtermap`		Total	
Pipe	Time	Procs	Time	Procs	Time	Procs
P3	178	3	299	4	301	7
P10	205	3	322	3	325	6

Table 8.9: Symarea: P3 vs P10 stage by stage

estimated times suggest a more nuanced picture. We examine the implementations P1-P7 versus P8-P14, which are pairwise identical, i.e. P1 corresponds to P8, P2 to P9 etc, with the exception that for P8-P14 transformation rule T6 is applied. Comparing P2 (391 ms) and P9 (354 ms) it seems that having applied transformation T6 swapping the two filters increases performance. However P3 (301 ms) and P10 (325 ms) indicates the contrary to be true. A similar effect is observed when comparing P5 (380 ms) and P12 (355 ms) against P7 (227 ms) and P14 (257 ms).

Tables 8.8 and 8.9 compares P2/P9 and P3/P10 looking deeper into the underlying calculations. The tables show the expected completion times and the number of processors for each stage of the pipelines, as well as the expected completion time and processors used for the pipeline as a whole. Note that a 1-processor implementation means an essentially sequential implementation, while an n-processor implementation, with n $\geq$ 3, means a farm implementation with 1 farmer and (n-1) workers. The reason why P9 is faster than P2 is that the `filter` stage of P2 acts as a bottleneck, while the swapping of the `filter`s in P9 gives a better balance of the load in the pipeline. The P3/P10 implementations are derived from P2/P9 respectively by transforming the `filter` and the `map` stages of P2/P9 into a single `filtermap` stage. Applying this transformation is favourable both for P3 and P10. The `filter` stage of P2 does however have a heavier work load than the corresponding stage of P9, which results in P3's performance model for `filtermap`

Comparing P5 and P12 stage by stage								
	`map`		`filter`		`map`		Total	
Pipe	Time	Procs	Time	Procs	Time	Procs	Time	Procs
P5	241	1	374	1	350	3	380	5
P12	241	1	292	1	350	3	355	5

Table 8.10: Symarea: P5 vs P12 stage by stage

Comparing P7 and P14 stage by stage						
	`map`		`filtermap`		Total	
Pipe	Time	Procs	Time	Procs	Time	Procs
P7	137	3	223	5	227	8
P14	241	1	253	4	257	5

Table 8.11: Symarea: P7 vs P14 stage by stage

suggesting 3 workers to be used in the farm vs 2 workers for P10. Thus, the performance models for P3 and P10's `filtermap` stages favour P3.

Tables 8.10 and 8.11 shows a similar scenario for P5/P12 vs P7/P14. P5's `filter` stage is a bottleneck compared to the faster swapped `filter`'s of P12. Transforming the `filter` and `map` stages into a single `filtermap` stage is again favourable. However, the heavier computational task of P5's `filter` allows 4 workers to be used in P7's `filtermap` stage, while only 3 workers are recommended for P14. As a result P7 offers a faster implementation than P14.

Note that all of this is subject to the quality of the performance models of the farm stages of the pipelines. It seems however that given the current performance models the transformation system has performed as expected in finding what it considers to be the best implementation.

8.4.5 Further Experiments

We conducted a few further experiments with a version of the P7 implementation. Our aim was to observe to what degree modified communication protocols would affect performance when using a varied number of worker processors. We used four different data sets, each

Performance with modified communication protocols								
	8 sides		15 sides		20 sides		10-16 sides	
Prog	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
seq	961	1.0	1912	1.0	2590	1.0	1639	1.0
P7-1	261	3.7	464	4.1	612	4.2	412	4.0
P7-2	250	3.8	439	4.4	577	4.5	383	4.3
P7-3	241	4.0	449	4.3	607	4.3	398	4.1
P7-4	219	4.4	424	4.5	571	4.5	367	4.5
P7-5	228	4.2	387	4.9	509	5.1	353	4.6

Table 8.12: Symarea: times (in milliseconds) with modified communication

with 200 polygons with from 8 to 20 sides, and with all polygons being symmetric. Six different implementations were tested:

- **seq**: A purely sequential program
- **P7-1**: The P7 implementation configured with 7 workers in the `map` farm, and 8 workers in the `filtermap` farm.
- **P7-2**: As P7-1, but with the communication of the lists of object coordinates as a loop rather than using a stack.
- **P7-3**: The P7 implementation configured with 3 workers in the `map` farm, and 4 workers in the `filtermap` farm.
- **P7-4**: As P7-3, but with the communication of each object coordinate as a single 2-`INT` (an occam2 `[2]INT` representing a `(real * real)` tuple) or 4-`INT` (an occam `[4]INT` corresponding to a `(real * real * (real * real))` tuple) message rather than sending these as 2 or 4 separate single `INT` messages. Note that when data is taken from a local heap or stored into a local heap this involves an overhead in packing and unpacking the messages.
- **P7-5**: As P7-4, but using 7 workers in the `map` farm, and 8 workers in the `filtermap` farm.

The results are summarised in table 8.12. Comparing P7-1 and P7-2 a small positive effect (improvement 4%-7%) can be observed by using a loop rather than a stack for list communication. A somewhat bigger improvement (12%-16%) over P7-1 is seen in P7-5 as a result of sending tuples as single messages. In these implementations a high number of worker processors is used, a total of 15 for the two farms. Using fewer workers, a total of 7, turns out favourably for P7-3 which is about 1%-7% faster than P7-1, while which is better of P7-4 and P7-5 depends on the input data.

The experiments performed with modified communication protocols shows that an increase in execution speed can be obtained. The effect is significant in the sense that it points out communication protocols as a natural candidate for optimisation in a future version of the compiler.

8.5 Summary

We have presented the results of applying the SkelML prototype compiler to three example programs. The compiler handled the compilation of the programs satisfactorily with respect to collection of profiling information, derivation of process networks, application of transformations and code production. For the resulting occam2 code speedups were achieved for all programs.

The performance estimates were of varying quality. Mostly the errors were less than 20%, and sometimes even below 10%. For the ray tracer the estimates were consistently lower than the measured results, for the the best match program they were consistently higher, while for the symmetric area application, both effects were observed. Overall we consider these results to be positive considering that both sequential time performance model constants and formulae as well as skeleton performance models are relatively unsophisticated. Thus we believe that there is scope for further refining the performance estimates to an even higher degree of accuracy.

The transformation system seems to work as expected, and as demonstrated in the symmetric area example, can contribute positively to the performance of the occam2 implementation.

A number of other points were also made in the chapter. We noted that the SkelML implementation compared reasonably well both with the PUFF and Concurrent Clean compilers for the ray tracer example.

For the best match program changes in the input list length did not affect the occam2 implementation's performance drastically. Finally, for the symmetric area program, optimisations in the communication protocols resulted in a small, but clear improvement in execution times.

Chapter 9

Conclusion

In this chapter we summarise the thesis, draw conclusions, and give pointers to future work. Section 9.1 reviews the thesis, pointing out the main issues investigated and the conclusions drawn in each chapter. Section 9.2 examines the outcome of the thesis with respect to our objectives, and assesses its contribution. A discussion of our work pointing out limitations and their relevance is given in section 9.3, while section 9.4 concludes the thesis with pointers to avenues of future research.

