Synthesising Parallel Functional Programs to Improve Dynamic Scheduling

David J. Parrott

A thesis submitted for the degree of Doctor of Philosophy in the University of London

March 1993
Abstract

This work investigates novel methods for improving the efficiency of evaluating lazy functional programs in parallel. We are specifically concerned with distributed memory architectures in which it is expensive for processors to communicate with each other via message passing.

Traditionally, improvements in parallel evaluation are found by experimental processes supported by intuition and simple mathematical models and much existing research has been based on improving the execution time of a number of small benchmark programs. A key contribution of this thesis is the development of a language for synthesising the low-level run-time characteristics of functional programs. Using the language, it is possible to construct large synthetic workloads in a much shorter time-scale than the equivalent functional programs. From an experimenter's viewpoint the behaviour of the synthetic workloads is more predictable than that of functional programs because it is not distorted by compile-time transformations; it is therefore simpler to obtain specific experimental behaviour. The language enables the granularity of workloads to be altered in a straightforward manner and can model, under controlled conditions, instabilities due to run-time input.

It is noted that functional language compilers ought to be able to estimate the expected time-costs of the individual functions of a program and that the time-cost information can then be used to guide dynamic scheduling strategies. Some research has taken place in this area but no existing research adequately deals with the problems of laziness (or, indeed, instabilities caused by run-time input). We investigate the effect on dynamic schedulers when laziness is not taken into account. The time-cost analysis is then improved to take account of lazy evaluation and we are able to show that the superior time-cost information can improve efficiency.

However, large deviations from mean time-costs due to lazy evaluation and run-time input can precipitate inaccuracies even in the more complex compile-time estimates. We develop a number of heuristic run-time techniques to cope with these inaccuracies and gauge their success by introducing controlled instabilities.
Acknowledgements

Many thanks are due to Chris Clack, my supervisor, whose insights, and editorial guidance have been invaluable throughout this period of research. Thanks are also given to Stuart Clayman with whom I have had many stimulating and fruitful discussions and whose talent for detecting obscure bugs in seemingly perfect code remains unsurpassed. I am indebted to Nigel Chapman, Stuart Clayman, Chris Clack, Simon Courtenage, and Derek Long for giving their time to proof-read this thesis, and for the many useful suggestions they have made. Funding for my research was provided by the Science and Engineering Research Council of Great Britain and the work benefitted greatly from the friendly surroundings of the Department of Computer Science at University College London.

For his unswerving moral support, graphics expertise, friendship, and generally calming influence in times of stress, I am most grateful to my brother Dennis. For a safe and comfortable haven from which to commute each day, and for their patience and enduring support, I give heartfelt thanks to my parents. I am also indebted to my aunt Vi for her financial support and encouragement. Finally, I would like to say a big thank you to the friends who have been responsible for my social well-being these past three years, and especially to Adam for providing a welcome escape to the fens on so many occasions.

Dave Parrott,
March 1993.
Contents

Abstract 2

Acknowledgements 3

Introduction 15
  Goals of the Research .......................................................... 16
  Contributions of the Work ..................................................... 17
  Overview of the Thesis ........................................................... 17

1 Background 19
  1.1 Research Topics for Functional Languages ......................... 19
    1.1.1 Language Design ....................................................... 19
    1.1.2 Program Analysis and Transformation .......................... 20
    1.1.3 Functional Language Implementation ............................ 22
    1.1.4 Support for Functional Programmers ............................ 22
  1.2 Parallel Architectures ...................................................... 24
    1.2.1 General Purpose Hardware ........................................... 25
    1.2.2 Special Purpose Functional Language Reduction Machines ... 25
    1.2.3 Gauging the Success of Special Purpose Hardware ........... 28
    1.2.4 Networked Processors ................................................ 29
  1.3 The Identification and Management of Parallelism ............... 30
    1.3.1 Identifying Parallelism by Hand .................................. 30
    1.3.2 Using Strictness Analysis to Obtain Parallelism in Lazy Functional Languages ........................................... 30
    1.3.3 Skeletons ..................................................................... 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.4</td>
<td>Managing Parallelism by Scheduling, Load Balancing, and Partitioning</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>Simulations and Experimental Stimuli</td>
<td>34</td>
</tr>
<tr>
<td>2.1</td>
<td>Simulations of Computer Systems</td>
<td>35</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Behavioural Modelling</td>
<td>37</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Stochastic Simulation</td>
<td>38</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Time-driven and Event-driven Simulations</td>
<td>39</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Emulation</td>
<td>40</td>
</tr>
<tr>
<td>2.1.5</td>
<td>Discussion</td>
<td>42</td>
</tr>
<tr>
<td>2.2</td>
<td>Generating Experimental Workloads</td>
<td>42</td>
</tr>
<tr>
<td>2.2.1</td>
<td>High-level Source Programs</td>
<td>43</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Deriving Workloads from Execution Traces</td>
<td>45</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Synthetic Workloads</td>
<td>46</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Examples of Synthetic Workloads</td>
<td>48</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Discussion</td>
<td>49</td>
</tr>
<tr>
<td>2.3</td>
<td>Monitoring Experimental Functional Workloads</td>
<td>50</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Measuring Task Sizes—System Implementors Versus Application Programmers</td>
<td>52</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Profiling Tools for Functional Programs</td>
<td>55</td>
</tr>
<tr>
<td>2.4</td>
<td>Summary</td>
<td>56</td>
</tr>
<tr>
<td>3</td>
<td>Modelling Functional Workloads</td>
<td>57</td>
</tr>
<tr>
<td>3.1</td>
<td>Specification of the Problem</td>
<td>57</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Experimental Requirements</td>
<td>58</td>
</tr>
<tr>
<td>3.1.2</td>
<td>The Experimental Environment</td>
<td>59</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Sharing and Delayed Evaluation</td>
<td>61</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Summary of Requirements</td>
<td>64</td>
</tr>
<tr>
<td>3.2</td>
<td>The Paragon Workload Description Language</td>
<td>65</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Task Template Graphs</td>
<td>66</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Synchronising Parent and Child Tasks</td>
<td>68</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Resource Consumption</td>
<td>69</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Describing Sharing and Delayed Evaluation</td>
<td>70</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Optimising Tail-Recursion</td>
<td>74</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Constructing Profilers</td>
<td>110</td>
</tr>
<tr>
<td>4.6</td>
<td>Example Algorithms</td>
<td>110</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Divide and Conquer Algorithms</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>An Example Divide and Conquer Algorithm: Quicksort</td>
<td>112</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Tail-Recursive Loops</td>
<td>118</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Stream Processing</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>Sequential Stream Processing</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>Parallel Stream Processing</td>
<td>121</td>
</tr>
<tr>
<td>4.7</td>
<td>Summary</td>
<td>123</td>
</tr>
<tr>
<td>5</td>
<td>Time-cost Analysis of Paragon Workloads</td>
<td>125</td>
</tr>
<tr>
<td>5.1</td>
<td>An Introduction to the Time-cost Analysis</td>
<td>125</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Dealing with Recursion</td>
<td>126</td>
</tr>
<tr>
<td>5.2</td>
<td>A Simple Time-Cost Analysis without Sharing</td>
<td>127</td>
</tr>
<tr>
<td>5.2.1</td>
<td>First Order Recursion</td>
<td>129</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Extending the Analysis to Tail Calls</td>
<td>129</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Limited Simultaneous Recursion</td>
<td>131</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Extending the Analysis to Task Templates with many spawn instructions</td>
<td>133</td>
</tr>
<tr>
<td>5.3</td>
<td>Reasoning about Sharing Semantics</td>
<td>133</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Probabilistic Events versus Mechanistic Actions</td>
<td>136</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Predicting Interaction Between Task Instances</td>
<td>137</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Evaluation Points and Evaluation Intervals</td>
<td>137</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Some Notation to Describe Lexical, Temporal, and Dependence Relations</td>
<td>140</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Reasoning about the Interface between Parent and Child Tasks</td>
<td>142</td>
</tr>
<tr>
<td>5.3.6</td>
<td>Some examples</td>
<td>142</td>
</tr>
<tr>
<td>5.4</td>
<td>An Analysis of Declared Sequence Evaluation</td>
<td>144</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Simplifications</td>
<td>144</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Temporal Relationships and Probabilistic Events</td>
<td>144</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Determining the Necessity of Formal Parameters and Locally Declared Sequences</td>
<td>145</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Temporal Relationships between Parent and Child Tasks</td>
<td>147</td>
</tr>
</tbody>
</table>
A.8 Program to demonstrate loss of parallelism due to throttling ............ 223

B Experimental Results ................................................................. 224
  B.1 Excessive Throttling can be Counter-productive ....................... 224

C Implementation Details for Lexical Profiling ............................... 226
  C.1 Synopsis of Lexical Profiling .................................................. 226
  C.2 Changes to the Compiler .......................................................... 228
  C.3 The Run-time Mechanisms ...................................................... 230
    C.3.1 Instantiating Profiled Supercombinators ............................. 233
    C.3.2 Instantiating Unprofiled Supercombinators ....................... 234
    C.3.3 Call-count Profiling ....................................................... 235
    C.3.4 Space Profiling ............................................................ 235
    C.3.5 Time Profiling ............................................................. 236
  C.4 A Single-Stepped Example of Lexical Profiling ....................... 237

Bibliography ................................................................................. 241
List of Figures

1.1 Four aspects of functional language implementation. ......................... 22

2.1 Using a simulation to simplify design and experimentation. ................. 35

2.2 Four components of a simulated distributed system. .......................... 36

2.3 A single node in a time-driven simulation network. ............................ 40

2.4 Event-driven simulation. ................................................................. 41

2.5 How applications programmers and systems programmers relate functional
programs to their run-time behaviour. .................................................. 54

3.1 The experimental environment. ......................................................... 61

3.2 A graphical example of non-strictness and sharing. ............................ 63

3.3 A directed, cyclic graph of task template definitions are expanded into a
call-tree of task instances at run-time. .............................................. 66

3.4 A first attempt at describing the template graph of Figure 3.3(a). .......... 67

3.5 A concrete interpretation of abstract work units. ................................ 82

4.1 The most general combination of spawn and block instructions. .......... 90

4.2 Equivalent parent/child synchronisation in a functional program. ....... 91

4.3 An example of shared computation in a Paragon model. .................... 96

4.4 Call tree depicting possible origins of declared sequence, \( Q_\infty \). .... 103

4.5 Paragon simulation of a divide and conquer algorithm. ....................... 112

4.6 Best-case and worst-case call trees for quicksort. ............................ 114

4.7 The distribution of total calls to the filter function. .......................... 116

4.8 The distribution of total calls to the ++ function. ............................. 117

4.9 A Paragon model of a tail-recursive loop with parallel sub-tasks. ....... 119

4.10 A Paragon model of a sequential pipeline. ...................................... 120

4.11 Timing diagram for a parallel pipeline of four functions. .................. 122
4.12 A Paragon approximation of a pipeline of interleaved functions................. 123
5.1 A small Paragon program with mutually recursive task templates.............. 128
5.2 The graph of the Paragon program shown in Figure 5.1................................. 128
5.3 A series of block instructions acting on a child task instance.................... 140
5.4 Dividing a task template into its evaluation intervals............................... 141
5.5 An example of a recursive Paragon program with declared sequence eval-
uation and parameter passing.............................................................................. 154
5.6 The graph, G, representing the Paragon program shown in Figure 5.5........... 155
5.7 Expanding G to G* by a collecting interpretation......................................... 156
5.8 Expanding G* to G' by path analysis............................................................... 157
5.9 The final mapping of costs to the original program graph............................ 158
5.10 Task templates with inconsistent tail-call modifiers.................................... 159
5.11 Coping with inconsistent tail-call modifiers................................................ 159
6.1 Distributing work to parallel processing elements....................................... 166
6.2 Alternative memory organisations for GRIP.................................................. 172
6.3 Asymptotic time-complexities for two algorithms......................................... 175
6.4 Exporting tasks in descending order of their time-costs.............................. 177
6.5 Compile-time scheduling may fail when tasks suspend................................. 179
6.6 Global scheduling outperforms localised scheduling when tasks suspend. . 181
6.7 The disruption of scheduling mechanisms by sharing.................................... 182
6.8 An effect of sharing on schedulers which use time-cost analysis.................... 184
6.9 Improving Dynamic Scheduling by Accounting for Sharing....................... 185
6.10 Tasks made unstable by sharing and run-time input..................................... 186
6.11 The equivalence between foraging insects and PEs looking for work............ 188
6.12 A possible source of intransitive time-cost comparisons............................. 189
6.13 Mean time-costs calculated for Program A.7................................................. 193
6.14 Risk aversion is at least as good and can be better than scheduling with
mean time-cost information.................................................................................... 194
6.15 Increasing the number of PEs per physical processor to improve latency-
tolerance.............................................................................................................. 197
6.16 The effects of varying the architectural configuration................................... 200
6.17 Mapping the technique to 1 PE per physical processor with many task pools per PE. .................................................. 203

B.1 Loss of parallelism when the throttling threshold is too low. .......... 224

C.1 The profiling framework. ................................................................. 228
C.2 An example of an unprofiled function which is shared by two profiled functions. ......................................................... 229
C.3 Duplicating the shared function. ......................................................... 230
C.4 Profiling the shared function in its own right. ..................................... 230
C.5 A naïve attempt at profiling lazy, higher-order functions. ................. 231
C.6 Lexical profiling using boxed arguments. .......................................... 232
C.7 A fully augmented graph cell. ............................................................ 232
C.8 Instantiating a profiled supercombinator. ......................................... 234
C.9 Instantiating an unprofiled supercombinator. .................................... 235
C.10 Lazy, higher-order profiling. ............................................................. 238
List of Tables

3.1 Paragon ranges ................................................................. 78
3.2 BNF syntax table for Paragon ............................................. 81
4.1 Counting primitive operations using lexical and dynamic profiling styles ............................................. 109
4.2 Summary of quicksort profiles .......................................... 113
4.3 Normalising the beta distribution .................................... 117
5.1 An initial set of qualified $T_i \rightarrow T_j$ symbols ................. 143
6.1 The real costs of executing the tasks given in Figure 6.7 .......... 183
B.1 Profile results for quicksort ........................................... 225
C.1 A summary of colouring information ............................... 240
Introduction

Declarative programming languages are designed for describing what is required of a programming solution rather than how to perform the low-level computations which ultimately achieve the solution [DGP91]. In particular, functional languages liberate the programmer from such concerns as memory management and the undesirable properties of side effects [Hug89]. The lack of side effects ensures that an expression can be replaced by its value at any time during the execution of a program without altering the final outcome. This feature, known as referential transparency, is one of the reasons that functional languages are promoted as good tools for programming parallel and distributed systems [Bur90a].

In the absence of side effects, concurrent threads of execution can be constructed automatically, without the need for explicit synchronisation and communications annotations in the source code. This allows programmers with no special knowledge of the underlying architecture to write parallel programs. Functional programs which execute on sequential processors should execute on parallel processors without modification, although to make best use of the hardware, inherently parallel algorithms are recommended. The importance of referential transparency, implicit memory management, and automatic synchronisation in reducing programming complexity is apparent when parallel programs written in functional languages are compared with those written in parallel imperative languages such as Occam [Ker87].

Most modern functional languages have lazy evaluation semantics. That is, an expression is evaluated only when it is guaranteed that the result of the evaluation is needed. This is a powerful tool for programmers because it allows algorithms to be expressed in elegant and novel ways [BW88, Hug89]. However, laziness is one of the greatest challenges for researchers who wish to build efficient implementations of functional languages. Understanding and coping with the effects of laziness play a large part in the work presented
in this thesis.

Goals of the Research

The ultimate goal of our research is to provide improved dynamic scheduling techniques for a newly developed parallel reduction machine called DIGRESS [Cla92a]. DIGRESS has a loosely-coupled architecture. To achieve this goal it is necessary to make a careful study of the way that functional workloads behave on loosely-coupled parallel machines. The complexity of automatically scheduling lazy functional workloads is especially challenging.

It was recognised in the initial stages of the research that it is unrealistic to develop many large-scale functional programs specifically for the purpose of providing workloads to stimulate the scheduler. This is a common problem faced by many researchers when attempting to evaluate the success of an infrastructure for executing functional programs in parallel. It is often the case that a limited number of trivial test programs are used to demonstrate general properties of implementations. We therefore attempt to provide a fast mechanism for building workloads by synthesising the run-time behaviour of functional programs. The aim is to be able to construct workloads whose behaviour is well-defined, and which take only a fraction of the time required to build real functional programs.

One of the primary methods proposed for the design of scheduling algorithms is to derive time-cost information at compile-time to guide decisions made at run-time. Recall that we are dealing exclusively with lazy functional workloads. Existing time-complexity analyses often claim to deal with lazy languages [BH89, San90, Wad88] but in truth they are limited to call-by-name languages because they fail to take sharing semantics fully into account [Mah90]. One of our aims is to exploit the fact that we have an abstraction of functional language behaviour to devise a time-cost analysis of synthetic workloads that will make sensible approximations in the presence of non-strictness, sharing, and parallel evaluation. We then intend to use the results of the analysis to investigate the utility of the improved information for scheduling purposes.

Finally, given that even the best compile-time analysis cannot be expected to have detailed knowledge of run-time inputs to the workload, we aim to develop and test a number of heuristics for making the use of time-cost information more reliable.
Contributions of the Work

In realising the goals stated above, the following contributions are made to research in parallel implementations of functional programming languages:

- We develop a method for rapidly constructing well-defined, low-level workloads with which to stimulate experimental architectures.

- Time-cost analysis of workloads in the presence of sharing semantics is explored. We are able to comment on the suitability of this analysis for improving scheduling algorithms, and consequently the efficiency of program execution.

- Scheduling algorithms for use with the chosen parallel architecture are designed and tested.

- The experiments conducted with scheduling algorithms and synthetic workloads enable us to quantify the minimum acceptable workload granularity for efficient parallel execution of functional programs to be achieved on our chosen architecture.

- A technique for profiling lazy functional programs is implemented. We use the resulting profiler to characterise real functional workloads in order to construct realistic synthetic workloads. We demonstrate the strengths and weaknesses of two different types of profiling tool in this context.

Overview of the Thesis

The main topics of this research are the design and analysis of synthetic functional workloads, and their use in developing new scheduling strategies for the DIGRESS parallel architecture. In Chapter 1 we provide a general background to current research topics in functional programming. We also discuss parallel architectures and to the way in which parallelism within functional programs is identified and managed. In Chapter 2 the discussion concentrates on the topics of simulating experimental computer systems and their workloads. Methods commonly employed to record the results of the associated experiments are also explored in this chapter.

In Chapter 3, we develop a workload description language called Paragon which will be used to provide synthetic workloads that model key features of the behaviour of real, lazy functional programs. The Paragon language forms the basis for much of the work
in the remaining parts of the thesis. In Chapter 4 we provide evidence that Paragon workloads perform as expected and demonstrate examples of synthetic workloads that behave in a similar manner to a number of common functional language constructs. A profiling tool for higher-order, lazy functional languages was developed to support this work, the implementation details of which are presented in Appendix C.

Chapter 5 is devoted to the theory of analysing lazy workloads at compile-time. We start with a small subset of the Paragon language which bans lazy evaluation and move towards an almost complete subset where laziness is tackled by a series of transformations on the original program graph. In Chapter 6 a number of dynamic scheduling algorithms are presented. These are related to existing algorithms and improvements are depicted graphically. Finally, in Chapter 7 the work is reviewed and conclusions are drawn.

At the conclusion of each chapter, a summary is given, including proposals for further work where appropriate. The proposals are reiterated, briefly, in the review of Chapter 7. Related work is discussed in the introductory chapters and in the introductions to each major section of the thesis. A summary of related work with references to the more complete discussions is given in Chapter 7.
Chapter 1

Background

To put our research into context, and to explain some of the terms used in the introduction, this chapter provides a brief description of the sort of research that is currently being conducted in connection with functional languages. General purpose parallel architectures and architectures constructed specifically for executing functional programs are also discussed. In the final part of the chapter, we deal with methods for identifying and managing parallelism within functional programs.

1.1 Research Topics for Functional Languages

Research into functional programming languages can be arranged into a number of broad categories such as language design, program analysis, functional language implementation, and programmer support. The work reported in this thesis is related to all four of these categories.

1.1.1 Language Design

There are many functional languages in existence and yet more currently in development. Some of the earlier languages were constructed primarily to learn about the nature of functional languages per se and to experiment with compilation techniques (e.g. Ponder [Fai85] and FP [Bac78]). Other languages such as ML [MCP93, HMM86, Wik87] and Haskell [HPJW+91] have been designed more with the end-user in mind, but have also been the subject of vast amounts of theoretical research. Haskell is the most recent of the large functional languages and its design is still incomplete at the time of writing. Its main innovation is a general purpose overloading mechanism [HPJW+91, NS91, Ber92]
which allows the same function name to be re-used for different types of input, with the support of a meta-type (or class) mechanism [HB91]. Novel features, such as functional arrays [HS85] have also been included into the Haskell definition.

It has long been recognised that functional languages are both elegant and concise. This is partly due to the nature of the lambda calculus on which functional languages are based and partly due to the efforts of language designers to ‘sugar’ the lambda calculus. For example, the following notational style is typically used to construct a list of values related by generator functions and constraints:

\[
[ (x, y) \mid x \leftarrow \text{ordinates}, y = f(x), 0 \leq x \leq 20 ]
\]

This expression describes a list of coordinate pairs \((z, y)\), based on the equation \(y = f(z)\). The expression very closely resembles the mathematical set notation

\[
\{(z, y) \mid z \in X, y = f(z), 0 \leq z \leq 20\}
\]

for the set of ordinates, \(X\).

The above is an example of a design feature which aids programming ‘in the small’. Another aspect of language design is the consideration of programming ‘in the large’; i.e. large scale issues such as the management of program modules and their associated interfaces. Both the ML and Haskell design committees have paid much attention to this, in order to make the languages attractive for serious software development employing teams of programmers.

For our research we are concerned primarily with the behaviour of functional programs at run-time rather than their high-level descriptions. When a program is compiled and executed it becomes a workload. The programming languages mentioned above are all used to provide semantic descriptions of the workload. Later (in Chapter 3) a language is developed which aims to provide pragmatic descriptions of workloads, paying no attention to their semantics. These will then be used in place of real functional workloads to drive our experiments.

1.1.2 Program Analysis and Transformation

There are potentially unlimited ways in which computer programs can be analysed. Functional programs are especially amenable to analysis because of their mathematical basis [FH88]. Some examples of analyses that can be performed on functional programs are:
Strictness analysis We attempt to discover which arguments of a function are definitely going to be required when the function is executed [CPJ85, HY86, JL89b, DW89]. This can improve efficiency because closures do not have to be constructed to represent the arguments in their unevaluated state [Myc81]. Strictness information is also important for obtaining parallelism (see Section 1.3, below).

Time-complexity analysis Predictions of the algorithmic complexity of a program [BH89, Le 85, Le 88, Ros89, San90, Wad88, Weg75] can help to determine the success or failure of a program transformation that was meant to improve efficiency. Also, if a measure of the absolute costs of the sub-components of a program can be determined then parallel execution can be made more efficient [Mah90] (see Chapter 6).

Sharing analysis Knowing whether or not the result of a computation is shared [Gol87] allows a compiler to optimise the object code for a program [BPJR88, FW87]. Knowledge of sharing properties can also be used to improve garbage collectors [Hud86]. We use sharing information in Chapter 5 to improve the quality of information derived from a time-cost analysis.

Partial evaluation A functional program which takes no run-time input can be evaluated in full at compile-time. Most functional programs do, however, read input at run-time but portions of many programs can be isolated from run-time input and can often be evaluated by the compiler [BHY89, HG84, Lau91, HS92, NN89, WCRS91].

And there are many others. The analyses most relevant to this thesis are those for time-complexity and sharing. A time-cost analysis for the synthetic workload language of Chapter 3 is developed in Chapter 5. This analysis takes sharing into account and the results are used to test dynamic scheduling algorithms of Chapter 6. We shall develop a number of workloads in Chapter 6 which correspond to workloads generated by functional language compilers. They represent the final output of the compiler and it is assumed that no further transformations take place.
1.1.3 Functional Language Implementation

Figure 1.1 shows the relationships between four aspects of functional language implementation. In this thesis we are mainly concerned with the aspect of parallelisation and, more specifically, the development of run-time strategies for improving the efficiency of parallel execution. The remaining aspects of theory, compiler technology, and hardware design under-pin the work on parallelisation. For our research we shall be using a newly designed architecture, based on a network of distributed workstations. To illustrate the context of the work, we shall examine a number of hardware configurations used for the execution of functional programs in Section 1.2. Methods for exploiting parallelism are presented in Section 1.3.

1.1.4 Support for Functional Programmers

Until recently, programmer support for functional languages was a poorly considered area. Far more effort has been expended on language design and compilation than has been apportioned for user support. With language maturity, however, comes the desire to improve the environment in which the language is used. Here, we look briefly at the issues of debugging and profiling.
Debugging

Unlike imperative languages where it is possible to insert 'print' statements to follow the actions of the program, functional languages are side-effect free and produce all of their output as the result of the whole program. An attempt to add side-effect output to functional programs in a referentially transparent manner was made in [PC90] but quickly abandoned because of the difficulty in interpreting the lazily generated output. It is often the case that the side-effect style of output bears scant resemblance to the order envisaged by the programmer because of the evaluation order imposed by lazy evaluation semantics [HHO90].

Another example of a debugging tool is that of [HO85] which provides a method for transforming a functional program into an equivalent program which carries debugging information in the form of shadow variables. Using this, traces of the execution can be constructed and examined on completion of the whole program or when breakpoints are reached. This work also suggests the use of an interactive environment in which functions can be redefined and tested an arbitrary number of times. An interactive environment is offered by many modern functional language compilers. The Standard ML of New Jersey compiler [TD90] and the Haskell [HPJW+91] compilers from the universities of Yale and Chalmers are now packaged with facilities for intelligent program editing, compiling, execution, and debugging via the GNU Emacs editor [Sta86].

More recently, algorithmic debugging has come to the fore [NF92]. Here, trace information gathered for a program run is used to drive a question and answer session to determine where an error might have occurred in the design of the algorithm. At each stage, the user is presented with a portion of the call-tree for the program, and the known results at that stage, and has the option of descending further into the call-tree if the results are incorrect or moving back up the tree if expected results are found. In this way, errors in the algorithm can be pin-pointed. The main problem with the technique is the large volume of information presented to the user [HHO90].

Profiling

Profiling differs from debugging in that summaries of a program executions are presented to the programmer rather than intermediate results, or complete traces. Traditionally, profilers report memory usage, execution times, and the number and variety of function-
calls made (e.g. [GKM82]). For higher-order, lazy functional languages there are many
problems associated with profiling. Consider, for instance, the possibility that a computa-
tion is described within the definition of one function, $A$, but at run-time is evaluated by
another function, $B$, (a very common occurrence under lazy evaluation). We must decide
whether the function-calls, memory-usage, and time taken to execute the stray compu-
tation should be attributed to $A$ or $B$ [CPC91]. Higher-order languages cause problems
for profiler implementation because it is not known at compile-time which function is to
be called when higher-order variables are used [ADM88].

Currently, research into profiling functional languages is particularly active. At
present, even the interface which allows the programmer to specify what is to be profiled
has not reached a steady state. The profiler of [SPJ92] chooses to annotate individual
expressions in the source code (in a manner similar to that of [PC90]) while that of
[CPC91] chooses to profile on function boundaries, reporting on functions whose names
are specified directly to the compiler. Both of these profilers report their measurements in
close correspondence to the source code (i.e. for the above example the cost of executing
the stray expression is attributed to function $A$).

The work by Runciman and Wakeling [RW90, RW92] has paid close attention to the
study of memory usage, making a clear distinction between the functions which produce
data objects and the constructor functions which define the shape of the objects. Also,
for the example above, the memory used by the stray expression is reported according
to the locality of its execution, thus attributing the memory usage to function $B$.

It is clear that much further work is required to standardise functional language
profilers, and to bring together the innovations that have been developed separately at
disparate research sites.

Profiling tools have an obvious use for the construction of synthetic workloads: we
wish to ensure that the pragmatic descriptions specified are equivalent to those generated
by real functional programs. This is demonstrated in Chapter 4 and implementation
details for the profiler described in [CPC91] are given in Appendix C.

1.2 Parallel Architectures

In order to experiment with the execution of functional programs in parallel it is first
necessary to choose a suitable architecture on which to base the research. Parallel com-
putations may belong to either of the following categories:

**Process-parallel** — a MIMD (multiple instruction, multiple data) architecture is required. Programs are decomposed into separate tasks whose control flow is mutually independent. Data dependencies may exist between tasks which imply synchronisations, but the execution of separate tasks is otherwise independent.

**Data-parallel** — a SIMD (single instruction, multiple data) architecture is required. In this case, complex data structures provide the vehicle for parallelism. A single instruction is applied simultaneously to a subset of the components of a data structure [HS86].

For certain special purpose parallel computations there is the possibility of using more exotic hardware such as systolic arrays and associative memory which implement fixed algorithms on data whose structure is pre-defined [QD84]. For the most part the work of this thesis is concerned with MIMD parallel architectures.

### 1.2.1 General Purpose Hardware

There are now many examples of general purpose parallel machines, ranging from multiprocessor workstations, for which individual programs have little or no control over the exploitation of parallelism, up to the large vector processing machines such as the Cray X-MP, software for which requires extensive vectorisation work [BH92]. Functional language implementations have been reported for the Intel iPSC hypercube [GH86, Gol88] and for transputers [Bur90a, KLB91, GWW89, SK84, MS90, MS87, BBC+90].

### 1.2.2 Special Purpose Functional Language Reduction Machines

In the 1950s and 1960s, high level languages were developed in response to a crisis in software construction (e.g. Lisp (1959), Fortran (1954–57), Cobol (1959–60), and Algol (1958–68) [Bar88, Wex81]). From then until the present day new languages have been constructed to solve different classes of problems. The primary purpose of the earliest high-level languages was to replace assembly language programming. Consequently, these languages encourage programmers to express algorithms in an imperative style which is similar to that of assembly code because compilers can more easily construct efficient assembly programs when the high level imperative source program behaves in a similar manner to the target program.
Functional programs pose new problems for compiler writers because they provide new facilities such as higher-order functions, delayed evaluation, and sharing semantics. At the same time, functional languages ban many of the features traditionally found in imperative languages such as assignment statements. The task of mapping lazy, higher-order functional languages to a machine language instruction set that is inherently imperative can therefore be complex and subject to inefficiency. A popular solution has been to construct special purpose hardware whose low level instruction sets are targeted towards the requirements of functional languages so that the mapping from high level functional language to machine level code is simplified. Examples of special purpose architectures are described below.

The SKIM Reduction Machine

The SKIM reduction machine [CGMN80] (and later, SKIM II [SCN84]) is an example of a SISD processor. It is not a parallel machine but is one of the earliest attempts to build specialised hardware for functional languages. It was designed at Cambridge following Turner's idea [Tur79] that S, K, and I combinators could be used as a low level representation of functional programs. The operations of a small set of combinators are microcoded into the hardware so that the machine language has applicative characteristics and is more suitable for the execution of functional programs than traditional hardware. Further optimisations centre on the way that memory is organised. Each memory cell contains a head and a tail part, which provides a natural data structure for functional language implementation, and a single bit that is used for garbage collection (a single bit is sufficient for mark/scan and one-bit reference count garbage collectors [Coh81]).

NORMA

The NORMA reduction machine [Sch86] is similar in nature to SKIM. It is a SISD architecture with specially configured memory to support data structures (all memory cells contain head and tail fields), garbage collection, and graph reduction techniques such as pointer reversal [CPJ86, PJ87b]. The machine is microprogrammable and the initial graph reduction implementation is based on extended Turner combinators. A novel feature of this architecture is that special hardware is provided to inspect status (or tag) bits in the memory cells to determine the next operation to be performed on
the graph. Traditionally, this testing imposes a significant overhead on graph reduction [PJ87a].

COBWEB

The COBWEB architecture [HOS85, AHK+87, BBK87] is designed to utilise the VLSI fabrication technique of wafer-scale integration. Normally, many identical integrated circuits are constructed on a single wafer and separated, tested, and packaged as individual components. With wafer-scale technology, the separation does not take place: processors constructed on a wafer remain in their original configuration and the whole wafer is used as a fine-grained parallel machine. The COBWEB reduction mechanism uses variable-sized tokens to encode function definitions, and final and intermediate results of reduction.

Alice and Flagship

The Alice project [DR81, CFR86] concentrated on the implicit parallelism within declarative programming languages. Programs are represented as a graph where each node is a packet containing a function applied to a vector of arguments. The packets are augmented with a number of status entries which determine how they are to be executed. The architecture consists of a complex interconnection (switching) network attached to a number of processing agents. The system is controlled by a central, shared processor responsible for the distribution of work between processing agents. A prototype Alice was implemented using transputer chips microcoded in Occam; the intention was to move to custom designed processors at a later stage.

The Alice project was later superseded by the Flagship project [WWW+88, WW87b, Det86]. This was also influenced by the Manchester Dataflow Computer [GKW85, Gur85], an architecture designed for the parallel evaluation of data-driven computations. Flagship continues to use the packet-based representation of functional programs designed for Alice but the hardware is modified so that memory is closely coupled to the processing elements in order to reduce the bottleneck of the interconnecting switching network. The Flagship hardware is designed to execute programs written in a number of different styles including functional, dataflow, and logic languages.
The GRIP Multiprocessor

The GRIP architecture [PJCSH87] was designed at University College London. It consists of a number of parallel processing elements, each with a small amount of local memory, and a number of intelligent memory units (IMUs), closely coupled via a futurebus interface. A GRIP machine consists of a number of boards, each of which contains four processing elements and a single IMU. Fast communication is achieved between the units of a single board via an internal bus. The IMUs are the main innovation of GRIP; rather than limiting memory operations to data storage and retrieval, the units are microcoded to perform high level tasks such as graph manipulation. This leads to interesting design decisions regarding the balance of intelligence between processing elements and IMUs.

Machine à Réduction Symbolique (MaRS)

The MaRS project [CCC88] combines many of the features of the architectures listed above. Like SKIM it uses a combinatorial machine language; like Alice it employs a tightly coupled interconnection network; and like GRIP the machine contains a number of memory processors which are intelligent enough to carry out dynamic allocation and garbage collection activities. The processing and memory nodes communicate via message passing. MaRS therefore resembles a distributed memory architecture.

1.2.3 Gauging the Success of Special Purpose Hardware

The special purpose graph reduction machines listed above have all been used as the basis for advanced research into graph reduction techniques (e.g. [Rob89, WW87b, HPJ91, HPJ92]). The GRIP multiprocessor, now at Glasgow, has even been made available to the wider research community for the parallel execution of experimental programs [HPJ90].

However, none of the machines have been adopted by a major manufacturer for mass production. There are many reasons for this, one of which is that it is time consuming (and therefore expensive) to adapt existing applications to run on the new hardware. Gradual evolution is preferable to sudden changes and is the reason that C is able to replace Fortran as the most widely used, general purpose, computer language. Many of the original Fortran libraries were easily converted to C and in some cases C and Fortran may be interfaced together so that the original libraries can be used without modification.
The move to specialised hardware is drastic and, unsurprisingly, is not well received by end users.

1.2.4 Networked Processors

Another type of parallel architecture is a set of processors loosely-coupled via a local area network (LAN). This is an interesting area because many educational, commercial, and industrial sites possess large numbers of processors connected via LANs, primarily because the connectivity allows physical resources such as file servers and printers to be shared between the processors. LANs also provide general communications facilities between the processors allowing, for example, a user process to be executed on a remote processor while another process is executed on the local processor. This is coarse grained parallelism requiring careful management of communications overheads.

Networks of workstations have been used to emulate parallel machines. For example, the DAPS project [Hud84, HG84] used workstations to emulate the topology of a special purpose hardware design. The DIGRESS project [Cla92a], whose LAN-based architecture is described in Chapter 3, moves beyond emulation towards using a network of workstations as a real parallel machine. Research effort is thus expended to optimise program execution for the network of workstations rather than for the machine being emulated by the network. One of the DIGRESS project's aims is to utilise idle workstations (especially overnight) for applications that would otherwise require access to large mainframe or supercomputer facilities.

Communication across local area networks is slow in comparison to tightly coupled parallel processing. Moreover, errors in transmission may occur and, for general purpose LANs, there will be other traffic using up the available bandwidth. Latency periods are therefore less predictable than for bus-bound communication. However, the advantage of gaining large scale parallel processing virtually for free is attractive enough to encourage further research.

Much of this thesis is concerned with the provision of synthetic, functional language workloads. These allow the issues raised by high and unpredictable latency to be explored in a systematic manner.
1.3 The Identification and Management of Parallelism

Given that functional programmers may be unaware of the low-level aspects of program execution, it is evident that support must be provided by the compiler and possibly some run-time system code to identify parallelism. It has been noted that to achieve reasonable speed-ups on parallel processors, parallel algorithms should be employed [Bus87, Roe89, HPJ92]. To supplement algorithmic parallelism, a great deal of research has been directed towards attempting to extract implicit parallelism from functional programs via methods such as strictness analysis (e.g. [CPJ85, HY86, BH86, WH87, JL89b, DW89, Wad87b, Hun90, Noc90, Jen91, etc]).