9.1 Thesis Summary

In the first three chapters of the thesis we gave the motivation for our work and reviewed a wide range of relevant approaches and proposals. In chapter 1 we noted that parallel computers are not widely used despite there being a strong demand for more processing power. Chapter 2 concluded that a low-level and explicit approach to parallel programming complicates the development of parallel software, while at the same time hampering portability. We advocated an implicit and high level approach to parallel programming, and suggested functional programming languages as a good choice in this respect. Functional languages have favourable characteristics for software development in general, and due to evaluation order independence, present good potential for parallelism. However, the approaches of dataflow and parallel graph reduction have so far not succeeded in fulfilling this potential. We therefore suggested skeletons as a more promising possibility, and in chapter 3 we reviewed and discussed a number of approaches to using skeletons for parallel processing. We concluded that by restricting the exploitation of parallelism to a fixed set of skeletons, by using profiling information as a basis for performance modeling, and by expressing skeletons as HOFs in a functional language, a skeleton-based compiler should be able to handle all major aspects of compilation with minimal programmer intervention.

In chapter 4 we outlined the overall structure of the proposed SkelML compiler in accordance with this, and presented implementation details of skeletons and communication protocols.

In the following three chapters we investigated the internals of the compiler. Chapter 5 dealt with performance modeling and prediction. We presented an operational semantics-based approach to profiling of sequential SML code, which gave satisfactory accuracy of predicted occam2 code execution times on our target platform. We also gave performance models for the supported skeletons, and explained how profiling information could be used for predicting communication times.

An internal representation of skeleton parallelism as a process network was suggested in chapter 6. We argued that the representation of skeletons as skeleton nodes contributed to a high level of abstraction in the network representation, to the benefit of internal manipulation and transformation, performance prediction and mapping. We also discussed the extraction of process networks from SML programs, and presented a scheme $\mathcal{N}$ describing our approach in some detail.

The problems involved in static resource allocation were discussed in chapter 7. We presented a scheme $\mathcal{M}$ to map a process network down to a physical architecture. We also discussed the role of transformations as an aid to optimisation, and highlighted problems associated with this.

In chapter 8 we applied the compiler to three example programs. We outlined the compilation process and discussed the results obtained, with the conclusion that the prototype implementation of the compiler delivered a promising performance in all respects.

9.2 Achievements and Contribution

In chapter 2 we stated that the overall objective of the thesis was to to explore skeleton-based compilation of functional programming languages. Following the review and discussion in chapter 3, we proposed a framework for such compilation in chapter 4. The main points of our approach, as described in the design of the SkelML compiler, can be summarised as:

- Parallelism is found in a fixed set of skeletons expressed as higher order functions in a functional language. Pipeline parallelism is extracted from function composition. No other ways of exploiting parallelism are supported.
- Other than expressing a problem using certain HOFs, the programmer is neither requested nor permitted to make decisions regarding the exploitation of parallelism.
- A profiling system extracts information about the program's run time behaviour with respect to execution times and sizes of data objects. This data forms the basis for the compiler's decisions regarding exploitation of parallelism and automatic resource management. The profiling mechanism is directly related to the structural operational semantics of SML, and should ease the re-targeting of the compiler.
- A set of performance formulae and constants are used in performance predicting for skeletons, communication and sequential execution.
- A process network representation is used for handling of the program internally in the compiler. This network is manipulated by optimising transformations, and is mapped down to the target platform.

The achievements of the thesis are the integration of the above points into a framework for skeleton-based compilation, and concrete proposals for addressing each point in a compiler implementation. Through a prototype implementation, all the essential aspects of the above mentioned points have been illustrated.

On this background our precise contribution is:

- a review and discussion of the the characteristics and properties relevant and desirable for skeleton-based compilation.
- the proposal of a framework for skeleton-based compilation. We have stated what components we believe to be the cornerstones of a skeleton-based compiler for a functional language: a performance prediction mechanism based on profiling information and performance models for skeletons, an internal representation allowing optimisations and automatic resource allocation, and an automatic or semi-automatic transformation system. We have outlined the responsibilities of each component as well as the general structure of the compiler.
- the integration of these components into a prototype compiler implementation. We have chosen a functional language with a fairly rich sets of features as our source language. The implemented profiling system is directly based on the operational semantics of this language. Through built-in performance models the compiler can predict the run time behaviour of skeletons, communication, and sequential pieces of code. From an internal process network representation of the program, the performance predictions are used for automatic resource allocation and for guiding the supported transformation system. occam2 code is produced to support the parallel evaluation of the program, where the compiler emits code for skeletons, communication protocols, and various administrative code, while the s2o translator is used to produce sequential code.
- the illustration through examples of how the compilation process works in practice, accompanied by performance results, showing a typical error in performance predictions of around 20% or less.

9.3 Limitations and Discussion

Specifying and implementing a full-scale parallelising compiler is a very large and complex task, and we have had to enforce several restrictions to comply with the resources at our disposal. In our choice of what to focus on, we have given priority to illustrating the overall compiler structure with emphasis on what we believe to be the central concepts of skeleton-based compilation. In this context the compiler is primarily meant as a research prototype. Each component in the compiler can be refined in greater detail, most notably the performance models. The results obtained using the prototype compiler seems to back up our claim that indeed, skeleton-based compilation is possible and fully feasible. This section points out some limitations of our work, and discusses their overall relevance.

A fundamental limitation of our approach to skeleton parallelism is that we do not consider the general nesting of skeleton structures. The only form of nesting we consider is that of skeletons figuring as elements of a pipeline. Nesting allows parallelism to be exploited at several levels simultaneously, which is certainly an important contributor to exploitation of massive parallelism. By disallowing nested parallelism, we diminish the degree of parallelism exploitable. On the other hand, by only exploiting parallelism at the topmost level of nesting, the grain of computation increases, which may compensate somewhat. For example if the argument function to a `map` contains a skeleton substructure, for example another `map`, the corresponding implementation would be a farm where each worker is itself a farm. By exploiting parallelism at the topmost level only, the outermost farm may be able to successfully support more workers than the nested structure could support sub-farms.

An interesting related issue is how to implement nested structures if parallelism at the innermost level is found to be too fine grain to be worth exploiting. Should parallelism then be exploited at the level of the next enclosing skeleton, or would the exploitation of parallelism at another level offer better performance? Accurate performance models may be able to provide satisfactory answers to these questions. Our main reason for not supporting nested parallelism is that it complicates the generation of code for skeleton harnesses. This poses no fundamental problem, but would have been too time demanding in the context of the project.

A further practical limitation is that we only provide a limited set of skeletons. Without doubt a divide and conquer or a geometric skeleton would have increased the power of the compiler. The compiler design is extensible in this respect, so that more skeletons could be added. What would be required for each new skeleton is its SML definition, a performance model, and a template occam2 harness implementation.

In earlier chapters we have also pointed out a few other practical limitations. The support for communication of strings is trivial to implement. So is the implementation of a HOF removal pre pass, which as it turns out would have been an useful feature. The lack of a garbage collector for s2o restricts the compiler's usefulness for compiling and running larger applications, but for the purpose of demonstrating the main issues of the approach taken, the lack of this feature is not significant. The same can be said for the restricted implementation of the mapping algorithm with respect to `PAR/SEQ` structures.

The need for making accurate performance predictions creates tight bounds between the compiler and the target architecture. This complicates the porting or re-targeting of the compiler, which is an important issue since a separate port is required for each variation of architecture. However, porting a compiler is a one-off effort which is paid back manifold when developing programs. This applies not only to a single compilation of a program, but throughout the program's lifetime, because compilation is an essential part in software development and maintenance. It may be seen as a limitation that we have implemented the compiler to target only one specific platform, but it would not have been practically feasible to consider another architecture. Having said this, we have throughout the thesis discussed implications for other architectures, and we will elaborate further on this in the "Future Work" section to follow.

9.4 Future Work

There are many appealing avenues to pursue for future work. With respect to our approach, we have already pointed out a number of limitations and their possible corrections in the previous section.

We have elaborated skeletons in the context of strict functional languages. Proponents of lazy languages will want to examine the use of skeletons with that evaluation strategy. Other language paradigms may also be interesting. In particular object-oriented languages seem to provide natural encapsulation mechanisms that fit well with the philosophy of skeletons. A point in favour of using the object-oriented paradigm is the general aversion in industry against using an "exotic" programming paradigm like the functional. While we consider this at best an inaccurate judgement, the use of a widely used language like C++ may present a more realistic approach for bringing skeletons into mainstream industrial use.

As we believe software portability to be crucial to the successful mainstream use of parallel computers, the ease with which skeleton-based compilers can be re-targeted is a relevant issue. The concept of relating profiling information to the program semantics rather than to the model of evaluation seems important in this respect. This could allow profiling information to be permanently associated with the program, and would avoid further profiling runs whenever the program was subsequently ported. We have indicated the storage of a vector of inference rule frequencies as a possible implementation of this idea, but further elaboration is needed in order to establish if this would be a practical approach.

Regarding which and how many skeletons to provide support for, this is certainly an area that needs more investigation. We have adopted an approach based on a few general purpose skeletons, but there may be problems for which more specialised skeletons would be preferable. We believe this to be one of the main open questions in skeleton-based parallel processing.

Another interesting area is program transformation. Here it appears that functional skeleton-based systems have an important advantage over other approaches to parallel programming. The rich body of transformation work already existing for functional languages could here be used in a new context. Our chosen approach of an exhaustive search works well for small programs or when program transformation is used for optimising only a restricted range of structures, for example pipelines. However, an exhaustive search strategy becomes computationally infeasible when dealing with program transformation in a more general form. We suggested the use of heuristics to guide the search, but whether efficient heuristics can be found still needs further investigation. Alternatively, semi automatic transformation systems could be used, where the programmer and transformation system collaborate in deriving the transformed program.

For both of the above points the Bird-Meertens Formalism seems to be a good choice as an underlying basis for skeleton-based parallel programming. Its formal underpinning and natural incorporation of equational laws provides a promising framework in which to further investigate transformations. Also with respect to the choice of particular skeletons BMF has been advocated as well suited because a form of completeness can be guaranteed in the set of operators provided for a given type.

A general problem in parallelising compilers, and one which applies strongly to skeleton-based compilers such as SkelML, is the programmer's perception of the quality of the compiler's decisions. Unless the programmer is fully convinced that the compiler produces efficient code, it will always be tempting to use an explicit language and build applications by hand. An analogous dilemma occurred with the first compilers for imperative languages, where many in the beginning preferred to use assembly languages. Nowadays of course, traditional compilers are so good that it is only in very rare cases that machine-level programming is used. Functional languages still suffer from this, in that despite their acknowledged good characteristics for software development, performance is still viewed as a problem. For parallelising compilers the problem is not only to produce efficient code for a given parallel program, but also to make the many decisions regarding when and how to exploit parallelism. Thus a compiler that programmers would be willing to trust and use must not only produce correct and efficient code with respect to functionality, but it must also make correct performance predictions.

With respect to future programming of parallel computers it is certain that the need for a well suited programming methodology is paramount. Issues of fundamental long term importance like ease of programming and portability must be addressed satisfactorily. We have argued that even though skeletons provide a high level of abstraction, this in no way conflicts with high performance. Skeletons do however impose a restricted way of exploiting parallelism as a price to pay for better structural properties. Similar developments towards a higher level of abstraction have proved to be a successful recipe in other areas of science. Will this also apply to the area of parallel processing? In our opinion the answer to this question must be a clear "yes"!

Appendix A

Auxiliary Function Definitions in SML

```
fun map f [] = [] |
    map f (h::t) = f h :: map f t;

fun filter f [] = [] |
    filter f (h::t) =
      if f h then h :: (filter f t)
             else filter f t;

fun mapfilter f p [] = [] |
    mapfilter f p (h::t) =
      if p h then (f h)::(mapfilter f p t)
             else mapfilter f p t;
(* equivalent to ((map f) o (filter p)) *)

fun filtermap p f [] = [] |
    filtermap p f (h::t) =
       let val fh = f h in
          if p fh then fh::(filtermap p f t)
                  else (filtermap p f t)
       end;
(* equivalent to ((filter p) o (map f)) *)

fun fold (f:'t->'t->'t) a [] = a |
    fold f a (h::t) = f h (fold f a t);
(* This is foldr: requires that f is associative AND commutative *)

fun foldmap (f:'t->'t->'t) a g [] = a |
    foldmap f a g (h::t) = f (g h) (foldmap f a g t);
(* equivalent to (fold f a) o (map g) *)

fun foldr f a [] = a |
    foldr f a (h::t) = f h (foldr f a t);

fun foldr1 f l = foldr f 1 l;

fun id x = x;
```

Appendix B

Transformation Rules

```
(* T1 *)
trans {f,g,l} map f (map g l) == map (fn x=>f (g x)) l;

(* T2 *)
trans {f,p,l} map f (filter p l) == mapfilter f p l;

(* T3 *)
trans {f,p,l} filter p (map f l) == filtermap p f l;

(* T4 *)
trans {f,g,a,l} fold f a (map g l) == foldmap f a g l;

(* T5 *)
trans {p1,p2,l} filter p2 (filter p1 l) ==
                filter (fn x => (p1 x) andalso (p2 x)) l;

(* T6 *)
trans {p1,p2,l} filter p1 (filter p2 l) == filter p2 (filter p1 l);
```

Appendix C

SML code listsings

C.1 sumn

```
(* Computes sum of integers between 1 and n *)
(* Used for profiling purposes only *)

fun sumn 0 = 0 |
    sumn n = n + sumn (n-1);
```

C.2 primes

```
(* Finds the n'th prime number *)
(* Used for profiling purposes only *)

fun append [] l = l |
    append (h::t) l = h::(append t l);

fun reverse1 [] l = l |
    reverse1 (h::t) l = reverse1 t (h::l);

fun reverse l = reverse1 l [];

fun new n [] = true |
    new n (h::t) = if (n mod h) = 0 then false else new n t;

fun gen x n x1 build =
      if x=x1
      then build
      else if (new n build)
           then (gen x (n+1) (x1+1) (append build [n]))
           else gen x (n+1) x1 build;

fun primes n = gen n 2 0 [];

fun primeno n = hd (reverse (primes n));
```

C.3 sort

```
(* Sorts n integers using a binary tree as the intermediate structure *)
(* Used for profiling purposes only *)

fun append [] l = l |
    append (h::t) l = h::append t l;

datatype 'a TREE = NIL |
                   NODE of ('a TREE * 'a * 'a TREE);

fun insert x NIL _ = NODE (NIL,x,NIL) |
    insert x (NODE (l,v,r)) grth =
       if (grth x v)
       then NODE (l,v,insert x r grth)
       else NODE (insert x l grth,v,r);

fun build1 [] T  _ = T |
    build1 (h::t) T gr = insert h (build1 t T gr) gr;

fun build l gr = build1 l NIL gr;

fun unbuild NIL = [] |
    unbuild (NODE (l,v,r)) = append (unbuild l) (v::(unbuild r));

fun sort1 l gr = unbuild (build l gr);

fun intgr (x:int) y = x>y;

fun intsort l = sort1 l intgr;

fun random x = ((x + 77777) mod 15737, x);

fun ranlist1 0 seed = [] |
    ranlist1 n seed =
       let val (newseed,rnd) = random seed in
          rnd::(ranlist1 (n-1) newseed)
       end;

fun ranlist n = ranlist1 n 187;

fun sort n = intsort (ranlist n);
```