Irrespective of how parallelism is obtained from functional programs, another interesting and challenging problem is how to map the resulting threads of execution onto the available processing elements. Unlike more general problems such as the analysis and transformation of functional programs, the mapping issue is very closely related to the underlying architecture. There is a likelihood that different behaviour will be obtained when different architectures are studied. It is essential, therefore, for implementors of functional systems to develop a thorough understanding of the way that parallel functional programs interact with specific architectures.

1.3.1 Identifying Parallelism by Hand

One of the simplest techniques for identifying parallelism within a functional program is to use annotations and functions with special parallel evaluation properties [Bur84, HPJ92, Kel87, Roe89, PJ89]. This places the onus directly on programmers to find the parallelism within programs. We have already mentioned in Section 1.1.4 that the behaviour of functional programs is often counter-intuitive for many programmers. Therefore, explicit specification of parallel activity can be very difficult to get right [Cla93].

1.3.2 Using Strictness Analysis to Obtain Parallelism in Lazy Functional Languages

In parallel implementations of lazy functional languages, strictness information plays a large part in determining when parallel evaluation can take place [CPJ85]. Strictness information specifies that a value will definitely be required at some time in the future and thus allows the lazy, demand-driven evaluation mechanism to be short-circuited.
This improves the efficiency of graph reduction because fewer closures are required to represent unevaluated objects. It also aids parallelism. All of the strict arguments of a function can be evaluated in advance of (or concurrently with) the evaluation of the function itself and, because referential transparency ensures that the arguments can be evaluated in any order, they can be evaluated in parallel with each other.

Modern strictness analysis techniques are able to approximate the extent to which non-flat data structures such as lists can be evaluated strictly [JL89b, Wad87b]. List structures are heavily used in functional programming, hence it is useful to classify their strictness properties as accurately as possible, e.g.:

- A function which needs to inspect the contents of every list cell that it visits is said to be **head strict** in that list.
- A function which needs to inspect the whole list structure (but not necessarily the contents of the cells) is said to be **tail strict**.
- A function is **hyper-strict** in its argument if every sub-component of the argument is to be evaluated in full (this definition applies to all types of a structured arguments, not just lists).

The hyper-strictness property is especially useful for parallelising programs which employ pipelines of functions of the form \( f_n(f_{n-1}(\ldots f_1(\text{list})\ldots)) \). If the functions \( f_1 \ldots f_n \) are executed lazily then a large amount of synchronisation takes place because every component of the result is demanded individually and calculated only upon receipt of the demand [PJ87b]. The demands must pass all the way along the pipeline before the intermediate results can be calculated and passed back. If the values are hyper-strict then the functions can execute more efficiently in parallel because the calculation of the whole structure begins immediately at each stage of the pipeline (see Section 4.6.3).

There are, of course, many applications for which these simple classifications are insufficient and it is often possible to build a more complete strictness profile of complex data structures [HW87, JL89b]. Recent innovations using projections [WH87, DW89, Hun90] (based on continuations [Hug87a]) are leading to a better understanding of the problem and better classifications.

Most strictness analysis techniques are based on abstract interpretation [FH88, Chapter 20] or projection analysis\(^1\) over finite domains. Unfortunately many recursive data

\(^1\)Interesting comparisons between the two methods are given in [Bur90b] and [DW90] (and an alter-
structures can only be described by infinite domains and therefore approximations have
to be made. The situation can be improved, however, if contextual information is taken
into account at run-time. Burn proposed evaluation transformers [Bur87, Bur90a, Bur91,
KLB91] which deal with non-flat data structures (specifically lists) and allow the strict­
ness information held about an object to be improved at run-time when more is known
about its context.

1.3.3 Skeletons

One method of managing parallel behaviour is provided by skeletons [DFH^]. These are
predefined templates with well-defined function semantics and also well-defined behaviour
for a range of parallel architectures. It is intended that programmers code with the
skeleton library in mind, expressing algorithms so that they correspond to the form of
one or more skeletons. The compiler knows about skeletons in advance and is able to
generate code for efficient parallel evaluation. Skeletons express notions such as pipelines
of processes, divide and conquer algorithms, and data-parallel operations [HS86].

For some skeletons, at least, a degree of knowledge of the underlying architecture is
required of the programmer in order to select the most appropriate skeleton. To ease
this, work is taking place to transform between skeletal forms [DFH^].

1.3.4 Managing Parallelism by Scheduling, Load Balancing, and Parti­
tioning

For programs to execute efficiently in parallel, both interprocess communications and the
amount of time that each processor spends idle must be minimised. A processor becomes
idle when it runs out of work and has either to search for more work or to wait for work to
be sent, according to the chosen load balancing strategy. Work is supplied to processors
in the form of tasks.

The three topics of load balancing, partitioning, and scheduling are closely related: all
three are concerned with obtaining optimal execution times on parallel machines. Load
balancing algorithms (e.g. [Hud84, Sar87]) attempt to ensure that every processor in a
parallel machine has an equal amount of work to perform at all times. Partitioning (e.g.
[Gol88, HG85]) is the art of determining the subdivisions of a program to be executed as
native method is proposed in [Noc90]).
parallel tasks. Scheduling (e.g. [CK88, Mah90, BBC90, MS90, PJ89, HPJ92, KLB91]) is the ordering of run-time activities so that the most efficient behaviour is obtained.

Each of these can be managed statically, at compile-time, or dynamically, at run-time. Typically, there is more information available at run-time with which to make educated decisions, but at the sufferance of run-time overheads for the gathering, storage, and analysis of the information. We deal with this problem in more detail in Chapter 6.
Chapter 2

Simulations and Experimental Stimuli

Modelling and simulation are effective ways to explore both software and hardware design spaces without having to commit resources to the construction of real prototypes [RS92]. By simulating a system, researchers can enter into the experimental design cycle shown in Figure 2.1 which leads to a better understanding of the problem before a real implementation is attempted (e.g. [HOS85, AHK^87]). Booch states in [Boo91] that:

"...most software systems are highly unique, and therefore their developers have only a restricted basis of experience from which to draw. ...the best we can do during the design process is to take a stab at the design, step back and analyze it, then return to the products of the design and make improvements based on our new understanding."

He suggests that the process is repeated until confidence in a correct, complete design is achieved, and describes the whole cycle as round-trip gestalt design. Models allow the initial iterations of the design cycle to take place more quickly because it is possible to simplify the problem and to experiment with successively larger subsets of the final system as experience is gained. Furthermore, models can be distorted in many ways and made to perform beyond the current capabilities of real implementations. This enables the design space to be explored far more comprehensively than would otherwise be possible.

This chapter explores three aspects of experimentation:

1. techniques for simulation [CS89],

2. the generation of experimental stimuli (or workloads), and
Figure 2.1: Using a simulation to simplify design and experimentation.

3. the monitoring of experiments.

For the investigations into dynamic scheduling techniques presented in Chapter 6, we require workloads that typify higher-order, lazy functional programs. We therefore examine the techniques for generating workloads in the context of experimenting with functional language systems. In the section on workload monitoring, particular attention is paid to the way in which different classes of experimenters interpret the activities of functional programs.

2.1 Simulations of Computer Systems

In this thesis we are interested in the use of distributed computer systems, specifically for executing functional programs in parallel. A computer system can be simulated at several different levels of abstraction according to which of its characteristics are to be studied. Figure 2.2 shows four components of a distributed computer system, which can be analytically modelled, simulated in software, or implemented in full.

For a given system configuration we may wish to study the effects of different classes of workloads. In this example it may be best just to synthesise the workloads and to execute a simplified abstract model of computation on the target hardware. Alternatively, if the effects of altering the configuration of the machine are to be studied (e.g. by altering the interprocessor communications infrastructure) then it might be better to simulate the hardware components. There are many reasons for modelling a computer system, including:
Figure 2.2: Four components of a simulated distributed system.

1. To develop a general understanding of how a proposed system configuration will behave prior to its construction (e.g. [WWW+88]).

2. To debug a real system. The system is implemented in full but is known to be problematic. An idealised mathematical model is constructed so that the actual behaviour of the real system can be compared with its expected behaviour. The behavioural model can then be tuned until the observed and expected behaviour are coincident. At this point the functionality of the real system is fully described [Bat88].

3. To allow the system to be monitored without affecting its behaviour. It is well known that as soon as measurement probes are attached to a real system then its behaviour changes. By modelling the whole system, performance measurements can be isolated more easily from the execution part of the model. A good example of this is Hudak's diffused combinator experiment [HG84] where an emulation, originally using a network of workstations, had to be simulated because time-stamping and statistics gathering had too great an effect on the execution profile.

4. To investigate the effect of using an arbitrary number of parallel processing elements. A simulation is not limited by the availability of physical processors.
Therefore, theoretical experiments can be executed with any size of machine (e.g. see [Des89]).

Traditionally, analytical models have been seen as a cost effective way of evaluating computer systems [HL84] but, recently, simulations have become more common because computer time is less expensive. Also, simulations can form the basis of experiments for which the equivalent analytical representation is intractable.

The remainder of this section discusses four common techniques used to simulate computer systems and how the above goals are achieved. The discussion provides a basis for deciding the most appropriate form of modelling for our proposed experiments with dynamic scheduling techniques.

2.1.1 Behavioural Modelling

A behavioural model is a mathematical description of the expected activity in a system. A simple example of a behavioural model can be found in a parser for a computer language. Here, behaviour equates to syntax, and the behavioural model is a set of syntax rules which describe the grammar of the language. If a program contains an error then the parser detects a deviation from the underlying model and is able to analyse the problem and report a sensible error message to the programmer. This model is easily described by a finite state automaton (FSA) that accepts only those input streams which conform to a regular expression corresponding to the grammar [ASU86].

Although finite state automata do not have the computing power of Turing Machines, they can be used to verify an implementation of a Turing Machine given a suitable set of state transition rules (a behavioural model). The input to the FSA includes all read accesses that the Turing Machine makes to its memory. For example, the actions of a sequential Four Stroke Reduction Engine implementation can be described by a simple FSA [CPJ86]. Theoretically the reduction engine has unbounded memory in which the program graph is stored, but there are a finite number of reduction actions. The choice of which action to execute next is partially governed by the contents of the graph so the graph state must form part of the input to the FSA. An FSA cannot replace the reduction engine because it has no ability to make updates to the graph. It can, however, verify the choice of reduction action at each stage of the execution.

The above technique is described in [Bat87a] where an implementation of a computer system is executed and compared to a behavioural model. Again, deviations from the
underlying mathematical model can be analysed and the results used to debug the im-
plementation. For parallel systems the situation is not quite so straightforward. The
execution may still be deterministic but the execution sequences are partially ordered.
Therefore a simple FSA is insufficient to describe the behaviour. In [Bat87b], Bates
addresses this problem with the use of shuffle automata which are an extended form of
FSA, able to cope with partial orderings.

2.1.2 Stochastic Simulation

A stochastic simulation is also based on a mathematical model of the system under inves-
tigation and relies on a sequence of pseudo-random numbers to determine the run-time
behaviour. The components of the system are represented by (stochastic) functions and
the stimuli for independent components are provided by independent random variables.
Typically, the 'computational model' component of Figure 2.2 is simulated by a set of
mathematical relations and the 'processing elements' component is simulated by a queu-
ing model (e.g. [EL86]); this is a hybrid simulation [Mac87], combining experimental and
analytic techniques.

For this type of simulation it is important [Jai91] that the sequence of pseudo-random
numbers used to drive the experiments have the following properties:

1. For a given experiment, the random sequence must have a sufficiently long period
   of non-repeating numbers.

2. Consecutive numbers in the random sequence must be sufficiently independent so
   that the correlation between consecutive numbers is small. For example, it should
   not be possible to state boundary conditions for a number given the preceding list
   of numbers.

3. The distribution of the whole sequence must be sufficiently uniform.

4. The generating algorithm should be efficient.

It is recognised that many of the random number generators supplied as part of the
libraries of proprietary machines are deficient in one or more of the above properties
[Rip87]. It is sometimes necessary to construct a special purpose generator for stochastic
experimentation.
As for the analytical model, it is necessary to characterise the components of the simulated system very carefully. [Sve90] argues that the common practice of assuming statistical independence between random variables is often not justifiable. For example, if a workstation provides virtual memory facilities, the number of page swaps performed (which limits the useful processing time available to user processes and increases the amount of local disc I/O) is closely related to the individual memory requirements of workloads running on the workstation. Such dependencies should be reflected in the stochastic model which describes the loading of the system.

Examples where stochastic simulation is used include the identification of bottlenecks in networked computer systems [Jai91, part VI] and for experimenting with fault tolerance (e.g. [Puc90, BKA90]).

2.1.3 Time-driven and Event-driven Simulations

The processing elements of a distributed computer system can be simulated by time-driven or event-driven processes. By time-slicing the activity of the processes, a single simulation program is able to represent the operation of an arbitrary number of parallel processing elements. Large simulations are likely to require a correspondingly large amount of processing time. Therefore, they are ideal candidates for execution on multiprocessor machines [RS92]. This should not, however, be confused with emulation techniques (see Section 2.1.4) which attempt to match the parallel execution of the model with the parallel execution of the machine being modelled.

In a simulated system, an event is defined to be a change of state in the system. By characterising a system in terms of its state changes, a simulation can be constructed as a set of atomic events which represent those changes. The events are then scheduled so that their behaviour is representative of the parallel system as a whole.

For time-driven simulation, experiments are divided into equal time periods. At each clock tick of the simulated machine a new system state is calculated from the existing state. A simulated parallel computer system, for example, can be represented as a network of functional units whose state is defined by a collection of time-stamped packets, \( P_i \), (also known as tokens) travelling between the units. These will contain workload descriptions and other information required by the chosen computational model.

Figure 2.3 illustrates a single functional unit, \( F \). The global clock-time for the whole simulation is given by \( T_{\text{sim}} \) and the incoming token \( P_i \) is time-stamped with the time,
T_i, at which it was created. When \( T_{\text{sim}} \geq T_i \), the token can be processed by \( F \) and a response \( F(P_i) \) is output. The time-stamp for \( F(P_i) \) is set to \( T_{\text{sim}} + \Delta F \) where \( \Delta F \) is the simulated time for \( F \) to perform its function. When all such functional units in the simulation have been visited, the next clock tick occurs and the process repeats.

In an event-driven simulation of a distributed computer network, each processing element, \( PE_i \), has a current time, \( T_i \), and a list of events, \( (E_{ij}) \), whose execution is pending. In this example we shall denote the time required to execute an event \( E_{ij} \) by \( \Delta_{ij} \) and define \( B \) to be the set of all processing elements whose list of pending events is non-empty.

A single step of the simulation proceeds as illustrated in Figure 2.4. The simulator chooses the processing element \( PE_i \) with the earliest current time, \( T_i \), i.e. such that \( T_i \leq T_j \) for all \( PE_j \in B \). The event \( E_{i1} \) is removed from the head of \( PE_i \)'s list of pending events and is executed. \( T_i \) is then incremented by \( \Delta_{i1} \). The occurrence of an event may cause other events to be inserted into the event lists of the same or another processing element. Therefore, the set \( B \) is redefined after each event is executed. The position at which a new event is inserted into a list is governed by the time-stamps of the new event and of the events already in the list. Execution of events proceeds until the last remaining event is complete and all pending event lists are empty.

2.1.4 Emulation

Simulation requires a considerable amount of work to ensure that the simplified description of the system being modelled is sufficiently accurate. An alternative method of synthesising a system is to emulate it using resources whose operations approximate those of the real system. For example, it is common for transputer networks (e.g. the emulation of [GWW89]) or workstations connected via local area networks (e.g. [HG84]).
to be used to emulate a target distributed architecture. Real inter-processor communications are used, thereby alleviating the problem of modelling the hardware characteristics. Components of the emulator are mapped as closely as possible to components of the real system. This corresponds to emulating the 'hardware/network infrastructure' component of Figure 2.2.

Another level of emulation is to utilise the real target hardware and to emulate the 'abstract machine' component of Figure 2.2. The advantage of using the target environment is that experiments are subject to the external influences experienced by the real system. For example, to emulate the hardware of a time-sharing system, assumptions would have to be made about competing processes. By emulating at the abstract machine level only, real processes will be competing with the experiment.

The processing elements of a hardware or abstract machine emulator differ significantly from those of the stochastic simulation described in Section 2.1.2. For the emulator, processing elements are driven by executable workloads in a similar manner to a real implementation. This is in contrast to the abstract mathematical functions used to drive the stochastic simulation.
2.1.5 Discussion

A number of methods for simulating computer systems have been outlined in this section. Of these, one is to be chosen for conducting experiments on dynamic scheduling techniques for loosely-coupled, parallel graph reduction. Now, the DIGRESS system under active development at University College London and Athena Systems Designs [Cla92a] provides the infrastructure necessary to construct a real distributed processing system using loosely-coupled workstations. Given that we already have a prototype of the target architecture on which the final dynamic scheduling algorithms are to be employed, the method of emulation seems the most appropriate for exploiting our available resources. A significant advantage of utilising the prototype DIGRESS architecture is that the design of the prototype itself can be evaluated simultaneously with the scheduling algorithms. This fits in well with the idea of the iterative design process mentioned at the beginning of the chapter.

Returning to the terminology of Figure 2.2, the chosen experimental framework will use a real network infrastructure and real processing elements. The emulation will be of the computational model. By using the prototype DIGRESS system, we avoid having to characterise the hardware components mathematically.

It is expected that one of the major factors influencing the success or failure of a scheduler (and, indeed, the whole graph reduction project) is the cost of sending messages between processing elements. The costs experienced with an emulator built on the target hardware will thus be identical to those experienced when the final system is constructed. It will be important, therefore, that the speed and granularity of emulation be very close to that of running a real system, otherwise the magnitude of communications costs experienced with the emulation will be disproportionate. A useful corollary of this observation is that we can use the emulator to discover the optimum granularity for workloads. The real system can then be constructed with this in mind (see Chapter 6).

In the next section we discuss methods of generating workloads to experiment within the proposed framework.

2.2 Generating Experimental Workloads

In order to obtain valid results from an experiment, suitable and well considered stimuli are required [Jai91]. Given that one of the goals of our research is to find improved
methods for executing lazy functional programs in parallel, we need to make a careful examination of the nature of the workloads generated from functional programs. For distributed memory parallel systems, functional programs are decomposed into well defined tasks which are executed individually as sequential threads of computation. A test workload is therefore a collection of tasks which, individually, execute as sequential threads and, in combination, perform the actions specified by a functional program.

In this section, three routes commonly used for generating test workloads for functional programming experiments are presented. In the discussion at the end of this section we consider, for each route, the suitability of using the resulting workloads to driving the proposed system emulation.

2.2.1 High-level Source Programs

It is common (especially in functional programming literature) for a small set of high-level programs to be used as benchmarks for measuring the performance of systems. The following are examples of ubiquitous functional language benchmark programs:

- **nfib** calculates the \( n \)th number (for integers, \( n \geq 0 \)) in the series: 1, 1, 3, 5, 9, ... (a variation of the more famous Fibonacci series: 1, 1, 2, 3, 5, ...):

\[
\begin{align*}
nfib :: & \quad \text{num} \to \text{num} \\
nfib \ n = & \begin{cases} 
1, & \text{if } n < 2 \\
1 + nfib(n-1) + nfib(n-2), & \text{otherwise}
\end{cases}
\end{align*}
\]

- **nqueens** finds all possible solutions to placing \( n \) queens onto the squares of an \( n \times n \) chessboard such that no queen occupies the same row, column, or diagonal as any other:

\[
\begin{align*}
nqueens :: & \quad \text{num} \to \text{num} \to [[\text{num}]] \\
nqueens \ n \ 0 = & \ \ [] \\
nqueens \ n \ (m+1) = & \ [ q:qs | qs <- nqueens \ n \ m; \\
& \q <- [1..n]; \ \text{safe} \ 1 \ q \ qs ] \\
\where \\
\text{safe} \ d \ q \ [] = & \ \text{True} \\
\text{safe} \ d \ q \ (x:xs) = & \ q \neq x \ & \ \text{abs} \ (q-x) \neq d \ & \\
& \ \text{safe} \ (d+1) \ q \ xs
\end{align*}
\]
• \texttt{tak} is a tail recursive function whose three arguments are also calculated by recursive calls:

\begin{verbatim}
tak :: num -> num -> num -> num
tak x y z = z, if y \geq x
    = tak a b c, otherwise
    where
        a = tak (x-1,y,z)
        b = tak (y-1,z,x)
        c = tak (z-1,x,y)
\end{verbatim}

• \texttt{pfac} calculates the factorial of a number \( (n! = \text{pfac } n \ 1) \) using a parallel, recursive algorithm:

\begin{verbatim}
pfac :: num -> num -> num
pfac n z = 1, if n < z
    = n, if n = z
    = pfac n (mid+1) * pfac mid z, otherwise
    where mid = (n + z) div 2
\end{verbatim}

• \texttt{sort}; any one of a number of sorting algorithms, often utilising a divide and conquer algorithm such as quicksort:

\begin{verbatim}
qsort :: [num] -> [num]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
    where smaller = filter (<x) xs
        larger = filter (\geq x) xs
\end{verbatim}

Most benchmark programs of this kind are described in just a few lines of code, using a modern functional language, and even relatively complex problems such as \texttt{nqueens} require fewer than about ten lines. One failing with the use of small benchmarks is that, individually, they each exercise limited aspects of the system under test. Another failing is that these programs are often fine grained. Tricks are required to experiment with the effects of granularity, such as defining multiple copies of the same function, only one of which is allowed to spawn parallel tasks (e.g. see [HPJ91]); or adding clauses to the program which allow parallel execution to occur when a predefined threshold is reached [LV91].

The use of larger, more realistic programs for benchmarking has begun to become slightly more popular (see for example [RW92], [BH92], and [CC91]) but has still not reached the maturity exhibited by research into imperative language systems (for example, see [Luc92] where a climate model, electronic circuit simulation, and x-ray tomography are used as test programs). There remains, however, a fundamental problem
with employing real programs. In order to make general statements about the behaviour of a strategy for parallel execution, the behaviour of the test workloads should be fully understood first. It is then possible to qualify the results of performance experiments by describing the type of workload used, and limits beyond which desirable performance is no longer obtained. Qualification is relatively trivial for the simple benchmark programs listed above but not so straightforward for real programs that were originally constructed for purposes other than benchmarking.

A good example of the problem of qualifying functional workloads is given by the degree to which lazy evaluation affects run-time behaviour. In Chapter 6 it is shown that laziness can have a profound effect on the accuracy of compile-time predictions about a workload. To gauge confidence in a prediction it may be useful to express how lazy a program might be; perhaps in terms of the proportion of work shared or avoided due to laziness. For large programs this is not trivial to assess. It is especially difficult to characterise workloads when programs undergo transformations during compilation because the behaviour of the resultant tasks may no longer correspond directly to the original definitions.

[AHPJT91] provides an interesting example of a functional database program that results in a more substantial workload whose parallel operations are relatively well understood. However, this program was crafted specifically for benchmarking purposes and is by no means typical.

2.2.2 Deriving Workloads from Execution Traces

In between using high-level source programs and synthetic workloads (see Section 2.2.3) as experimental stimuli is the technique of tracing real workloads [HL84, Sve90, Des89]. A trace is a record of every action performed during the execution of a real workload on an existing computer system. The demands of the workload on the system are measured (e.g. the number and size of disc reads, or the number of attempted accesses to cache memory) but the response of the system need not be recorded. The trace is used during the subsequent experiments to drive a simulation of a different computer system from that on which the workload was originally executed. The demands of the trace will be identical to those of the original execution but the responses of the simulated system will differ from those of the real system.

This method is guaranteed to give realistic workloads and, because a trace can be a
simplified representation of the original execution profile, it is amenable to straightforward statistical analysis. However, traces still suffer from some of the problems associated with real programs. Firstly, writing the original program to generate a trace is labour intensive and altering a trace requires the source program to be re-written and executed again to construct a new trace from scratch. Secondly, for experiments based on the parallel execution of functional programs, the problem of relating changes in the source program to changes in the operation of the parallel tasks is persistent.

A further disadvantage of using traces is that they occupy a large amount of storage space. This is likely to enforce a physical limit on the size and complexity of the workloads that can be generated [Des89].

2.2.3 Synthetic Workloads

Real workloads may contain more detail than is necessary for the proposed experiments. This involves a degree of unwanted complexity and, when measurement tools are added, will inevitably increase the experimental execution time. This is in direct contravention of the requirement given in Section 2.1.5 that the speed of the emulation is close to that of the real system.

An alternative is to synthesise the workloads, thereby allowing the computational model to be simplified and, it is to be hoped, providing more control over the speed of execution. To make simplifications, the key factors of the real workloads must be identified and characterised. The choice of factors is heavily influenced by the primary interests of the experimenter. It is prudent, therefore, to verify the behaviour of the model against that of real workloads to demonstrate that the simplifications do not distort the experimental results.

Synthetic workloads have many advantages over real workloads. For example, the behaviour of a synthetic workload can be specified explicitly by definition (and is therefore well understood) whereas a real workload may require complex retrospective analysis to determine its characteristic behaviour. Only those aspects of workloads which are relevant to the experiment need to be represented [CGS89] because the primary function of a synthetic workload is to act as an experimental stimulus, not to process input data. This frees the experimenter from the semantic complexity of real functional programs, allowing her to focus, instead, on the pragmatic properties of the workload.

By simplifying the workload specification, compile-time analyses can be developed
with greater ease than would be possible for real workloads. This is especially relevant when the real workload is a functional program which combines lazy evaluation with higher-order function calls (e.g. see the work of [Les89b, LM91, NN89, San90]). Having implemented a new analysis for synthetic workloads, experiments can be performed to determine whether the improvement in the quality of information can be used to improve execution times. If an improvement is observed then there is a strong case to argue for developing an equivalent analysis for real workloads.

It is straightforward to construct a broad spectrum of well understood synthetic workloads, each of which is only slightly different from its predecessor. This encourages thorough experimentation with families of workloads in preference to using a number of unrelated experimental stimuli. The construction of workload families is further enhanced by making synthetic workloads scalable. It is then a trivial exercise to alter the grain size of a modelled program in order to experiment on the effects of granularity. Also, to minimise inter-processor communications in a distributed memory system the task granularity must be large [Gol88, Mah92, PJ89]. Consequently, each task represents many lines of source code. By employing a high-level description language, the effort to describe the tasks of a synthetic workload will therefore be significantly less than that for writing the equivalent source code for real workloads. This also means that synthetic workload descriptions are far more compact than workloads based on traces of real executions, allowing potentially unbounded computations to be specified in a finite space.

For parallel functional programs, the size of tasks generated from a real program is dependent on the partitioning algorithm. To change the granularity of real tasks would therefore require the partitioning program to be re-written. We can use the performance results obtained from experiments with scalable synthetic workloads to guide the future development of real task partitioning algorithms because the behaviour of the synthetic tasks is well defined. Improvements can be achieved by adjusting the partitioning algorithms to obtain real tasks that exhibit similar behaviour. Again, for real tasks it is difficult to experiment with a full range of behavioural patterns because the partitioner must be re-coded for each pattern.

Finally, for real workloads, parallel execution is largely dependent upon having a good parallel algorithm [Amd67, Cul86, Roe89]. A model side-steps this requirement by allowing parallelism to be specified explicitly whenever it is needed by an experiment.
A disadvantage of synthesising workloads is that it is possible to construct workloads which do not correspond to those generated naturally. This could give rise to misleading experimental results and it is important to verify that the experimental workloads are realistic. Tools to aid workload validation are discussed in Section 2.3 and the issue is dealt with in some depth in Chapter 4.

2.2.4 Examples of Synthetic Workloads

User Processes Executing on a General Purpose Distributed System

In [Kun91], Kunz uses executable synthetic workloads to represent user processes running on a general purpose distributed computer. The aim of the experiment is to balance the workload across the system by allowing processes to migrate to remote processors when the local processor is busy. In this model, four aspects of the workload are identified as crucial to the experiment:

1. how frequently the processes are executed (the process arrival rate),
2. how much CPU time each process requires,
3. the amount of I/O performed by each process, and
4. how much memory each process consumes.

A workload is therefore a collection of executable tasks which interface directly with the target hardware, and simulates the way in which a real program consumes the above-mentioned resources. Each factor is governed at run-time by random variables which conform to specified probability distributions and, to characterise the factors correctly, real workloads were observed at length prior to experimentation. In Kunz's experiments, CPU time is simulated by memory reads and writes, I/O is performed by reading and writing to a file, and standard system calls are made to dynamic memory allocation routines to consume heap space.

The model also requires that some of the variables exhibit statistical dependence (for example, memory consumption is made to increase in proportion to the length of time that a process has been running). For this workload it was not necessary to provide a mechanism for interprocess communication because user processes typically execute in mutual isolation.
Process-based Imperative Programs

The system implemented at Edinburgh University to model the statistical behaviour of concurrent, imperative programs [CFPS92, CPS92] is intended both to explore the behaviour of general classes of parallel programs and to improve the performance of an individual program. In order to satisfy both of these goals different types of workloads are required.

In the initial Edinburgh model, synthetic workloads were defined using the Simula language which allows general classes of program to be described. The underlying architecture for this system is a simulation of a network of transputers and real workloads are coded in Occam. To study the behaviour of specific programs a translator was constructed [CGS89]. This converts the Occam source into Simula programs which model just the aspects of the workload that are relevant to the simulation:

1. interprocess communication, and
2. the amount of CPU time consumed by each process.

A more controlled approach to generating workloads for this system is given in [PS91]. Here, a special purpose modelling language (an Event Definition Language, or EDL) is presented in which workloads are defined as directed, labelled graphs. The nodes of the graph represent tasks and the arcs represent the communication channels. The structure of the graph is fixed for the duration of each experiment but different graphs can be generated from a single EDL script to vary the experiments in a determined manner.

Various attributes are specified for each process, including the granularity of CPU usage per loop of the process, the ratio of idle (sleep) time to useful work time, and the amount of storage space that is used. Random variables decide the frequency and length of messages at compile-time. Random variables are not employed at run-time. Instead, the compiler ensures that the behaviour of each node is either fixed for the duration of the experiment or varies deterministically over time.

2.2.5 Discussion

Three methods of generating experimental workloads have been described. Of these, the method of synthesising workloads has numerous advantages when many well-defined workloads of varying complexities are required for experimentation. The price to be
paid for flexibility is the need for workload validation. A problem with real functional programming languages as experimental workloads is that the compiler and abstract machine have significant influence on performance. For our work, we currently wish to avoid this complication so have decided to develop synthetic workloads. With adequate description tools a synthetic workload will behave in precisely the manner required, will be fast to construct, and is easily adapted.

For synthetic workloads, the ‘computational model’ component of Figure 2.2 is an abstract machine which performs a subset of the operations of a real abstract machine. The subset is determined by the characterisation of real functional workloads on which the synthetic workloads are based.

2.3 Monitoring Experimental Functional Workloads

In this section we consider what experimental data is to be gathered, and how, when functional programs and their synthetic counterparts are executed on distributed memory, parallel machines. To collect the information, monitoring tools must be provided. The simplest form of monitor has already been mentioned in Section 2.1.4 and exploits existing operating system tools to obtain the necessary information. Sometimes, however, this solution is inadequate. For example, in [Cla92b], it is reported that the intelligent memory units of the GRIP multiprocessor have no access to a real-time clock, hence there are no ‘system calls’ which will provide timing information. An alternative solution is therefore proposed whereby a register is updated at regular intervals to approximate a clock. This is just one example where the information required by an experimenter is not readily available in the desired form and therefore demands careful thought.

Experiments dealing with distributed graph reduction may need to measure a number of attributes. For the DIGRESS system [Cla92a], the underlying communications subsystem [GB91] provides a library of measurement facilities, and others can be obtained from the operating system. Beyond these, all other measurements must be coded into the workloads themselves. The following are examples of characteristics which might be used as metrics for comparing performance:

- The total execution time of the whole program, and time spent per processor performing graph reduction. The total number of reductions performed on each processor may also be measured, but has limited meaning if there are many different
types of reduction, each taking different amounts of time to execute.

- The time taken to deliver individual processor to processor messages (communications latency).

- The total amount of idle time for each processor. This can be used in conjunction with the overall program execution time to calculate the average processor utilisation. Large idle times indicate a high degree of latency intolerance [Sar87] which will be important when message passing is expensive.

- The total number of messages passed between processing elements [Mah90]. The average size of messages might also be important, and it may be useful to subdivide these according to which messages request work, contain work packets, are used for garbage collection purposes, and request remote values.

- The number of tasks in the task pool (measured dynamically throughout the execution). If task pools are over-stocked for much of the program run then there is obviously too much parallelism for the machine to handle.

- The execution times for individual tasks.

- The number of tasks which migrate from one processing element to another. When locality of reference is to be studied [Mah90, Sar87, Cla89], it may be appropriate to record sufficient information so that later analyses can reconstruct the complete migration patterns for the program.

- The number of garbage collections, locally (for individual processors) and globally (synchronised across the whole system).

- The number of times (and the reasons) that tasks are suspended.

- Traditional resource utilisation (such as I/O to discs or display terminals). This information proves useful in [NS88] for dynamically load balancing a parallel computation according to the resources being consumed per processor.

Further information can be calculated from the raw measurements. For instance, the total time-overhead incurred due to message passing, processor synchronisation, servicing remote requests, and garbage collection is important when experimenting with scheduling
algorithms because a certain algorithm may distribute work perfectly evenly between processors but suffer large overheads.

A simple index for task granularity can be calculated directly from the total time spent reducing the graph and the number of tasks created during the lifetime of the program. However, a more detailed analysis might take into account the ratio of (i) the elapsed time between running out of work and new work arriving at a processor to (ii) the time required to execute the work, thus measuring granularity with respect to the size of system overheads. The statistical distribution of the grain size of individual tasks may also be important because it can have an important rôle to play in dynamic scheduling algorithms (see Chapter 6).

2.3.1 Measuring Task Sizes—System Implementors Versus Application Programmers

A fundamental difference exists between the way in which application programmers and system implementors naturally reason about the execution costs of lazy functional languages. In a strict language the source-level definition of expressions corresponds closely to the order of their evaluation at run-time. For example, the run-time behaviour of:

\[ z = f_{\text{strict}}(exp_1, \ldots, exp_n) \]

is described as follows:

1. All of the arguments \( exp_1, \ldots, exp_n \) are evaluated (this is independent of the definition of \( f_{\text{strict}} \)). Depending on the source language it may also be possible to state the order in which the arguments are evaluated.

2. The function, \( f_{\text{strict}} \), is executed. Any subexpressions contained within \( f_{\text{strict}} \) and relevant to the result are evaluated.

3. The result is bound to \( z \).

In a lazy language, delayed evaluation has the effect of transferring the evaluation of an expression from the position in the code where it was declared to the point where its value is required. In the following example:

\[ y = f_{\text{lazy}}(exp_1, \ldots, exp_n), \]
without a definition for \( f_{\text{lazy}} \) and further information about the context in which \( y \) is used, the run-time behaviour cannot be determined beyond stating that the function will be called and the result bound to \( y \). Any combination of the arguments may be evaluated or left unevaluated by the function call. Moreover the result may contain references to some unevaluated arguments which may then be evaluated at a later time. Consider the following definition of \( f_{\text{lazy}} \):

\[
f_{\text{lazy}}(\text{arg}_1, \ldots, \text{arg}_n) = (\text{arg}_1, \text{arg}_n)
\]

Here, the result is a pair containing the first and last arguments passed to \( f_{\text{lazy}} \). The pair can be constructed without evaluating any of the arguments. If the result, \( y \), later occurs in an expression of the form:

\[
z = (\text{fst } y) + (\text{snd } y)
\]

then the values of \( \text{exp}_1 \) and \( \text{exp}_n \) will finally be needed and the expressions will be evaluated. This is a simple example and yet it is quite difficult to reason about where expressions are evaluated. In large functional programs the examples become much more complex and unevaluated objects may be passed through many function calls before evaluation occurs.

The difference between the strict and lazy cases is significant. For straightforward performance evaluation, application programmers find it simpler to reason about the behaviour of strict programs than for lazy programs. As long as lazy programs provide them with correct results, why should applications programmers have to know where evaluations occur at run-time? The issue, which is further complicated by program transformations, is discussed in depth in [Cla93] and [CPC92]. Clayman concludes that the cost of evaluating expressions should be reported to application programmers with respect to the lexical structure of the source program. The program is still evaluated lazily but the call-trace is constructed as if strict evaluation had occurred.

A system implementor takes a very different view of the situation. The actual behaviour of the program at run-time is more important than the lexical relationships within the original source code. The effects of lazy evaluation must be reported exactly as they occur so that the knowledge can be used to improve compile-time and run-time heuristics (e.g. dynamic scheduling).

Despite having different requirements and viewpoints, both application programmers and system implementors need to make a connection between the definition of lazy func-
Figure 2.5: How applications programmers and systems programmers relate functional programs to their run-time behaviour.

The application programmer observes the run-time behaviour of the program and attempts to map this back to the original definitions in the source code.

The system implementor analyses the source program and attempts to use the analysis to predict its run-time behaviour. The run-time behaviour is also observed by the implementor, but only so that the success of the prediction can be verified. Analysis of the run-time behaviour is made with respect to the run-time domain, hence it is not necessary to map the run-time behaviour back to the lexical structure of the source code.