C.4 euler

```
(* Based on euler benchmark program in Augustsson and Johnsson,
   Graph Reduction with the <V,G>-machine, FPCA 89 *)

fun length [] = 0 |
    length (_::xs) = 1+length xs;

fun map f [] = [] |
    map f (x::xs) = f x :: map f xs;

fun filter f [] = [] |
    filter f (x::xs) =
       if f x
       then x::filter f xs
       else filter f xs;

fun sum [] = 0 |
    sum (x::xs) = x+sum xs;

fun fromto i j =
       if i>j
       then []
       else i::fromto (i+1) j;

fun gcd x 0 = x |
    gcd x y = gcd y (x mod y);

fun relprime x y = gcd x y = 1;

fun euler n = length (filter (relprime n) (fromto 1 (n-1)));

fun sum [] = 0 |
    sum (x::xs) = x+sum xs;

fun top n m = sum (map euler (fromto n m));
```

C.5 ray

```
(* Ray tracer by D Busvine/P H J Kelly *)

datatype Object = Polygon of int * (real*real*real) * ((real*real*real) list);
(* Polygon(i,N,Vs) is a polygon with identifier i,
   with normal vector N and vertices Vs *)

datatype Impact = IMPACT of real * int |
                  NOIMPACT;
(* Impacts: IMPACT(r,i) indicates an impact at distance r with object i
            NOIMPACT indicates no impact *)

fun FirstImpact (os:Object list) (r:((real*real*real)*(real*real*real))) = ...;
(* Finds first impact, or NOIMPACT, between ray and objects *)

fun FindImpacts rays objects = map (FirstImpact objects) rays;
(* Finds all first impacts between objects and rays *)

fun GenerateRays (Detail,X,Y,Z) = ...;
(* generates a list of Detail*Detail rays emanating from the point X, Y, Z.
   The rays are formed by projecting from the view point (X,Y,Z)
   through a grid of points (of side Detail) held at a distance of 4 units
   from the viewpoint.
   The grid is positioned so that rays in the centre of the view
   are directed toward the origin. *)

fun top (Detail,XView,YView,ZView) Scene =
              FindImpacts (GenerateRays (Detail,XView,YView,ZView)) Scene;
(* Toplevel function *)

(* Example call:
val sc =
[Polygon(1,(1.0,0.0,0.0),
[(1.0,0.0,~0.5),(1.0,0.0,0.5),(1.0,~1.0,0.5),(1.0,~1.0,~0.5),
 (1.0,0.0,~0.5)]), ... ];


top (15,10.0,7.0,6.0) sc;
*)
```

C.6 bestmatch

```
(* By T Bratvold *)

val jot = 1.0E~3;
fun sqr (x:real) = x * x;
fun length [] = 0 |
    length (_::t) = 1 + length t;
fun distance (x1,y1) (x2,y2) = sqrt (sqr(x2-x1) + sqr(y2-y1));
fun scalefactor p1 p2 p1' p2' = (distance p1 p2)/(distance p1' p2');
fun equal v1 v2 = (abs (v1-v2)) < jot;
fun pointok scale p0 p1 p2 p0' p1' p2' =
       (equal (distance p1 p2) ((distance p1' p2') * scale)) andalso
       (equal (distance p0 p2) ((distance p0' p2') * scale));

fun lastelm (h::[]) = h |
    lastelm (_::t) = lastelm t;

(* objR = reference object, obj = obj from main input list *)
(* scale = relation obj = objR * scale *)

fun match2 obj (p1::p2::[]) _ = true |
    match2 (p1::p2::p3::T) (p1'::p2'::p3'::T') scale =
       (pointok scale p1 p2 p3 p1' p2' p3') andalso
       (match2 (p2::p3::T) (p2'::p3'::T')) scale;

fun match1 _ _ 0 = false | (* have tried all rotations *)
    match1 obj objR n =
       match2 obj objR
          (scalefactor (hd obj) (hd (tl obj)) (hd objR) (hd (tl objR))) orelse
       match1 (tl obj) objR (n-1);

fun match obj objR =
    (length obj = length objR) andalso
    match1 (obj@obj) ((lastelm objR)::objR) (length objR);

(* matchtop toplevel function for map *)
(* returns a list of frequencies [0,0,0,...,1] (1 in pos i = 1 obj#i found) *)

fun matchtop [] obj = [] |
    matchtop (objR::T) obj =
       if (match obj objR)
       then [1]
       else 0::(matchtop T obj);

(* findmfo zips with + two lists of freqeuncies *)
fun findmfo ([]:int list) L = L |
    findmfo L [] = L |
    findmfo (h1::t1) ((h2:int)::t2) = (h1+h2)::(findmfo t1 t2);

(* findmax picks best candidate from list of frequencies *)
fun findmax2 _ [] (bestC,bestF) = (bestC,bestF) |
    findmax2 [] _ (bestC,bestF) = (bestC,bestF) |
    findmax2 (Ch::Ct) (Fh::Ft) (bestC,(bestF:int)) =
       if (Fh > bestF)
       then findmax2 Ct Ft (Ch,Fh)
       else findmax2 Ct Ft (bestC,bestF);

fun findmax candidates frequencies = findmax2 candidates frequencies ([],~1);
```

```
(* toplevel *)
fun bestmatch (candidates:(real*real)list list)
              (problems:(real*real)list list) =
   findmax candidates
      (fold findmfo [] (map (matchtop candidates) problems));
```

C.7 symarea

```
(* By G Michaelson *)

val jot = 1.0E~4;
fun sq (x:real) = x*x;

fun dists (ax,ay) (bx,by) =
       let val dsq = (sq (bx-ax))+(sq (by-ay)) in
          (sqrt dsq,dsq)
       end;

(* ax * ay -> bx * by -> length * length squared *)

fun lengths1 (ax,ay) [] = [] |
    lengths1 (ax,ay) [(Nx,Ny)] = [(Nx,Ny,dists (Nx,Ny) (ax,ay))] |
    lengths1 (ax,ay) ((ix,iy)::(jx,jy)::t) =
     (ix,iy,dists (ix,iy) (jx,jy))::(lengths1 (ax,ay) ((jx,jy)::t));

(* pass 1st point down to last point *)

fun lengths [] = [] |
    lengths ((ax,ay)::t) = lengths1 (ax,ay) ((ax,ay)::t);

(* x * y list -> x * y * (side length * sl squared) list *)

fun samesides1 d [] = true |
    samesides1 d ((ax,ay,(ab,_))::t) =
       abs (d-ab)<jot andalso samesides1 d t;

fun samesides ([]:(real*real*(real*real))list) = false |
    samesides ((ax,ay,(ab,_))::t) = samesides1 ab t;

fun cosine (ax,ay,(ab,absq)) (bx,by,(bc,bcsq)) (cx,cy,_) =
       let val (ac,acsq) = dists (ax,ay) (cx,cy) in
          (absq+bcsq-acsq)/(2.0*ab*bc)
       end;

fun samecosines1 a b cos1 [c,d] =
       abs (cos1-(cosine c d a))<jot andalso
          abs (cos1-(cosine d a b))<jot |
    samecosines1 a b cos1 (a1::b1::c1::t) =
       abs (cos1-(cosine a1 b1 c1))<jot andalso
          samecosines1 a b cos1 (b1::c1::t);

fun samecosines [] = false |
    samecosines [c] = false |
    samecosines [b,c] = false |
    samecosines (a::b::c::t) = samecosines1 a b (cosine a b c) (b::c::t);

fun tarea (ax,ay,_) (bx,by,_) (cx,cy,_) =
       let val ((ab,absq),(bc,bcsq)) =
                       (dists (ax,ay) (bx,by),dists (bx,by) (cx,cy))
           val cos1 = cosine (ax,ay,(ab,absq)) (bx,by,(bc,bcsq))
                                     (cx,cy,(0.0,0.0)) in
              0.5*ab*bc*(sqrt (abs (1.0-cos1*cos1)))
       end;

fun area1 a b [] = 0.0 |
```

```
    area1 a b (c::t) = (tarea a b c)+(area1 a c t);

fun area ([]:(real * real * (real*real)) list) = 0.0 |
    area [c] = 0.0 |
    area [b,c] = 0.0 |
    area (a::b::c::t) = area1 a b (c::t);

fun symarea l =
       map area (filter samesides (filter samecosines (map lengths l)));
(* Toplevel function *)
```

Bibliography

[1] FAST: Functional Programming for Arrays of Transputers - The Collected Papers. Report CSTR 93-15, Declarative Systems Laboratory, Electronics & Computer Science, University of Southampton, UK, 1993. Also available as Report Number DOC 93/47 from Department of Computing, Imperial College, London, UK.