In the first case the programmer performs a backward mapping from run-time behaviour to the source program and in the second case performs a forward mapping from the source program to run-time behaviour. The difference between the two viewpoints has a large effect on the way that profiling and monitoring tools are constructed. In
Chapter 4, Section 4.5.1, we show that a more precise description of workload behaviour can be obtained when both viewpoints are applied simultaneously.

2.3.2 Profiling Tools for Functional Programs

In addition to the special purpose monitoring tools built into system implementations, profilers provide a more general purpose mechanism for instrumenting the behaviour of programs. Profiling helps programmers to identify hot-spots in source programs in order to improve the performance of applications software [GKM82]. A profiler can also provide information from trial executions of a program so that partitioning, dynamic scheduling, and load balancing algorithms can be made to work more effectively for future executions [Mah90, BP92, SH86]. If an implementation is known to be correct with respect to the original program specification, a greater understanding of the program can be obtained by making theoretical predictions about its behaviour and verifying or disputing these using a profiler. Conversely, if the correctness of an implementation has not been proved but the theoretical behaviour of the program is well understood, then a profiler can demonstrate that the memory and time consumption, and the shape of the call-graph, are consistent with the theory. In Section 4.5.1 the latter technique is used to verify the behaviour of synthetic workloads with respect to the measured behaviour of real workloads.

Several research sites have recently been involved with the design of profiling tools for functional programming languages. Some aspects of the tools have been based on the gprof profiler [GKM82] which is used to monitor the behaviour of imperative languages such as C. In [ADM88] and [RW90] the following problems associated with profiling functional languages are identified:

1. Compile-time transformations can dramatically reduce the correspondence between the run-time representation of a program and the original source code.

2. Higher-order semantics [BW88, FH88] allow function calls to be anonymous. Functions are therefore chosen at run-time from a set of alternatives and cannot be determined at compile-time.

3. Resource management routines such as garbage collection [Bak78, Coh81] are expensive and it is not clear how timings should be attributed to different parts of the source program.
4. If an expression is shared then eliminating its evaluation from one place in a program will cause the evaluation to move to another (possible quite remote) position. Therefore the removal of hot-spots is a non-trivial exercise.

5. The methods employed by imperative profilers such as gprof often cause recursive function calls to be handled badly. It is common for a set of mutually recursive functions to be treated as a single profiling unit. This is clearly undesirable for functional programs where recursive functions occur frequently.

These issues have been pursued at length (e.g. see [CPC91], [RW92], and [SPJ92]), and new profiling techniques have emerged. The implementation details of one such profiling technique are given in Appendix C. The resulting profiler is used in Chapter 4 to demonstrate equivalent algorithmic behaviour between synthetic workloads written in the Paragon language\(^1\) and real functional programs.

2.4 Summary

This chapter has explored a range of techniques of simulating and experimenting with computer systems and workloads. To explore dynamic scheduling technique for use with the DIGRESS distributed multiprocessor, it was decided to emulate the activities of a parallel graph reduction machine, and to drive the emulation using synthetic workloads. The decision to construct an emulator using the target hardware and the target communications subsystem was guided by the following:

- the factors influencing experiments are many and complex,
- the factors can be provided by an existing system implementation,
- some of the factors fall outside the experimental domain (e.g. user processes competing with the graph reduction engine for processor time).

The decision to use synthetic workloads in preference to real functional programs, or traces of real programs, is based on the many advantages listed in Section 2.2.3.

---

\(^1\)Paragon is described in Chapter 3 and in [PC92a, PC92b]
Chapter 3

Modelling Functional Workloads

By isolating key run-time attributes of lazy functional languages it is possible to construct simulated workloads that exhibit just those features which are relevant to the proposed experimental investigations. We can ensure that simulated workloads are both well understood and well behaved. To speed the process of workload construction, a special purpose description language is developed in this chapter.

3.1 Specification of the Problem

Functional programming languages possess the quality of referential transparency which means that an expression can be replaced with its value at any time without changing the semantics of the computation. This allows expressions to be evaluated in any order. Parallel evaluation can thus take place without the need for explicit synchronisation in the functional source code. To evaluate a program in parallel, it is partitioned into sequential code segments known as tasks. Many tasks execute simultaneously to produce the required parallelism. The workload to be modelled is therefore a collection of tasks whose internal behaviour and interaction are determined by functional language semantics.

The goal of modelling workloads is to achieve a simplified representation of real workloads so that it becomes possible to experiment more freely with the system under test. The key to a successful model is to determine which characteristics of the real workload are important and which can be ignored [Bat88]. Primarily, we intend to use synthetic workloads to model the tree of calls made by equivalent functional tasks and the synchronisation properties that take place between functional tasks. By artificially producing
these characteristics, a new method is provided for exercising the components of a parallel system implementation whose performance we wish to measure.

The system under test consists of hardware and software components for executing the workload and for conveying messages between processing elements. It also encompasses task scheduling, garbage collection, and any other housekeeping mechanisms, including techniques to locate and manage parallelism. The model workload does not form part of the system under test. The call-tree and synchronisation properties of the workload are defined by the experimenter using a workload modelling tool, whereas the properties of the system under test are measured empirically.

3.1.1 Experimental Requirements

Model functional workloads are employed because they provide advantages over using real workloads for experimental purposes. Functional languages pose many practical problems to the experimenter, mainly because higher-order functions and lazy evaluation semantics make it extremely difficult to predict the nature of the run-time stimulus provided by real functional programs. Simple examples such as the benchmarks listed in Section 2.2.1 are reasonably straightforward to construct and analyse, but larger examples require a great deal of time and effort. One of the main experimental requirements of model workloads is therefore that they circumvent this bottleneck: they must be simple to specify and easily understood.

One purpose of the experiments presented in this thesis is to determine key properties inherent in our chosen parallel environment. This is motivated by the desire to build an optimising compiler for real functional programs which tailor workloads to the characteristics of the underlying architecture (for an example of this, see [Koo90]). To conduct experiments to find the best classes of workloads for the environment under test, the test workloads must be readily adaptable.

We intend to run experiments to assist the development of improved dynamic scheduling algorithms. One of the planned approaches is to use advanced analyses to extract more information from source code than is currently available and to measure the improvements thereby achieved. The model must be sufficiently simple that the analyses can be developed with minimal effort, and to allow experimental evidence to be gathered quickly. Only when the utility of an analysis has been demonstrated for the model will it be recommended for further development in the domain of real functional programs.
Standard experimental requirements apply to model functional workloads. These include the ease of workload construction, repeatability of experiments, and a close mapping from the model to the real world. Beyond this we have the requirement that model workloads can be modified systematically so that trends can be established and, for a large number of experiments based on similar issues (e.g. the effect of laziness or granularity on a number of problem areas), the ability to re-use and adapt existing workloads is likely to be useful.

An issue that is often ignored when using real functional languages for experimental purposes is that of run-time input. We shall see in Chapter 6 that run-time input can have a large influence on the effectiveness of dynamic schedulers which make use of time-cost predictions made at compile-time. It must be possible to introduce the effects of run-time input into the model in a controlled manner.

Finally, it is required that modelled workloads do not introduce any new features that are not present in real functional language stimuli. An example of this is deadlock. For an implementation of a real functional language to be correct, it must guarantee to return the results of all terminating programs. If communication deadlocks can occur in a parallel implementation then the implementation is not correct. To predict the correctness of an implementation using a model stimulus, the model must not be able to introduce new deadlock conditions into an otherwise deadlock-free environment. (It is acceptable, however, for a model workload to fail to terminate if it is modelling a non-terminating program.)

A second example is that of memory consumption and space leakage [Wad87a]. Functional implementations usually employ garbage collectors (e.g. [GRW88, Hug85, Hug87b, Les89a, Rud86]) which attempt to ensure that dynamically allocated memory which is no longer active is returned to the memory allocator for re-use. Similarly, when a model stimulus is used, the implementation should again provide a garbage collector to allow memory to be recovered when its contents are no longer active. Model workloads should therefore be amenable to garbage collection.

### 3.1.2 The Experimental Environment

The environment chosen for this work is an experimental parallel architecture called DIGRESS [Cla92a]. DIGRESS is a loosely-coupled, distributed memory architecture intended for the parallel execution of functional programs and is built on top of a network
of Unix workstations connected via an ethernet. Processing elements (PEs) are software entities executing on the workstations and there may be an arbitrary number of PEs per workstation. At the heart of the design is a custom built, reliable, asynchronous message passing protocol which typically achieves a data transfer rate of 300k bytes per second [GB91]. The protocol requires each workstation to be executing a local system manager process (or Ism) which keeps track of all of the processing elements (and their host workstations) that form the current virtual reduction machine. Processing elements may join and leave the system arbitrarily\(^1\) which indicates the necessity for effective dynamic scheduling algorithms.

Figure 3.1 is a schematic representation of the DIGRESS system used in conjunction with simulated workloads. All processing elements in a DIGRESS virtual machine have equal connectivity and PE to PE communication (and communication between PEs and special units such as the program loader) is performed via message passing. The 300kBs\(^{-1}\) data transfer rate for messages is high with respect to the effective ethernet bandwidth; however, message passing still imposes a large overhead with respect to CPU processing times and it is expected, therefore, that DIGRESS workloads will always be coarse grained. The issue of granularity is sufficiently important for it to be given special status in the workload model and Figure 3.1 illustrates the facility to artificially scale the grain size of modelled functional tasks so that the effects of granularity on the performance of the system can be investigated easily. Also shown in the figure is the need for comprehensive run-time logging facilities to be incorporated into the simulated processing elements.

When Deschner used trace-based workloads in [Des89], he chose to ignore operations at the level of abstract machine instructions in favour of characterising workloads by the higher level operations which were directly relevant to his study. In contrast, the work of King [Kin90] reduces the operations of workloads for a number of different abstract machines to an extremely low level in order to make direct comparisons between the behaviour of the machines. When experimenting with the DIGRESS architecture, and specifically with dynamic task scheduling issues, the former approach appears to be more attractive. The coarse granularity expected of DIGRESS workloads implies that tasks will contain a large number of low-level abstract machine instructions to be executed\(^1\) Usually under controlled conditions, but it is also intended that the system be tolerant towards workstation failure during the lifetime of a computation.

60
sequentially and that the need for expensive PE to PE communication will be kept to a minimum. We are therefore interested mainly in characterising the synchronisation properties of tasks which are responsible for message passing. Work performed internally, within the confines of the sequential code of a task, can be characterised trivially by a simple resource consumption mechanism.

3.1.3 Sharing and Delayed Evaluation

Ideally, functional tasks would have no cause to communicate with each other beyond that required for parents to create child tasks and for children to return results to their parents. However, for lazy evaluation to work properly, tasks must perform some degree
of extra synchronisation, and will therefore require extra inter-task communications. Consider the function definition:

\[ \lambda e_1 \, e_2 \, e_3. \text{if } e_1 \text{ then } e_2 \text{ else } (e_3 + e_3). \]

This demonstrates the following two aspects of laziness which both require synchronisation:

- **non-strictness**: when the function is applied to three argument expressions, \( e_1, e_2, \) and \( e_3 \), the evaluation of the second and third arguments is delayed because only one of their values will ultimately be required.

- **sharing**: the third argument occurs more than once in the body of the function expression. Sharing ensures that the corresponding argument expression \( e_3 \) will be evaluated only once.

The property of non-strictness is central to lazy evaluation because non-terminating expressions can be passed safely as arguments as long as they are never evaluated. The sharing property is not semantically important because referential transparency also works in reverse, allowing values to be replaced by the expression which originally described the value. A consequence of this is that shared expressions can be copied and evaluated any number of times, without affecting the final value of the program. However, evaluating an expression multiple times incurs an obvious performance penalty and is to be avoided whenever possible. Figure 3.2 shows the graphical result of instantiating the example function with expressions \( e_1, e_2, \) and \( e_3 \): only one instance of \( e_3 \) is present in the graph and none of the arguments have been evaluated at this stage.

This is well known functional language implementation theory and is fully discussed in texts such as [PJ87b, FH88, PJL92]. Its relevance to building synthetic workloads is that non-strictness and sharing can have significant effects on the cost of tasks both in terms of message passing and the amount of work performed internally. In turn, this will affect scheduling strategies which rely on cost forecasts. One of the aims of creating synthetic workloads is to test the degree of susceptibility of various strategies to this problem. The model must therefore be able to closely reflect the functional characteristics.

Shared and delayed expressions may or may not form tasks in their own right. For example, consider the expression \((e_1 + e_2)\). A compiler might decide that the addition should be executed as a sub-computation of the currently evaluating task but that the
sub-expressions $e_1$ and $e_2$ are sufficiently expensive for one of them to be computed on a remote processor, as a separate task. A method must therefore be found for modelling the expression as a computation which may be transferred from one task to another and which may spawn tasks in its own right.

If an expression is shared by two or more tasks then lazy semantics requires that only one task will execute the code to evaluate the expression. For efficiency, the expression will not be duplicated and will be held locally within the task that created it until an external task requires its value. Therefore extra message passing is needed to fetch the value. If the expression is unevaluated when the request for its value arrives then the code will be transferred to the requesting task. Similarly, delayed (but unshared) evaluation may require the code for an expression to be left with the task which created it until its value is required by another task. This would be sensible, for example, if the expression contained a large amount of state. Here, the cost of transferring the unevaluated expression is best avoided until it is known to be necessary.

Message passing is required in both of the above cases and the amount of work performed internally by the donor and recipient tasks changes dynamically as expressions are moved around. To model this, the following mechanisms are required:

1. a mechanism to define and name shared and delayed workloads,
2. a mechanism to trigger the evaluation of a shared or delayed workload,
3. a mechanism to pass shared and delayed workloads between tasks.
3.1.4 Summary of Requirements

A model of a functional language workload for experimenting with dynamic task management algorithms on the DIGRESS architecture should exhibit the following qualities:

- Model workloads should be representative of real workloads that could reasonably be generated from functional programs and must operate in the same loosely-coupled, distributed memory environment as the real workloads. An important aspect of this requirement is that message passing activities and synchronisation properties of model workloads must correspond to those expected of functional programs executed in the given environment.

- The specification of test workloads should be simple so that experiments may be built quickly and without the likelihood of mistakes being introduced due to unnecessary complexity.

- Where the utility of an advanced analysis (such as time-cost analysis) is to be investigated, the analysis must be possible for the modelled workloads even if the corresponding analysis for real workloads is not currently available.

- We have decided that, within the boundaries of a task, it does not matter how a program spends its time. Therefore, it is sufficient to model only the fact that resources are consumed. Moreover, for scheduling experiments we are mainly concerned with the amount of time between successive task synchronisations and their associated message passing. To do this it is sufficient to model internal work as a time delay.

- Inter-task synchronisation is the single most important factor of the model and we need to specify: how new tasks are invoked, how tasks are suspended and subsequently resumed, and how computations are transferred between tasks.

The above list does not mention explicit representation of PE to PE communication. Communication is a consequence of inter-task synchronisation. Therefore, it need not be included as an independent parameter of the model. Also omitted from the list are methods for modelling garbage collection techniques, hardware characteristics, or the consumption of resources other than CPU cycles. These are not immediately required for experimenting with dynamic scheduling techniques on the DIGRESS architecture and

64
have not been dealt with at this stage. They are, however, considered as candidates for future work in Section 3.4.

The list of requirements given above matches very closely the list of advantages gained by synthesising workloads given in Section 2.2.3. We choose, therefore, to use synthetic workloads defined by a special purpose description language. Workload descriptions are text files which are simple to adapt, to include into other descriptions, and to use as input for automatic analysis, thus fulfilling a number of the other requirements listed above. It remains to define the language and to show that the synthetic workloads generated in this way are valid with respect to the actions of real functional programs. The remainder of this chapter is devoted to developing the workload description language, and validity is demonstrated in Chapter 4.

3.2 The Paragon Workload Description Language

In the previous section it was decided that synthetic workloads would be generated from textual definitions written in a task description language. The workloads thus described bear some resemblance to the trace-based workloads of [Des89] but the methods of description differ significantly in the following respect. Traced-based workloads are expressed as a complete call-tree in which every task to be executed is specified explicitly. In contrast, the workload description language developed in this chapter allows a single template definition to define the operations of a whole family of task instances. Furthermore, by introducing recursive task templates, terse descriptions are able to generate workloads with many task instances. Figure 3.3 illustrates this graphically. Figure 3.3(a) shows a finite, directed, cyclic graph of task templates, representing a program that uses a divide and conquer algorithm. Figure 3.3(b) shows the run-time representation of the program, where the recursive definition is unfolded into a call-tree of potentially infinite depth.

In this section the Paragon language is developed. We begin with a minimal tool for describing connectivity between tasks and introduce additional features to describe the other aspects of functional language behaviour that we wish to model.
Figure 3.3: A directed, cyclic graph of task template definitions are expanded into a call-tree of task instances at run-time.

3.2.1 Task Template Graphs

The primary building block of the Paragon language will be the task template. Our first concern then is to provide syntax to describe the template graphs of Figure 3.3(a). Templates descriptions take the following form:

\[
\text{tname} \{ \\
\text{instruction}_1 \\
\vdots \\
\text{instruction}_n \\
\} 
\]

where the instructions will be defined in the following sections. A collection of template definitions forms the nodes of the program graph, but there is currently no mechanism for defining the arcs.

Tasks are spawned when it is decided that work can be performed by parallel threads of computation. Typically this occurs when strictness analysis or user annotations result in code which creates tasks to evaluate the strict arguments of a function in advance of the function call. To model the creation of a task instance, the following instruction is introduced:

\[
\text{SPAWN} \rightarrow \text{tname} 
\]
This uses the template *tname* to create a child task instance. The *spawn* thus describes the arc of the graph between the parent and child instances. The template graph of Figure 3.3(a) can thence be represented by the example program given in Figure 3.4.

Let us now attempt to define a semantic meaning for the template graph. In a real functional program, the arcs of the graph represent dataflow in two directions. Firstly, arguments are passed from the parent task to its child in the form of an environment [PJS89, PJ92]. Secondly, when the child completes, it returns a result to its parent. However, at this stage we are attempting to define a minimal model and have no requirement to represent these data values explicitly. Implicit argument passing and the implicit returning of results will therefore be assumed.

The *spawn* instruction is adequate for describing arcs of the template graph, but it says nothing about the synchronisation properties between the parent and the child. It is not clear whether the parent must wait for its children to complete prior to its own completion, or if it should return immediately. In a real functional program, the parent task will have need to use the results of the child so will have to await its completion. Also, it may be the case that one set of children must complete before another can be spawned. For example, in the expression:

\[ \text{if } (\text{exp}_a + \text{exp}_b) > 2 \text{ then } 0 \text{ else } (\text{exp}_c + \text{exp}_d) \]

the value of \((\text{exp}_a + \text{exp}_b)\) must be known prior to the possible evaluation of \((\text{exp}_c + \text{exp}_d)\). A mechanism is clearly required to state at which points synchronisation occurs.
3.2.2 Synchronising Parent and Child Tasks

Consider the actions of a parallel evaluator when confronted with a function application $f \ e_1 \ldots \ e_n$ where $f$ is strict in one or more of its arguments. Expressions bound to the strict formal parameters are spawned as tasks for parallel evaluation and the function is entered. At some point during the execution of $f$ the value of the strict argument will be required by a primitive function and one of several actions will follow. If the argument has already been evaluated (in parallel) then there is no more to be done. If the argument has not been evaluated then the current processing thread must be interrupted and the evaluation forced (as discussed in Section 3.3). If the evaluation is *currently* taking place then, typically, the parent task must suspend until the evaluation is complete. Various suspension/resumption mechanisms are possible such as maintaining a list of suspended tasks that are awaiting the value [CPJ86] or for child tasks to explicitly inform their parents of their return [WW87b].

To specify points at which a parent task synchronises with one or more of its children, it must first be possible for a parent to identify the children uniquely. It is insufficient to make reference to the task template used by the child because several child task instances may have been created from a single template. The `spawn` instruction, as described above, is therefore extended with a facility to uniquely label each of the children:

```
SPAWN label -> tname
```

A blocking instruction is now introduced which suspends the current thread of execution until specified child tasks have completed. The instruction takes the form:

```
BLOCK label_1 \ldots label_n
```

where the labels are those of *preceding* `spawn` instructions within the current template definition. We can now attempt to represent the functional if-then-else expression shown above. Currently, there is no mechanism for representing conditional behaviour (this is dealt with later in Section 3.2.7). A task template will therefore be built to represent just the test and the else branch of the expression. Assuming that only one argument to each `+` operator will be spawned as a parallel task, the following definition
approximates the behaviour of the expression:

```java
if_then_else {
    Spawn a -> exp_a
    Block a
    Spawn c -> exp_c
    Block c
}
```

However, the calculations for the values $exp_b$ and $exp_d$ are conspicuously absent from this approximation. These form part of the computational work performed by the task and are therefore responsible for the consumption of machine resources. To model the calculations, some thought must be given to modelling resource consumption.

### 3.2.3 Resource Consumption

The simplest form of computational activity that we can model is the consumption of CPU cycles. This represents the amount of time taken to compute the value of an expression. In Section 3.4 other types of resource consumption are considered as candidates for future work, but for now we concentrate on timing issues only. The following instruction is therefore sufficient to specify resource consumption:

```
WORK t
```

where $t$ is a number of abstract time units. A concrete interpretation is given for abstract time units in Section 3.3.2.

The characterisation of the example `if-then-else` expression now becomes:

```java
if_then_else {
    Spawn a -> exp_a
    Work $t_b$
    Block a
    Spawn c -> exp_c
    Work $t_d$
    Block c
}
```

such that the local calculation of expressions $exp_b$ and $exp_d$ is represented by the two `WORK` instructions.

The amount of time spent executing `WORK` instructions can be used a measure of *useful* work. All other run-time activity, such as task spawning and context switching when tasks suspend, is the overhead incurred by parallel evaluation. The ratio of computational time to total processing time therefore expresses the efficiency of an execution.
3.2.4 Describing Sharing and Delayed Evaluation

In Section 3.1.3 three mechanisms were listed for the representation of shared and delayed evaluation properties of function programs: (i) the definition, (ii) the forced evaluation, and (iii) the communication of shared and delayed objects. None of the Paragon instructions so far defined fulfil these roles. A child task can only be referenced by one parent, so two parents cannot share the implicit results of a single child. Therefore, synchronisation is currently limited to a straightforward parent/child relationship.

Consider the following function application:

\[ f(g_1 e_1) \cdots (g_n e_1) \text{ where } e_1 = \ldots \] (3.1)

If \( f \) is strict in one or more of its arguments (for \( n \geq 1 \)) then a corresponding number of child tasks will be spawned. If none of \( g_1 \ldots g_n \) are strict in their argument then the expression \( e_1 \) will not be evaluated in advance and we assume that a closure is passed to each of these functions indicating the location of the code and environment to compute \( e_1 \) \cite{PJ87b, FW87, WW87b, AJ89b, PJS89, PJ92, etc}. If \( n = 1 \) then the closure represents a delayed value and no calculation or data transfer takes place until it is deemed to be necessary. If \( n > 1 \) then \( e_1 \) is not only delayed but is also shared. A shared expression is evaluated at most once (by the first task to require its value) and subsequent requests for the value must wait until it is computed.

At present we have no notion of an explicit environment (remember that argument and result passing between tasks is implicit), but the notion of code is given by sequences of Paragon instructions. Therefore, closures can be synthesised by the following instruction:

\[
\text{DECLARE } \text{sequence-name} \{ \\
\quad \text{instruction}_1 \\
\quad \vdots \\
\quad \text{instruction}_n \}
\]

Which names the synthetic closure and describes the actions of the code. In this definition, \( \text{instruction}_i \) (for \( 1 \leq i \leq n \)) is a \textit{SPAWN}, \textit{BLOCK}, \textit{EVAL}, or \textit{WORK} instruction (the definition of \textit{EVAL} will be given shortly). These named sequences of instructions are given the title \textit{declared sequences}.

When an instance of a task is created from a task template, a set of unevaluated code sequences are built in accordance with the \texttt{DECLARE} instructions present in the template.
Just like a real closure, each code sequence can be referenced an arbitrary number of times but will be executed at most once. When they are evaluated they behave exactly as any other sequence of instructions. Just like the functional language where construct, the sequence declaration does not imply that the evaluation of the enclosed instructions takes place; the DECLARE instruction merely defines the sequence.

The next stage in modelling expression (3.1) is to modify the SPAWN instruction so that declared sequences can be passed explicitly from a parent to its children:

\[
\text{SPAWN } label \rightarrow tname \ <seq_1, \ldots, \ seq_n> \]

The SPAWN instruction now possesses a list of declared sequence references delimited by angle brackets. These form a list of explicit actual parameters which are additional to the implicit parameters that are assumed to pass from parent to child when a SPAWN is executed. A declared sequence is referenced by specifying its name. Further methods of reference will be explained in due course.

Using this syntax, expression (3.1) can be described thus:

\[
\text{apply } t \{ \\
\quad \text{Declare } e_1 \{ \ldots \} \\
\quad \text{Spawn } \text{arg}_1 \rightarrow g_1 \ <e_1> \\
\quad \vdots \\
\quad \text{Spawn } \text{arg}_{n-1} \rightarrow g_{n-1} \ <e_1> \\
\quad \text{Work } t_{g_n} \\
\quad \text{Block } \text{arg}_1 \ldots \text{arg}_{n-1} \\
\quad \text{Work } t_f
\}
\]

However, there is currently no mechanism either for the child tasks to access their formal parameters or for forcing the evaluation of declared sequences.

The first problem is solved by introducing the syntax $x$ to reference the $x$th sequence in the list of actual parameters supplied by the parent. If fewer than $x$ actual parameters were passed then an unshared, empty code sequence is assumed by default. A $x$ parameter reference is valid wherever a declared sequence name may be used. For example, $x$ can be included in the list of actual parameters of another SPAWN instruction. This passes the $x$th formal parameter of the current task on to its child, thereby allowing the synthesis of a function definition of the form:

\[
f \ x_1 \ldots x_n = \ldots (g \ x_i) \ldots \quad \text{(for } 1 \leq i \leq n)\]

where $x_i$ is described by a closure.
The addition of explicit parameters introduces an explicit environment to task instances. By definition, the environment of a declared sequence (which does not take explicit parameters in its own right) is set to that of its defining task instance. Declared sequences now correspond very closely to the traditional \((\text{code, environment})\) definition of closures.

To force the evaluation of a declared sequence, another new instruction is required. The instruction takes the form:

\[
\text{EVAL seq}_1 \ldots \text{seq}_n
\]

and, if they are not already evaluated, causes the evaluation of the declared sequences bound to the references \(\text{seq}_1 \ldots \text{seq}_n\). The evaluation proceeds from left to right.

These mechanisms enable expression (3.1) to be represented in full. The templates \(g_1 \ldots g_{n-1}\) used in the Paragon description, above, can now be defined. They contain an \text{EVAL} instruction at the point at which the value of the shared expression is required:

\[
g_i \{ \\
\quad \vdots \text{Eval } \$1 \\
\quad \vdots \\
\}
\]

The \text{Work} \(t_{g_n}\) instruction of the apply\_f template should also be replaced with the three instructions:

\[
\text{Work } t_{g_{n,1}} \\
\text{Eval } e_1 \\
\text{Work } t_{g_{n,2}}
\]

such that apply\_f also requires the value of the shared expression when it evaluates the \(n\)th argument supplied to \(f\). The first task to request the evaluation of the expression will be the task which incurs the expense of the evaluation.

The representation of closures is almost complete. In Section 3.2.1 we stated that dataflow takes place between tasks in two directions. For example, in the following functional code:

\[
\begin{align*}
f & \ (g_1 \text{ intermediate}) \ (g_2 \text{ intermediate}) \ expr \\
& \quad \text{where} \\
& \quad \text{intermediate} = h \ e_1 \ldots e_n
\end{align*}
\]

the intermediate result of evaluating \(h\) is passed as a shared argument to both \(g_1\) and \(g_2\). The flow of closures from parent to child tasks has been covered, but not that from
child to parent. Therefore, the following instruction is introduced to allow synthetic
closures to be passed explicitly as the result of a child task to its parent:

\[
\text{RETURN } \text{seq}_1 \ldots \text{seq}_n
\]

The label attached to the \texttt{SPAWN} instruction which created the child can then be used
to make reference to the returned closure. The above example is modelled in Paragon as
follows:

\[
h \{ \\
    \text{Declare result \{ \ldots \}} \\
    : \\
    \text{Return result} \\
\}
\]

\[
\text{apply}\,\text{f} \{ \\
    \text{Spawn intermediate } \rightarrow \text{h} \\
    \text{Spawn arg1 } \rightarrow \text{g1 <intermediate>} \\
    \text{Spawn arg2 } \rightarrow \text{g2 <intermediate>} \\
    \text{Work } t_{\text{expr}} \\
    \text{Block arg1 arg2} \\
    \text{Work } t_f \\
\}
\]

The synchronisation property here is such that task instances \text{arg1} and \text{arg2} will suspend
if they reach an \texttt{Eval} $1$ instruction before task instance \text{intermediate} has completed
and returned its value.

A Summary of the Declared Sequence Mechanisms

In summary, declared sequences can be executed, passed as arguments to child task
instances, and returned as results from children to their parents. References are made to
declared sequences in the following ways:

1. By explicitly specifying the \textit{sequence-name} given in the \texttt{DECLARE} instruction.

2. By specifying the label attached to a preceding \texttt{SPAWN} instruction of the current
task or declared sequence. The label identifies the declared sequences returned
as the result of the associated child task instance. If the child does not return
an explicit sequence then an unshared, empty sequence is assumed by default. If
the child returns more than one sequence then the collection behaves like a single,
compound sequence. The evaluation of a compound sequence takes place as if each

73
of the sequences had been specified separately within an \texttt{eval} instruction (i.e. they are evaluated from left to right).

3. By referencing a formal parameter of the current task. The syntax for a formal parameter is $z (z \geq 1)$, indicating the $z$th actual parameter passed to the current task.

Declared sequence evaluation provides another method of synchronisation between tasks beyond that of parents blocking to await the completion of their children. An attempted evaluation of a declared sequence will cause the current task to suspend if:

1. the declared sequence is shared and is in the process of being evaluated by another task or

2. the return value of a child task is to be evaluated but it is not yet known (either the child task has not completed or its return value is bound to the result of another task which is not yet complete).

A suspended task can resume at any time after the condition which caused the suspension no longer persists.

3.2.5 Optimising Tail-Recursion

An optimisation often applied to functional implementations is that of tail calls. A tail call occurs when a function has completed all of its processing apart from a single call to another function. In functional languages, tail calls are often caused by functions tail-calling themselves. As we are dealing with low-level run-time specifications it would be sensible to specify tail-recursive calls explicitly and to give them special status in the implementation. This can be done using the following instruction:

\begin{verbatim}
TAIL t <seq_1, ..., seq_n>
\end{verbatim}

where $t$ specifies the expected number of tail calls to be made for this task at run-time. A tail call counter is maintained for each task and is inspected when a \texttt{TAIL} instruction is executed. If the counter is greater than zero then it is decremented and the instruction creates a new task instance, in place of the existing instance, and immediately begins to execute the new instance. The argument list allows the completing task instance to pass
synthetic closures to the new instance. If the tail call counter is zero then no action is taken.

This mechanism preempts the \texttt{RETURN} instruction described in Section 3.2.4. The \texttt{RETURN} instruction is ignored for all tail-recursive instances \textit{except} for the last in the cycle. When the last task instance completes, the \texttt{RETURN} instruction is then executed, and the result is returned to the task which spawned the first task instance in the cycle.

\texttt{TAIL} instructions are useful for simulating list processing where successive tail calls deal with successive items in the list. The typical list length is characterised by the value of $t$. However, it might be desired to employ a single task template to represent a number of processes which operate on lists that have different lengths. This is achieved by adding a \texttt{modifier} to the \texttt{SPAWN} instruction:

\begin{verbatim}
SPAWN label -> tname <seq$_1$, \ldots, seq$_n$> m
\end{verbatim}

If the child task created with the \texttt{SPAWN} contains a \texttt{TAIL} $t$ instruction, the number of tail calls performed by the child is given by the product $m \times t$.

### 3.2.6 Summarising the Paragon Instruction Set

In summary, task templates are defined in Paragon as follows:

\begin{verbatim}
template-name {
  declaration$_1$
  
  declaration$_m$
  simple-instruction$_1$
  
  simple-instruction$_n$
  tail
  return
}
\end{verbatim}

where:

- \textit{declarations} ($m \geq 0$) are \texttt{DECLARE} instructions which define named code sequences. These model \textit{closures} which are used in functional language implementations to provide both sharing and delayed evaluation. It will be shown in Section 4.3.5 that these also model higher-order behaviour.
simple-instructions \((n \geq 0)\) are \texttt{SPAWN, BLOCK, EVAL, or WORK} instructions, which provide a pragmatic definition of run-time activity. They create new child task instances, synchronise parent and child tasks, force the evaluation of synthetic closures (declared sequences), and synthesise primitive work, respectively.

\texttt{tail} is an optional \texttt{TAIL} instruction that forces task instances to make recursive tail calls. Only one \texttt{TAIL} instruction may be specified per template and it may only be followed by a \texttt{RETURN} instruction or (if there is no explicit \texttt{RETURN}) the closing brace of the definition.

Note that a recursive tail call differs from a recursive spawn in the following ways:

- The subsequent \texttt{RETURN} instruction is ignored, until the very last tail call has completed.
- The recursively spawned child overwrites the current task and inherits its parent rather than existing as a separate entity.
- The tail call count for the child is calculated by decrementing the current tail call counter. There is no modifier present in a \texttt{TAIL} instruction, so the counter is not recalculated by the \(m \times t\) formula given in Section 3.2.5.

\texttt{return} is an optional \texttt{RETURN} instruction for binding the result of a task to a synthetic closure. There is at most one \texttt{RETURN} instruction per task template and if there are none, the return value is bound to an empty sequence.

This syntax is extended further in Section 3.2.7, to include facilities to model run-time input. After this, a full BNF syntax is given for the language.

### 3.2.7 Simulating Run-Time Input

In algorithmic terms, the only mechanism to be affected by input data is that of decision making. If we ignore low-level issues such as floating point versus integer arithmetic then, as long as the expression contains no conditionals, the processing effort required to compute the expression will be constant, irrespective of the input. When conditionals are introduced, however, alternative actions are selected dynamically, according to the input over which the conditionals are resolved. This can have one of two effects with respect to the activities of a task:
1. A conditional expression may cause the amount of work performed internally within
a task to vary, but does not alter the degree of synchronisation between the current
and other tasks.

2. A conditional branch selects one of a number of alternative actions that are responsi­
sible for different amounts of inter-task synchronisation.

We can model the first of these effects simply by extending the syntax of the Paragon
WORK instruction to accept a random variable:

\[
\text{WORK } r
\]

where \( r \) is one of the \textit{ranges} defined in Table 3.1. For example, task instances constructed
from the following template:

\[
\text{simple} \{ \\
\quad \text{WORK NORMAL 50.34 4.2} \\
\}
\]

perform primitive work for a time period whose duration has a normal distribution about
a mean of 50.34 time units with a standard deviation of 4.2 units.

A Paragon \textit{range} specifies the probability density function (pdf) of a random vari­
able. Values are expressed in units and can be applied whenever it is useful for a numeric
value to be generated stochastically (for example, in specifying the tail call counts and
the corresponding modifiers described in Section 3.2.5). Table 3.1 contains just a small
number of methods for specifying random variates, the simplest of which is as an unqual­i­
ified constant, \( x \), whose pdf is just a delta function. The table can be extended by adding
new pdf constructor functions to Paragon whenever the need arises (e.g. CHISQUARE \( \nu \)).

By default, random variables are re-calculated every time a new value is required.
However, in some cases it may be useful for a random value to be generated once only
for a template and for the result to be used as a fixed number by every task instance
constructed from the template. A new value will be generated for each execution of
the workload and will remain fixed for the whole execution period. 'Fixed' variates are
indicated by placing an asterisk immediately after the \textit{range} constructor (e.g. BETA* a b
l h).

To reflect the more complicated case when run-time input affects tasks synchroni­
sations, the instructions for dealing with synchronisation must be modified. This can
be done with a single mechanism which extends the \texttt{BLOCK}, \texttt{EVAL}, \texttt{RETURN}, \texttt{SPAWN},
Range | Description | Mean | Variance |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$z$</td>
<td>$z$ is a real number specifying a fixed number of units.</td>
<td>$z$</td>
<td>0</td>
</tr>
<tr>
<td>UNIFORM $a, b$ or $[a..b]$</td>
<td>$z$ units where $z$ is a random variable uniformly distributed in the range $a \leq z \leq b$ where $a$ and $b$ are real numbers ($a \leq b$).</td>
<td>$\frac{a + b}{2}$</td>
<td>$\frac{(b - a)^2}{12}$</td>
</tr>
<tr>
<td>NORMAL $\mu, \sigma$</td>
<td>$z$ units where $z$ is a random variable normally distributed with variance $\sigma^2$ about the mean value $\mu$ (where $\sigma$ and $\mu$ are real numbers).</td>
<td>$\mu$</td>
<td>$\sigma^2$</td>
</tr>
<tr>
<td>BETA $a, b, l, h$</td>
<td>$z$ units where $z$ is a random variable ($l \leq z \leq h$) whose probability density function is given by: $f(z) = \frac{z^{a-1}(1 - \bar{z})^{b-1} \Gamma(a + b)}{\Gamma(a) \Gamma(b)}$ where $a, b, l, h$ are real numbers such that $a &gt; 0, b &gt; 0, h &gt; l$; and $\bar{z} = (z - l)/(h - l)$.</td>
<td>$\frac{a(h + l)}{2(a + b)} + l$</td>
<td>$\frac{ab(h - l)^2}{(a + b)^2(a + b + 1)}$</td>
</tr>
</tbody>
</table>

Table 3.1: Paragon ranges.

and TAIL instructions with alternative branches which are selected according to explicit probabilities. Each of the instructions is extended to take the form:

```
instruction  alternative_1 (pp_1\%)
  | alternative_2 (pp_2\%)
  | \vdots
  | alternative_n (pp_n\%)
```

The choice of which branch of the instruction to execute is made according to the percentage probabilities, $pp_i\%$, for $1 \leq i \leq n$. The probabilities express a discrete pdf over the range of alternative actions. For example, consider the functional expression:

```
a + (if x then y else z)
```
If y and z are to be represented by synthetic closures, and the probability of x evaluating
to true is determined to be 0.25 then the expression can be represented in Paragon by:

```
example {
    EVAL $1 (25%) | $2 (75%)
    Work t1
}
```

where y corresponds to the first explicit parameter of example, and z corresponds to the
second.

If the total percentage probability is less than 100% then it is possible for none of the
branches to be selected. Conveniently this allows terminating, recursive programs to be
written. Previously, all instructions were unconditional and thus the divide and conquer
example given in Figure 3.4 would never terminate because new task instances would be
generated ad infinitum.

Two slight modifications are made to the general purpose branching syntax given
above. The first is for the `TAIL` instruction. Here, the tail call counter already determines
whether or not the tail call takes place. Therefore, if none of the branches of the tail call
are selected probabilistically then a tail call is still made (as long as the counter is greater
than zero), but with an empty list of actual parameters. The second modification is for
`SPAWN` instructions. The full branching syntax for `SPAWN` is as follows:

```
SPAWN label
  -> tname1 <seq1,1,..., seq1,m1> mod1 (p1%) |...
   | tname2 <seq2,1,..., seq2,m2> mod2 (p2%)
   | ... |...
   | tnamen <seqn,1,..., seqn,mn> modn (pn%)
  -> work
```

such that the final alternative is given by a `WORK` instruction that is executed if none of
the other branches are taken. This is convenient when there is a choice between creating
a child task and performing some work locally (e.g. see Section 4.6.1 in Chapter 4).

### 3.2.8 Formal Paragon Syntax

The formal BNF syntax of Paragon programs is given in Table 3.2. Terminals (literal
characters) are printed in the table in courier font and non-terminals are printed in
emphasised font. Alternates are specified using the | character and optional syntax is
enclosed in []'s. Notice that Paragon is not sensitive to the case of reserved words (e.g. instruction names and random variable constructors such as NORMAL).

3.3 Implementation Issues

In this section topics concerned with the compilation and implementation of Paragon workloads are discussed.

3.3.1 Choosing Between Alternate Branches of an Instruction

The general purpose syntax adopted in Section 3.2.7 for specifying alternative run-time behaviour is:

```
instruction alternative_1 (pp_{1}{\%})
| alternative_2 (pp_{2}{\%})
| ...
| alternative_n (pp_{n}{\%})
```

The following inequalities must hold for correct compilation:

\[ 0 < pp_i < 100, \quad \text{where } 1 \leq i \leq n \]  
\[ \sum_{i=1}^{n} pp_i \leq 100. \]  

(3.2) \hspace{1cm} (3.3)

To simplify model building, probabilities may be omitted and calculated automatically using the following formula:

\[ pp_i = \frac{100 - \sum_{p p_j \in P} pp_j}{i + w - |P|}, \quad \text{for all } pp_i \notin P \]  

(3.4)

where P is the set of probabilities which are specified explicitly. The w term is normally zero except for the SPAWN instruction where it is used to take account of an extension to the syntax for alternative actions:

\[ w = \begin{cases} 
1, & \text{if there is a work item in the SPAWN} \\
0, & \text{otherwise.} 
\end{cases} \]

The residual probability after the explicit probabilities have been summed is divided equally between those alternatives that had implicit probabilities and (if w ≠ 0) the w item. If there are no implicit probabilities then the residual probability remains unassigned. It is therefore possible for none of the branches of an instruction to be selected at run-time. In this case, no action is performed by that instruction.
\begin{table}
\begin{center}
\scriptsize
\begin{tabular}{ll}
\texttt{description} & \texttt{\to taskdefs} \\
\texttt{taskdefs} & \texttt{\to taskdef [taskdefs]} \\
\texttt{taskdef} & \texttt{\to template-name \{ [decls] units [tail] [return] \} } \\
\texttt{decls} & \texttt{\to DECLARE d-label \{ units \} [decls]} \\
\texttt{units} & \texttt{\to unit [units]} \\
\texttt{unit} & \texttt{\to block | eval | spawn | work} \\
\texttt{block} & \texttt{\to BLOCK blocklist} \\
\texttt{blocklist} & \texttt{\to s-list [probability] | blocklist} \\
\texttt{eval} & \texttt{\to EVAL eval-list} \\
\texttt{eval-list} & \texttt{\to seqs [probability] | eval-list} \\
\texttt{spawn} & \texttt{\to SPAWN s-label \to spawnlist \to work} \\
\texttt{spawnlist} & \texttt{\to spawnitem [probability] | spawnlist} \\
\texttt{spawnitem} & \texttt{\to template-name [range]} \\
& \texttt{| template-name actuals [range]} \\
\texttt{template-name} & \texttt{\to name} \\
\texttt{\to work} & \texttt{\to WORK range} \\
\texttt{tail} & \texttt{\to TAIL [range] \to tail-list} \\
\texttt{tail-list} & \texttt{\to actuals [probability] | tail-list} \\
\texttt{return} & \texttt{\to RETURN retvals} \\
\texttt{retvals} & \texttt{\to seqs [probability] | retvals} \\
\texttt{probability} & \texttt{\to ( number \%)} \\
\texttt{actuals} & \texttt{\to < seqlist >} \\
\texttt{seqlist} & \texttt{\to seq [, seqlist]} \\
\texttt{seqs} & \texttt{\to seq [seqs]} \\
\texttt{seq} & \texttt{\to s-label | d-label | $posint$} \\
\texttt{range} & \texttt{\to number | [ number .. number ] | distribution} \\
\texttt{distribution} & \texttt{\to UNIFORM [*] number number} \\
& \texttt{| NORMAL [*] number number} \\
& \texttt{| BETA [*] number number [number number]} \\
\texttt{s-list} & \texttt{\to s-label [s-list]} \\
\texttt{d-label} & \texttt{\to name} \\
\texttt{s-label} & \texttt{\to name} \\
\texttt{name} & \texttt{\to initial-char subs-chars} \\
\texttt{subs-chars} & \texttt{\to initial-char | digit} \\
\texttt{initial-char} & \texttt{\to small | large | _} \\
\texttt{small} & \texttt{\to a | b | . . . | z} \\
\texttt{large} & \texttt{\to A | B | . . . | Z} \\
\texttt{number} & \texttt{\to [-] posint [ . posint [o [-] posint]]} \\
\texttt{posint} & \texttt{\to digit [posint]} \\
\texttt{digit} & \texttt{\to 0 | 1 | . . . | 9} \\
\end{tabular}
\end{center}
\caption{BNF syntax table for Paragon.}
\end{table}
3.3.2 Making Abstract Time Units Concrete

For the abstract time units used by \texttt{WORK} instructions to be meaningful, the magnitude of a work unit must be made concrete. It would be ill-advised to specify work units as an absolute measure of time because this takes no account of the power of the underlying hardware or (for multi-threaded systems) the loading of the machine. Instead, we use the piece of C code shown in Figure 3.5 to simulate processing. Assuming the code is not optimised, it executes 100\(g\) function calls and 100\(g\) integer increments per work unit where \(g\) is the process granularity. The granularity is a dimensionless coefficient and is represented by a real number.

3.3.3 Restrictions on Declared Sequence References

In total, three modes of reference to declared sequences were given in Section 3.2.4. Any of these modes can be used wherever synthetic closures are required, except for the following situations. Firstly, no instruction within a declared sequence may contain explicit references to its own label or the labels of any other declared sequences. This is called the \textbf{direct reference rule} and is needed to avoid cyclic graphs and deadlocks (see Chapter 4 for further discussion of deadlocks). Secondly, the label of a \texttt{SPAWN} instruction may only be referenced within the scope of the declared sequence or task in which the \texttt{SPAWN} occurs, and only by \textit{subsequent} instructions. This is called the \textbf{return reference}
rule and avoids cyclic references between declared sequences.

3.3.4 Evaluating Child Tasks Locally

When there is insufficient processing capacity for a child task to be evaluated remotely, there are a number of different ways in which the implementor could force it to evaluate locally. The Paragon language places no restrictions on the implementation; the instructions for creating child tasks merely indicate that parallelism is available in the synthetic workload. It is left to the implementor to determine exactly how tasks are exported to remote processing elements for parallel evaluation and how parallelism is dynamically throttled when there is insufficient processing capacity for parallel evaluation to occur.

For example, one implementation may cause an unevaluated child task to be subsumed into its parent task for sequential evaluation (this is the technique described in [HPJ92]). Another implementation may take the view that if a child is unevaluated when its value is required then the parent must block and start evaluating another task from its local task pool. The child task might then be evaluated locally or might still be picked up by a remote processor. In both cases the parent will be released from its block as soon as the value becomes available. The relative merits of these techniques will not be discussed here but it should be noted that implementations of Paragon can be made to resemble whichever real functional language implementation is favoured by the experimenter.

3.3.5 Speculative Parallelism

The Paragon specification does not insist that a child task must be the subject of an unconditional BLOCK instruction. If the end of a parent task is reached and synchronisation has not taken place between the parent and one or more of its children then speculative parallelism is being modelled and the runaway children are now irrelevant. The current Paragon implementation does not attempt to kill off irrelevant tasks in the way suggested by [Hud84], but leaves them running and merely assures that their eventual return (if they are terminating processes) is handled properly. The experimenter should be aware of this because irrelevant tasks compete for resources with useful tasks.

If speculative parallelism is not to be modelled, it is the responsibility of the workload designer to ensure that the completion of all child tasks is synchronised properly by the use of BLOCK instructions.
3.4 Further Work

The Paragon language presented in this chapter has a number of omissions that are possible candidates for future work. Perhaps the most pressing of these is the ability to model the consumption of resources other than CPU cycles. Specifically, we do not model memory consumption, beyond that required to describe task instances. A straightforward reference-count garbage collector is sufficient to collect task instances because they form a call-tree in which there can be no cycles. However, it would be more realistic to introduce a method of allocating memory in order to study the behaviour of other types of garbage collectors and the effect that they have on dynamic scheduling algorithms.

Currently, the workloads are managed by real implementations of dynamic task management algorithms; it is envisaged that garbage collection experiments would be conducted in the same manner, using an extended Paragon syntax to introduce garbage and a real implementation of a garbage collector to reclaim the memory. For this approach to be effective, the nature of garbage must be studied closely and an accurate model constructed. Beyond memory allocation, traditional I/O resource modelling [Jai91] could be added to Paragon workloads if future investigations so demand.

The other major omission from Paragon is the ability to model hardware elements. The reason for this is that Paragon workloads are designed to stimulate a real hardware platform such as the DIGRESS system on which the experiments of this thesis are based. Two possible approaches can be taken in future work that includes hardware modelling. Firstly, the hardware model can be kept separate from the workload model, in which case Paragon workloads would require no alterations to be executed on synthetic hardware. Alternatively, an approach similar to that of [PS91] might be adopted by extending the Paragon language to include instructions which model the hardware.
Chapter 4

Reproducing Functional Language Behaviour

The mechanisms given in Chapter 3 for describing synthetic workloads are useful to experimenters only if the resulting workloads correctly model the behaviour of functional programs. In this chapter we verify that Paragon is sound and show that it is complete with respect to the key features of functional language workloads that we have chosen to model. Procedures for verifying and validating individual Paragon workloads are also presented, and example models are illustrated which mimic the actions of common algorithmic forms.

4.1 Introduction to Paragon Verification and Validation

In [Jai91, p.413], Jain states that:

"During the development of the simulation model, you must ensure that the model is correctly implemented and that it is representative of the real system. These two steps are called model verification and validation, respectively."

For Paragon workloads there are, in fact, four stages of verification and one of validation to consider:

1. Verifying that the Paragon language is sound. This demonstrates that Paragon scripts produce workloads that are equivalent to those which can be generated from real functional programs.
2. Verifying that the Paragon language describes all that we want it to describe. This is the property of completeness.

3. Verifying that the implementation of the Paragon language does not introduce inconsistencies (i.e. debugging the implementation).

4. Verifying that the pragmatic description contained within each Paragon script stimulates the system under test in the way that was intended in the original specification. Thus we remove logical errors in the composition of individual workloads.

5. Validating individual synthetic workloads with respect to real functional programs. If a particular algorithm is being modelled, this demonstrates that the workload specification will stimulate the system under test in the same way as a real implementation of the algorithm.

The first three stages are necessary before any workloads are constructed and are dealt with in Sections 4.2, 4.3, and 4.4. The fourth stage is relatively straightforward. Whenever a model workload is to be employed, the experimenter should have a clear idea of the required stimulus. The actions of the stimulus are traced when the model is executed and verification is achieved by comparing these with the expected actions. If they differ then the experimental results obtained with the invalid workload are rejected and the original model is refined. The comparison/refinement process is repeated until the traced actions are consistent with those expected of the workload.

The final stage of validating workloads is the most difficult to perform. If it were practicable to measure a real stimulus, a real functional program must have been constructed in the first instance, and it appears that there is no longer a need to construct a model. However, altering the stimulus produced by a real program may well require that its semantics are changed. One of the claims of our workload model is that it removes the experimental overhead of writing programs in the presence of computational semantics. Therefore, it is useful to construct an initial functional program against which a model workload can be validated, and then to make perturbations to the model, rather than to the real program. In this way we can rapidly measure a comprehensive spectrum of responses from the system under test. These issues are presented in Section 4.5 and examples of workloads based on common algorithmic forms are illustrated in Section 4.6.
4.2 Soundness

Paragon is a language for modelling lazy functional language workloads. Therefore, to show that Paragon is sound, it is sufficient to show that any behaviour that can be described by Paragon can also be achieved by a real functional program.

We claim that Paragon models the following three aspects of real functional language workloads:

1. Arbitrary computation is modelled by the consumption of CPU cycles.

2. The dynamic call-tree of a Paragon program matches that of a real functional program.

3. The inter-task synchronisation properties of Paragon programs are a subset of those pertaining to real functional workloads.

In the following sections we show that the Paragon language is sound in each of these aspects by demonstrating that for each modelling construct, there is at least one functional language expression that can produce an equivalent workload.

For the five instructions which can take an arbitrary number of alternative branches (i.e. BLOCK, EVAL, RETURN, SPAWN, and TAIL), the proofs are broken into two parts. Initially, unconditional versions of the instructions are considered, and in Section 4.2.5 we show that the conditional branching mechanism can be modelled by a functional program.

4.2.1 Primitive Work

The simplest Paragon activity is described by the

\[ \text{WORK } r \]

instruction which causes an amount of computation to take place. The time required by the computation is determined by the value of a random variable \( r \) (see Table 3.1 on page 78). An equivalent workload can be generated by the following functional expression:

\[ \text{length (read "foo.data")} \]

where \( \text{length} \) is the function which takes the length of a list, \( \text{read} \) is the function which takes input from a file, and \( r \) characterises the probability density function for the length of the run-time input.

87
4.2.2 The Dynamic Call-Tree

The arcs of the dynamic call-tree of a paragon program are determined by the TAIL and SPAWN instructions. First consider the TAIL instruction. From Section 3.2.5 we know that the number of recursive tail calls made by a task is determined by the product of two random variables. The first, $m$, is supplied by the SPAWN instruction which created the task, and the second, $t$, is a property of the task itself. There is therefore one value of $t$, but there may be many values of $m$. Now, the following is an example of a real tail-recursive function:

\[
\begin{align*}
\text{trmap} \, [\,] \, \text{list} & = \text{list} \\
\text{trmap} \, (x:xs) \, \text{list} & = \text{trmap} \, xs \, ((\text{transform} \, x):\text{list})
\end{align*}
\]

This function can be compiled to generate exactly the call-tree obtained from a task template which contains a TAIL instruction. Any other activity within the task is generated by the transform function. If we then add the following function definitions:

\[
\begin{align*}
data & = \text{read} \, "\text{foo.data}" \\
f_1 & = \text{trmap} \, (\text{transform}_1 \, \text{data}) \, [] \\
\vdots \\
f_n & = \text{trmap} \, (\text{transform}_n \, \text{data}) \, []
\end{align*}
\]

the number of tail-calls made by trmap is determined by the amount of data in the file foo.data and the resulting length of the lists after transformation by functions \text{transform}_1 \ldots \text{transform}_n. This corresponds directly to the two-part characterisation for the Paragon TAIL instruction. Therefore, the call-trees specified by the Paragon TAIL instruction can be reproduced by real functional programs.

Now consider the SPAWN instruction. Spawns can occur from either tasks or declared sequences that are being evaluated during the lifetime of a task. Section 4.2.4 deals with the issue of locality of evaluation for synthetic closures, and the soundness of the synchronisation properties are demonstrated in Section 4.2.3. Therefore, if closures are treated as separate nodes in the dynamic call-tree then it is sufficient in this section to show that it is sound for each task and declared sequence template to possess an arbitrary
number of `spawn` instructions. Consider the following template definition:

```plaintext
f {
    ...
    Spawn label1 -> t1 <args...>
    ...
    Spawn labeln -> tn <args...>
    ...
}
```

A node with an identical out-degree can be constructed in the call-tree of a real functional program by a task (or closure) containing a function call with \( n + 1 \) strict arguments, as follows:

```plaintext
f = g (t1 args...) ··· (tn args...) (tn+1 args...)
```

When \( f \) is evaluated, the first \( n \) arguments to \( g \) are spawned as parallel computations, and the \( (n + 1) \)th argument is kept for local evaluation.

All dynamic call-trees created by the execution of Paragon programs, such that each node has an arbitrary out-degree, can therefore be reproduced by executing real functional programs.

### 4.2.3 Synchronisation due to `block` Instructions

Figure 4.1 illustrates the most general arrangement of `spawn` and `block` instructions. In Section 3.2.2, the restriction was made that `block` instructions may only reference preceding `spawn` instructions within the current template. Therefore, each `label-listi` is a list of spawn labels which form a subset, \( L_{i} \subseteq \{ label_{j,k} \mid 1 \leq j \leq i \leq n, 1 \leq k \leq n_{j} \} \). For completeness, we define \( L_{m+1} = \emptyset \).

Figure 4.2 shows a functional program segment from which it is possible to obtain a real workload with identical synchronisation properties to those of Figure 4.1. The desired synchronisation occurs when the compiler ensures that the last \( n_{i} \) arguments to functions \( f_{i} \) (for \( 1 \leq i \leq m + 1 \)) are spawned as parallel tasks (in the figure, these arguments are represented by arbitrary expressions \( expr_{i,j} \)). After spawning the arguments for \( f_{i} \), the current task enters the code for \( f_{i} \). For functions \( f_{1} \ldots f_{m} \) the code begins to execute the addition by first evaluating the left hand side. The functions \( g_{i} \) (for \( 1 \leq i \leq m + 1 \)) demand the values of a subset of their arguments such that there is a one-to-one mapping between this subset and the subset, \( L_{i} \), defined above.
Figure 4.1: The most general combination of SPAWN and BLOCK instructions.

It is therefore possible to arrange for a functional program to exhibit the same parent/child synchronisation properties as any legal arrangement of SPAWN and BLOCK instructions.

4.2.4 Synthetic Closures

Synthetic closures are created with DECLARE instructions and may contain BLOCK, EVAL, SPAWN, and WORK instructions in their own right and make reference to the environment of their defining task (see Section 3.2.4). Here we shall show that it is valid for closures to contain these instructions, and that all possible scenarios in which synthetic closures are evaluated can be modelled by a real functional program.

The following function definition demonstrates the validity of all of these mechanisms. We assume that \( f \) is being evaluated in a head strict context [Bur90b, WH87] so that
\[ f_0 = f_1 (expr_{1,1}) \cdots (expr_{1,n_1}) \]

\[ f_1 x_{1,1} \cdots x_{1,n_1} \]
\[ = (g_1 x_{1,1} \cdots x_{1,n_1}) \]
\[ + (f_2 x_{1,1} \cdots x_{1,n_1} (expr_{2,1}) \cdots (expr_{2,n_2})) \]
\[ \vdots \]

\[ f_m x_{1,1} \cdots x_{1,n_1} x_{2,1} \cdots x_{2,n_2} \cdots x_{m,1} \cdots x_{m,n_m} \]
\[ = (g_m x_{1,1} \cdots x_{1,n_1} x_{2,1} \cdots x_{2,n_2} \cdots x_{m,1} \cdots x_{m,n_m}) \]
\[ + (f_{m+1} x_{1,1} \cdots x_{1,n_1} x_{2,1} \cdots x_{2,n_2} \cdots x_{m,1} \cdots x_{m,n_m}) \]
\[ (expr_{m+1,1}) \cdots (expr_{m+1,n_{m+1}}) \]

\[ f_{m+1} = g_{m+1} \]

Figure 4.2: Equivalent parent/child synchronisation in a functional program.

the first item of the resulting list is evaluated immediately, within this task.

\[ f \ x_1 \cdots x_n \]
\[ = [\text{lhs} + \text{rhs}, c_1, \ldots, c_m] \]
where
\[ \text{lhs} = g_1 (c_1, \ldots, c_m) x_1 \cdots x_n \]
\[ \text{rhs} = g_2 (c_1, \ldots, c_m) x_1 \cdots x_n \]
\[ c_1 = h_1 x_1 \cdots x_n \]
\[ \vdots \]
\[ c_m = h_m x_1 \cdots x_n \]

If the first argument of both \( g_1 \) and \( g_2 \) is non-strict then closures will be constructed for the expressions, \( c_1 \ldots c_m \). The closures make reference to the arguments of \( f \). Therefore it is valid for a Paragon synthetic closure to reference the environment of the task in which it is defined. If \( h_i \) (for \( 1 \leq i \leq m \)) requires the value of any of its (non-strict) parameters then the evaluation of the parameter will be forced. Thus it is valid for synthetic closures to contain `EVAL` instructions. If any of the parameters to \( h_i \) are strict, then child tasks may legitimately be spawned to evaluate them in parallel. Thus it is valid for synthetic closures to contain `SPAWN` and `BLOCK` instructions. Trivially, \( h_i \) may contain any amount of primitive work that does not require tasks to be spawned. Therefore, it is valid for synthetic closures to contain `WORK` instructions.

A compiler will typically arrange for the evaluation of \( f \) to proceed by spawning one
of the arguments to the addition as a parallel task, and for the other argument to be evaluated by the current task. Therefore it is valid for Paragon tasks to evaluate their own declared sequences, and for them to be passed as arguments and evaluated by child tasks. Given that \( f \) is being evaluated in a head strict context then only the first item in the list will be calculated. Therefore, if any of \( c_1 \ldots c_m \) are not evaluated by \( g_1 \) or \( g_2 \) then their closures will be retained for possible evaluation later. Thus it is valid for Paragon tasks to return one or more of their closures to their parents for subsequent evaluation.

4.2.5 Conditionals

The five instructions, block, eval, return, spawn, and tail, each take the following general form:

\[
\text{instruction } \begin{cases} \text{alternative}_1 (pp_1^\% ) \\ \text{alternative}_2 (pp_2^\% ) \\ \vdots \\ \text{alternative}_n (pp_n^\% ) \end{cases}
\]

We know from Equation (3.3) that \( \sum_{i=1}^{n} pp_i \leq 100 \), so we can model this behaviour using a conditional expression constructed in the syntax of the Haskell kernel [HPJW+91]:

\[
f \text{choice} = \begin{cases} \text{alternative}_1 (pp_1^\% ) & \text{if } p < pp_1 \\ \text{alternative}_2 (pp_2^\% ) & \text{if } p \geq pp_1 \land p < (pp_1 + pp_2) \\ \vdots \\ \text{alternative}_n (pp_n^\% ) & \text{if } p \geq pp_{n-1} \land p < (pp_1 + pp_2 + \cdots + pp_n) \\ \text{default} & \text{otherwise} \end{cases}
\]

where choice is randomly distributed in the range \( 0 \leq \text{choice} < 1 \), and the default expression is a constant and thus costs nothing to evaluate. The cases are selected in one-to-one correspondence with the cases of the conditional Paragon expression. The expressions \( expr_1 \ldots expr_n \) are arbitrary functional expressions. We have already demonstrated that workloads constructed using unconditional block, eval, return, spawn, and tail instructions may be modelled by real functional programs; hence the conditional forms may also be modelled by functional programs by combining the previous definitions with the above case expression.

\[ ^1 \text{NB: just as Paragon does not insist that spawned tasks are evaluated in parallel, parallel functional language implementations such as [CPJ86, HPJ92] also allow spawned tasks to be evaluated sequentially when there is insufficient capacity in the machine to evaluate them in parallel.} \]
4.2.6 Summary

It has now been demonstrated that the following aspects of Paragon programs can be modelled by real functional programs:

- The number of CPU cycles consumed by a work instruction.
- The shape of the dynamic call-tree of tasks.
- The synchronisation properties of a parent task awaiting the completion of one or more of its children.
- The ability of synthetic closures to execute block, eval, spawn, and work instructions.
- The evaluation of synthetic closures, locally by their defining tasks, or remotely, by a descendent or ascendant task.
- The combination of alternative branches of block, eval, return, spawn, and tail instructions.

It is therefore concluded that workloads produced by Paragon programs can always be modelled by real functional workloads.

4.3 Can Paragon Describe all of the Required Stimuli?

In this section we show that Paragon can model all of the experimental features specified in Section 3.1. Recapping briefly, the key behavioural features of functional programs on which it has been decided to concentrate are:

Referential transparency The order in which events occur has no effect on the value of the final result of the program. This implies that the order in which parallel tasks are evaluated can be altered at run-time without compromising the integrity of the workload.

Non-strict evaluation Increased communication overheads are experienced when argument evaluation is non-strict.

Shared computation Laziness (as opposed to non-strictness) is a major cause of difficulty when attempting to predict the behaviour of functional programs (e.g. see
[Mah90, San90] which both avoid the issue). In Section 6.3.5 we see that sharing, especially, can have drastic effects on dynamic scheduling strategies. The order-independence afforded by referential transparency accentuates the instability introduced by sharing. It is required that the evaluation of shared objects produces the same behaviour as for real programs with respect to task inter-task synchronisation.

Data dependencies The order in which events occur is partially constrained in many functional programs by data dependencies [Gol88]. These provide synchronisation points at run-time and sequentialise the execution.

Dynamic instability Real functional programs operate on data supplied at run-time and to which the compiler has no access. Many algorithms are fundamentally dependent on the characteristics of the run-time input, thereby reducing the degree to which the behaviour of the functional program can be predicted at compile-time.

Higher-order functions Typically, functional programming languages provide higher-order functions. These complicate static analysis enormously but are primarily programmers' tools and can often be transformed away at compile-time. For completeness, however, we express an interest in modelling at least a limited degree of higher-order behaviour.

Recall from Chapter 3 that, in order to simplify the model, we allow explicit parameter passing between tasks to be ignored unless it is necessary to express one or more of the above characteristics.

4.3.1 Referential Transparency

The key concern of referential transparency is that the order in which tasks are evaluated should have no effect on the computed result. In a fully lazy environment, where shared expressions are computed at most once, and unused expressions are never computed, this is equivalent to reproducing the same nodes in the dynamic call-tree irrespective of the order in which the tasks are evaluated. The arcs of the call-tree may differ, however, depending on where the shared expression is evaluated.

Now, the run-time stimulus produced by Paragon tasks and declared sequences is fixed at compile-time in all respects apart from the following:

1. the actions determined by random variables, and
2. the **eval** instruction which may or may not cause the evaluation of declared sequences, depending on their run-time history.

Consider each of these in turn:

Case 1: Variation in behaviour due to random variables models a variation in the input to the program. But referential transparency makes no claims for consistency when the (code, data) closed form of the program is altered. We therefore assume that the actions of all instructions containing random variables are fixed for this investigation. This is equivalent to tracing the execution of a Paragon workload, and replacing it with a program that contains no random variables but which performs exactly the actions recorded in the trace.

Changes in execution order from one program run to another must have other causes such as a change in the number of processing elements employed for the calculation.

Case 2: In the absence of variation due to random variables, consider a task instance, \( T \), which contains a number of **eval** instructions operating on a set of declared sequences, \( Q_T \). From Section 3.2.4 we know that an instance of a declared sequence is evaluated at most once. Therefore, if \( Q \in Q_T \) has been evaluated prior to the execution of the eval instruction in \( T \) then it will not be evaluated by \( T \). On completion of \( T \), then, all of \( Q \in Q_T \) will have been evaluated exactly once, irrespective of their history.

Consider a declared sequence, \( Q_{i,q} \), created by task instance \( T_i \). During one program run, \( Q_{i,q} \) is executed by task instance \( T_j \), and during a second run it is executed by task instance \( T_k \), such that \( j \neq k \). For correctness we require that the behaviour of \( Q_{i,q} \) is consistent with referential transparency. We know from Sections 3.2.4 and 3.3.3 that a declared sequence may only reference other declared sequences via the formal parameters of the task instance which created it. In this case, and regardless of whether \( Q_{i,q} \) is evaluated by \( T_j \) or \( T_k \), the eval instructions in \( Q_{i,q} \) refer to \( Q_p \subseteq T_p \) where \( T_p \) are the formal parameters of \( T_i \). As before, on completion of \( Q_{i,q} \), all of \( Q \in Q_p \) will have been evaluated, irrespective of their history.

We conclude that the presence of a node in the dynamic call-tree generated by a Paragon workload is independent of the order in which tasks are executed.
4.3.2 Sharing and Non-Strictness

Figure 4.3 contains an example of the general sharing properties provided by Paragon. The declared sequence `comp` is shared by both parent and child and can be evaluated in one of three different places—the table within Figure 4.3 lists the probabilities for `comp` to be evaluated in each place. The shared computation can only occur once, hence the probabilities are exclusive and can be summed to determine the overall probability that `comp` is evaluated.

The Paragon `SPAWN` instruction always passes declared sequence parameters non-strictly. Only the `EVAL` instruction has the ability to cause declared sequence evaluation: strict argument passing can be simulated by preceding a `SPAWN` instruction with an `EVAL`.

The sharing properties of Paragon models are important. They require implicit inter-task communication and will cause suspensions to occur if two or more tasks are attempting to evaluate the same declared sequence. Suspensions are released when the evaluation completes. This is exactly the behaviour described for implementations of real functional programming languages in [PJCS89, CPJ86, AJ89b, WW87b], etc. In Section 4.4.4, the
current implementation of the Paragon task suspension and resumption mechanism is compared with that of the HDG-machine [KLB91].

4.3.3 Data Dependencies

Data dependencies occur when one expression cannot be evaluated without the result of another. Evaluation is usually forced in lazy functional programs when a primitive function (e.g. +) is applied, or when pattern matching forces at least the structure of a result to be computed before it can be passed as an argument to another function. In the following example, the order in which function evaluation occurs is fixed by data dependencies (assuming that each processing function pattern matches on its input):

\[
\begin{align*}
  f \text{ state} &= \text{new state} \\
  \text{where} \quad \text{state}_1 &= \text{process}_1 \text{ state} \\
  \text{state}_2 &= \text{process}_2 \text{ state}_1 \\
  \vdots \\
  \text{new state} &= \text{process}_n \text{ state}_{n-1} 
\end{align*}
\]

This is modelled in Paragon by the BLOCK instruction. Assuming no shared or higher-order data objects, the above example can be expressed as:

\[
\begin{align*}
  f \{ \\
  \text{Spawn child}_1 \rightarrow \text{process}_1 \\
  \text{Block child}_1 \\
  \vdots \\
  \text{Spawn child}_n \rightarrow \text{process}_n \\
  \text{Block child}_n \\
  \}
\end{align*}
\]

(When shared or higher-order data objects are introduced, they are modelled by declared sequences in the normal way and passed as parameters by the SPAWN instructions.)

4.3.4 Dynamic Instability

The inclusion of random variables in the Paragon specification provides the ability to model instability in a controlled way. To reason about the effects of instability it is necessary to be able to quantify the degree to which variable input data supplied at runtime can affect the behaviour of a program. With the exception of DECLARE and WORK, Paragon instructions may contain any number of alternative branches to be selected under the control of random variables. For instructions of the form:

\[
\begin{align*}
  \text{instruction} \quad \text{alternative}_1 \quad (pp_1\%) \\
  \vdots \\
  \| \quad \text{alternative}_q \quad (pp_q\%)
\end{align*}
\]
the probabilities, \( p_1 \% \ldots p_n \% \), express a very specific discrete random distribution for executing the alternative branches of the instruction. All other random variates (such as those used in the \texttt{work} instruction) are continuous distributions specified by ranges (see Section 3.2.7). The set of theoretical distributions is limited only by the current implementation of the Paragon compiler and may be extended whenever necessary.

In Section 4.6.1 an example is given where random variates are used to characterise the distribution of calls to partitioning and combining functions in a divide and conquer algorithm. They are also used to control the number of recursive calls made with respect to average expected input. In this way, the example workload is made to vary from one run to the next, such that key statistical characteristics observed over many experiments can be predicted in advance.

\subsection*{4.3.5 Higher-Order Behaviour}

A functional program contains a finite number of function definitions. This implies that specialised versions of functions can be constructed at compile-time and higher-order parameters eliminated. Moreover, the collecting interpretations of, e.g., [HY91, BHY89, Hud86] allow this to be optimised so that the number of specialised functions are minimised. However, for partially applied functions, especially those which are shared, the higher-order values are traditionally represented at run-time by the same closure mechanism which is used to describe unevaluated objects. Closures are synthesised in Paragon by declared sequences (see Section 3.2.4). Consider the higher-order function definition:

\begin{verbatim}
compose f g = composition
where composition x = f (g x).
\end{verbatim}

This function takes two arguments which are themselves functions, each taking one argument. The result of \texttt{compose} is also a function of one argument. When that function is executed it first applies \( g \) to its argument and then \( f \) to the result.

For the purposes of task modelling we might be unconcerned about the explicit representation of the parameters of \( g \) and \( f \). But, if \( f \) is strict in its argument then there is a data dependency and the order of events is such that \( g \) executes first, followed by \( f \). The following Paragon task template models this by equating the higher-order parameters \( f \)
and g with explicit template arguments, $1$ and $2$, respectively:

```plaintext
Compose {
  Declare result {
    Eval $2$ $1$
  }
  Return result
}
```

This synthesises the desired closure-based representation because explicit arguments are bound to declared sequences. The parameters to Compose are packaged into a lazy code sequence which is then returned in the unevaluated state. In essence this models second-order behaviour because the current Paragon definition restricts explicit parameter passing to tasks. No mechanism is available to parameterise declared sequences on synthetic closures in their own right. The arguments $1$ and $2$, and the result, therefore correspond to first-order functions.

A limited form of higher-order function passing can be modelled by adding a third explicit argument, thus:

```plaintext
Compose2 {
  Declare result {
    Eval $3$ (p%)
    Eval $2$ $1$
  }
  Return result
}
```

Compose2 might be used, for example, to model

```plaintext
compose2 g' f x = composition
  where composition y = f (g' x y).
```

where $g'$ is strict and $x$ is not a higher-order function. The percentage probability, $p$, determines the chance for the third argument to be evaluated by either of the functions $f$ or $g'$.

The main limitation of the Paragon representation of higher-order functions is that placed on partial applications. In the above example, the argument $x$ of compose2 is expressed explicitly by argument $3$ of Compose2. If fewer than three actual parameters are supplied when Compose2 is spawned then the remaining synthetic closures are automatically assumed to be unshared, empty declared sequences (see Section 3.2.4). However, for workload models, we are more interested in studying the effects on efficiency caused by the presence of higher-order functions at run-time, rather than describing specific functional language constructs. The simple mechanisms shown above for synthesising closures is sufficient for current modelling purposes.
4.4 Verifying the Implementation

There are a number of implementation issues to be addressed before synthetic workloads are constructed. Some of these issues are directly related to models which encompass speculative parallelism.

Active tasks and declared sequences are represented in the Paragon implementation by a descriptor which maintains a complete record of the state of the task. The state of a task is described by the following tuple: \((\text{template, tail-calls, program-counter, parent, local-seqs, parameters, return-values, active-children, block-count})\), and a number of minor, administrative items. The descriptor for a declared sequence is almost identical to this, but differs in that it has no entries for tail-calls or local-seqs, and is extended with an extra item, block-list, listing the descriptors that are blocked, awaiting completion of the current sequence.

In the above tuple, the template identifies which template was used to construct the current descriptor. The tail-calls counter indicates the number of recursive tail calls still to be executed for this task instance. The program-counter indicates the current instruction within the template.

The parent entry of the tuple is the task or declared sequence which created the current descriptor and to which results are to be sent. The local-seqs are references to the declared sequences bound locally within the current task; parameters are those sequences supplied as actual parameters to the current task; and return-values are those sequences returned by child task instances.