[2] William B Ackerman. Data Flow Languages. *IEEE Computer*, pages 15–24, February 1982.

[3] Shakil Ahmed, Nicholas Carriero, and David Gelernter. A Program Building Tool for Parallel Applications. In *Proceedings of the DIMACS Workshop on Specifications of Parallel Algorithms, Princeton University*, May 1994.

[4] A V Aho, J E Hopcroft, and J D Ullman. *Data Structures and Algorithms.* Addison-Wsley, 1983.

[5] A V Aho, R Sethi, and J D Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley, 1986.

[6] Brad Alexander. Mapping Adl to the Bird-Meertens Formalism. Technical Report 94-18, The University of Adelaide, Australia, 1994.

[7] Andrew W Appel, Bruce F Duba, and David B MacQueen. Profiling in the Presence of Optimization and Garbage Collection. Technical Report CS-TR-197-88, Princeton University, Dept. Comp. Sci., Princeton, NJ, USA, November 1987.

[8] Andrew W Appel and David B MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, New York, August 1991. Springer-Verlag.

[9] Arvind and David E Culler. Dataflow Architectures. *Annual Reviews in Computer Science*, (1):225–253, 1986.

[10] Arvind and Kattamuri Ekanadham. Future Scientific Programming on Parallel Machines. In *Proceedings of Supercomputing: 1st International Conference, Athens, Greece*, pages 639–686, 1987.

[11] Arvind and Rishiyur S Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the Conference on Parallel Architectures and Languages Europe, Eindhoven, The Netherlands*, pages 1–29. Springer Verlag, LNCS 259, 1987.

[12] L Augustsson and T Johnsson. Parallel Graph Reduction with the $<\nu,G>$-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 202–213, 1989.

[13] L Augustsson and T Johnsson. The Chalmers Lazy ML Compiler. *The Computer Journal*, 32(2):127–141, 1989.

[14] Bruno Bacci and Marco Danelutto *et al.* P^3L: a Structured High-level Parallel Language, and its Structured Support. Technical Report HPL-PSC-93-55, Pisa Science Center, Hewlett-Packard Laboratories, Pisa, Italy, May 1993.

[15] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, August 1978. 1977 ACM Turing Award Lecture.

[16] F Baiardi and M Danelutto *et al.* Pisa Parallel Processing Project on general-purpose highly-parallel computers. Technical Report HPL-PSC-91-23, Hewlett-Packard Laboratories, Pisa Science Center, Pisa, Italy, May 1991. Presented at the Fifteenth Annual International Computer Software And Applications Conference, September 11-13, 1991, Tokyo, Japan.

[17] Peter Bailey and Malcolm Newey. An Extension of ML for Distributed Memory Multicomputers. In G Gupta, G Mohay, and R Topor, editors, *Proceedings of the Sixteenth Australian Computer Science Conference*, pages 387–396, February 1993.

[18] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David Padua. Automatic Program Parallelization. CSRD Report 1250, CSRD, University of Illinois at Urbana-Champaign, November 1992.

[19] H P Barendregt. *The lambda calculus - its syntax and semantics.* North-Holland, 1984.

[20] M L Barja *et al.* Semantics Based Implementation of a Deductive Object-Oriented Database Programming Language. *Journal of Programming Languages*, 2:93–108, 1994.

[21] Richard Bird. A Calculus of Functions for Program Derivation. Technical Monograph PRG-64, Oxford University Computing Laboratory, UK, December 1987.

[22] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[23] Richard Bird. An introduction to the theory of lists. In M Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer-Verlag, 1987.

[24] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N Turner. The ML Kit Version 1. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, U.K., March 1993.

[25] Guy E Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, PA, USA, April 1993.

[26] Guy E Blelloch and Siddhartha Chatterjee. VCODE: A Data-Parallel Intermediate Language. In *Proceedings Frontiers of Massively Parallel Computation*, October 1990.

[27] Karsten Bohlmann, Rita Loogen, and Yolanda Ortega-Mallén. Concurrent Functional Processes. In *Proceedings of the Fifth International Workshop on Implementation of Functional Languages, Nijmegen, the Netherlands*, pages 35–50, September 1993.

[28] Shahid H Bokhari. On the Mapping Problem. *IEEE Transactions on Computing*, C-30(3):207–214, March 1981.

[29] Tore A Bratvold. Determining Useful Parallelism in Higher Order Functions. In *Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages, Aachen, Germany*, pages 213–226, September 1992. Aachener Informatik-Berichte Nr 92-19.

[30] Tore A Bratvold. A Skeleton-Based Parallelising Compiler for ML. In *Proceedings of the Fifth International Workshop on Implementation of Functional Languages, Nijmegen, the Netherlands*, pages 23–34, September 1993.

[31] R A A Bruce, S R Chapple, N B MacDonald, and A S Trew. CHIMP and PUL: Support for Portable Parallel Computing. Technical Report EPCC-TR93-07, Edinburgh Parallel Computing Centre, March 1993. Presented at the Fourth Annual Conference of the Meiko User Society, Southampton, April 1993.

[32] P C Burkimsher. Combinator Reduction in a Shared-memory Multiprocessor. *The Computer Journal*, 30(3):215–222, 1987.

[33] Geoffrey L Burn. Evaluation Transformers - A Model for the Parallel Evaluation of Functional Languages (Extended Abstract). In G Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 446–470. Springer-Verlag LNCS 274, 1987.

[34] R M Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[35] R M Burstall, D B MacQueen, and D T Sanella. Hope: an experimental applicative language. Technical Report CSR-62-80, Department of Computer Science, University of Edinburgh, 1980.

[36] F W Burton and M R Sleep. Executing Functional Programs on a Virtual Tree of Processors. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 187–194. ACM, 1981.

[37] David Busvine. Transformation Techniques for Functional Programs. Technical Report TR 90/3, Department of Computer Science, Heriot-Watt University, Edinburgh, July 1990.

[38] David Busvine. Translation of SML to Sequential Occam2. Technical Report TR 91/7, Department of Computer Science, Heriot-Watt University, Edinburgh, 1991.

[39] David Busvine. *Detecting Parallel Structures in Functional Programs.* PhD thesis, Dept of Computing and Electrical Engineering, Heriot-Watt University, October 1993.

[40] David Busvine. Implementing Recursive Functions as Processor Farms. *Parallel Computing*, 19:1141–1153, 1993.

[41] David Cann. Retire FORTRAN? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[42] M Capel, J M Troya, and A Palma. Distributed Active Objects: A Methodological Proposal and Tool for Distributed Programming with Transputer Systems. *Microprocessing and Microprogramming*, (38):197–204, 1993.

[43] L Cardelli. Compiling a Functional Language. In *Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin*, pages 208–217, August 1984.