Prior to a child's return its entry in the list of return-values can also take a special value to indicate that the current task is blocked, waiting for the child to complete. The active-children are the number of child task instances that were previously spawned but have not yet returned. Finally, the block-count is the number of children on which the execution of the current descriptor is blocked. This counter is reduced whenever a child returns and is found to be one of those responsible for the suspension. A blocked descriptor is allowed to resume execution when its block-count reaches zero.

In the following verification, we shall make reference to the mechanisms employed by the implementation of the HDG-machine [KLB91], Flagship [WW87b], and the Four Stroke Reduction Engine [CPJ86]. The well documented implementations of these machines make them ideal candidates on which to base the comparison between Paragon and
real functional language implementations. The HDG-machine is one of the most recent parallel implementations for functional languages and embodies many of the innovations suggested by earlier implementations.

4.4.1 Memory Consumption

In Section 3.1.1 it was stressed that Paragon workloads must not consume unbounded amounts of memory when the equivalent functional program would not be expected to do so. Examining the tuples which record the state of tasks and declared sequences, we find that all entries, barring the block-list, have fixed size. Also, the list of Paragon instructions describing the actions of the task or declared sequence are stored uniquely in the original task template and are not copied to individual instances. This corresponds directly to the implementation of the HDG-machine [KLB91] which uses variable application (Vap) nodes. Vap nodes record a fixed amount of state and contain the equivalent of a block-list. (Lists of blocked tasks are found in many parallel implementations of functional languages from the Four-Stroke Reduction Engine [CPJ86] onwards.)

To model laziness, the actual parameters passed to a task by its parent, and the values returned to the task by its children, are initially represented by local or remote pointers [Hug85] which indicate the origin of the associated declared sequences. (There is nothing inherent in the Paragon language which requires the sequences to be duplicated, although this approach may be adopted if the experimenter is interesting in investigating the effects of duplicating shared workloads.) Therefore, to represent a task, the Paragon implementation requires approximately a constant multiple of the amount of space required by the HDG-machine. Moreover, real functional languages produce data objects which remain in existence after the tasks that created them have been garbage. Since there is currently no analogue for these in Paragon, the memory consumption of a Paragon workload is at most as big as a constant factor times that of a corresponding real program. This statement is only true, however, if the Paragon implementation garbage collects memory when it is no longer active.

A task descriptor can be garbage collected as soon as the end of the task is reached and there are no active (speculative) children. However, the locally-declared sequences must remain intact if any references to them still exist. This will be the case if the sequences are returned as the result of a task to its parent, or were passed as arguments to speculative descendants which have not completed.
The garbage collection of declared sequences is managed by a weighted reference-counting scheme [Bev87, WW87a, KLB91, Les89a, GRW88]. Normal reference-counting [Coh81] is unsuitable for message passing environments because increment and decrement messages might both be in transit simultaneously. This leads to the possibility of a race. If an increment message is followed by a decrement message and the latter overtakes the former in transit, it is feasible that the decrement will reduce the count on the object to zero. The object will then be garbaged prematurely. For weighted reference-counting, weights are associated with each object and with pointers to the object. The sum of the pointer weights is always equal to the weight on the object itself. When a pointer is deleted, its share of the weight is returned to the object. When a pointer is copied, its share of the weight is divided between itself and the copy. Only decrement message are ever transmitted to the object, thereby avoiding the race.

4.4.2 Guaranteeing the Absence of Cycles

Reference-count garbage collection will reclaim all inactive declared sequence descriptors, only if it is not possible to form cyclic dependencies between declared sequences. We must therefore demonstrate that cycles will not occur. All parametric, returned, and direct (named) references to declared sequences are bound to concrete instances of declared sequences created by task instances. To prove that cycles cannot occur it is sufficient, therefore, to show that, for any two distinct declared sequence instances, Q and Q', when Q makes a direct or indirect reference to Q', it is not possible for Q' to make a direct or indirect reference to Q. To clarify the proof we define the relation X Q to mean that the task or declared sequence X makes a (direct or indirect) reference to declared sequence Q. The above statement is therefore expressed by

\[ \forall Q, Q' [Q \neq Q' \land Q \sim Q' \Rightarrow Q' \not\sim Q] \]

Also, for task instances, T and T', define the relation T T' to mean that T' is a direct descendant of T.

**Proof**: Let \( T_i \) be an instance of a task and \( Q_{i,q} \) be a declared sequence created by \( T_i \). Let \( X_A \) be an ancestor of \( T_i \) (\( X_A \leftarrow T_i \)) such that the lineage of \( T_i \) originates from the 8th SPAWN instruction of \( X_A \) (\( \text{spawn}_{X_A,e} \)). Let the sequences returned as the result of \( T_i \) be \( R_i \), and the formal parameters of \( T_i \) be \( P_{i,1} \ldots P_{i,n} \).
From Section 3.3.3 we know that $Q_{i,q}$ may make references to other declared sequences via the following routes only:

1. $Q_{i,q}$ may reference the formal parameters, $P_{i,1} \ldots P_{i,n}$, of the task, $T_i$.

2. $Q_{i,q}$ may reference the return value, $R_j$, of one of its own child task instances, $T_j$.

Consider Case 1 where $Q_{i,q} \sim P_{i,z}$, for $1 \leq z \leq n$. To construct a cycle of references via this route, there must exist a $P_{i,z}$ such that $P_{i,z} \sim Q_{i,q}$. Now, the scope of direct references to $Q_{i,q}$ is limited to $T_i$. Furthermore, the binding of $P_{i,z}$ to a declared sequence takes place outside of $T_i$. Therefore, for $P_{i,z} \sim Q_{i,q}$, the latter must be returned to the ancestors of $T_i$ as part of $R_i$. However, by the return reference rule of Section 3.3.3, $P_{i,z}$ cannot be bound directly to $Q_{i,q}$ because $R_i$ can only be specified in the list of actual parameters of \texttt{SPAWN} instructions that occur subsequent to that which created $T_i$. The only remaining way in which $P_{i,z} \sim Q_{i,q}$ is if $P_{i,z}$ is bound to a declared sequence, $Q_\infty$, such that $Q_\infty \sim R_i$. There are three possible cases (illustrated in Figure 4.4):

1(a) $Q_\infty = Q_{k,s}$ which was created and returned by $T_k$, where $T_k$ is a descendant of the task or declared sequence $X_A$ such that:

$$(Q_{k,s} \sim R_i) \land (X_A \leftarrow T_k) \land (T_k \neq T_i) \land (T_k \not\sim T_i) \land (T_i \not\sim T_k).$$

When $T_k \leftarrow T_i$, set $X_A = T_k$ and use case 1(b). When $T_i \leftarrow T_k$, set $X_A = T_i$, $T_i = T_k$, and again use case 1(b).
Now, for $Q_{k,s} \sim R_i$, a formal parameter, $P_{k,z}$, of $T_k$ must be bound to $R_i$, and $Q_{k,s} \sim P_{k,z}$. $R_i$ must therefore be passed as an actual parameter by a spawn instruction, $\text{spawn}_{X_A,\sigma'}$, of $X_A$, because $X_A \leftarrow T_k$. But,

$$T_k \neq T_i \Rightarrow \sigma \neq \sigma'.$$

If $\sigma < \sigma'$ then $R_i$ is available to $\text{spawn}_{X_A,\sigma'}$ but, by the return reference rule, $Q_\infty$ is not available to $\text{spawn}_{X_A,\sigma'}$, hence $P_{i,z} \not\rightarrow Q_\infty$. Similarly, if $\sigma > \sigma'$ then $Q_\infty$ is available to $\text{spawn}_{X_A,\sigma'}$ but $R_i$ is not available to $\text{spawn}_{X_A,\sigma'}$, hence $Q_\infty \not\rightarrow R_i$. Therefore, no cycle of references can occur in this case.

1(b) There exists a task instance $X_A$ such that $X_A \leftarrow T_i$ and $Q_\infty$ is one of its locally-declared sequences. By the return reference rule, the return value $R_i$ can only be referenced directly by the task or declared sequence which spawned the associated child task instance. Therefore:

$$X_A \leftarrow T_i \Rightarrow Q_\infty \not\rightarrow R_i,$$

and no cyclic references occur in this case.

1(c) There exists a declared sequence $X_A = Q_\infty$ such that $X_A \leftarrow T_i$. Declared sequences have no locally-declared sequences of their own and do not return results. In this case, $R_i$ can only be referenced by $X_A$ and the declared sequences created by its descendants. Case 1(a), above, rules out cycles when $Q_\infty$ is created by a descendant of $X_A$, therefore $Q_\infty \sim R_i \Rightarrow Q_\infty = X_A$. If $P_{i,z}$ is bound to $Q_\infty$ then $Q_\infty$ must be passed as a parameter of $\text{spawn}_{X_A,\sigma'}$. However, the direct reference rule of Section 3.3.3 bans references to $X_A$ from within $X_A$. Therefore $Q_\infty \neq X_A$ and, again, no cyclic reference occurs.

The proof that no cycles occur via Case 2 is exactly that given in Case 1(c), above. We conclude, therefore, that cycles cannot be generated either via Case 1 or via Case 2. □

From the above analysis it is evident that cycles of references between declared sequences will never be generated and that reference count garbage collection is sufficient to ensure that all possible garbage is collected.
4.4.3 Deadlocks

Deadlocks are caused whenever there are two activities, $A$ and $B$, such that activity $A$ cannot proceed until activity $B$ has completed, and activity $B$ cannot proceed until activity $A$ has completed. Let us examine the activities of the Paragon instruction set.

Declare instructions are purely declarative. They perform no actions and, therefore, do not depend on the completion of other activities.

Work instructions are self contained. The implementation given in Figure 3.5 illustrates that no external dependencies exist to impede the completion of a work instruction.

Return, Spawn, and Tail instructions make reference to declared sequences. However, the references are constructed lazily such that the corresponding sequence descriptors are not required when the instructions are executed. This means that even if a reference is made to a child instance which has not yet returned its result, these instructions can proceed unhindered.

Consider the behaviour of the RETURN instruction. The descriptor of the parent to which a child sends its result cannot be garbage collected until all of its children have completed. Therefore it (or one of its subsequent tail calls) is guaranteed to exist when the child returns. The communication subsystem forcibly interrupts the activity of the parent to deliver the return value; hence the RETURN cannot be blocked.

Next, consider the SPAWN instruction which creates a new child task instance. This cannot fail to complete unless there is insufficient memory, at which point the whole program fails as would be expected for any real functional program that exhausts the heap.

Finally, consider the TAIL instruction which overwrites the current descriptor with a new instance of the same task. The implementation ensures that the return-values part of the state of the overwritten descriptor is carried forward to the new descriptor. When a child created by the overwritten descriptor returns, the new descriptor recognises it and deals with it appropriately. Therefore, the TAIL instruction does not have to wait for speculative children to complete before it
overwrites the current descriptor. Hence, TAIL cannot be blocked and is free from deadlock.

**Block** instructions suspend the execution of the current descriptor until the specified child task instances have completed. The completion of a descendant depends only on the actions of an ancestor if both are attempting to evaluate a shared declared sequence. However, in this case we know that the parent is executing a **BLOCK** instruction so cannot possibly be evaluating a declared sequence. **BLOCK** instructions are therefore free from deadlock.

**Eval** instructions suspend execution of the current task or declared sequence, X, if the evaluation of a declared sequence, Q, is requested, but Q is already being executed elsewhere, by another **EVAL** instruction. If X is itself a declared sequence then it appears that a cyclic dependency might be possible. If it were the case that X contains an **EVAL** instruction which causes the evaluation of another sequence Q, and that Q contains an **EVAL** instruction which requires the evaluation of X then deadlock would result. The first sequence to be evaluated will cause the second to start executing and will suspend until it is completed. When the second tries to cause the execution of the first, it finds that it is already executing and also suspends. However, we have already seen in Section 4.4.2 that it is impossible to form such cycles of declared sequence references.

The absence of deadlocks in request/response message passing is ensured by adopting the following protocol. When a task sends a request message, it suspends itself before listening for responses. When a request arrives for a sequence that is **not** being evaluated, an immediate response is sent. When a request arrives for a sequence which is already evaluating then the response is delayed until the evaluation is complete.

We conclude that the current Paragon language implementation is free from deadlocks.

**4.4.4 Suspension and Resumption**

Tasks and declared sequences suspend their execution for two reasons. Firstly, a **BLOCK** instruction forces the current instance to wait for the completion of one of its children. Secondly, an **EVAL** instruction attempts to execute a declared sequence which is already
being evaluated elsewhere: the current instance cannot continue until the evaluation is complete. In both cases, the current instance must suspend execution until it is re-awoken when the pending event occurs. The Paragon implementation maintains two separate task pools: one for newly sparked tasks, awaiting execution, and a second for resumed tasks. In the tradition of [WW87b, KLB91], only newly sparked tasks may be migrated to remote processing elements.

To facilitate resumption of a parent, when the children on which it is blocked have completed, we arrange for each child to send an explicit return message to the parent, as in the Flagship model [WW87b]. Since a parent can block on several children simultaneously, the receipt of a single return message may not be sufficient for the parent to resume immediately. Following the Flagship example once more, blocked parents maintain a count of the children on which they are still blocked. When the count reaches zero, the parent is resumed.

The second case differs from the first in that many descriptors can be blocked awaiting the completion of the same declared sequence. A common technique [CPJ86, WW87b, KLB91] is to maintain a list of tasks blocked on a computation. A separate list of blocked descriptors is maintained for every declared sequence. If a task (or declared sequence) attempts to execute a declared sequence, \( Q \), which is already executing, it is added to \( Q \)'s list of blocked tasks and suspends. When the execution of a declared sequence is complete, the list is checked and all of the blocked descriptors are resumed.

4.5 Validating Individual Workloads

In Chapter 2 (Section 2.2.3) one of the advantages given for using synthetic workloads in preference to real functional programs is the ability to construct large workloads with minimal effort. We also note that real functional workloads are more difficult to understand than synthetic workloads because the former are specified semantically and the latter, pragmatically. For these reasons, it is often laborious or impracticable to construct a real functional program which corresponds to the synthetic workload.

An alternative is to construct a collection of small functional programs which represent a set of standard algorithms and programming techniques. For each functional program, a synthetic workload is built and verified to produce similar run-time behaviour. The verified workloads can then be used as a basis for constructing new workloads. One
method of doing this is to take an existing experiment, based on real functional programs, and to reconstruct it using synthetic workloads. Perturbations can then be made to the synthetic workloads to satisfy a number of "what if?" scenarios.

4.5.1 Validation by Profiling

To ensure that the behaviour of a synthetic workload corresponds to that of a real program, measurements of both must be taken and compared. In Section 2.3.2, profilers were discussed briefly and the difference in application and system programmers' viewpoints was highlighted. The difference is mainly in the way that lazy evaluation is viewed. When laziness delays the evaluation of an expression, the system programmer is interested in the point at which the expression is evaluated, and therefore requires a profiler of the style suggested by [ADM88, RW92]. The application programmer, however, simply wants to know whether an expression is evaluated, and is happiest when the information is presented in terms of the lexical affinities of the original source program. Application programmers are therefore in need of lexical profiling tools, as suggested by [CPC91, SPJ92].

We can exploit this difference by providing both types of profiler which, used together, form a more comprehensive picture of the program's behaviour. For example, consider the following short programs:

Program 1

\[
\begin{align*}
  f & = (g \ x) / 18 \\
  & \text{where } x = (\text{expression}) \\
  g \ x & = (h \ x) * 10 \\
  h \ x & = x + 32
\end{align*}
\]

Program 2

\[
\begin{align*}
  f & = (g \ 10) / 18 \\
  g \ y & = (h \ x) * y \\
  & \text{where } x = (\text{expression}) \\
  h \ x & = x + 32
\end{align*}
\]

Program 3

\[
\begin{align*}
  f & = (g \ x) / 18 \\
  & \text{where } x = (\text{expression}) \\
  g \ x & = x * (h \ 10) \\
  h \ x & = x + 32.
\end{align*}
\]

The three programs are similar but differ in where the expression, \( x \), is declared and evaluated. Assuming that the programs are evaluated lazily, Table 4.1 shows the number of primitive operations (+, *, or /) counted for the functions of each of these programs.

If we consider just the count of primitive operations using the lexical profiling style, the results given in Table 4.1 for programs 1 and 3 are indistinguishable. Similarly, if
Table 4.1: Counting primitive operations using lexical and dynamic profiling styles.

<table>
<thead>
<tr>
<th>Program</th>
<th>function in which x is...</th>
<th>Number of primitive operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...declared ...reduced</td>
<td>lexical profile</td>
</tr>
<tr>
<td>1</td>
<td>f</td>
<td>1 + (p_x)</td>
</tr>
<tr>
<td>2</td>
<td>g</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>1 + (p_x)</td>
</tr>
</tbody>
</table>

\(p_x\) = number of primitive operations to evaluate x

we count just the primitive operations reported dynamically then programs 1 and 2 are indistinguishable. Therefore, when attempting to demonstrate that the behaviour of a synthetic workload is comparable to that of a real workload, the results will be more convincing if both types of profiling are employed. Where the effects of sharing and delayed evaluation are not being modelled explicitly, there will be no difference in the reports obtained from lexical and dynamic profilers and a single profile will suffice.

Notice that, when sharing is present in a program, changes in reduction order will affect the outcome of dynamic profilers. Consider the following program:

\[
f = (g \, x) + (h \, x)
\]

where \(x = \langle \text{expression} \rangle\)

\[
g \, x = x \times 10
\]

\[
h \, x = x + 32
\]

If the evaluation order for + is left to right then a dynamic profiler will credit g with the evaluation of x. If the evaluation order is right to left then h will be credited. If \(f\) is evaluated on a parallel processor then the reduction order may well be nondeterministic, fluctuating under the influence of factors such as the loading of individual processing elements and the reliability of inter-processor communications. A dynamic profiler will therefore return different results each time and must be executed a number of times to obtain an average behaviour. Lexical profilers do not suffer from this problem because the credit for an expression is always given to the function which lexically contained the expression in the source program. This is clearly a static relationship.
4.5.2 Constructing Profilers

In total, four profilers are required to provide both lexical and dynamic profiling for synthetic and real workloads. Profiling tools for Paragon workloads are comparatively easy to construct because task and declared sequence descriptors can be enlarged to contain any amount of information required to construct a complete profile (taking care not to compromise the memory consumption properties discussed in Section 4.4.1). However, profilers for real functional languages are less straightforward.

Dynamic profiling is the simplest of the two techniques to implement for real functional languages. A dynamic profiler simply requires code to keep track of the current function [ADM88], measuring time and memory allocations between successive function transitions. Higher-order functions can be a problem because profiling code added at compile-time does not know which function will be called at run-time. The solution given by [ADM88] is to profile higher-order parameters as separate functions but we can improve on this by applying techniques developed for lexical profiling in Appendix C.

Lexical profiling is more complex because individual expressions as well as whole functions are traced. The results of Table 4.1 illustrate that, under lazy evaluation, the function which evaluates an expression is not necessarily the one in which the expression was declared. For lexical profiling, the executable code must be amended so that extra information is carried with expressions at run-time. Appendix C gives full implementation details of the lexical profiling tool constructed to profile the real functional workloads of this chapter. The extra data added to a functional program provides correct results for higher-order functions without having to profile higher-order parameters separately.

4.6 Example Algorithms

A small set of basic algorithms and standard functional constructions are now presented with equivalent Paragon representations. For the first example, a detailed workload characterisation is also illustrated. Similar studies can be made for the remaining examples although they are not included here.

4.6.1 Divide and Conquer Algorithms

Divide and conquer is a popular method for implementing parallel algorithms (for example, the ZAPP project [BS81, SK84] is based solely on this class of algorithm). The
divide and conquer method divides a problem into sub-problems. If the solution to a sub-problem is not trivial then it too is divided into further sub-problems. By repeated division, trivial problems are eventually obtained whose solutions are known or can be calculated directly. At each level of the division, the results of the sub-problems are combined to produce the solution for that stage.

We shall now attempt to characterise a divide and conquer workload using Paragon. Figure 4.5 shows a Paragon program constructed from two templates. The first template, divide and conquer, represents the three stages of a single recursive step of a general divide and conquer algorithm. The work items at the start and end of the template simulate the average work required to divide the problem and to combine the results, respectively. The spawn items simulate the recursive steps and the percentage probabilities, \( r_i \), determine the characteristics of the input data (e.g. how well it is balanced on average). In representing the quicksort algorithm, \( n = 2 \) and, for perfectly random lists, \( r_1 = r_2 \).

**Hypothesis 4.1** To guarantee termination, the condition \( \sum_i r_i < 100 \) must hold.

**Proof:** Initially there is only one instance of the divide and conquer template; therefore set \( t_0 = 1 \). An instance spawns an average of \( c \) recursive child instances, where \( c = \sum_i r_i / 100 \). Hence, the number of instances executing by the \( n \)th recursive generation is given by \( t_n = \sum_{j=0}^{n} c^j \). The time taken to execute individual instances is fixed and finite. Therefore, the total time to execute the whole simulation is finite iff there are a finite number of instances. The sum to infinity, \( t_\infty \), is given by \( \sum_{j=0}^{\infty} c^j \) which converges to \( 1/(1-c) \) when \( c < 1 \) and diverges otherwise. \( \square \)

The second template of Figure 4.5, leaf, represents the invocations for which no sub-problems are generated. For quicksort this is the case when the input list has zero length. The recursive spawns can be optimised by employing the following syntax:

```
Spawn rec1 -> divide and conquer (r1\%)
          -> Work w1
```

In this case the leaves of the call tree are not represented by tasks in their own right but simply by work instructions executed by the 'parent' task.

The Paragon definition makes no statement about the run-time execution strategy. For example, the implementation is at liberty to arrange for one of the spawned children to be executed as part of the current task, without placing a separate task descriptor into a task pool. This is discussed in greater detail in Section 3.3.4.
An Example Divide and Conquer Algorithm: Quicksort

A good example of the divide and conquer technique is provided by the quicksort algorithm for sorting a list of numbers into ascending order:

```haskell
qsort : [num] -> [num]
qusort [] = []
qusort (x:xs) = qsort smaller ++ [x] ++ qsort larger
  where smaller = filter (<x) xs
        larger = filter (>=x) xs
```

The algorithm is based on the following three steps:

**Divide the problem** A pivot point is chosen (in this case the first item in the list) and all the numbers in the list which are (i) smaller than the pivot and (ii) larger than the pivot are found.

**Solve the sub-problems** Quicksort is called recursively on sub-parts (i) and (ii) respectively.

**Combine the results** The ++ operator is used to concatenate the sorted sub-lists.

The first step in constructing a synthetic version of the workload is to characterise the typical behaviour of the real workload. The quicksort program was executed for sixty different input lists each containing 100 randomly generated integers, and uniformly distributed in the range 0 ≤ i < 100. The complete set of results is shown in Table B.1, in Appendix B, and are summarised here in Table 4.2. We shall make an analysis of the qsort algorithm, paying particular attention to the expected behaviour of the
partitioning and combining functions. The analysis provides theoretical limits and is supplemented by the experimental results to help determine sensible random distributions for the $w_c$ and $w_p$ variates of Figure 4.5. The resulting synthetic workload is then executed and its behaviour is compared with that of the real workload.

It can be inferred from Table B.1 that the total number of calls made to `qsort` is given by \(2n + 1\) where \(n\) is the length of the input list.\(^2\) It is evident, however, that the number of recursive calls made by the partitioning function, `filter`, is dependent on the contents of the input list. The number of recursive calls made to the combining function, `++`, is also variable.

Quick sort is most efficient when its call tree is perfectly balanced, as in Figure 4.6(a), and is least efficient when the nodes of its call tree form a single spine, as in Figure 4.6(b) [AHU83]. We can use this to determine upper and lower bounds for the total number of calls made to `filter` during the lifetime of the algorithm. The time-complexity of `filter` is \(O(l)\) where \(l\) is the length of its input list. At each (non-leaf) node in the `qsort` call tree, `filter` is invoked twice on the tail of the current list. Therefore, the amount of work performed by `filter` during the lifetime of `qsort` is proportional to \(\sum_{\text{nodes}} 2l\).

Consider the worst-case call tree of Figure 4.6(b). Summing the calls to `filter` over the nodes of the tree we get

\[
P_{\text{max}} = \sum_{i=1}^{n} 2i = n(n + 1).
\]

\(^2\text{PROOF: by induction on sublists of length } m \text{ and } n - m - 1, \text{ constructed by a call to } \text{qsort} \text{ with a list of length } n. \text{ When } n = 0, \text{ qsort returns immediately, satisfying } 2n + 1 = 1. \text{ Extending the input from } n \text{ to } n + 1 \text{ elements gives sublists of length } m + 1 \text{ and } n - m - 1 \text{ which require a further } 2(m + 1) + 1 \text{ and } 2(n - m - 1) + 1 \text{ calls to } \text{qsort}, \text{ respectively. Summing the calls completes the induction:}

\[1 + (2(m + 1) + 1) + (2(n - m - 1) + 1) = 2(n + 1) + 1.\]
Contrast this with the perfectly-balanced tree of Figure 4.6(a). Here, \( n = 2^z - 1 \), hence \( z = \log_2(n + 1) \). At each node and leaf, \( l = l_i = 2^z - 1 - 1 \), where \( i \) is the distance of the node from the root of the tree. The total number of calls to filter at each node is given by

\[
2L_i = 2(2^z - 1) - 1.
\]

For a perfectly-balanced binary tree, there are \( 2^i \) nodes (or leaves) at distance \( i \) from the root. Therefore, the total of calls to filter is given by

\[
\sum_{nodes} 2L = \sum_{i=0}^{z-1} 2^i \cdot 2(2^z - 1) = 2[2^z(z - 1) + 1].
\]

And, substituting for \( z \), we get

\[
P_{\text{min}} = 2[(n + 1)(\log_2(n + 1) - 1) + 1]. \tag{4.2}
\]

A similar analysis is performed for the combining function, ++. Firstly, we note that ++ is used by qsort in an expression of the form:

\[
\text{left} ++ [x] ++ \text{right} \tag{4.3}
\]

where left and right are sorted sublists. The expression is invoked by calls to qsort which form the (non-leaf) nodes of its call tree. Therefore, exactly \( 2n \) calls are made from qsort to ++, as confirmed by the results of Table B.1. However, the number of recursive calls made by ++ depends on the length of its left-hand argument. Therefore, the minimum number of calls to ++ occurs when left is empty for every call to ++. If this
is the case then (assuming left-precedence for ++) zero recursive calls are made by the
first ++ of Expression 4.3 and one recursive call is made by the second ++. The minimum
number of calls to ++ during the lifetime of qsort is thus

\[ c_{\text{min}} = 3n. \quad (4.4) \]

As for filter, the maximum number of calls to ++ occurs when the call tree for
qsort is that of Figure 4.6(b). It is given by

\[ c_{\text{max}} = 2n + \sum_{i=1}^{n} i + (i + 1) = n(n + 4). \quad (4.5) \]

For the experimental results reported in Table 4.2, \( n = 100 \). Hence, from Equations 4.1
4.2, 4.4, and 4.5:

\[
\begin{align*}
\text{p}_{\text{min}}(100) &= 1145 \\
\text{p}_{\text{max}}(100) &= 10100 \\
c_{\text{min}}(100) &= 300 \\
c_{\text{max}}(100) &= 10400
\end{align*}
\]

We now have theoretical limits on the total number of calls made to the partitioning
and combining functions. The next step is to obtain a suitable function to describe the
distribution of their call counts so that random variables can be constructed to synthesise
the resulting work. Table 4.2 contains the sample mean and sample deviation recorded
for the experiment. The histograms shown in Figures 4.7 and 4.8 respectively illustrate
the observed frequency with which the number of calls to filter and ++ occurred. It is
appropriate to approximate the bell-shaped, observed distributions with probability density functions (pdfs) based on the beta distribution [Jai91, p.484] [Kre83, p.A-57] because
the number of calls is bounded in both cases. The normalised pdfs are superimposed on
the histograms in Figures 4.7 and 4.8 and are calculated as follows.

For the pdf, \( f(z) \), distributed according to the beta distribution with shape parameters \( a \) and \( b \),

\[ f(z) = \frac{z^{a-1}(1-z)^{b-1}}{\beta(a,b)}, \quad \text{where} \quad \beta(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}. \]

The mean of \( f(z) \) is given by \( a/(a + b) \) and the variance by \( ab/[(a + b)^2(a + b + 1)] \)
[Jai91]. However, \( f(z) \) is defined for \( 0 \leq z \leq 1 \); thus, for the observed call-frequencies,
each call-count, \( z \), must be scaled to \((z - z_{\text{min}})/(z_{\text{max}} - z_{\text{min}})\). Therefore, let

\[ m = \frac{z - z_{\text{min}}}{z_{\text{max}} - z_{\text{min}}}. \]
For a sample size of $n$, sample variance is given by $s^2 = \sum_{i=1}^{n} n(x_i - \bar{x})^2/(n - 1)$. Hence the corresponding scaling for variance is given by

$$v = \frac{s^2}{(x_{\text{max}} - x_{\text{min}})^2}.$$

Solving $m = a/(a + b)$ and $v = ab/[(a + b)^2(a + b + 1)]$ simultaneously for $a$ and $b$, we get

$$a = m \left[ \frac{m(1 - m)}{v} - 1 \right] \quad \text{and} \quad b = (1 - m) \left[ \frac{m(1 - m)}{v} - 1 \right].$$

For filter: $a = a_p = 7.959$, $b = b_p = 190.7$, and

$$f(p) = \frac{p^{6.959}(1 - p)^{189.7}}{2.844 \times 10^{-15}}.$$

For ++: $a = a_c = 29.72$, $b = b_c = 946.1$, and

$$f(c) = \frac{c^{28.72}(1 - c)^{945.1}}{7.955 \times 10^{-59}}.$$

Equations 4.7 and 4.8 can be used in conjunction with the mean profiled execution times recorded in Table 4.2 for filter and ++ to determine suitable random variates for $w_c$.  

Figure 4.7: The distribution of total calls to the filter function.
and $w_p$. (The functions superimposed over the histograms of Figures 4.7 and 4.8 are normalised to match the results obtained experimentally (see Table 4.3)).

For the real $qsort$ program, the costs of partitioning a problem into sub-problems, and of combining the results, diminish as the problems get progressively smaller. However, apart from recursive tail-call specification, there is currently no mechanism in Paragon for relating the value of a variable in a task instance to the value of a variable belonging to its parent. Therefore, we must approximate the relationship between successive recursive

![Figure 4.8: The distribution of total calls to the ++ function.](image)

<table>
<thead>
<tr>
<th>Function</th>
<th>$f(x)$ Eqn.</th>
<th>Area under $f(x)$ ($= A$)</th>
<th>Bucket size ($= B$)</th>
<th>Normalisation ($= B/A$)</th>
<th>Peak $f(x)$ plotted at</th>
<th>Normalised peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>(4.7)</td>
<td>8954.9</td>
<td>100</td>
<td>$1.117 \times 10^{-2}$</td>
<td>(1460,30.1)</td>
<td>(1460,0.336)</td>
</tr>
<tr>
<td>++</td>
<td>(4.8)</td>
<td>10099.4</td>
<td>36</td>
<td>$3.565 \times 10^{-3}$</td>
<td>(596,73.4)</td>
<td>(596,0.262)</td>
</tr>
</tbody>
</table>

Table 4.3: Normalising the beta distribution.
calls to qsort. Two relatively straightforward options are available. The first is to ran-
randomly generate individual partitioning and combination costs each time qsort recurses.
This provides the expected degree of variability, albeit with arbitrary incidence. Unfor-
fortunately, the sum of random variates does not conform to the required beta distribution.
A simpler approximation is made by employing the Paragon range: BETA* a b l h. The
* ensures that a single random variate is generated when the workload begins executing
and remains constant throughout the lifetime of the workload. The beta distribution
over the whole workload is achieved by calculating w_p and w_c as follows:

\[ w_p = \text{BETA} \ast a \ b \ \frac{t_p \cdot P_{\text{min}}}{n} \ \frac{t_p \cdot P_{\text{max}}}{n} \]

and

\[ w_c = \text{BETA} \ast a \ b \ \frac{t_c \cdot C_{\text{min}}}{n} \ \frac{t_c \cdot C_{\text{max}}}{n} \]

The factors t_p and t_c are determined experimentally and produce overall timings which
coincide with the mean times attributed to filter and ++ respectively.

The values r_1 and r_2 of the divide_and_conquer template are still to be calculated.
We know that the number of calls to qsort is 2n+1 for input lists of length n. Examining
Figure 4.6, we see that n+1 calls are represented by the leaves of the call trees and that n
calls are represented by nodes. For a single execution of the workload, 1/(1-(r_1+r_2)/100)
instances of the divide_and_conquer template are constructed. When r_1 = r_2 = r, the
average number of nodes is given by n = 1/(1 - 2r/100), hence

\[ r_1 = r_2 = r = \frac{50(n-1)}{n} . \tag{4.9} \]

For n = 100 we have r_1 = r_2 = 49.5.

The final variable to be defined is w_l. This simulates the work executed by the n+1
calls to qsort which are represented by leaf nodes in the call tree. As for t_p and t_c, w_l is
determined experimentally.

4.6.2 Tail-Recursive Loops

A very common construct in functional programming is that of the tail-recursive loop.
For example, many standard function definitions such as dropWhile and foldl (e.g. see
the definitions given for the Haskell prelude [HPJW+91]) take the general form:

\[
\begin{align*}
\text{f}_{\text{tail}} \ x_1 \ldots \ x_n &= g_0 \ x_1 \ldots \ x_n \\
&= \text{f}_{\text{tail}} \ (g_1 \ x_1 \ldots \ x_n)\ldots(g_n \ x_1 \ldots \ x_n) &\text{for case 1} \\
\end{align*}
\]

for case 2
Parallelism can be obtained from the above function definition by strictness analysis over the functions \( f_{\text{tail}} \) and \( g_0 \ldots g_n \). It should be noted that this is a very restricted form of parallelism that leads only to a fixed number of parallel tasks, the evaluation of which will usually be expected to complete prior to the tail-call. The resulting activity will thus be the sequential generation of batches of parallel tasks and will be most useful when used in conjunction with inherently parallel techniques such as divide and conquer. For example, Figure 3.3 (on page 66) shows the run-time instances of tasks of a divide and conquer algorithm. At each stage of the recursion, a number of sub-processes are created, any or all of which might take the form of a tail-recursive task that spawns batches of parallel children.

Tail-recursion is modelled explicitly in Paragon using the \texttt{TAIL} instruction (see Section 5.2.2). Where \( f_{\text{tail}} \) and \( g_i \) (for \( 0 \leq i \leq n \)) are strict in all of their arguments, the behaviour of a call to \( f_{\text{tail}} \) can be modelled by the Paragon templates shown in Figure 4.9. Here, \texttt{CallFtail} spawns an initial instance of \texttt{Ftail}, corresponding to the initial call to \( f_{\text{tail}} \). The synthetic closures \( q_1 \ldots q_n \) represent the initial arguments to the function.

\texttt{Ftail} spawns parallel tasks to evaluate its arguments. It then performs some work, and spawns parallel tasks to calculate the next set of arguments. Finally, \texttt{Ftail} creates a new instance of itself with a tail-recursive call, passing the new argument values lazily. A total of \( t \) tail-calls are made and, after the final call, \texttt{Ftail} returns control to \texttt{CallFtail}.
which then spawns an evaluation of the final set of arguments in parallel with an instance of template $G_0$.

4.6.3 Stream Processing

Lazy functional languages are ideally suited to processing continuous (and possibly infinite) streams of data by using the composition of a series of functions [Hug89] (this is sometimes called a pipeline of functions). Function composition is expressed with the higher-order operator $\circ : (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$, defined as follows:

$$f \circ g = \text{comp}$$

where $\text{comp} \ x = f(g \ x)$.

Sequential Stream Processing

Consider the function $f_0 : [\tau_0] \to [\tau_n]$, defined thus:

$$f_0 = \text{map} (f_n \circ f_{n-1} \circ \cdots \circ f_1)$$

where $f_i : \tau_{i-1} \to \tau_i$, for $1 \leq i \leq n$. This produces a simple, inherently sequential form of composition where $n$ subsidiary functions are called each time map recurses. The Paragon templates shown in Figure 4.10 model the data dependency between successive subsidiary functions, resulting in sequential behaviour. Every subsidiary function, $f_i$ is represented by a separate task template $F_i$ and all but $F_1$ contain a \textsc{spawn} instruction which immediately blocks pending the child's return. This will cause the child process to be executed on the local processor and the parent to be resumed when it returns.

The initial template, $F_n$, also contains a recursive tail-call, modelling the activity of the \textsc{map} function. The length of the input list is characterised by the random variable $l$. 

![Figure 4.10: A Paragon model of a sequential pipeline.](image)
The calculation can also be modelled by a single, tail-recursive loop as shown in Section 4.6.2. This is equivalent to a compiler combining the actions of the main function \( f_0 \) and the subsidiary functions, \( f_1 \ldots f_n \) into a single task.

**Parallel Stream Processing**

Another form of function composition occurs when \( f_0 \) is defined in the following manner:

\[
f_0 = f_n \circ f_{n-1} \circ \ldots \circ f_1
\]

where, for \( 1 \leq i \leq n \), \( f_i : \mathbb{N}_{i-1} \to \mathbb{N}_i \) and

\[
f_i \, \text{xs} = \text{map} \, g_i \, \text{xs}
\]

where \( g_i = \text{processing-function} \).

When \( f_0 \) is applied to an input list, \( l_0 \), the intermediate lists, \( l_1 \ldots l_{n-1} \), will be constructed by the subsidiary functions \( f_1 \ldots f_{n-1} \) and a result list, \( l_n \), will be constructed by \( f_n \). Under lazy evaluation only those parts of the lists that are required for the result will be generated [PJ87b]. Therefore, if the result list is consumed one item at a time then the evaluation order will be such that, when the first item of the result list is demanded, just the first item of each of the lists \( l_1 \ldots l_n \) will be calculated, in sequence. This behaviour continues for each item demanded from the result list until the end of the input list is reached. The execution of the subsidiary functions and the generation of the intermediate lists is thus interleaved.

The demand-driven behaviour described in the previous paragraph assumes the absence of strictness information (see Section 1.3.2). If it is known in advance that every item in the result list, \( l_n \), is going to be demanded at some stage then it is safe to calculate the whole list as soon as possible. This is an example of *hyper-strictness* and, by utilising a mechanism such as Burn's evaluation transformers [Bur87], it is possible to force all of the intermediate lists to be hyper-strict also.

In the presence of hyper-strictness, every \( f_i \) can be made to execute as a separate task and so each stage of the calculation could be executed on a separate processor. Some synchronisation between the processors is required because, for \( f_i \) \( (1 < i \leq n) \) to calculate the \( x \)th element of the list \( l_i \), the function \( f_{i-1} \) must have finished calculating the \( x \)th element of \( l_{i-1} \). If the value is not yet available then the process executing \( f_i \) must suspend and if there is no other work available the processor will idle. There will also be a communications overhead because data flows between successive stages. However,
when the processing time for the successive stages is sufficiently high (with respect to the communications latency) the parallelism achieved will result in an overall speedup.

Figure 4.11 illustrates these principles for an example where \( n = 4 \) (i.e. four processors are used) and where an input list of three items is supplied. This example shows a relatively skewed timing graph because there are fewer items per list than there are processing elements. When larger input lists are used the idle times at the beginning and end of the timing period account for a much smaller fraction of the processing time.

This is a form of data-parallelism [HS86] and can be approximated [HM88] using the Paragon model shown in Figure 4.12. Here, all of the functions \( f_i \) are modelled by the Process task template.

The Main template starts off the simulation by spawning an instance of the tail-recursive template ParThreads. In turn, ParThreads spawns a child instance of Thread, passing its first argument as a parameter to the child. Immediately after spawning the child, the instance of ParThreads tail-calls a new instance of itself, lazily passing on the result of the child. In this manner a number of parallel child threads are created, each of which has access to the result of the preceding thread. On completion of the last tail-called instance of ParThreads, a return is made to Main, which then attempts to evaluate the result of the final thread. It therefore blocks until the final thread has completed. (Notice that the results of the threads are not bound to declared sequences. By default, they are bound to an unshared, empty declared sequence. This produces
the correct synchronisation because, as for a real declared sequences, the empty result is returned only when the thread completes.

Each thread models a pipeline of functions, $g_n \circ g_{n-1} \circ \cdots \circ g_1$, operating on a single element of the input list. The pipeline is similar to that shown for sequential stream processing, above. The data dependencies of the functions are therefore represented, but what of the synchronisations required between elements of the resulting lists? This is obtained by the arguments that are passed from thread to thread. The \texttt{EVAL} instruction of each instance of \texttt{Thread} first waits for its child to complete and then attempts to evaluate the result of the previous thread. However, this will not be available until the previous thread has completed. Consequently, and in accord with the timing diagram of Figure 4.11, the completion of threads will be skewed.

Although is it possible to reproduce the end effects of the synchronisations between threads, a model of synchronisation on intermediate lists is not attempted. For correct simulation of data-parallel processing, the Paragon language must be extended to synthesise compound data structures (see Section 3.4).

4.7 Summary

In this chapter it was shown that workloads constructed using the Paragon language are able to model the behaviour of lazy functional programs as far as the requirements listed in Chapter 3. It was also demonstrated that Paragon programs always describe
workloads that could have been generated with lazy functional programs.

Implementation issues have been discussed, and it has been shown that the Paragon language is inherently free from deadlock and is amenable to reference count garbage collection because cyclic references are not constructed. The use of profiling tools as a method for validating workloads was described, and in the final section, a number of example workload constructions were demonstrated. These correspond to general algorithmic techniques employed by functional programmers and can be extended and combined as necessary to produce complete workloads.
Chapter 5

Time-cost Analysis of Paragon Workloads

A time-cost analysis of Paragon programs is developed in this chapter. The work demonstrates that it is possible to estimate the time-cost of programs which contain shared components and which are executed using a lazy evaluation strategy. The ultimate aim of the analysis is to derive an ordering over task templates based on the expected execution times for the associated task instances. In Chapter 6 we show how this information can be used at run-time to aid dynamic scheduling decisions. Issues of parallel execution such as inter-processor communications latency are also dealt with in Chapter 6. The analysis presented in this chapter deals solely with task execution times.

5.1 An Introduction to the Time-cost Analysis

A good starting point for calculating the cost of a task is given by the amount of primitive work performed by the instance. This is defined by the WORK instructions in the template (and the default WORK branch of SPAWN instructions). Secondly, EVAL instructions cause declared sequences to be evaluated; since these may also contain WORK instructions they too add to the cost of the task instance. Declared sequences can however be shared, hence it is not always certain that executing an EVAL instruction will result in work being performed. Sharing severely complicates the analysis so we choose to ignore the problem for the time being and return to it in Section 5.3 having first established a simpler analysis for programs which contain no EVAL instructions. RETURN, DECLARE, and BLOCK instructions do not make a direct contribution to the cost of a task. Therefore,
these shall also be ignored in the initial analysis, which leaves only the creation of child
task instances by spawn instructions. The effect of child instances on the expected cost
of a task demands further consideration.

To clarify the contribution that child tasks make to the cost of a parent it is useful to
contrast the demands of a dynamic scheduler with those of a compile-time algorithm that
partitions a functional program into task units. A compile-time partitioning strategy may
try to regulate the size of individual task units in terms of the primitive work performed
within the boundary of each task. Here, the cost of a child task has no bearing on the
cost of the parent because it falls outside the boundary of the parent and is treated as a
separate entity. When dealing with the use of time-cost analysis for dynamic scheduling
purposes, however, the approach is different. Here the aim is to ensure that, when a
processor requests work, it should be kept sufficiently busy that repeat requests for work
are minimised. Taking as an example a partitioning and scheduling strategy used on the
Grip multiprocessor [HPJ92] we find that program partitioning is performed by placing
spark annotations in the code. If a spark is acted upon at run-time then a child task
is created as a separate entity. A run-time throttling strategy is employed which causes
sparks to be thrown away whenever it is useful to do so (e.g. the availability of work on
the local processor is low or it is simply faster to execute a child process locally). In this
case the work that would have been performed by a child task is subsumed by its parent.
Therefore the potential cost of the parent must include that of the child. This applies to
any scheduling strategy that allows the work of a child task to be executed on the same
processor as its parent.

In Paragon models, task templates contain spawn instructions to indicate which child
tasks are to be activated. These correspond directly to the grip spark annotations and
are used either to create parallel children or to generate local work, depending on the
chosen run-time scheduling strategy.

5.1.1 Dealing with Recursion

Since Paragon templates can be defined recursively, the time-cost analysis must be able
to cope with recursion. The termination conditions of recursive Paragon definitions are
specified probabilistically. Therefore the analysis is simpler than an equivalent analysis
for real functional programs because, for these, it is often possible to detect a base-case
only in the presence of run-time input. Consider, for example, the following function
which takes a list of numbers and returns the list up to (but not including) the first zero encountered:

\[
\begin{align*}
\text{before} :: [\text{num}] & \rightarrow [\text{num}] \\
\text{before} & [\text{num}] = [\text{num}] \\
\text{before} (x:x:\text{s}) & = [\text{num}], \quad \text{if } x = 0 \\
& = x : \text{before } x \times \text{s}, \quad \text{otherwise}
\end{align*}
\]

The order-of-magnitude time-complexity of this function is \(O(n)\) in the length of the list up to the first zero. However, the time cost cannot be estimated without further information about typical run-time input. If before is evaluated in a hyper-strict context then its result is demanded in full. This behaviour is modelled by the following Paragon template:

\[
\text{before} \{ \\
\quad \text{Work } w_1 \\
\quad \text{Spawn } \text{recurse} \rightarrow \text{before } (p\text{s}%) \\
\quad \text{Block } \text{recurse} \\
\quad \text{Work } w_2 \\
\}
\]

The time-cost of the model workload can be estimated because the probability, \(p\text{s}\%), of recursively spawning fully characterises typical run-time input.

## 5.2 A Simple Time-Cost Analysis without Sharing

A simple time-cost analysis is developed in this section in the absence of shared workloads (the \texttt{DECLARE} and \texttt{EVAL} instructions are ignored in the analysis until Section 5.3).

Formally, a Paragon program can be represented as a directed, cyclic graph, \(G\), made up of vertices, \(V\) (which constitute the set of task templates), and of edges, \(E\) (which describe the set of all possible spawns). An instance of template \(T_i \in V\) is free to spawn a task using template \(T_j \in V\) if there is at least one corresponding edge \(e_{(ij)} \in E\) (the subscript \(k\) may be omitted if there is exactly one edge connecting \(T_i\) to \(T_j\)). Figure 5.1 shows a simple Paragon program containing three mutually recursive task templates. The graph for this program is shown in Figure 5.2. There is a one-to-one correspondence between the edges, \(e_{(ij)}\), and the branches of \texttt{SPAWN} instructions in the program. For each edge there is an associated probability, \(p_{(ij)}\).

Now, let \(T\) be a subset of \(V\) and \(E_T = \{e_{(ij)} \mid e_{(ij)} \in E \land T_i \in T \land T_j \in T\}\). \(T\) describes a mutually recursive set of tasks if and only if \((T, E_T)\) forms a strongly connected component of \(G\). We say that an SCC, \(S\), is \textit{maximal} if and only if \(S' \supseteq S\).
Figure 5.1: A small Paragon program with mutually recursive task templates.

Figure 5.2: The graph of the Paragon program shown in Figure 5.1.
S ⇒ S' = S, for all SCCs \( S' \subseteq G \) (in Figure 5.2 the whole graph forms a maximal SCC). Subsets of \( V \) whose graphs \( (T, E_T) \) are maximal SCCs of \( G \) will be denoted by a subscript, \( \rho \) (e.g. \( T_{\rho} \)).

### 5.2.1 First Order Recursion

The before function from Section 5.1 exhibits first order (or linear monadic) recursion [Bus87, chapter 4]. For this type of recursion, \(|T_R| = 1\) for all \( T_R \subseteq V \). Therefore, no maximal SCC contains more than one vertex.

The cost expression, \( cT \), for a template, \( T \), which contains a single \texttt{SPAWN} instruction of the form:

\[
\texttt{SPAWN} \rightarrow T \ (ps\%),
\]

and no \texttt{TAIL} or \texttt{EVAL} instructions, is given by:

\[
cT = w + \frac{ps}{100} cT = w \sum_{i=0}^{\infty} (ps/100)^i.
\]

\( w \) is the sum of primitive work performed by the \texttt{WORK} instructions of the template (for the before template \( w = w_1 + w_2 \)). For convergence, the recursive call is not unconditional (i.e. \( ps < 100 \)), whereupon:

\[
cT = \frac{100w}{100 - ps}.
\]  

### 5.2.2 Extending the Analysis to Tail Calls

The Paragon \texttt{TAIL} instruction is designed to model the behaviour of recursive tail calls (see Section 3.2). Although the run-time behaviour of \texttt{TAIL} differs from that of \texttt{SPAWN} the cost of making a tail call is identical to that of the first order recursive spawn described above. To calculate the cost of the tail call it is therefore sufficient to replace the \texttt{TAIL} instruction by a recursive \texttt{SPAWN} which has the same cost, and then use Equation 5.1.

From Section 3.2.5 we know that when an instance of template \( T_i \) spawns a child using template \( T_j \), the number of tail calls performed by the child is given by the product of two random variables:

\[1^\text{st The mean of a product of two random variables is equal to the product of means if and only if the two random variables are independent [Jai91].}^\text{st} \]
1. The range item in the tail call instruction of $T_j$; this specifies an unmodified number of tail calls. Define $\bar{r}_j$ to be the mean value of this range calculated according to Table 3.1.

2. The mean value of the mod item in the spawn alternative which caused $T_i$ to invoke $T_j$. Define $\bar{m}_{ij}$ to be the mean value of this range.

Therefore:

$$tc_j = \bar{r}_j \times \bar{m}_{ij}.$$  \hspace{1cm} (5.2)

The tail instruction causes the child to recurse $tc_j$ times, terminating after a total of $tc_j + 1$ invocations. The same average cost is obtained from a recursive spawn instruction whose probability of success is given by:

$$pts_j = \frac{tc_j}{tc_j + 1},$$

therefore the tail call instruction:

**TAIL** range

\[\rightarrow\ <seq_1, \ldots, seq_{1,m_1}> \ (pt_1\%)\]

\[\vdots\]

\[\rightarrow\ <seq_n, \ldots, seq_{n,m_n}> \ (pt_n\%)\]

is equivalent to:

**SPAWN**

\[\rightarrow\ T_j <seq_1, \ldots, seq_{1,m_1}> \ (pts_1\%)\]

\[\vdots\]

\[\rightarrow\ T_j <seq_n, \ldots, seq_{n,m_n}> \ (pts_n\%)\]

\[\rightarrow\ T_j\]

where (for $1 \leq k \leq n$),

$$pts_k = pt_k \frac{tc_j}{tc_j + 1}.$$  \hspace{1cm} (5.3)

The final branch of the spawn ensures the correct default behaviour for tail calls: if none of the preceding alternatives are chosen at run-time then a tail call is made with an empty parameter list.

(Notice that for a tail-call of template $T_j$ to be re-written as a spawn instruction, all modifiers, $m_{ij}$, must be the same. This restriction will be lifted in Section 5.5.3.)
5.2.3 Limited Simultaneous Recursion

The first order recursive case is simple but describes only a small subset of Paragon programs. In this section the analysis is extended to describe a restricted form of simultaneous (or mutual) recursion. Again only work instructions and at most one spawn instruction are permitted but the spawn is of the more general form:

\[
\text{spawn} \rightarrow T_{i_1} \ (ps_{j_1}) \ | \ \cdots \ | \ T_{i_n} \ (ps_{j_n}).
\]

A maximal SCC (of G), S = (\(T_R, E_T\)), containing \(\tau\) vertices (\(|T_R| = \tau\)) can be represented by the transition matrix:

\[
P = \begin{bmatrix}
P_{11} & P_{12} & \cdots & P_{1\tau} \\
P_{21} & P_{22} & \cdots & P_{2\tau} \\
\vdots & \vdots & \ddots & \vdots \\
P_{r1} & P_{r2} & \cdots & P_{r\tau}
\end{bmatrix}.
\]

The spawn instruction from template \(T_i\) defines a row, \(P_i = [p_{i1} \ p_{i2} \ \cdots \ p_{ir}]\), of the matrix, where \(p_{ij}\) is the probability (0 \(< p_{ij} \leq 1\)) for \(T_i\) to spawn a child task instance using template \(T_j\):

\[
p_{ij} = \sum_k p_{s(i,j)k}.
\]

Inequality 3.3 (with \(pp_j = 100p_{ij}\)) ensures that \(\sum_{j=1}^{\tau} p_{ij} \leq 1\) for all \(i\).

The cost, \(cT_i\), of executing a task instance spawned using template \(T_i \in T_R\) is given by:

\[
cT_i = w_i + \sum_{j=1}^{\tau} p_{ij} cT_j, \quad \text{for } 1 \leq i \leq \tau. \tag{5.4}
\]

Expanding (5.4) by one recursive step we get:

\[
cT_i = w_i + p_{i1} \left( w_1 + \sum_{k=1}^{\tau} p_{1k} cT_k \right) + \cdots + p_{ir} \left( w_r + \sum_{k=1}^{\tau} p_{rk} cT_k \right)
\]

\[
= w_i + \sum_{j=1}^{\tau} p_{ij} w_j + \sum_{k=1}^{\tau} \sum_{j=1}^{\tau} p_{ij} p_{jk} cT_k.
\]

Expanding (5.4) to the limit gives the following (where, for notational convenience, \(t_0\) is defined to be equal to \(i\)):

\[
cT_i = w_i + \sum_{\pi=1}^{\infty} \sum_{t_1=1}^{\pi} \cdots \sum_{t_n=1}^{\pi} \left( \prod_{\rho=1}^{\pi} p_{t_{\rho-1},t_{\rho}} \right) w_{t_n}.
\]

131
The mutually recursive relationship expressed in (5.4) is expressed more naturally in matrix form. Given that the primitive work performed during a single call to a task is $w_i$ and the total cost of calling a task is $cT_j$, we define the vectors:

\[
W = \begin{bmatrix}
w_1 \\
\vdots \\
w_r
\end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix}
cT_1 \\
\vdots \\
cT_r
\end{bmatrix}.
\]

Equation (5.4) can then be expressed as $C = PC + W$. This recursion relation has an iterative solution $C^{(k+1)} = PC^{(k)} + W$ where $C^{(0)}$ is an initial and $C^{(k)}$ is the $k$th approximation for $C$. By expansion, we have:

\[
C^{(k+1)} = P^{k+1}C^{(0)} + \sum_{j=0}^{k} P^j W.
\]

(5.5)

To solve Equation (5.5) we can use the eigenvalues, $\lambda_1 \ldots \lambda_r$, and the eigenvectors, $v_1 \ldots v_r$, of the transition matrix $P$. The analysis requires that the eigenvalues are distinct so that the corresponding eigenvectors are linearly independent. It is therefore assumed that these can be calculated, otherwise the analysis cannot proceed. This is acceptable because the probability that the eigenvectors of a randomly chosen matrix are calculable and distinct can be approximated to 1 [Mag85, §5.2.18] and since $P$ is obtained from an experimental workload then in the unlikely event that the analysis is thwarted it is possible to change the input model to allow the analysis to proceed. Moreover, the eigenvalues of a nonsymmetric matrix can be very sensitive to small changes in the matrix elements [PFTV88, p.382]. Therefore, it is likely that only a small change to a rogue workload is necessary to yield values that are suitable for analysis.

We know from the theory of eigenvectors (see e.g. [Mag85, ch.5]) that if $P$ is expressed:

\[
P = V \Lambda V^{-1}
\]

where $V = [v_1 \ldots v_r]$ and $\Lambda = \begin{bmatrix}
\lambda_1 \\
\vdots \\
\lambda_r
\end{bmatrix}$

then $Pv_i = \lambda_i v_i$. Thus, if $W$ is expressed as a linear combination of the eigenvectors:

\[
W = \sum_{i=1}^{r} a_i v_i, \quad \text{then} \quad P^j W = \sum_{i=1}^{r} a_i P^j v_i = \sum_{i=1}^{r} a_i \lambda_i^j v_i.
\]

Substituting for $P^j W$ in (5.5) and choosing $C^{(0)} = \emptyset$ (valid when $k$ is large) we get:

\[
C^{(k+1)} = \sum_{i=1}^{r} \sum_{j=0}^{k} a_i \lambda_i^j v_i = W + \sum_{i=1}^{r} \sum_{j=1}^{k} a_i \lambda_i^j v_i = W + \sum_{i=1}^{r} a_i v_i \sum_{j=1}^{k} \lambda_i^j, \quad \text{for } k \gg 1
\]

132
The term $\sum_{j=1}^{k} \lambda_j^j$ is simply the sum of a geometric progression which converges to $\lambda_i/(1 - \lambda_i)$ as $k \to \infty$ on condition that $|\lambda_i| < 1$, for $1 \leq i \leq \tau$; therefore we get:

$$\lim_{k \to \infty} C = W + \sum_{i=1}^{\tau} \frac{a_i v_i \lambda_i}{1 - \lambda_i}. \quad (5.6)$$

The total cost of executing a task $T_i$ is given by the $i$th row of $C$.

5.2.4 Extending the Analysis to Task Templates with many SPAWN Instructions

In the previous section, an analysis of Paragon programs was developed for mutually recursive task templates that were limited to at most one SPAWN instruction. This section deals again with templates $T \in T_R$ but they may now contain any number of SPAWN instructions. The evaluation of declared sequences is still banned.

The previous limitation of one SPAWN instruction per template led to the property that $p_{ij} \leq 1$ for $1 \leq i, j \leq \tau$ but since this property was not required to derive Equation (5.6) the condition can be removed. Collectively, the SPAWN instructions from template $T_i$ define the $i$th row of the matrix:

$$S = \begin{bmatrix}
    s_{11} & s_{12} & \cdots & s_{1\tau} \\
    s_{21} & s_{22} & \cdots & s_{2\tau} \\
    \vdots & \vdots & \ddots & \vdots \\
    s_{\tau 1} & s_{\tau 2} & \cdots & s_{\tau \tau}
\end{bmatrix}$$

where $s_{ij}$ is the average number of times that a single instance of $T_i$ uses template $T_j$ to create a child task and is given by:

$$s_{ij} = \sum_k p_{(i,j)k}.$$ Equation (5.6) developed in Section 5.2.3 holds for generally recursive Paragon programs if $\lambda_i$ and $v_i$ (for $1 \leq i \leq \tau$) are the eigenvalues and eigenvectors, respectively, of $S$.

5.3 Reasoning about Sharing Semantics

Until now the time-cost analysis has not taken the evaluation of declared sequences into account. Declared sequences complicate time-cost analysis in the following ways:

1. Because of sharing the history of a declared sequence determines whether or not an EVAL instruction causes the evaluation of the sequence. Declared sequences can therefore alter the expected cost of tasks dynamically.
2. The order in which tasks are executed (hence the order in which their \textit{eval} instructions are encountered at run-time) is relevant to where declared sequences are evaluated.

3. In the previous analysis it did not matter whether two possibilities for spawning children were specified by alternative branches of a single \texttt{spawn} instruction or by two distinct \texttt{spawn} instructions. This is because the average number of calls to each child is identical in both cases. However, in the presence of sharing it is necessary to know whether or not it is possible for two children to execute in parallel with each other. Alternative branches of a single \texttt{spawn} cannot do this because they are mutually exclusive but children spawned by separate instructions have no restriction. Parallel threads introduce uncertainty in deciding which task will evaluate a certain declared sequence thus a distinction is henceforth made between the two cases.

4. Sharing semantics can introduce statistical dependencies into the expressions which describe the history of a declared sequence. For example, consider the following task template:

\begin{verbatim}
dependency {
    Declare seq1 { ··· }
    ·
    Eval seq1 (p1\%)
    ·
    Eval seq1 (p2\%)
    ·
}
\end{verbatim}

The outcome of the first \textit{eval} instruction partly determines whether the second \textit{eval} instruction causes the evaluation of seq1. This example is relatively straightforward but more complex dependencies are likely if sequences are passed as arguments to child task instances and returned as results to parents. For this reason, expressions denoting probabilistic events are left in algebraic form throughout the analysis.

5. A single task template can be used by many \texttt{spawn} instructions to construct different task instances. Each \texttt{spawn} may contain a different set of actual parameters,
therefore the instances differ because their formals are bound to instances of declared sequences constructed from disparate templates. Task instances must be characterised, therefore, in terms of the task template and the declared sequence templates used by their formals.

As for the simple time-cost analysis described in Section 5.2, each edge, \( e_{ij} \in E \), of the program graph, \( G \), is associated with a probability of spawning, \( p_{ij} \). Moreover, because we are now dealing with parameterised task templates, each edge, \( e_{ij} \), is also associated with a list, \( R_{ij} \), of declared sequence references².

The strategy for analysing the time-cost of Paragon programs is divided into five stages as follows:

Stage 1: Perform a sharing analysis over the task templates of the program to decide which formal parameters and which locally-declared sequences are evaluated by the \texttt{EVAL} instructions of each task. This analysis is described in Section 5.4.

Stage 2: Transform the program graph, \( G \), into a new graph, \( G' \) (made up of nodes \( V' \) and edges \( E' \), such that \( G' \) exhibits the following characteristics:

- There is a one to many mapping from \( T_i \in V \) to \( T'_{ij} \in V' \) and from \( e_{ij} \in E \) to \( e'_{ij} \in E' \). Expressed simply, each task of the original program is represented by one or more nodes in \( G' \).
- All and only those call-trees that may be generated by executing \( G \) are produced by executing \( G' \).
- For all nodes, \( T'_{ij} \in V' \), the set of declared sequences which may be bound to the formal parameter, \( \xi_{i,x} \), must contain exactly one element (which may be either a declared sequence, \( Q_{k,q} \), or the special value, \texttt{unbound}).
- All edges, \( e'_{ij} \), entering a node \( T'_{ij} \) must correspond to the same branch of the same spawn instruction, \( T_i, \sigma_{\pi} T_j \), in the original program.

Stage 3: Replace the \texttt{EVAL} instructions of every node in \( G' \) with in-line macro expansions of the declared sequences which they evaluate (see Section 5.5.2). Each instruction in the expansion is suitably adjusted to reflect the probability of the evaluation.

The program graph has now been transformed to one which has an equivalent

²See Section 3.2.4 for a definition of sequence references.
time-cost to the original but which no longer contains shared expressions. The new graph will not be used at run-time, it is merely required to perform time-cost calculations.

Stage 4: Use the analysis described in Sections 5.2 and 5.1 to calculate the time-costs of the task templates in the new program graph.

Stage 5: Map the resulting time-costs back onto the edges of the original program graph.

Before continuing with the analysis of sharing and declared sequence evaluation it is prudent to pause in order to examine the nature of parallel evaluation in the presence of shared workloads. Firstly, key terms used in the subsequent analysis are defined. The Paragon instruction set is then categorised according to which instructions cause inter-task synchronisation and which do not. Thirdly, the structure of task templates is expressed in terms of well defined regions. This simplifies the analysis in the presence of sharing. Finally, some notation is introduced for reasoning about intra- and inter-task relationships. The notation will be used extensively in Section 5.4.

5.3.1 Probabilistic Events versus Mechanistic Actions

The term event is used in the mathematics of simulation and in probability theory to describe two different but related concepts. Consider an experiment where a pair of dice are thrown a number of times. In simulation, each dice throw might constitute a separate mechanistic event. Probabilistically, however, we may define an event, \( A \), as 'the sum of two dice is less than seven' and calculate that the probability of the event \( A \) is \( \frac{15}{36} \).

In the context of analysing Paragon programs, this dichotomy is potentially confusing. We shall therefore reserve the term event for the probabilistic sense and use the term action to denote mechanical activity. For example, the simulated spawning of a task is an action while the phrase 'a child is spawned by the \( \sigma \)th spawn instruction of task \( T_i \)' describes an event with a probability of, say, \( p_\sigma \).

In the following analysis, two classes of events will be encountered:

atomic events are those whose probability can be determined without reference to other events. For example, consider the instruction:

\[ \text{SPAWN} \ label \to \ T_j \ (50\%). \]
The event that 'the \textit{spawn} instruction creates a child task using template $T_j$' is atomic and has a probability of 50 per cent. The probability of an atomic event is independent of the probabilities of all other atomic events.

\textbf{compound events} are combinations of atomic and other compound events. Two compound events may both refer to a common sub-event, hence statistical dependencies may exist between them. To cope with dependencies, compound events are described algebraically using set notion. To obtain an estimate of the time-cost of a Paragon task, an arithmetic expression must be constructed from the algebraic representation of the time-cost, taking note of any statistical dependencies that are present ([BE87] provides a good introduction to this topic).

\subsection*{5.3.2 \textbf{Predicting Interaction Between Task Instances}}

Paragon instructions fall into two categories: those whose actions affect just the current task instance (we shall call these \textit{passive} instructions) and those which interact with other task instances and consequently necessitate message passing (we shall call these \textit{active} instructions). The following table illustrates which instructions fall into each category:

\begin{table}[h]
\centering
\begin{tabular}{ll}
\hline
Passive & Active \\
\hline
Declare & Block \\
Tail & Eval \\
Work & Return \\
Spawn & \\
\hline
\end{tabular}
\end{table}

When workloads are shared by multiple tasks, the time-cost analysis is mostly concerned with the actions of the active instructions. These will be examined in some detail in the following sections.

\subsection*{5.3.3 \textbf{Evaluation Points and Evaluation Intervals}}

When an instance of a task, $T_i$, or a declared sequence, $Q_{i,q}$, spawns a child task using template $T_j$, the child may or may not begin to execute immediately. The precise temporal relationships between tasks executing in parallel are dependent on the loading of the system, the number of processing elements assigned to the calculation, the reliability of the communication infrastructure, and other attributes that are outside the scope of Paragon specifications. The Paragon \texttt{block} instruction does, however, limit the lifespan of each task instance with respect to specific points in its parent.
In the absence of more detailed information we are forced to make the following assumptions and simplifications when analysing a task template, $T_i$:

1. The entire execution period of a child task instance (which was constructed using template $T_j$), including the period required to execute the descendants of the child, is considered to occur at a single point in time. This point is hereafter referred to as the evaluation point of $T_j$. An evaluation point is equivalent to an abstract mechanical action that occurs instantaneously.

   This simplification allows the evaluation properties of a child task, $T_j$, to be applied atomically.

2. The probability for two instantaneous actions to occur simultaneously is assumed to be zero.

3. In the absence of blocking, the evaluation point of a child, $T_j$, may occur, with equal likelihood, at any point during its parent's lifetime.

   This assumption is needed because, other than the simple blocking mechanism, Paragon contains no instructions to constrain when tasks are dynamically scheduled and at which precise time they will complete.

4. The likelihood that the evaluation point of an instance of $T_j$ occurs during a time interval measured with respect to its parent, is proportional to the probability that the child has not yet completed in that interval.

   This assumption is a heuristic. It allows the probability that an instance of $T_j$ reaches its evaluation point during a given execution interval of $X_i$ to be related to the BLOCK instructions in $X_i$ which lexically precede the interval.

5. In the absence of blocking, the parallel actions $a_1 \ldots a_n$ (respectively performed by a collection of tasks and declared sequences, $X_1 \ldots X_n$, executing in parallel) may take place in any order with equal likelihood.

   This assumption defines the behaviour of parallel child tasks while $X_i$ is executing within an interval that is bounded by, but does not contain, BLOCK instructions.

6. It is assumed that speculative parallelism [BZ90, SY90, PJ89] is not being modelled and that all child tasks are meant to terminate prior to their respective parents. To enforce this assumption, the closing brace of a task template is treated as an
unconditional BLOCK instruction which suspends the parent until every child task has completed.

Consider the action of the following BLOCK instruction on the task instance identified by label:

\[ \text{BLOCK} \quad \text{label} \quad (pb\%) \] .

If the probability that the instance had not already completed immediately prior to the BLOCK is given by \( p \) then there is a probability \( (100 - pb)p/100 \) that it has still not completed immediately after the BLOCK.

Figure 5.3 shows a progression of BLOCK instructions acting on a labelled task instance. The expression attached to each stage indicates the probability that the instance is still evaluating after immediately after the preceding BLOCK instruction has been executed. The probabilities will be used in Section 5.4.5 as weighting factors to estimate when the evaluation point of the child instance is reached with respect to the instructions of the parent task. In the figure, the Greek letter \( \beta \) is used to range over the BLOCK instructions and we use a caret (\(^\)\) to indicate the largest value in a range (e.g. \( 1 \leq \beta \leq \beta' \)). In accordance with Simplification 6, above, there are \( \beta - 1 \) explicit BLOCK instructions and the closing brace of the template represents \( \text{BLOCK}_\beta \).

If a parent and child task make references to the same declared sequence (the sequence is shared) then, by estimating where the evaluation point of a child task will occur with respect to the parent, probabilities can be calculated to determine which of the tasks will force the evaluation of the sequence. The probabilities are influenced by the EVAL instructions within the two tasks. It is convenient, therefore, to express the evaluation point of the child with respect to the EVAL instructions of the parent. To facilitate this, the parent's template is divided into intervals delimited by EVALs. These shall be called evaluation intervals. The Greek letter \( \alpha \) will be used to range over the EVAL instructions and their corresponding intervals.

Figure 5.4 shows a task divided by \( \alpha - 1 \) EVAL instructions into \( \alpha \) evaluation intervals. This is analogous to 'fences and gate-posts' in that there is always one fewer EVAL instruction than there are intervals. If the very last instruction of a template is an EVAL then the \( \alpha \)-interval is empty.
The probability that the child has not yet completed at each stage of the parent's execution:

```plaintext
parent_templt {
  ...
  Spawn child_instance -> child_templt
  ...
  Block child_instance (pb_1\%)
  ...
      \[ \frac{100 - pb_1}{100} \]
  Block child_instance (pb_2\%)
  ...
      \[ \frac{(100 - pb_1)(100 - pb_2)}{100^2} \]
  ...
  Block child_instance (pb_{\beta-1}\%)
  ...
      \[ \frac{\prod_{\beta=1}^{\beta-1}(100 - pb_{\beta})}{100^{\beta-1}} \]
  ...
  0
}
```

Figure 5.3: A series of BLOCK instructions acting on a child task instance.

5.3.4 Some Notation to Describe Lexical, Temporal, and Dependence Relations

To decide which BLOCK instructions occur within an evaluation interval (and thereby derive the probability of a child's evaluation point occurring prior to the associated EVAL instruction) it is necessary to be able to specify the lexical order of the instructions:

**Definition 5.1** The syntactic relation, \(<\), is defined over any two Paragon instructions belonging to the same task template. \(I_1 < I_2\) states that instruction \(I_1\) lexically precedes instruction \(I_2\). The inverse relation is denoted by \(>\).

Where no ambiguity arises, expressions such as BLOCK\(\beta < EVAL_\alpha\) will be written \(\beta < \alpha\).

It will also be necessary to describe temporal relationships between the evaluation points of child tasks and the actions of the parent. These are functions of the run-time behaviour and are expressed thus:

**Definition 5.2** The temporal relation, \(<\), is defined over all mechanical actions, such that \(a_1 < a_2\) denotes the event 'action \(a_1\) occurs prior to action \(a_2\)'. The inverse relation is denoted by \(>\).
template {
  DECLARE instructions
  SPAWN, WORK, and BLOCK
  instructions
  Eval1...
  SPAWN, WORK, and BLOCK
  instructions
  Eval2...
  ...
  Eval_{k-2}...
  SPAWN, WORK, and BLOCK
  instructions
  Eval_{k-1}...
  SPAWN, WORK, BLOCK, TAIL, and RETURN instructions
  Eval_k...
}

1-interval
2-interval
\((\alpha - 1)\)-interval
\(\alpha\)-interval

Figure 5.4: Dividing a task template into its evaluation intervals.

From Assumption 4 on page 138 we know that, in the absence of blocking, every ordering of actions is equally likely. Furthermore, from Assumption 2 we know that two instantaneous actions will not take place simultaneously. Therefore, for two instantaneous actions \(a_1\) and \(a_2\), which may occur in parallel,

\[
Pr(a_1 < a_2) = Pr(a_2 < a_1) = 0.5. \tag{5.7}
\]

Logical combinations of events are provided by the following operators:

- The \(\cap\) operator expresses the conjunction of events.
- The \(\cup\) operator expresses the disjunction of events.
- The unary operator \(\sim\) expresses the failure of an event (i.e. \(Pr(\sim e) = 1 - Pr(e))\).

By a relaxation of syntax, an action may be specified whenever an event is expected. This represents the probabilistic event that the action occurs.
5.3.5 Reasoning about the Interface between Parent and Child Tasks

Child task instances are created by the **spawn** instruction which is responsible for determining which declared sequences are passed as arguments to the children. Spawn instructions have many attributes and to describe these in a consistent manner the notation

$$T_i \xrightarrow{id \_{\text{attr}}} T_j$$

is introduced. This symbol denotes a **spawn** instruction where:

- $T_i$ is the task template in which the **spawn** instruction occurs.
- $T_j$ is the task template used by the **spawn** instruction to create a child task instance.
- $id$ is an identifier that indicates precisely which **spawn** instruction (or sub-part thereof) is of interest. Typically, $id$ is set to $\sigma$ to identify the $\sigma$th **spawn** instruction of $T_i$, and set to $\sigma \eta$ to refer specifically to the $\eta$th branch of the instruction.
- $attr$ is an optional attribute of the call and qualifies the meaning of the whole symbol. If $attr$ is not specified then $T_i \xrightarrow{id \_\text{attr}} T_j$ indicates the **spawn** instruction per se. An initial set of qualified symbols are defined in Table 5.1; further qualifiers will be defined as they are needed.

This notation is easily extended to describe **spawn** instructions which belong to declared sequences. Declared sequence definitions appear only within task templates and the symbol $Q_{i,q}$ denotes the sequence defined by the $q$th **declare** instruction of template $T_i$. A **spawn** instruction belonging to $Q_{i,q}$ is described by

$$Q_{i,q} \xrightarrow{id \_{\text{attr}}} T_j$$

which is a direct analogue of the $T_i \xrightarrow{id \_\text{attr}} T_j$ symbol. Definitions which are valid for both $Q_{i,q} \rightarrow T_j$ and $T_i \rightarrow T_j$ symbols are specified by employing a generic symbol: $X \rightarrow T_j$.