[44] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989.

[45] Thomas L Casavant and Jon G Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Sy stems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[46] Lawrence W Chan. Algorithmic Skeletons in Concurrent ML. Technical Report EPCC-SS92-07, Edinburgh Parallel Computing Centre, September 1992.

[47] K Mani Chandy and Carl Kesselman. Parallel Programming in 2001. *IEEE Software*, pages 11–20, November 1991.

[48] A Church. An unsolvable problem in the elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[49] A Church. *The Calculi of λ-conversion*. Princeton University Press, 1941.

[50] Paul D Coddington. The Application of Transputer Arrays to Scientific Problems. In *Australian Transputer and OCCAM User Group Conference Proceedings, RMIT, Melbourne, 23-24 June 1988*, pages 5–10, 1988.

[51] Charles Cohen. Computer-Assisted Microanalysis of Programs. *Communications of the ACM*, 25(10):724–733, October 1982.

[52] Murray Cole. List Homomorphic Parallel Algorithms for Bracket Matching. Technical Report CSR-29-93, Department of Computer Science, The University of Edinburgh, August 1993.

[53] Murray Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. Technical Report CSR-25-93, Department of Computer Science, The University of Edinburgh, May 1993.

[54] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT, 1989.

[55] A Contessa, E Cousin, C Coustet, M Cubero-Castan, G Durrieu, B Lecussan, M Lemâitre, and P Ng. MaRS: a combinator graph reduction multiprocessor. In E Odijk, M Rem, and J-C Syre, editors, *PARLE'89 Parallel Architectures and Languages Europe, Vol I*, pages 176–192. Springer-Verlag LNCS 365, 1989. More information on MaRS is available on the WWW, URL: http://www.cert.fr/~castan/mars.html.

[56] David E Culler, Klaus Erik Schauser, and Thorsten von Eicken. Two Fundamental Limits on Dataflow Multiprocessing. In *Proceedings of the IFIP Working Group 10.3 (Concurrent Systems) Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*. Elsevier Science Publishers, January 1993.

[57] H K Curry and R Feys. *Combinatory Logic*. North-Holland, Amsterdam, The Netherlands, 1958.

[58] Marco Danelutto and Roberto Di Meglio *et al.* A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8:205–220, August 1992. North Holland.

[59] Marco Danelutto and Susanna Pelagatti. Parallel Implementation of FP using a Template-based Approach. In *Proceedings of the Fifth International Workshop on Implementation of Functional Languages, Nijmegen, the Netherlands*, pages 7–21, September 1993.

[60] J Darlington, A J Field, P G Harrison, D Harper, G K Jouret, P J Kelly, K M Sephton, and D W Sharp. Structured Parallel Functional Programming. In Hugh Glaser and Pieter Hartel, editors, *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 31–51. Department of Electronics and Computer Science, University of Southampton, 1991. Technical Report Series CSTR 91-07.

[61] J Darlington, M Ghanem, and H W To. Structured Parallel Programming. In *Working Conference on Massively Parallel Programming Models: Suitability, Realization, and Performance*, Berlin, Germany, 1993.

[62] J Darlington and M Reeve. ALICE - A multi-processor reduction machine for the parallel evaluation of applicative languages. In *Proc. ACM Conf. Functional Programming Lang. Comput. Arch.*, pages 65–75, 1981.

[63] J Darlington and H W To. Building Parallel Applications without Programming. In *Presented at the Second Workshop on Abstract Machine Models for Highly Parallel Computers, Leeds, UK*, April 1993.

[64] J Darlington *et al.* Parallel Programming Using Skeleton Functions. In A Bode, M Reeve, and G Wolf, editors, *PARLE '93 Parallel Architectures and Languages Europe, Munich, Germany*, pages 146–160. Springer-Verlag, LNCS 694, 1993.

[65] J Darlington *et al.* A Functional Programming Environment Supporting Execution, Partial Evaluation and Transformation. In E Odijk, M Rem, and J-C Syre, editors, *PARLE'89 Parallel Architectures and Languages Europe, Vol I*, pages 286–305. Springer-Verlag LNCS 365, 1989.

[66] Antony J T Davie and David J McNally. CASE - A Lazy Version of an SECD Machine with a Flat Environment. Revised Report CS/90/19, Department of Mathematical and Computational Sciences, University of St. Andrews, North Haugh, St. Andrews, Fife KY16 9SS, U.K., 1990.

[67] Alan L Davis and Robert M Keller. Data Flow Program Graphs. *IEEE Computer*, pages 26–41, February 1982.

[68] I P de Guzmán, P G Harrison, and E Medina. Pipelines for Divide-and-Conquer Functions. *The Computer Journal*, 36(3):254–268, 1993.

[69] Jack B Dennis. Data Flow Supercomputers. *IEEE Computer*, pages 48–56, November 1980.

[70] Antoni Diller. *Compiling Functional Languages.* John Wiley and Sons Ltd, 1988.

[71] Kattamuri Ekanadham. A Perspective on Id. In *[163]*, 1991.

[72] Pieter H Hartel *et al.* Pseudoknot: a Float-Intensive Benchmark for Functional Compilers (DRAFT). In John Glauert, editor, *Proceedings of the 6th International Workshop on the Implementation of Functional Languages, Norwich, U.K.*, pages 13–1 – 13–34, September 1994.

[73] Martin S Feather. A System for Assisting Program Transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1982.

[74] D Feldcamp, H V Sreekantaswamy, A Wagner, and S Chanson. Towards a Skeleton-based Parallel Programming Environment. In A Veronis and Y Paker, editors, *Transputer Research and Applications 5*, pages 104–115. IOS Press, 1992.

[75] D Feldcamp and A Wagner. Parsec - A Software Development Environment for Performance Oriented Parallel Programming. In S Atkins and A S Wagner, editors, *Transputer Research and Applications 6*, pages 247–262. IOS Press, 1993.

[76] David Fernández-Baca. Allocating Modules to Processors in a Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1427–1436, November 1989.

[77] Anthony J Field and Peter G Harrison. *Functional Programming.* Addison-Wesley, 1988.

[78] I Foster and D Walker. Paradigms and Strategies for Scientific Computing on Distributed Memory Concurrent Computers. In *Proceedings of the High Performance Computing 1994 Conference.* Society for Computer Simulation, San Diego, CA, USA, April 1994.

[79] Guang Gao, Jean-Luc Gaudiot, and Lubomir Bic. Special Issue on Dataflow and Multithreaded Architectures - Guest Editors' Introduction. *Journal of Parallel and Distributed Computing*, 18:271–272, 1993.

[80] Milind Girkar and Constantine D Polychronopoulos. Automatic Extraction of Functional Parallelism from Ordinary Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, March 1992.

[81] Benjamin Goldberg. Multiprocessor Execution of Functional Programs. *International Journal of Parallel Programming*, 17(5):425–473, 1988.

[82] Benjamin Goldberg and Paul Hudak. Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor. In Joseph H Fasel and Robert M Keller, editors, *Graph Reduction, Proceedings of a Workshop, Santa Fé, New Mexico, USA*, pages 94–113. Springer-Verlag LNCS 279, 1986.

[83] S A Green and D J Paddon. An Extension of the Processor Farm Using a Tree Architecture. In C Askew, editor, *Proceedings of the 9th occam User Group Technical Meeting*, pages 53–70. IOS Press, 1988.

[84] Andrew S Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. *IEEE Computer*, pages 39–51, May 1993.

[85] Jai Prakash Gupta, Stephen C Winter, and Derek R Wilson. CTDNet-A Mechanism for the Concurrent Execution of Lambda Graphs. *IEEE Transactions on Software Engineering*, 15(11):1357–1367, 1989.