Finally, a simple extension to the Paragon $z$ syntax for specifying formal parameters provides a method for referencing the formal parameters of a task template: the symbol $s_{i,x}$ denotes the $x$th formal parameter of template $T_i$.

5.3.6 Some examples

The following examples demonstrate the way in which qualified $X \rightarrow T_j$ expressions will be used to make statements about Paragon programs.
<table>
<thead>
<tr>
<th>$T_i \xrightarrow{\sigma_{\text{Ret}}} T_j$</th>
<th>This denotes the declared sequence that is returned as the result from a child task. The child was an instance of template $T_j$ and was spawned by the $\sigma$th SPAWN instruction of template $T_i$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_i \xrightarrow{\sigma_{\text{E}}} T_j$</td>
<td>This denotes the evaluation point of the child instance.</td>
</tr>
<tr>
<td>$T_i \xrightarrow{\sigma_{\Delta \eta}} T_j$</td>
<td>The $\delta$th actual parameter of the specified alternative, $\eta$, of the $\sigma$th SPAWN instruction of template $T_i$. (The use of the $\Diamond$ operator is inspired by the syntax for passing parameters in SPAWN instructions where the parameters are enclosed in $&lt;&gt;$ brackets.)</td>
</tr>
<tr>
<td>$T_i \xrightarrow{\sigma_{\text{Label}}} T_j$</td>
<td>Refers to the identification label attached to the $\sigma$ SPAWN instruction of template $T_i$.</td>
</tr>
</tbody>
</table>

Table 5.1: An initial set of qualified $T_i \rightarrow T_j$ symbols.

1. The expression:

   $$T_i \xrightarrow{\sigma_{\text{E}}} T_j \equiv \mathfrak{S}_{i,x}$$

specifies that when the $\sigma$th SPAWN instruction of $T_i$ spawned a child task, (a) it used template $T_j$ and (b) the $y$th actual parameter passed to the child was bound to the $z$th formal parameter of $T_i$.

2. Probabilistic reasoning is central to Paragon analysis; the following expression:

   $$\Pr \left( \left( T_i \xrightarrow{\sigma_{\text{E}}} T_j \right) < \text{EVAL}_x \right)$$

indicates the probability that the child task, spawned by the $\sigma$th SPAWN instruction of task $T_i$, reaches its evaluation point prior to the time that $T_i$ executes its $\alpha$th EVAL instruction.
5.4 An Analysis of Declared Sequence Evaluation

The aim of this section is to develop an analysis which can determine the likelihood that a task causes the evaluation of one of its locally-declared sequences or formal parameters. The analysis will then be used in Section 5.5 to replace \texttt{EVAL} instructions by the bodies of the declared sequences which they are known to evaluate, suitably adjusted to reflect the probability of the evaluation.

5.4.1 Simplifications

To simplify the analysis we shall continue to use a subset of the full Paragon language. It is, however, a much larger subset than previously allowed and one from which sensible workloads can be constructed (e.g. see Chapter 6 where all of the experimental workloads belong to this subset). The full language is accepted with the exception of the following restrictions:

1. \texttt{RETURN} instructions are not allowed.

2. Declared sequences may not access the formal parameters of their defining tasks.

The first simplification means that the map of shared references is in the form of a tree rather than a graph because references can only be passed from older to newer generations of task instances. The second simplification effectively bans all references to shared workloads from declared sequence templates (remember that a declared sequence, \(Q_{i,q}\), cannot defined its own local sequences and that it may not reference the declared sequences of its defining task). Therefore declared sequences have no use for \texttt{EVAL} instructions and, although they may spawn child task instances, their \texttt{SPAWN} instructions cannot pass actual parameters. This restriction allows declared sequences and formal parameters to be analysed in mutual isolation.

5.4.2 Temporal Relationships and Probabilistic Events

The analysis concentrates on the temporal relationships between parent and child tasks and the following properties, each of which is described by a collection of probabilistic events:

- **Necessity** is the property that the value of an object is required. This corresponds to a declared sequence being referenced by an \texttt{EVAL} instruction. The events which
describe this property have the generic name ‘Need’.

**Intactness** is the property that a formal parameter or a locally-declared sequence has not already been evaluated. The probabilistic events which describe intactness properties are given the generic name ‘Intact’ and rely heavily on the synchronization properties described in Section 5.4.4.

**Locality of evaluation** combines the previous properties to describe which task instances cause the evaluation of which declared sequences, thus fulfilling the aim of the section. The events which describe locality of evaluation have the generic name ‘Eval’.

Firstly, expressions for necessity of formal parameters and locally-declared sequences are defined. Temporal relationships between parent and child tasks are then explored prior to defining expressions for intactness over formals and declared sequences. Finally, these expressions are used to define the probabilistic events that a task causes the evaluation of its locally-declared sequences and formal parameters. The evaluation properties will be used in Section 5.5 to transform eval instructions into the bodies of the declared sequences which they evaluate.

### 5.4.3 Determining the Necessity of Formal Parameters and Locally Declared Sequences

In this section, expressions are given for the compound events that the values of formal parameters and locally-declared sequences are required by the eval instructions or children of a task. If a value is necessary and the declared sequence (either defined locally or bound to a formal parameter) has not yet been evaluated then the evaluation will be triggered. If the sequence has already been evaluated then no action will be taken. Necessity does not therefore determine whether an evaluation takes place, it merely places a limit on the unevaluated state of a sequence. The probabilistic events for necessity properties are defined as follows.

- The necessity of a declared sequence within an eval instruction is given directly by the instruction’s definition. In Chapter 3, the eval instruction is defined with a number of alternative branches, each of which may require the value of a number of locally-declared sequences and formal parameters. We shall formally identify each
sequence reference by $\mathcal{V}_{\alpha \delta}$ to indicate that we are dealing with the $\eta$th branch of the $\text{EVAL}_\alpha$ instruction and, specifically, the $\delta$th entry within the branch. The necessity of a locally-declared sequence, $Q_{i,q}$, is given by the event that the branch of $\text{EVAL}_\alpha$ which is chosen at run-time contains a reference to $Q_{i,q}$:

$$\text{Need}_{Q_{i,q}}(T_i, \text{EVAL}_\alpha) = \bigcup_{\eta} \left( \text{EVAL}_{\alpha \eta} \cap \bigcup_{\delta} (\mathcal{V}_{\alpha \delta} \equiv Q_{i,q}) \right)$$

and similarly, the necessity of a formal parameter, $s_{i,z}$, is given by:

$$\text{Need}_{s_{i,z}}(T_i, \text{EVAL}_\alpha) = \bigcup_{\eta} \left( \text{EVAL}_{\alpha \eta} \cap \bigcup_{\delta} (\mathcal{V}_{\alpha \delta} \equiv s_{i,z}) \right) .$$

1. The necessity of a declared sequence with respect to a child task is similar to the previous case for $\text{EVAL}$. The $\text{SPAWN}$ instruction which creates the child task contains a number of alternative branches, each of which can pass declared sequence references to the child as actual parameters. The event that the child needs the value of a declared sequence is therefore defined as the conjunction of the following events:

1. The $\eta$th branch of the $\text{SPAWN}$ is chosen.
2. The declared sequence reference (either $Q_{i,q}$ or $s_{i,z}$) is bound to the $\delta$th actual parameter of the spawn.
3. The value of the $\delta$th parameter is necessary for the evaluation of the child task created by the spawn.

The expression for the necessity of a locally-declared sequence with respect to a child task instance is therefore given by:

$$\text{Need}_{Q_{i,q}}(T_i \rightarrow_{\sigma} T_j) = \bigcup_{\eta} \left( (T_i \rightarrow_{\sigma} T_j) \cap \bigcup_{\delta} \left( (T_i \rightarrow_{\sigma} T_j \equiv Q_{i,q}) \cap \text{Need}_{s_{i,z}}(T_i, \text{SPAWN}) \right) \right)$$

where $\text{Need}_{s_{i,z}}(T_i)$ is defined by Equation (5.12) below. The equivalent expression for the necessity of a formal parameter, $s_{i,z}$, with respect to a child task is:

$$\text{Need}_{s_{i,z}}(T_i \rightarrow_{\sigma} T_j) = \bigcup_{\eta} \left( (T_i \rightarrow_{\sigma} T_j) \cap \bigcup_{\delta} \left( (T_i \rightarrow_{\sigma} T_j \equiv s_{i,z}) \cap \text{Need}_{s_{i,z}}(T_j) \right) \right) .$$
Finally, the necessity of a formal parameter, \(s_{i,a}\), with respect to the task in which it is bound, depends on the necessity of the parameter with respect to (i) the EVAL instructions of the task and (ii) the children of the task:

\[
\text{Need}_{s_{i,a}}(T_i) = \bigcup_{\alpha} \text{Need}_{s_{i,a}}(T_i, \text{EVAL}_{\alpha}) \cup \bigcup_{\sigma} \text{Need}_{s_{i,a}}(T_i \xrightarrow{\sigma} T_j).
\]

(5.12)

### 5.4.4 Temporal Relationships between Parent and Child Tasks

Synchronisation between parent and child tasks is based primarily on BLOCK instructions which limit the lifespan of child tasks with respect to their parents. In this section blocking properties are defined for the evaluation intervals of task templates (as illustrated in Figure 5.4 on page 141); from these, expressions can be derived for the probability that a child task has not completed when its parent reaches a specified evaluation interval. Also, expressions are given for the temporal relationships between the evaluation point of a child task and the BLOCK instructions of its parent.

These relationships govern the intactness properties of formal parameters and locally-declared sequences and will be used in the definitions of Section 5.4.5.

#### The Blocking Properties of an Evaluation Interval

The blocking properties of an evaluation interval of a task, up to and including the BLOCK\(\beta\) instruction (which must occur in the interval) can be summarised by a set of pairs defined thus:

\[
\text{BLOCK}_{\beta} = \{ (\sigma, \text{label}_{\beta\eta}) \mid \text{label}_{\beta\eta}\delta \equiv (T_i \xrightarrow{\text{label}_{\beta\eta}} T_j) \wedge \alpha - 1 < \beta' \leq \beta < \alpha \}
\]

where \(pb_{\beta\eta}\) is the probability for the \(\eta\)th branch of BLOCK\(\beta\) to be executed and \(\text{label}_{\beta\eta}\delta\) is the \(\delta\)th entry in that branch. The set which summarises the blocking properties of the whole of the \(\alpha\) evaluation interval is called BLOCK\(\alpha\) and is equal to BLOCK\(\beta\) for the largest \(\beta < \alpha\):

\[
\text{BLOCK}_{\alpha} = \text{BLOCK}_{\beta} \iff \{ \beta' | \beta' < \alpha \wedge \beta' \geq \beta \} = \emptyset.
\]

For the child task instance spawned by \(T_i \xrightarrow{\sigma} T_j\), the event that the child is subject to the action of a BLOCK instruction during the \(\alpha\) evaluation interval of \(T_i\), up to and including BLOCK\(\beta\), is expressed by:

\(T_i \xrightarrow{\text{BLOCK}_{\beta}} T_j\).
Applying the blocking properties shown in Figure 5.3 on page 140, the probability for the event is given by:

$$\Pr\left(T_i \frac{\sigma}{B_\beta^\alpha} T_j\right) = 1 - \left(\prod_{(\sigma,p_b^\alpha) \in \text{BLOCK}_\alpha} \left(1 - \frac{p_b^\alpha}{100}\right)\right).$$

The symbol $T_i \frac{\sigma}{B_\beta^\alpha} T_j$ is used to express the blocking event for the whole of the $\alpha$ evaluation interval of $T_i$ and there is an analogous expression for its probability given in terms of $\text{BLOCK}_\alpha$. For completeness, the probabilities:

$$\Pr\left(T_i \frac{\sigma}{B(\beta=0)} T_j\right) = \Pr\left(T_i \frac{\sigma}{B(\alpha=0)} T_j\right) = 0$$

are also defined.

**The Temporal Relationship between Evaluation Points and BLOCK Instructions**

Assuming a child has been spawned from $T_i$ using template $T_j$, the qualified symbol $T_i \frac{\sigma}{B_\beta^\alpha} T_j$ expresses the event that, immediately prior to the $\beta$th BLOCK instruction of $T_i$, the child task has not been blocked by any of the preceding BLOCK instructions of $T_i$. If $\beta < \sigma$ then the spawn cannot yet have occurred and the probability of the event is zero. When $\beta > \sigma$, the probability is given by:

$$\Pr\left(T_i \frac{\sigma}{B_\beta^\alpha} T_j\right) = \left(1 - \Pr\left(T_i \frac{\sigma}{B(\beta-1)} T_j\right)\right) \times \prod_{\alpha < \beta} \left(1 - \Pr\left(T_i \frac{\sigma}{B_\beta^\alpha} T_j\right)\right).$$

The qualified symbol $T_i \frac{\sigma}{B_\beta^\alpha} T_j$ expresses the event that the evaluation point of the child task occurs after its parent has executed its $\text{BLOCK}_{\beta-1}$ instruction (or the start of the task if $\beta = 1$) but before it has executed its $\text{BLOCK}_\beta$ instruction. Given that the child will reach its evaluating point at some time during its parent’s lifetime, we have:

$$\sum_{\beta > \sigma} \Pr\left(T_i \frac{\sigma}{B_\beta^\alpha} T_j\right) = 1,$$

and, by Assumption 4 on page 138, we know that the likelihood of a child reaching its evaluating point at a certain time is proportional to the likelihood that the child can be executing at that time:

$$\Pr\left(T_i \frac{\sigma}{B_\beta^\alpha} T_j\right) = k \times \Pr\left(T_i \frac{\sigma}{B_\beta^\alpha} T_j\right),$$

(5.14)
therefore, from (5.13) and (5.14),

\[ \text{Pr} \left( T_1 \rightarrow \mathcal{E}_\beta \rightarrow T_j \right) = \frac{\text{Pr} \left( T_1 \rightarrow \sigma \rightarrow T_j \right)}{\sum_{\beta' \succ \sigma} \text{Pr} \left( T_1 \rightarrow \sigma \rightarrow T_j \right)}. \tag{5.15} \]

5.4.5 Intactness Properties of Formal Parameters and Locally Declared Sequences

Intactness is the property that a formal parameter or a locally-declared sequence has \textit{not} been evaluated prior to a given point of reference. It is often useful to deal with intactness in two stages: firstly, to consider actions which occur sequentially, prior to the reference point and, secondly, to consider actions which occur in parallel with the reference point and to determine the possibility for any of these to trigger the evaluation before the reference point is reached. The analysis is simplified by concentrating on one evaluation interval at a time.

In the following analysis, expressions of the form:

\[ \left( T_1 \rightarrow \sigma \rightarrow T_j \right) \preceq \text{EVAL}_\alpha \]

will be used to denote the probabilistic event that the evaluation point of a specified child task is reached prior to a specified \text{EVAL} instruction. From Equation (5.7) we know that this event has probability 0.5 when \( \beta - 1 < \alpha < \beta \). Similarly, expressions of the form:

\[ \left( T_1 \rightarrow \sigma \rightarrow T_j \right) \preceq \left( T_1 \rightarrow \sigma' \rightarrow T_j \right) \]

are used to denote the probabilistic event that one evaluation point occurs prior to another. Again, from Equation (5.7), the event has probability 0.5 when \( \sigma' = \beta \).

Probabilistic events for intactness properties of locally-declared sequences and formal parameters are defined as follows.

- The event that a declared sequence, \( Q_{i,q} \), remains unevaluated when task \( T_i \) completes its \text{BLOCK}_\beta instruction depends on two conditions:

  1. none of \( T_i \)'s children have both reached their evaluation point \textit{and} consequently required the value of \( Q_{i,q} \); and
  2. \( T_i \) has not itself required the value of \( Q_{i,q} \).
The event is expressed as follows:

\[
\text{Intact}Q_{i,q}(T_i, \text{BLOCK}_{\beta}) = \\
\bigcap_{\sigma < \beta} \left( \text{Need}Q_{i,q} \left( T_i, T_j \right) \cap \bigcup_{\sigma < \beta' \leq \beta} \left( T_i, T_j \right) \right) \cap \\
\bigcap_{\alpha < \beta} \text{Need}Q_{i,q}(T_i, \text{EVAL}_{\alpha}) \cdot \cdot (5.16)
\]

Notice that, if \( \beta = 0 \), then \( \text{Pr} \left( \text{Intact}Q_{i,q}(T_i, \text{BLOCK}_{\beta=0}) \right) = 1 \). The corresponding event for the formal parameter, \( S_{i,z} \), is given by:

\[
\text{Intact}S_{i,z}(T_i, \text{BLOCK}_{\beta}) = \\
\bigcap_{\sigma < \beta} \left( \text{Need}S_{i,z} \left( T_i, T_j \right) \cap \bigcup_{\sigma < \beta' \leq \beta} \left( T_i, T_j \right) \right) \cap \\
\bigcap_{\alpha < \beta} \text{Need}S_{i,z}(T_i, \text{EVAL}_{\alpha}) \cdot \cdot (5.17)
\]

The event that a declared sequence, \( Q_{i,q} \), remains unevaluated at the point when the \( \text{EVAL}_{\alpha} \) instruction of \( T_i \) is executed is given by the conjunction of the events:

1. That \( Q_{i,q} \) remained intact after the preceding \( \text{BLOCK} \) instruction. (This is an analysis of sequential behaviour and is expressed by Equation (5.16).)
2. That no \( \text{EVAL}_{\alpha'} \) instruction between the preceding \( \text{BLOCK} \) instruction and \( \text{EVAL}_{\alpha} \) evaluates \( Q_{i,q} \). (This is also sequential behaviour.)
3. That \( Q_{i,q} \) is not evaluated by any of the child tasks which reach their evaluation points between the preceding \( \text{BLOCK} \) and \( \text{EVAL}_{\alpha} \). (This is an analysis of parallel behaviour.)

The event is therefore expressed by:

\[
\text{Intact}Q_{i,q}(T_i, \text{EVAL}_{\alpha}) = \\
\text{Intact}Q_{i,q}(T_i, \text{BLOCK}_{\beta-1}) \cap \\
\bigcap_{\beta-1 < \alpha' < \alpha} \text{Need}Q_{i,q}(T_i, \text{EVAL}_{\alpha'}) \cap \\
\bigcap_{\sigma < \alpha} \left( \text{Need}Q_{i,q} \left( T_i, T_j \right) \cap \left( T_i, T_j \right) \right) \cap \\
\left( T_i, T_j \right) \cap \left( T_i, T_j \right) \cdot \cdot \cdot (5.18)
\]

such that \( \beta - 1 < \alpha < \beta \).

150
The corresponding event for the intactness of a formal parameter, $S_{i,x}$, when the \textsc{eval$_\alpha$} instruction of task $T_i$ is executed is similarly given by:

\[
\text{Intacts}_{i,x}(T_i, \text{eval$_\alpha$}) = \text{Intacts}_{i,x}(T_i, \text{block$_{\beta-1}$}) \cap \\
\bigcap_{\beta-1 < \alpha' < \alpha} \sim \text{Need}_{i,x}(T_i, \text{eval$_\alpha$}) \cap \\
\bigcap_{\alpha' < \alpha} \left( \text{Need}_{i,x}(T_i, \text{eval$_\alpha$}) \cap \left( T_{i, \sigma' \cdot T_j} \right) \cap \\
\left( T_{i, \text{block$_{\beta}$} \cdot T_j} \right) \cap \left( T_{i, \text{eval$_\alpha$} \cdot T_j} \right) \cap \left< \text{eval$_\alpha$} \right> \right) \right). (5.19)
\]

• If we know that a child task, $T_j$, reaches its evaluation point between the block$_{\beta-1}$ and block$_{\beta}$ instructions of its parent, $T_i$, then, to decide whether $T_j$ evaluates the locally-declared sequence $Q_{i,q}$, we need an expression for the event that $Q_{i,q}$ has not already been evaluated. As before, the sequential part of the expression is dealt with by Equation (5.16) but we must also account for the following parallel events:

1. That other child tasks reach their evaluation points between block$_{\beta-1}$ and block$_{\beta}$ and require the value of $Q_{i,q}$ prior to $T_j$.
2. That an eval$_\alpha$ instruction occurs between block$_{\beta-1}$ and block$_{\beta}$ and requires the value of $Q_{i,q}$ prior to $T_j$.

The event is therefore given by:

\[
\text{Intact}_{Q_{i,q}}(T_i, \text{eval$_\alpha$}) = \\
\text{Intact}_{Q_{i,q}}(T_i, \text{block$_{\beta-1}$}) \cap \\
\bigcap_{\sigma' < \beta} \sim \left( \text{Need}_{Q_{i,q}}(T_i, \text{eval$_\alpha$}) \cap \left( T_{i, \sigma' \cdot T_j} \right) \cap \\
\left( T_{i, \text{eval$_\alpha$} \cdot T_j} \right) \cap \left< \text{eval$_\alpha$} \right> \right) \cap \\
\bigcap_{\beta-1 < \alpha < \beta} \left( \text{Need}_{Q_{i,q}}(T_i, \text{eval$_\alpha$}) \cap \left( \text{eval$_\alpha$} \cdot T_j \right) \cap \left< \text{eval$_\alpha$} \right> \right) \right). (5.20)
\]

and the corresponding event for a formal parameter to remain intact in the same circumstances is given by:

\[
\text{Intacts}_{i,x}(T_i, \text{block$_{\beta-1}$}) = \\
\text{Intacts}_{i,x}(T_i, \text{block$_{\beta-1}$}) \cap \\
\text{Intacts}_{i,x}(T_i, \text{eval$_\alpha$}).
\]
\[
\bigcap_{\sigma' < \beta, \sigma' \neq \sigma} \left( \text{Need} \delta_{i,x} \left( \sigma', T_j \right) \cap \left( T_1 \sigma'_e T_j \right) \right) \cap \\
\left( \left( T_1 \sigma'_e T_j \right) < \left( T_1 \sigma'_e T_j \right) \right) \cap \\
\bigcap_{\beta-1 < \alpha < \beta} \left( \text{Need} \delta_{i,x} \left( T_1, \text{EVAL}_\alpha \right) \cap \left( \text{EVAL}_\alpha < \left( T_1 \sigma'_e T_j \right) \right) \right). \tag{5.21}
\]

- We can now use Equation (5.20) to express the event that a locally-declared sequence, \( Q_{i,q} \), is intact when a child task reaches its evaluation point, whenever this may be with respect to its parent’s execution. The event is defined as follows:

\[
\text{Intact} Q_{i,q} \left( T_1 \sigma'_e T_j \right) = \bigcup_{\beta} \left( \left( T_1 \sigma'_e T_j \right) \cap \text{Intact} Q_{i,q} \left( T_1 \sigma'_e T_j \right) \right) \tag{5.22}
\]

and the corresponding expression for the equivalent intactness property of a formal parameter, \( \delta_{i,x} \), is given by:

\[
\text{Intact} \delta_{i,x} \left( T_1 \sigma'_e T_j \right) = \bigcup_{\beta} \left( \left( T_1 \sigma'_e T_j \right) \cap \text{Intact} \delta_{i,x} \left( T_1 \sigma'_e T_j \right) \right). \tag{5.23}
\]

- The intactness properties for a formal parameter, \( \delta_{i,x} \), given in Equations (5.17), (5.19), and (5.21) are defined only within the context of the task, \( T_i \), which binds \( \delta_{i,x} \). They assume that the actual parameter, \( T_k \sigma_\delta T_i \), passed to \( T_i \) by its parent, \( T_k \), has not been evaluated prior to \( T_i \) reaching its evaluation point. An expression is therefore required to describe the event that the assumption holds:

\[
\text{Intact} \delta_{i,x} \left( T_k \sigma_\delta T_i \right) = \\
\bigcup_{\eta} \left( \left( T_k \sigma_\delta T_i \right) \cap \right. \\
\left. \left( \left( T_k \sigma_\delta T_i \right) \equiv \delta_{k,x} \right) \cap \text{Intact} \delta_{k,x} \left( T_k \sigma_\delta T_i \right) \right) \cup \\
\left( \left( T_k \sigma_\delta T_i \right) \equiv Q_{k,q} \right) \cap \text{Intact} Q_{k,q} \left( T_k \sigma_\delta T_i \right) \right). \tag{5.24}
\]

5.4.6 Evaluation Properties

Evaluation properties determine whether a task instance will be responsible for executing the instructions of a declared sequence (either one that is declared locally within the task or one that is bound to a formal parameter of the task). These properties will be used to calculate how the cost of evaluating a declared sequence is divided between the tasks which share the sequence.
• For a task to cause the evaluation of one of its locally-declared sequences, the sequence, \( Q_{i,q} \), must be unevaluated immediately prior to the execution of an \texttt{eval} instruction. Furthermore, the \texttt{eval} instruction must demand the value of the sequence. These events are expressed as follows:

\[
\text{Eval}_{i,q}(T_i) = \bigcup_{\alpha} \left( \text{Intact}_{i,q}(T_i, \texttt{eval}_\alpha) \cap \text{Need}_{i,q}(T_i, \texttt{eval}_\alpha) \right). \tag{5.25}
\]

• The equivalent expression for a formal parameter can only be given with reference to the parent of the current task because intactness partially depends on the actions of the parent:

\[
\text{Eval}_{i,x}(T_k \rightarrow T_i) = \\
\text{Intact}_{i,x}(T_k \rightarrow T_i) \cap \\
\bigcup_{\alpha} \left( \text{Intact}_{i,x}(T_i, \texttt{eval}_\alpha) \cap \text{Need}_{i,x}(T_i, \texttt{eval}_\alpha) \right). \tag{5.26}
\]

5.5 Attributing the Time-Cost of Shared Workloads to Task Templates

Using the expressions presented in Section 5.4 we can immediately derive recursion equations for the following properties:

• that a task, \( T_i \), evaluates the locally-declared sequence \( Q_{i,q} \) itself (using Equation (5.25)), and

• that a task, \( T_i \), needs the value of its formal parameter, \( i,x \) (using Equation (5.12)).

This is possible because the required equations are all defined solely in terms of the actions of task \( T_i \) and its children. It is not yet possible, however, to determine whether the necessity of a formal parameter leads to the evaluation of the declared sequence bound to the parameter (i.e. whether the sequence is unevaluated at the time \( T_i \) requires the value). The difficulty arises because \( T_i \) may be invoked by many \texttt{spawn} instructions, each of which binds the formal parameters of \( T_i \) to a different set of declared sequences. The analysis must therefore be performed separately for each set of sequences which may be passed as actual parameters to an instance of template \( T_i \). The analyses will then be collected together to create an overall estimate of the time-cost for \( T_i \).
```
#define abl 60
#define ab2 100
#define bal 20

A {
    Declare QA1 { Work 100 }
    Declare QA2 { Work 500 }
    Work 1000
    \[\alpha = 1\]
    Eval QA1 (15%) | QA2 (25%)
    \[\sigma = 1\]
    Spawn child1 -> B <QA1, QA2> (abl%)
    Work 200
    \[\beta = 1\]
    Block child1 (50%)
    \[\alpha = 2\]
    Eval QA1 (35%)
    \[\sigma = 2\]
    Spawn child2 -> B <$2, $1> (ab2%)
    Work 50
    \[\beta = 2\]
    Block child1 child2
    \[\alpha = 3\]
    Eval QA1 QA2
    \[\beta = 3\]
}

B {
    \[\sigma = 1\]
    Spawn child -> A <$1, $2> (ba1%)
    Work 50
    \[\beta = 1\]
    Block child (30%)
    \[\alpha = 1\]
    Eval $1
    \[\beta = 2\]
    Block child
    \[\alpha = 2\]
    Eval $1 (35%) | $2 (45%)
    \[\sigma = 2\]
    Tail 1 -> <$2, $1>
    \[\beta = 3\]
}
```

Figure 5.5: An example of a recursive Paragon program with declared sequence evaluation and parameter passing.

5.5.1 The Evaluation of Non-local Declared Sequences

To determine which declared sequences it is possible to pass as the actual parameters of an instance of $T_i$ we can perform a collecting interpretation [HY91, Hug87a, BHY89, Hud86] on the whole program.

A collecting interpretation is easily demonstrated by an example. Consider the mutually recursive template definitions shown in Figure 5.5 (the lines of which are annotated with $\alpha$, $\beta$, and $\sigma$ numbers to help clarify the analysis). Template $T_A$ contains two declared sequence definitions which may be passed to an instance of template $T_B$ by $T_A$'s $\text{spawn}_{\sigma=1}$ instruction. $T_A$ also contains a second $\text{spawn}$ which invokes an instance of $T_B$ with the formal parameters of $T_A$ passed in reverse order. The formal parameters of $T_B$ may be passed on to future instances of $T_A$ by $T_B$'s $\text{spawn}_{\sigma=1}$ instruction. Finally, the
The parameter passing activities of the tasks are illustrated by the program graph, $G$, shown in Figure 5.6. Here the `TAIL` instruction of template $T_B$ has been transformed into a `SPAWN` using the analysis of Section 5.2.2. By Equation (5.3), we calculate that the probability, $t_b\%$, of the equivalent recursive spawn is 50%.

The straightforward program graph, $G$, allows us to trace the call-paths of the program and to generate a second graph, $G^*$. There is a one to many mapping of nodes $T_i \in G$ to nodes $T'_i \in G^*$ and the formal parameters of each node in $G^*$ are now associated with a concrete set of declared sequences. However, $G^*$ is still insufficient for a declared sequence evaluation analysis because the edges entering a single node $T'_j$ may represent different `SPAWN` instructions. For example, edges $e_{A_2B_3}^*$ and $e_{B_2B_3}^*$ in Figure 5.7 both enter $T'_j$: the first edge represents the `SPAWN_{σ=2}` instruction of $T_A$ and the second edge represents the `SPAWN_{σ=2}` instruction of $T_B$. Since the declared sequences passed by the two `SPAWN` instructions have different evaluation histories, their chances of intactness will differ from one instance of $T_B$ to the other.

The solution is to expand the graph once more, this time ensuring that each node, $T'_i$, 

Figure 5.6: The graph, $G$, representing the Paragon program shown in Figure 5.5.
in the final graph, \( G' \), is associated with a unique combination of both formal parameter bindings and originating SPAWN instruction. Figure 5.8 completes the example, showing the \( G' \) graph derived from the original templates \( T_A \) and \( T_B \). The components of the \( T_i \rightarrow T_j' \) symbols labelling the nodes indicate the following: \( T_j \) indicates that the node represents instances of \( T_j \); \( T_i \) indicates the template of the task which invoked the instance of \( T_j \); and \( \sigma \) identifies which of \( T_i \)'s SPAWN instructions created \( T_j \). In examples where SPAWN instructions have several branches, it would also be necessary to distinguish nodes according to which branch invoked \( T_j \) by specifying the branch number, \( \eta \).

Now, the recursion equations for a task instance to evaluate one of its formal parameters, \( q_{i,\nu} \), given by Expression (5.26) can be calculated for each node in \( G' \) because the precise SPAWN instruction are specified and, therefore, the evaluation histories of the declared sequences can be derived using Expression (5.24).

5.5.2 The Final Transformation and Analysis of \( G' \)

The final analysis follows and is based on the expanded program corresponding to \( G' \).

Given the recursion equations which express the event that a certain declared sequence, \( Q_{i,\nu} \), is evaluated by an instance of a task \( T_j \) (where \( i \) and \( j \) may or may not be the same), these are transformed into equivalent equations expressing the probability
of the evaluation and solved numerically. The instructions of the declared sequence can now be modified to account for the probability of the evaluation and inserted into \( T_j \), just as if they had occurred as an unshared part of the task.

If the probability of the evaluation is \( p_e \) then the amount of work performed by \texttt{work} instructions in the macro-expansion of \( Q_{i,q} \) must be reduced by \( p_e \). Similarly, the branches of \texttt{spawn} instructions within \( Q_{i,q} \) must have their percentage chances reduced by \( p_e \). When a \texttt{spawn} instruction is added to a task definition, a new edge is correspondingly added to \( G' \). Simplification 2 on page 144 states that the \texttt{spawn} instructions of declared sequences are guaranteed to pass no actual parameters to child tasks. Therefore, the formal parameters of the resulting child task instances are all unbound (equivalent to a binding to an unshared, empty declared sequence). The evaluation history of all unbound parameters is trivially identical, therefore it is sufficient for the new edge to proceed from the current node to any node which represents the correct child template and whose formal parameters are all unbound. If no such node currently exists within \( G' \) then it must be created.

The \texttt{declare}, \texttt{eval}, and \texttt{block} instructions of the task templates (and \texttt{block} instructions of \( Q_{i,q} \)) have no further role to play in the calculations because sharing has now been transformed away; so they are removed. Finally, since task parameters have

Figure 5.8: Expanding \( G^* \) to \( G' \) by path analysis.
(a) Summing costs over the nodes
(b) Summing costs over the arcs

Figure 5.9: The final mapping of costs to the original program graph.

no influence, all references to declared sequences in SPAWN instructions are deleted. This leaves $G'$ in exactly the form required for the simple time-cost analysis of Section 5.2. The simple time-cost analysis is performed on $G'$ and the results mapped back to the original program. The time-cost for a node $T_i \in G$ is given simply by the sum of the time-costs for all the nodes in $G'$ which are derived from $T_i$.

A more accurate mapping is also possible if costs are summed for the arcs of $G$ rather than for the nodes. Figure 5.9 illustrates the difference. By annotating the arcs, the results can be used by the static scheduling technique developed by Maheshwari in [Mah90] and described here in Section 6.3.2.

5.5.3 Notes on the Transformed Program Graph, $G'$

The program graph, $G'$, represents an expanded Paragon program whose task templates map to those of $G$ and which exhibit the same average time-costs as those of $G$. The run-time behaviour of the two programs are very different, however. During an execution of $G'$ no task suspensions would occur due to sharing. Also, the time-costs of individual components of $G'$ are fixed because the shared workloads are distributed evenly. The time-costs of the corresponding components of $G$ vary from one execution to another depending on which component actually executes the whole of the shared workload (which in turn depends on external factors such as processor loading). The transformation from $G$ to $G'$ is therefore solely for the purpose of analysing mean time-costs. All run-time
measurements will be taken from executions of the original program graph.

In Section 5.2.2, the following restriction was placed on the tail-call modifiers of \texttt{SPAWN} instructions: \textit{if several spawns create child instances using the same template then all of the spawns must modify the tail-calls of their child tasks by the same amount.} This allows the \texttt{TAIL} instruction in the child template to be replaced with a recursive \texttt{SPAWN} instruction that is valid for all instances of the child. However, the restriction can be lifted if we generate a separate tail-call node, in $G'$, for each modifier.

For example, consider the task templates defined in Figure 5.10. Both \texttt{SPAWN} instructions of $T_A$ create child instances using template $T_B$ and both modify the number of tail-calls made by their child. By Equation (5.3) the number of tail-calls made by the first and second child instances are:

$$
tb_1 = \frac{m_1 t_B}{m_1 t_B + 1} \quad \text{and} \quad tb_2 = \frac{m_2 t_B}{m_2 t_B + 1},$$

respectively. These are different when $m_1 \neq m_2$, hence two tail-call nodes are required in $G'$ to represent the tail-calls of $T_B$ by recursive spawns; this is illustrated in Figure 5.11.
5.6 Summary

In this chapter a time-cost analysis has been presented for a slightly restricted form of the Paragon language defined in Chapter 3. The analysis was initially developed for a subset of Paragon in which the declared sequence mechanism is banned. We then showed how a program graph that includes sharing can be transformed into a graph that exhibits the same average time-costs as the original but does not include sharing. The analysis is then applied to the second graph and the results mapped back to the original.

The transformation which removes sharing from a Paragon program graph relies on the constraints imposed on the order of evaluation by the Paragon synchronisation features, such as the BLOCK instruction. For unconstrained regions of the graph, it is assumed that all possible orderings are equally likely. Of course, only one of the possible orderings will occur per execution of the workload, and this will result in one specific task evaluating a shared sequence of instructions. Over many executions, however, it is expected that the predicted average behaviour will be observed.
Chapter 6

Techniques for Dynamic Task Management

In this chapter we explore heuristic techniques for dynamically scheduling lazy functional programs on parallel architectures. In particular, interest is focused on distributed memory machines. Test workloads are constructed using the Paragon language which allows the performance of the heuristic techniques to be measured against a wide range of potentially problematic characteristics of lazily evaluated programs.

New heuristics are developed to improve dynamic scheduling in the presence of sharing. The problem of scheduling when run-time input destabilises run-time costs is also considered. Finally, we investigate a new technique for latency tolerance in coarse-grain, message-passing architectures.

6.1 Introduction to Dynamic Task Management

Dynamic task management is a general term encompassing a number of techniques for controlling parallelism at run-time. Ideally, all parallelism would be detected at compile-time and, using an accurate model of the parallel computer on which the program will be executed, the compiler would pre-determine an optimum schedule for executing program tasks on the processing elements of the parallel machine [BP92]. In reality, however, this is infeasible because of factors such as run-time input which mean that the dynamic execution profile of the program is unpredictable [Vra90, §4.6]. Also, higher-order functions and lazy evaluation complicate compile-time analysis enormously. In general, an exact analysis might be intractable (e.g. [HY86, Hol91]), or the time needed to compile
programs might simply be unreasonable (e.g. [HH91]). In all of these circumstances an approximate analysis must be performed. Fortunately, we can offset compile-time approximations by using dynamic task management at run-time to make more effective use of the available information.