[86] J R Gurd, C C Kirkham, and I Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[87] Kevin Hammond. Parallel Functional Programming: An Introduction (invited paper). In Hoon Hong, editor, *Proceedings of PaSCo94, Linz, Austria*, pages 181–193. World Scientific Publishing Company, September 1994.

[88] Kevin Hammond and Simon L Peyton Jones. Profiling scheduling strategies on the GRIP parallel reducer. In H Kuchen and R Loogen, editors, *Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages, Aachen, Germany*, September 1992. Aachener Informatik-Berichte Nr 92-19.

[89] P G Harrison and M J Reeve. The Parallel Graph Reduction Machine Alice. In Joseph H Fasel and Robert M Keller, editors, *Graph Reduction, Proceedings of a Workshop, Santa Fé, New Mexico, USA*, pages 181–202. Springer-Verlag LNCS 279, 1986.

[90] Rachel Harrison. *Abstract Data Types in Standard ML*. John Wiley & Sons, 1993.

[91] Anthony J G Hey. Experiments in MIMD Parallelism. In E Odijk, M Rem, and J-C Syre, editors, *PARLE'89 Parallel Architectures and Languages Europe, Vol II*, pages 28–41. Springer-Verlag LNCS 366, 1989.

[92] James Hicks, Derek Chiou, Boon Seong Ang, and Arvind. Performance Studies of Id on the Monsoon Dataflow System. *Journal of Parallel and Distributed Computing*, 18:273–300, 1993.

[93] Jonathan M D Hill. The *AIM* is Laziness in a Data-Parallel Language. In *Proceedings of the Glasgow Workshop on Functional Programming*, 1993.

[94] Paul Hudak and Benjamin Goldberg. Distributed Execution of Functional Programs Using Serial Combinators. *IEEE Transactions on Computers C-34(10)*, October 1985.

[95] Paul Hudak. Para-Functional Programming. *Computer*, 19(8):60–70, August 1986.

[96] Paul Hudak *et al.* Report on the Programming Language Haskell, A Non-strict, Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[97] J Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, April 1989.

[98] J Hughes. Graph Reduction with Super Combinators. Technical Monograph PRG–28, Oxford University Computer Laboratory, 1982.

[99] Kai Hwang. *Advanced Computer Architecture - Parallelism Scalability Programmability*. McGraw-Hill, 1993.

[100] Thomas Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction, Montreal*, pages 58–69, 1984.

[101] Thomas Johnsson. Parallel Evaluation of Functional Programs: The $<\nu$,G$>$-machine approach (Summary). In E H L Aarts, J van Leeuwen, and M Rem, editors, *PARLE'91 Parallel Architectures and Languages Europe, Vol I*, pages 1–5. Springer-Verlag LNCS 505, 1991.

[102] Geraint Jones and Michael Goldsmith. *Programming in occam2*. Prentice Hall, 1988.

[103] John A Keane. An overview of the Flagship system. *Journal of Functional Programming*, 4(1):19–45, January 1994.

[104] Paul H J Kelly. *Functional Programming for Loosely-coupled Multiprocessors*. Pitman/MIT, 1989.

[105] Marco Kesseler. Communication Issues Regarding Parallel Functional Graph Rewriting. In *Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages, Aachen, Germany*, pages 139–157, September 1992. Aachener Informatik-Berichte Nr 92-19.

[106] Marco Kesseler. Reducing Graph Copying Costs - Time to Wrap it Up. In Hoon Hong, editor, *Proceedings of PaSCo94, Linz, Austria*, pages 244–253. World Scientific Publishing Company, September 1994.

[107] Dongseung Kim and Byung-Guoen Yi. A two-pass scheduling algorithm for parallel programs. *Parallel Computing*, 20:867–885, 1994.

[108] Stephen Cole Kleene. *Introduction to Metamathematics*. North-Holland, 1967.

[109] Donald E Knuth. *The Art of Computer Programming*, volume 1 Fundamental Algorithms. Addison-Wesley, 1968.

[110] Donald E Knuth. Mathematical Analysis of Algorithms. In C V Freiman, editor, *Proceedings of the IFIP Congress 71*, pages 19–27. North-Holland Publishing Company, 1972.

[111] H T Kung. Computational models for parallel computers. In R J Elliott and C A R Hoare, editors, *Scientific Applications of Multiprocessors*, pages 1–15. Prentice-Hall, 1989.

[112] Peter J Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1964.

[113] Hugh Lester, David R Kingdon, and Geoffrey L Burn. The HDG-machine: a highly distributed graph-reducer for a transputer network. *The Computer Journal*, 34(4):290–301, 1991.

[114] Virginia Mary Lo. Heuristic Algorithms for Task Assignment in Distributed Systems. *IEEE Transactions on Computers*, 37(11):1384–1397, November 1988.

[115] Perng-Yi Richard Ma, Edward Y S Lee, and Masahiro Tsuchiya. A Task Allocation Model for Distributed Computing Systems. *IEEE Transactions on Computers*, C-31(1):41–47, January 1982.

[116] Andreas Maaßen. Parallel programming with data structures and higher order functions. *Science of Computer Programming*, (18):1–38, 1992. Elsevier Science Publishers B.V. 1992.

[117] James S Mattson Jr. Performance of Parallel Schedulers for Distributed Graph Reduction. In *Proceedings of the Fifth International Workshop on Implementation of Functional Languages, Nijmegen, the Netherlands*, pages 173–183, September 1993.

[118] James R McGraw. The VAL Language: Description and Analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, January 1982.

[119] Daniel Le Métayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[120] Greg Michaelson. *An introduction to functional programming through Lambda calculus*. Addison-Wesley, 1989.

[121] Richard Miller. A Constructive Theory of Multidimensional Arrays. MSc to DPhil transfer dissertation, Programming Research Group, University of Oxford, February 1993.

[122] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[123] Rishiyur S Nikhil, Keshav Pingali, and Arvind. Id Nouveau. Computation Structures Group Memo 265, MIT Laboratory for Computer Science, Cambridge, Massachusetts, USA, July 1986.

[124] E G J M H Nöcker, J E W Smetsers, M C J D van Eekelen, and M J Plasmeijer. Concurrent Clean. In E H L Aarts, J van Leeuwen, and M Rem, editors, *PARLE'91 Parallel Architectures and Languages Europe, Vol II*, pages 202–219. Springer-Verlag LNCS 506, 1991.

[125] Michael G Norman and Peter Thanisch. Models of Machines and Computation for Mapping in Muliticomputers. Technical Report EPCC-TR92-15, Edinburgh Parallel Computing Centre, 1992.

[126] Salvatore Orlando and Marco Vanneschi. Compiling Data-Parallel Paradigms through Basic Communication Mechanisms. Technical report, Dipartimento di Informatica, University of Pisa, Italy, 1993.

[127] Susanna Pelagatti. *A methodology for the development and the support of massively parallel programs*. PhD thesis, Università di Pisa-Genova-Udine, Italy, March 1993. TD 11/93.

[128] Simon L Peyton Jones. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, 32(2):175–186, 1989.

[129] Simon L Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[130] Simon L Peyton Jones, Chris Clack, Jon Salkild, and Mark Hardie. GRIP - A High-Performance Architecture for Parallel Graph Reduction. In G Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 98–112. Springer-Verlag, Berlin, September 1987.

[131] Simon L Peyton Jones and David Lester. *Implementing Functional Languages.* Prentice Hall, 1992.

[132] Simon L Peyton Jones *et al.* The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 1993.

[133] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.

[134] D J Pritchard, C R Askew, D B Carpenter, I Glendinning, A J G Hey, and D A Nicole. Practical Parallelism Using Transputer Arrays. In *Proceedings of the Conference on Parallel Architectures and Languages Europe, Eindhoven, The Netherlands*, pages 278–294. Springer-Verlag, LNCS 258, 1987.

[135] D J Pritchard. Mathematical Models of Distributed Computation. In Traian Muntean, editor, *Parallel Programming of Transputer Based Machines, Proceedings of the 7th occam User Group Technical Meeting, Grenoble, France 1987*, pages 25–36. IOS Press, 1987.

[136] F A Rabhi and J Schwarz. 'Paradigm-Oriented' Design of Parallel Iterative Programs Using Functional Languages. In *Proceedings of Applications of Supercomputers in Engineering, Bath, UK*, 1993.

[137] F A Rabhi and J Schwarz. POPE: A "Paradigm-Oriented" parallel Programming Environment for SIT algorithms. In John Glauert, editor, *Proceedings of the 6th International Workshop on the Implementation of Functional Languages, Norwich, U.K.*, pages 37–1 – 37–21, September 1994.

[138] F A Rabhi. Exploiting Parallelism in Functional Languages: a Paradigm-Oriented Approach . In T Lake and P Dew, editors, *Abstract Machine Models for Highly Parallel Computers.* Oxford University Press, 1993.

[139] John Reekie. Towards Effective Programming for Parallel Digital Signal Processing. Research Report 92.1, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, May 1992.

[140] John Reekie and John Potter. Transforming Process Networks. In *Proceedings of the 1992 Massey Functional Programming Workshop, Massey University, Palmerston North, New Zealand.*

[141] John Reekie and John Potter. Process Network Transformation. In *Parallel Computing and Transputers, Brisbane, Australia.* IOS Press, 1993.

[142] John H Reppy. Concurrent Programming with Events - The Concurrent ML Manual. Technical report, Dept of Comp Sci, Cornell University, NY, USA, November 1990.

[143] C Runciman and R Wakeling. Profiling Parallelism. In *Glasgow Workshop on Functional Programming, Ayr*, pages XVII–1 – XVII–12, 1993.

[144] Patrick M Sansom and Simon L Peyton Jones. Time and space profiling for non-strict higher-order functional languages. Dept of Computing Science, University of Glasgow, UK, 1994.

[145] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* Pitman/MIT Press, 1989.

[146] Vivek Sarkar. PTRAN - The IBM Parallel Translation System. In *[163]*, 1991.

[147] Vivek Sarkar and John Hennessy. Compile-Time Partitioning and Scheduling of Parallel Programs. In *Proceedings of SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, CA*, pages 17–26, 1986.

[148] M Schönfinkel. Über die Bausteine der Matematischen Logik. *Matematische Annalen*, 92:305–316, 1924.

[149] Wolfgang Schreiner. A Para-Functional Programming Interface for a Parallel Computer Algebra Package. In Hoon Hong, editor, *Proceedings of PaSCo94, Linz, Austria*, pages 346–355. World Scientific Publishing Company, September 1994.

[150] Wolfgang Schreiner. Compiling a Para-Functional Language to Parallel PACLIB C. Technical report 94-44, RISC-Linz, Johannes Kepler University, Linz, Austria, August 1994. 1994.

[151] C C Shen and W H Tsai. A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion. *IEEE Transactions on Computers*, C-34(3):197–203, 1985.

[152] Jay M Sipelstein and Guy E Blelloch. Collection-Oriented Languages. *Proceedings of the IEEE*, 4(79):504–523, April 1991. Also as CMU-CS-90-127, School of Computer Science, Carnegie Mellon University, PA, USA.

[153] Stephen Skedzielewski and John Glauert. IF1 - An Intermediate Form for Applicative Languages, July 1985.

[154] Stephen Skedzielewski. Sisal. In *[163]*, 1991.

[155] David B Skillicorn. Models for Practical Parallel Computation. *International Journal of Parallel Programming*, 20(2):133–158, April 1991.

[156] David B Skillicorn. Questions and Answers about Categorical Data Types. In *Bulk Data Types for Architecture Independence (Seminar Program)*, London, May 1994. British Computer Society, Parallel Processing Specialist Group.

[157] David B Skillicorn and W Cai. A Cost Calculus for Parallel Functional Programming. Dept of Computing and Information Science, Queen's University, Canada, April 1994.

[158] David B Skillicorn. Architecture-Independent Parallel Computation. *IEEE Computer*, 23(12):38–51, December 1990.

[159] Mike Spivey. A Categorical Approach to Theory of Lists. In *Mathematics of Program Construction*, pages 399–408. Springer-Verlag LNCS 375, June 1989.

[160] G L Steele and G J Sussman. The Revised Report on Scheme. AI Memo 452, MIT, January 1978.

[161] Holger Stoltze. *Implementierung einer parallelen functionalen Sprache mit algorithmischen Skeletten zur Lösung mathematisch-technischer Probleme.* PhD thesis, Der Mathematisch-Naturwissenschaftlichen Fakultät der RWTH Aachen, Germany, 1994.

[162] Holger Stoltze and Herbert Kuchen. Parallel Functional Programming Using Algorithmic Skeletons. In Joubert, Trystram, Peters, and Evans, editors, *Parallel Computing: Trends and Applications*, pages 651–654. North-Holland Elsevier Science B.V., 1994.

[163] Boleslaw K Szymanski, editor. *Parallel Functional Languages and Compilers.* Frontier Series. ACM Press, New York, 1991.

[164] Chris Thornborrow and Chris Feagle. Parallel Modules in Modular Visual Environments. In *Parallel Processing for Graphics and Scientific Visualisation, Conference Proceedings.* University of Edinburgh, May 1993.

[165] R W S Tregidgo and A C Downton. Processor Farm Analysis and Simulation for Embedded Parallel Processing Systems. In Stephen J Turner, editor, *Tools and Techniques for Transputer Applications, Proceedings of the 12th occam User Group Technical Meeting*, pages 179–189. IOS Press, 1990.

[166] David A Turner. Miranda: A Non-Strict Functional Programming Language. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture, LNCS 201*, pages 1–16. Springer-Verlag, 1985.

[167] David A Turner. A New Implementation Technique for Applicative Languages. *Software - Practice and experience*, 9:31–49, September 1979.

[168] J D Ullman. NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[169] J D Ullman. *Elements of ML Programming.* Prentice Hall, 1994.

[170] A M Wallace, G J Michaelson, P McAndrews, K G Waugh, and W J Austin. Dynamic Control and Prototyping of Parallel Algorithms for Intermediate- and High-Level Vision. *IEEE Computer*, 25(2), February 1992.

[171] I Watson, J Sargeant, P Watson, and V Woods. Flagship Computational Models and Machine Architecture. *ICL Technical Journal*, (5):555–574, 1987.

[172] Kevin Waugh. Parallel Imperative Programs from Functional Prototypes. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages.* Department of Electronics and Computer Science, University of Southampton, 1991. CSTR 91-07.

[173] Kevin Waugh, Patrick McAndrew, and Greg Michaelson. Parallel Implementations from Functional Prototypes. Technical Report TR 90/4, Department of Computer Science, Heriot-Watt University, Edinburgh, 1990.

[174] Ben Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, September 1975.

[175] Åke Wikström. *Functional Programming Using Standard ML*. Prentice Hall, 1987.

[176] S C Winter. The CTDNet2 Graph Reduction Machine. Research report, School of Electronic Systems Engineering, Polytechnic of Central London, UK, 1990.

[177] Peter Woods. Persistent Functional Programming Language. B.Sc. (hons) Dissertation, Dept of Computing and El.Eng., Heriot-Watt University, Edinburgh, UK, 1993.