The work presented in this chapter assumes that a compiler has suitably annotated executable programs to indicate the availability of parallelism (but that the annotations do not enforce parallel execution). Methods for doing this, such as strictness analysis were mentioned in Chapter 1 and further discussion is given in Section 6.1.1, below. We deal here primarily with the issues of making decisions at run-time based on the compiler-supplied information and further information obtained dynamically. Given an annotated program, run-time mechanisms are therefore required to monitor the state of the parallel machine and to determine when parallel execution is desirable. Parallel execution then occurs whenever parallelism is both available and desirable.

In Section 6.3 the information demanded from the compiler is of a more complex nature but the approach is still to use a mixture of compile-time analysis and run-time information to decide when and where parallel execution will occur. Decisions made at run-time can be grouped into three main categories:

1. how to propagate parallelism (discussed in Section 6.1.2),

2. when to discourage parallelism (discussed in Section 6.1.3), and

3. given a set of tasks that are available for parallel execution, in which order should they be evaluated? (discussed in Section 6.1.2).

These issues are not necessarily orthogonal because the order of evaluation can have an effect on the amount of parallelism available in the system at specific points in time (e.g. see the discussion of LIFO/FIFO scheduling strategies in [RS87] and in Section 6.1.2 below). An inappropriate scheduling decision will then activate the mechanisms for discouraging the generation of parallel tasks.

Section 6.1.4 completes this introduction by discussing the relationship between high-level system design and the effectiveness of dynamic task-control strategies. A short case study is presented to demonstrate that task management strategies are often tuned to a particular architecture and in general are not universally applicable.
6.1.1 Compile-time Decisions Alone are Insufficient for Efficient Parallel Execution

Compile-time analysis can tell us a number of useful things about a functional program including: the asymptotic time-complexity of its functions [Le 85, Le 88, San89, San90]; which function arguments can be evaluated strictly [BH86]; and which expressions are shared and where they are referenced [BPJR88, Gol87, HG85, JL89a]. Each of these can be employed at compile-time to estimate the most desirable run-time behaviour for efficient parallel execution in given circumstances.

Strictness analysis by abstract interpretation [Myc81] is perhaps the best known of the above techniques and has long been used to determine when it is safe to evaluate expressions in parallel. However, it is not possible to find all of the parallelism in a program by compile-time analysis because that would be equivalent to a solution of the halting problem [HY86]. In [Bur87], Burn introduced the notion of evaluation transformers (see also Section 1.3.2) which allow estimates of strictness to be refined at run-time as more exact information becomes available. Evaluation transformers require a small run-time overhead but provide improved execution efficiency because strictness can be detected in more cases than by straightforward compile-time analysis.

Similarly, run-time information can be advantageous when attempting to use the other types of analysis to control the behaviour of parallel program execution. For example, in [Mah90] Maheshwari uses time-complexity information to specify the order in which the strict arguments of individual functions will be made into parallel tasks and demonstrates that improved reduction times are consequently achieved. In Section 6.3 we show that a further improvement can be made over the fixed compile-time orderings when the order is decided at run-time. This is possible because compile-time decisions cannot take into account the context switches that will be made between separate functions because exact execution profiles cannot be determined prior to executing the program (again, this would amount to solving the halting problem [Le 85]).

To support automatic compile-time analysis, it has been advocated that programmers should supply annotations in their source code. Annotations can provide information that it would be impossible for the compiler to detect automatically, such as an insight into the type of input likely to be supplied to the program at run-time. An example annotation is that for specifying which expressions are most likely to benefit from being
evaluated in parallel (e.g. the \{!\} annotation of [PJ89], the `par` annotation of [Roe89], and the @_P parallel-application operator of [Bur84]). It is tempting to imagine that by combining programmer knowledge and compile-time analysis, sufficient information will be available at compile-time to render redundant decisions made at run-time. Even with annotations, however, programs can still benefit from delaying certain decisions until run-time. The programmer might easily provide incorrect information, resulting in degenerate run-time behaviour or may simply underestimate the consequences of placing too many (or too few) annotations. Clayman reports in [Cla93] that injudicious placement of parallel annotations when using the GRIP multiprocessor [PJCSH87] via its remote access mail server [HPJ90] have led to *significant reductions* in run-time efficiency. For programs which have thousands of lines of source code, a trial and error approach to finding suitable annotations would be too great a burden to place on an applications programmer who has little or no knowledge of the expected behaviour of the program.\(^1\) Beyond this section, no further discussion will be given to the subject of annotations, save to say that their over- or under-use ought to be tempered by run-time checks to counteract degenerate behaviour in the light of programmer misapprehensions.

In summary, the case for delaying decisions until run-time is a strong one for many applications. Equivalence to the halting problem is often cited as the main reason why an approximate compile-time analysis is required. Also, uncertainties introduced by allowing programs to operate on input supplied at run-time mean that compilers cannot predict exact run-time behaviour. Programmer annotations might help to overcome some of the problems but cannot be relied upon to provide the whole solution. The remainder of this chapter will therefore work towards improving the quality of information available at run-time and the algorithms for using the information, specifically in the realm of dynamic task management.

### 6.1.2 Arranging for Parallel Execution

For parallel execution to take place, work must be distributed between the processing elements of the parallel machine. A fundamental choice when distributing parallel tasks is whether:

\(^1\)To reinforce this, Clayman further argues that the amount of recent interest expressed in profiling tools for functional programming languages [CPC91, RW92, SPJ92] demonstrates that the behaviour of functional programs is generally *not* well understood.
(a) having created a new task, a PE makes an unsolicited transfer of the work to a remote PE, or

(b) newly created tasks remain with their original PE until a request for work is received from a remote PE.

We shall call the first method *forced scheduling* to denote the activity of forcing work on remote PEs, and the second method *demand scheduling* to denote the request-based transfer. For the purposes of experimentation with the DIGRESS system, we choose to use a demand scheduling strategy. This is prompted by architectural considerations discussed in Section 6.2. Both techniques are discussed below, and a number of other practices commonly used to balance the load of a parallel machine dynamically are also described.

(a) Forced Scheduling

Under forced scheduling, PEs maintain a pool of tasks which have arrived and are awaiting execution (it is common for task pools to be managed as queues [BP92, WWW+88]). When a new task arrives at a PE, and the task pool has sufficient capacity, the new task is added. If the task pool is full then the task must be refused, requiring extra message passing and an increased period of latency. When a PE finishes executing its current task, a pending task is extracted from the pool. If the pool is empty, however, the PE must await the arrival of a task from a remote processor. Figure 6.1(a) illustrates the four steps required to transfer work from a heavily loaded to a lightly loaded processor using forced scheduling.

Choices made at compile-time (e.g. [Lo88]) and by dynamic load-balancing algorithms (e.g. [Sar87, Kun91, Sta85]) will typically be used to decide which processing element will receive a newly sparked task. Also, using a specialised programming language, it is possible for programmers to insert placement annotations into the source code [HS85]. This simplifies the compile-time analysis but restricts portability and requires the programmer to have detailed knowledge of the underlying architecture.

Diffusion Scheduling

Propitious task placement can be achieved when global knowledge of the state of the machine is available. However, global knowledge is expensive to maintain, and much
work has been directed towards schedulers which require knowledge of just a limited number of processing elements [Hud84, Sar87, BP92]. Furthermore, a potential problem of global knowledge is that a lightly loaded processing element may become swamped with work because its low utilisation is known to a large number of busy processors. If many of these elect to place work to the same lightly loaded processor, the loading of the machine will become unbalanced. A solution to this is proposed in [Hud84, HG84, GH86], where the transfer of work is restricted to neighbouring processors only. This encourages the gradual diffusion of work from heavily loaded areas to lightly loaded areas, hence it is known as diffusion scheduling.
Task Migration

To reduce idle times due to poor initial placements, an algorithm may be employed to migrate tasks from busy to idle PEs (e.g. [NS88]). If the initial placements are extremely poor, migrating tasks between processing elements can produce some improvement in efficiency but it has been shown in [ELZ88] that migrating active tasks is not likely to provide significant performance improvements.

(b) Demand Scheduling

Under demand scheduling, newly created tasks are not sent directly to the PEs on which they will be executed. Instead, they are either held locally in a task pool maintained by the creating PE or are sent to a task pool maintained by a remote task manager. Idle PEs are responsible for locating a task pool with a surfeit of tasks and requesting the transfer of work. As for forced scheduling, global knowledge of the loading of the system can improve scheduling by ensuring that requests for work are always sent to the task pool which contains the largest supply of work.

Again, there is a risk associated with global knowledge because it is possible for the heaviest loaded PE to be swamped with requests. The scheduler ought to ensure that not all of the heaviest loaded PE's work is given up, leaving that PE idle and needing to search for work. Global knowledge might also lead to the heaviest loaded PE spending all of its time answering requests and performing no useful work. A balance must be struck between the cost of updating the global information frequently and the penalty paid if too many requests are sent to a single processing element. In [MS90] (where the parallel machine is constructed from a network of transputers) the problems are minimised by sending requests to neighbouring processors only, thereby effecting a diffusion scheduling mechanism.

When a request for work arrives from a remote PE, a task is removed from the local task pool and sent in response to the request. If the task pool is empty, however, a denial will be sent in response and the requesting processor must search elsewhere. By ensuring that an idle PE sends only one request at a time, there is no danger that the idle PE will become swamped with work, and consequently starve other PEs of work. This also means that incoming tasks need never be refused because there will always be sufficient capacity in the local task pool. There is, however, a possibility that idle times
will be protracted because of the latency period of inter-processor communications and the time required to export a task once a request is received. Fortunately, idle times can be reduced if the scheduling scheme is made latency tolerant [Sar87]. When there is no spare work among the busy processors, the idle processors ought to be discouraged from sending requests. These points are discussed further in Section 6.6.

Figure 6.1(b) illustrates the five steps required to transfer work from a heavily loaded to a lightly loaded processor using demand scheduling.

Choosing the next Task to Schedule

Whenever there is a pool of tasks awaiting export an interesting question is which order the tasks should be dispatched for remote execution. The choice of task can have significant effects on the loading of the system and the efficiency of the overall reduction. A simple heuristic is to choose the least recently sparked task by managing the task pool as a FIFO queue [PJCS89, HPJ92]. This causes the program graph to be reduced in a breadth first manner which generates maximal parallelism. (If the machine becomes swamped with excessive parallelism then the FIFO strategy may be abandoned in favour of a LIFO strategy. The program graph is then traversed in a depth first manner, thereby generating minimal parallelism. We expand on the subject of restricting parallelism in Section 6.1.3.) A more complex strategy is to take account of the expected time-cost of the tasks in the task pool [Mah90, Mah92]. Using time-cost information, different selection criteria can be applied, depending on whether the request for work originated from the local or from a remote processing element. This technique is explored in some depth in Sections 6.3 and 6.5.

6.1.3 Restricting Parallelism

If the generation of parallel tasks incurs a run-time overhead and there are too few processors to take advantage of the parallelism then unnecessary overheads will occur. Therefore, whenever wasted parallelism is expensive, it is expedient to limit the creation of tasks. This can be done in several ways; for example:

1. The compiler generates code which produces fewer parallel tasks [HG85, Gol88].

2. A run-time throttling mechanism dynamically adjusts the behaviour of the code so that fewer tasks are constructed [HPJ92, MS90, PJCSH87, BG85].
3. Task generation is made lazy so that minimal overhead is incurred until the task is really going to be executed in parallel [CPJ86].

The first method places a significant burden on the compiler and, as discussed in Section 6.1.1, compile-time decisions are based on incomplete information and are less than optimal. If we intend to allow a dynamically configured machine (see, for example, the proposals of Section 6.2) then many different versions of the compiled program will be necessary for reasonable behaviour to be obtained from a variable number of processors. A lesser degree of compile-time throttling may be useful, however, for setting the minimum granularity of tasks. In general, the compiler should generate code whose sequential threads of execution (or tasks) are sufficiently large for their remote execution to be worthwhile [Gol88]. This is a problem of static program partitioning [Gol88, SH86] and is not tackled in this thesis.

The task-throttling approach is commonly used in modern parallel functional language implementations (e.g. [PJCS89, HPJ92, MS90, Sar87]) and most projects have demonstrated improved execution times for benchmark programs when throttling is used. Typically, throttling is obtained by:

1. Ignoring the option to generate parallel tasks when some pre-defined limit is reached [HPJ92].
2. Switching to a scheduling strategy which naturally produces fewer parallel tasks than the current strategy [RS87, BS81].
3. Choosing to execute versions of functions which generate less parallelism [BG85, Bus87].

One problem with throttling is the possibility that, having reduced the availability of tasks, many currently executing tasks complete and the machine is left with insufficient parallelism to keep all of its processors busy. This situation is particularly acute when a small number of processing elements are producers of tasks and a large number are consumers (see Experiment B.1 in Appendix B for a demonstration of this phenomenon). Another consequence of throttling is that schedulers which use time-cost information to improve execution efficiency (see Section 6.3) will not be able to make the best possible decisions because the throttling strategy acts first, removing potential tasks without consideration for their time-costs.

169
The approach explored in this thesis is that of lazy task generation: rather than reducing the amount of available parallelism, the technique advocates that the cost of building a parallel task is avoided at the time of the spark and delayed until the task is picked up for remote execution. We shall see in Section 6.1.4 that this method requires the system implementor to bear lazy task generation in mind when designing the structure of the reduction machine. As an example, we show that Clack's Four Stroke Reduction Engine [CPJ86] (as implemented on the GRIP multiprocessor [Cla92b]) allows lazy task generation, whereas a stricter regime is enforced by the design described in [HPJ91]. Both designs use the same basic hardware but employ different memory structures.

6.1.4 Architecture Influences Scheduling Policy

The scheduling policies adopted for reducing functional program tasks in parallel depend very much on the underlying machine architecture. This is easily demonstrated by comparing (i) an architecture which provides a flat address space for the whole of the machine's memory, such that all processors have equal access to the whole memory structure, with (ii) an architecture in which processors communicate via message passing.

For shared-memory architectures it is possible to maintain a single, global task pool into which all new tasks are placed when they are created, and from which all processing elements fetch work when they become idle. The overheads for placing a task into the pool and for removing the task for execution are minimal because the original task description is available to all processing elements. Therefore, the number of times that processors are required to fetch work from the pool, and the size of the work units, are relatively unimportant.

On distributed memory architectures, where each processor is tightly bound to its own local memory, a cost is incurred every time a task is exported from one processing element to another for remote execution. This has two main consequences:

1. A task should only be exported when the recipient processor needs the work, the donor has spare capacity, and the overhead of exporting the task is justified by the time required to execute the task.

2. Workloads should be arranged so that the number of exports required to keep every processing element busy is kept to a minimum.

The scheduling requirements of distributed memory architectures therefore differ quite
substantially from those of shared memory architectures. Between these two hardware extremes there is a complete design space for parallel architectures. For example, the GRIP multiprocessor [PJCSH87, PJCS89] described briefly in Chapter 1 is designed with a combination of shared and local memory and provides a useful case-study for examining scheduling policies. Recapping briefly, the GRIP hardware provides each processing element with its own private memory and access to a number of intelligent memory units (IMUs). A single IMU and four processing elements are physically grouped on each circuit board of the machine thus providing processing elements with a three-layer memory hierarchy (local memory, on-board IMU memory, and off-board IMU memory).

To date, at least two distinct memory organisations have been used with the GRIP hardware. The first was used by the interpreted Four Stroke Reduction Engine [CPJ86] and holds all graph cells on the IMUs. The local memory attached to the processing elements is small and used for maintaining an argument stack, abstract machine registers, some intermediate workspace for primitive instructions, and a small amount of code to govern the behaviour of the abstract machine. The task pools are stored in the IMUs and the action of sparking a task (so that it is visible to all processing elements in the machine) consists of placing just two pointers into the local portion of the distributed task pool. This memory arrangement is illustrated in Figure 6.2(a). The second is used by the compiled Spineless Tagless G-machine [PJS89] which holds the program code, the graph, and local task pool in the local processor memory. A distributed, global task pool is maintained in the IMUs [HPJ92] (see Figure 6.2(b)). For garbage collection purposes, pointers from IMU memory to local memory are banned. Therefore, to make a task globally available, its entire state, consisting of graph nodes and code pointers, must be flushed from local memory into the global task pool. Task descriptors are therefore large and expensive to create.

An advantage of the first memory model is that the overhead to spark a task is minimal. However, since all graph reduction is performed using IMU memory then heap access is expensive. A second advantage (specific to the GRIP architecture) is that, by using IMU memory to store the heap, the IMUs can be micro-coded to make high level heap transformations. In the second model, any heap stored in IMU memory is treated just as a task descriptor and is transformed in local PE memory only.

The second memory model speeds up heap access but sparking a task into a global task pool becomes expensive. When the task's description is flushed to the global task
pool, it by no means certain that the task will execute remotely. Therefore, work to export the task is performed non-lazily and there is now an urgent requirement for a throttling strategy to reduce the amount of wasted effort [HPJ92]. This is a direct result of banning pointers from IMU memory into PE memory.

It is apparent from this example that scheduling strategies can depend very much on initial design decisions such as the memory structure of a machine. It is important, therefore, to realise that new scheduling strategies may not be equally applicable to all types of machine.

6.2 The Environment Under Investigation

The scheduling techniques and experiments described in this chapter are based on experience with the DIGRESS distributed memory reduction architecture described in Section 3.1.2. In this section we explain the reasoning behind key scheduling decisions with respect to the DIGRESS architecture. The properties of the DIGRESS system which we consider to have an effect on scheduling strategies are:
1. Every processing element in a DIGRESS virtual machine is connected directly to every other processing element. Therefore one neighbourhood encompasses the whole machine and there is no natural concept of distinct groups of neighbouring elements.

2. All memory is local and private to individual processing elements. There is no globally addressable memory.

3. Inter-processor data transfer and synchronisation are carried out by message passing over a local area network and are therefore expensive.

4. The processor network is dynamically configurable: physical processors may leave and join the network arbitrarily.

5. The DIGRESS distributed system manager maintains global information about the loading of all processing elements.

6.2.1 Choosing between Demand and Forced Scheduling

The first decision is to choose between demand and forced scheduling strategies for disseminating work. If we consider a forced scheduling strategy then Property 5 immediately leads to the problem that processing elements might become swamped with work unless diffusion scheduling is employed (see Section 6.1.2). However, it can be concluded from Property 1 that diffusion scheduling techniques do not map to the DIGRESS hardware quite as naturally as they do on, say, the Alfalfa system [GH86] which uses a hypercube network, or ZAPP [MS90, MS87] which uses a network of transputers. It is possible to impose a notion of neighbouring processors but this somewhat undermines the DIGRESS principle of using global information to the fullest possible extent.

Consider next a demand scheduling strategy where tasks are exported solely in response to requests from idle processors. Using this scheme there is no danger of unwittingly swamping processors with work as long as idle PEs send just a single request for work at a time. Unfortunately, Property 3 means that we are likely to pay for the improved stability with relatively long idle times due to the high communications latency of the system. Also, Figure 6.1 shows that demand scheduling requires one extra message

---

7 Both of these architectures arrange processing elements as the nodes of sparsely connected graphs whose arcs represent direct communication paths. Two processing elements qualify as neighbours if they communicate directly.
to relocate work. This is partially offset by the extra message passing incurred by forced scheduling when workloads are refused by busy PEs and by the fact that latency tolerant heuristics can be used to overcome the latency of the extra message.

Ultimately the choice of strategy for DIGRESS is governed by the level of success in (i) avoiding a badly balanced machine under forced scheduling and (ii) reducing idle periods caused by request latency under demand scheduling. If we fail to solve (i) completely then the possibility remains that a force-scheduled system may reach a pathological state of imbalance. If we fail to solve (ii) completely then a demand-scheduled system will run a little less efficiently than it might otherwise, but no pathological case would be expected. In conclusion, we elect to take the safer option by choosing to implement a demand scheduling strategy and to address the problem of reducing the effect of request latencies.

6.3 Using Time-Cost Information to Guide Strategies for Demand Scheduling

When a request for work arrives at a task pool which contains a number of tasks awaiting execution, the task pool manager is free to choose any one of the tasks to send in reply to the request. If the task manager has advanced knowledge of the amount of time required to execute each task then it will be able to make an informed decision about which task to supply [Mah90, San90]. We define the term time-cost to mean the actual amount of time, in seconds, that a task will take to execute.

6.3.1 Selecting a Metric for Time-cost

Traditionally, the time required to execute an algorithm is expressed as a closed form expression (using 'big-O' notation) which specifies the order-of-magnitude of the expected time as a function of the size of the input [Weg75]. This is certainly appropriate for comparing the efficiency of alternative algorithms operating on the same input. However, it is an inappropriate metric for comparing the relative costs of a number of tasks in order to improve scheduling decisions.

Consider the time-complexity functions $time = k \times (4^n + n)$ and $time = kn \log_2 n$, for input size $n$ and constant factor $k$, plotted in Figure 6.3. The first (exponential) function might represent the expected complexity of an algorithm which first filters its
input (the linear part of the function) and then applies a transformation of exponential cost to the result (which, in this case, is typically $10^4$ times smaller than the original input list). The second (logarithmic) function is exactly that achieved, on average, by a number of common sorting algorithms such as quicksort.

It is clear from the figure that, in the limit, the exponential algorithm will cost more than the logarithmic algorithm. In general, however, the two tasks will operate on different finite input streams and, even if they do operate on the same input, there is a region of the graph in which the relative costs of the two algorithms are reversed. It makes no sense, therefore, to reason about concrete time-costs solely on the basis of asymptotic time-complexities; the actual input sizes and constants of proportionality are needed if reliable comparisons are to be made. This leads immediately to the question of whether or not it is possible to obtain the required values. There are two distinct cases to consider, depending on whether decisions are to be made at compile-time or at run-time.

Generally, programs read their input at run-time. Therefore, input sizes cannot be determined exactly at compile-time. The best that can be hoped for is an approximate characterisation of the input, either supplied by the programmer (in the form of annotations) or derived automatically from knowledge of previous input patterns (the latter approach is discussed in Section 6.5.3). In this case the input sizes are effectively known, thus concrete (albeit estimated) values can be calculated from the time-complexity expressions and can be represented as a simple numeric value.
At run-time the input to the program is available and, intuitively, it seems that it ought to be possible to construct information more accurately than that available at compile-time. It is reasonable to expect that the input to a task will be constructed lazily, as it is demanded, because we are dealing with lazy functional programs. Therefore it will not always be possible to determine the magnitude of some input prior to executing the task which consumes it. Consequently, we must continue to use the estimates of average input sizes which are derived at compile-time. The analysis of Chapter 5 does this for Paragon programs and the results are used to construct simple numeric values which represent the expected average time-costs of tasks.

6.3.2 An Example Scheduling Scheme Based on Time-Costs

In [Mah90], Maheshwari attempts to control parallelism in functional programs with the aid of time-complexity information at compile-time. For each function call, the strict arguments of the function are spawned as child tasks in an order which is expected to produce minimal idle times. This mechanism assumes, firstly, that the spawned tasks will be picked up by remote processing elements in the order in which they were spawned and, secondly, that all tasks will require greater execution time than the total time taken for it to be exported and for their result to be collected. The tasks created to evaluate the strict arguments of a function are exported in descending order of their time-costs for the following reasons:

1. In the event that some tasks are evaluated locally, the communications overhead is better justified by exporting the largest tasks.

2. By exporting large tasks, the remote processing element requesting the work will be kept busy for a longer period and will therefore send fewer work request messages.

3. When a processing element exports work, it must spend some time packaging and sending the tasks [Mah90, Ch.4]. The timing diagram in Figure 6.4 demonstrates that because of this overhead, it is better to export the most expensive of a number of tasks first.

We can state the last of these points more precisely. Let $C_{\text{export}}$ be the time taken to arrange for a task to be exported; $C_{\text{latency}}$ be the average network latency; $C_{\text{return}}$ be the time taken to package and return a result; and $C_{T_i}$ be the time-cost for task
In terms of the idle time experienced by the processor exporting work for remote execution, the timing diagram in Figure 6.4 shows that an idealised synchronisation can be achieved if $C_{T_{i+1}} - C_{T_i} = C_{\text{export}}$ for all tasks $T_i$ and $T_{i+1}$ exported consecutively, and $C_{T_i} = C_{T_x} + 2 \times C_{\text{latency}} + C_{\text{return}}$ where $T_l$ is the one task evaluated locally and $T_x$ is the least expensive of the exported tasks.

This assumes that we can relate communications costs and execution costs mathematically. To avoid having to quantify the relationship, Maheshwari suggests that tasks are exported in descending order of their time-cost, excepting the following heuristic:

"if [the cost of communication] $C_{\text{comm}}$ is very high compared to [the cost of creating n tasks] $\pi C_{\text{creation}}$, then the most complex argument should be kept for local execution
— if not, then the least complex one should be kept for local evaluation."

In reality, there is no guarantee that there will be sufficient processing capacity in the machine to execute all but one task in parallel. This will result in multiple tasks being executed locally. We do not implement the above heuristic because the sum of a number of small, locally evaluated, tasks may well prove to be greater than each of the larger tasks exported to remote processors. By keeping the smallest tasks local, the cost of exporting is always set against the most expensive tasks.

Child tasks might be spawned at any time during the lifetime of their parent and in any number of batches. In Section 6.3.3 we show that the simple, compile-time scheme can fail when this occurs and in Section 6.3.4 we present a solution to the problem which requires time-cost information to be carried at run-time.
6.3.3 Compile-Time Scheduling can Fail when Tasks Suspend

Lazy evaluation might result in tasks being temporarily suspended when there is contention for shared values. For example, a task, $A$, may demand the value of a shared expression, $E_s$, when $E_s$ is already being evaluated by another task, $B$. In this case, $A$ must suspend until $B$ has completed the evaluation. When the suspension is no longer required, task $A$ can be resumed. If another task, $C$, has begun execution in the meantime then resumption will not occur immediately. Instead, the resumable task, $A$, is placed in a special task pool and will be resumed as soon as task $C$ completes or suspends. We noted in Section 6.1.2 that performance gain is unlikely when active tasks are migrated [ELZ88]. Therefore, we keep resumable tasks separate from newly sparked tasks and do not allow them to be exported for remote execution.

Another cause of suspension during parallel evaluation is that a task may be blocked while a closure is fetched from another processor. The blocking mechanisms can make the compile-time scheduling strategy described in Section 6.3.2 behave badly. Figure 6.5 shows seven stages of a task management sequence to illustrate how task suspension disrupts the compile-time scheduling:

Stage 1: Task $A$ is currently executing.

Stage 2: Task $A$ requires the value of a remote expression. It is therefore temporarily blocked and must suspend. The processing element must now wait until more work is transferred from another PE because there are no tasks in the local task pool (the mechanism for transferring work between PEs is transparent to this example).

Stage 3: Task $B$ arrives from another PE and becomes the current task.

Stage 4: Task $B$ spawns a number of child tasks $B_1 \ldots B_n$ which are candidates for local or remote execution. They are spawned in the order of their relative time-costs according to the rules of the compile-time scheduling strategy.

Stage 5: The remote expression (or its value) on which task $A$ was blocked is now available. Task $A$ is therefore moved to the pool of resumable tasks.

Stage 6: Task $B$ now requires the value of a remote expression and suspends. The processing element resumes task $A$ (we assume that resumable tasks have priority over virgin tasks residing in the local task pool).
Figure 6.5: Compile-time scheduling may fail when tasks suspend.

Stage 7: Task $A$ spawns a number of child tasks $A_1 \ldots A_m$ which are candidates for local or remote execution. Again, these are spawned in the order of their relative time-costs according to the rules of the compile-time scheduling strategy.

In the final stage, the local task pool contains two distinct sequences of spawned tasks. The strategy ensures that the tasks within a single sequence are in the correct order but makes no claims about tasks taken from separate sequences. Therefore it is possible for the cost of $A_i$ to be greater than the cost of $B_j$ for any $i < m$, $j < n$. The compile-time scheduling strategy may therefore effect a smaller reduction in idle times and communications overheads than initially expected.

### 6.3.4 Making Time-cost Information Available at Run-time

Maheshwari's compile-time scheduling scheme can be improved by retaining time-cost information at run-time for all tasks. This requires a space overhead to store costs and a small time overhead to compare costs dynamically but allows tasks spawned by different parents to be scheduled correctly with respect to the order of their expected time-costs. The problem reported in the previous section is therefore solved and the improvement is easily demonstrated by experiment. To do this we implement three scheduling strategies
as follows:

**Random**: given a pool of unevaluated tasks, the choice of which task to execute next, either locally or remotely, is random.

**Localised**: the task pool is managed as a FIFO queue with respect to supplying tasks for remote execution and as a LIFO queue for supplying tasks for local execution. The compiler arranges for the strict arguments to a function to be spawned in descending order of their expected time-cost, according to Maheshwari's compile-time scheme [Mah92]. The queuing mechanism provides a correct time-cost ordering for the spawns of each individual function call unless the pool contains tasks spawned by more than one function call. In this case, a correct ordering cannot be guaranteed for the pool as a whole (hence the term *localised* scheduling).

**Global**: every task has a time-cost weighting attached to it at run-time. The algorithm is similar to the localised scheme except that it is possible to compare the expected time-cost of *any* two tasks at run-time. This allows the whole task pool to be maintained as a sorted list, thereby providing a correct ordering throughout.

The test workload given by Program A.1 in Appendix A was executed for the localised and global scheduling techniques, over a range of granularities. In the program, instances of the task templates Task_A and Task_B are equivalent to tasks A and B in Figure 6.5 and their respective children are equivalent to $A_1 \ldots A_m$ (where $m = 1$) and $B_1 \ldots B_n$ (where $n = 20$). The results are plotted in Figure 6.6 and show a significant reduction in run-time overheads when a global time-cost strategy is used in preference to the localised scheme.

### 6.3.5 Sharing Affects Decisions Based on Time-cost Information

Lazy evaluation will affect any scheduling strategy which is guided by time-cost predictions. Whenever two or more tasks share a sub-computation, the time-cost for evaluating the expression will be incurred by one task only. The actual time-cost of the other tasks will therefore differ from their potential time-costs. If expected time-costs are calculated and used without taking sharing into account, it is possible that the scheduling strategies given above may fail to improve efficiency. This is illustrated in Figure 6.7 where six tasks are held in a task pool, pending execution. The tasks, $A \ldots F$, are ordered according to
Figure 6.6: Global scheduling outperforms localised scheduling when tasks suspend.

their expected time-costs. If the tasks' time-costs include the cost of executing the shared work then the actual costs will be different because each piece of shared work is performed only once. If task A and task F are initially removed from the task pool for remote and local execution, respectively, then the real costs of the tasks are those given in column (i) of Table 6.1 which indicates that the ordering should have been A, F, D, B, E, C.

However, if the new ordering is used instead of A . . . F then F is no longer considered to be the least expensive task. C is now chosen for the initial local execution and therefore evaluates the shared work. Consequently C is now more expensive than F and we conclude that the effect of sharing in this case is to cause the scheduling strategy to behave paradoxically. Sharing has ensured that the task evaluated locally will cost more than other tasks executed remotely, thereby defeating the aim of the strategy.

The other two columns in Table 6.1 show the real costs of the tasks when the A . . . F ordering is used with different local and remote work requirements. Firstly, in column (ii), four local evaluations are performed prior to a remote evaluation (therefore, tasks F . . . C are evaluated before task A). Secondly, in column (iii), four remote evaluations are
performed prior to a local evaluation (therefore, tasks A ... D are evaluated before task F). In both of these cases the orderings are such that the cost of certain tasks executed locally exceeds the cost of other tasks executed remotely.

**An Experimental Demonstration of the Sharing Problem**

A single Paragon source is used to drive the experiment and conditional compilation is used to generate two related workloads:

1. with delayed (but not shared) argument passing (see Program A.2, page 217), and
2. with shared argument passing (see Program A.3, page 218).

Both workloads require the same amount of overall processing time and have the same dynamic call-graph; the difference between them is limited to the style of argument passing between tasks. Each workload is executed a number of times with granularity scales of 0.5, 1.0, and 2.0 (see Section 3.3.2 for a definition of unit sizes).

The mean overheads (measured with respect to useful work) are illustrated graphically in Figure 6.8. In the absence of sharing, both the localised and global scheduling strategies reduce the run-time overhead. In the presence of sharing, however, the correlation between the expected and actual time-cost of tasks is reduced and, as expected, the strategies fail to make an improvement. Therefore, we conclude that, for programs where
<table>
<thead>
<tr>
<th>Task</th>
<th>Expected Cost</th>
<th>Actual Cost (local/remote)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(i)</td>
<td>(ii)</td>
</tr>
<tr>
<td>A</td>
<td>2000</td>
<td>2000 (remote)</td>
</tr>
<tr>
<td>B</td>
<td>1800</td>
<td>800 (remote)</td>
</tr>
<tr>
<td>C</td>
<td>1600</td>
<td>600 (remote)</td>
</tr>
<tr>
<td>D</td>
<td>1400</td>
<td>900 (local)</td>
</tr>
<tr>
<td>E</td>
<td>1200</td>
<td>700 (local)</td>
</tr>
<tr>
<td>F</td>
<td>1000</td>
<td>1000 (local)</td>
</tr>
</tbody>
</table>

Table 6.1: The real costs of executing the tasks given in Figure 6.7.

the time-costs of shared workloads are significant with respect to the overall time-cost of individual tasks, the resulting instability in the actual time-cost of tasks will cause scheduling strategies based on time-cost analysis to fail to reduce communications overheads. This phenomenon is sufficiently important to warrant further investigation and we return to the issues of instability in Section 6.4.

6.3.6 Improving Time-cost Estimates by Taking Sharing into Account

The problem observed in the results of Figure 6.8 occurs because we failed to take sharing into account when time-cost predictions were made. When an expression is shared, its whole time-cost was wrongly attributed to every task that requires the value of the expression. Summing the time-cost of the tasks therefore gives an unrealistic prediction of the time-cost of the whole program because the shared work is counted many times over. A better solution would preserve the overall time-cost for the program. This can be achieved by dividing the time-cost for an item of shared work between the tasks which share it.

Chapter 5 showed a time-costs analysis that accounts for sharing in workloads generated using a subset of the Paragon task modelling language. In order to improve the accuracy of the division, the analysis estimates the probability for each task to evaluate the shared workload itself and apportions the workload's time-cost accordingly. A similar analysis for the full Paragon language is much harder, and for higher-order, lazy functional programs may require assumptions and simplifications to be made. (A naïve
approach would be to divide the shared time-cost \textit{equally} between all tasks which require
the value. This is inaccurate but would at least provide the correct \textit{overall} time-cost.)

Figure 6.9 shows the relative overheads measured for Program A.6 over a range of
granularity scales. The upper curve was obtained when the workload was dynamically
scheduled according to a time-cost analysis which failed to take sharing into account. The
lower curve was observed when the same workload was dynamically scheduled according
to the time-cost analysis of Chapter 5. The lower overheads obtained by the more
intelligent time-cost analysis are clearly preferable. The improvement is due to the fact
that overall costs of each task (including those of its children and shared expressions) are
more accurately represented.

A key feature of the workload used to stimulate the above experiment is that there are
no large deviations from the mean costs calculated by the analysis. In Section 6.5.4 we
show that even when sharing is analysed sensibly, large deviations from mean time-cost
predictions can still pose problems. This is addressed with further adjustments to the
dynamic scheduling scheme.
Figure 6.9: Improving Dynamic Scheduling by Accounting for Sharing.

6.4 Instability in Functional Workloads

The term *stability* is used here to describe the degree of conformity between the measured run-time behaviour of a functional program and the average behaviour predicted at compile-time. Conversely, *instability* is the degree of non-conformity between predicted and observed behaviour. Instability can be caused by any of the following factors:

**Run-time input** Where the size of input (e.g. the length of a list, or the magnitude of a number) dictates the number of recursive calls within a program segment. Also, conditional branching can be determined by run-time input (e.g. in an interactive, menu-driven program, user input is used to select one of a number of menu options). Each alternative branch may then result in a different amount of processing effort.

**Laziness** Where, as demonstrated in Section 6.3.5, the total cost of evaluating a task is dependent on whether or not other tasks evaluate shared expressions in advance.

**Degree of parallelism** Where different behaviour will be experienced according to the number of processors currently being employed to reduce the program.
The first two of these are illustrated by the Paragon Program A.4 given in Appendix A. In this program, a single tail-recursive parent spawns three children during each iteration, using task templates process1, process2, and process3, respectively. Fifteen tail-calls are made, giving sixteen iterations in total.

The results of the experiment are shown graphically in Figure 6.10. For each iteration, two bars are plotted, illustrating the cost of executing process3 relative to the costs of process1 and process2, respectively. It is clear that, on average, process3 is more expensive than process2 and less expensive than process1. However, during the sixth, eleventh, and final iterations, the relative costs are altered. Also, during the sixth iteration, the relative costs of process1 and process2 are reversed, with respect to all other iterations. The effects observed for this example are due to the combination of shared evaluation with run-time behaviour influenced by simulated run-time input.

In the next section we attempt to improve dynamic scheduling by taking instabilities into account.

### 6.5 Further Improvements to Time-Cost Schedulers

So far it has been demonstrated that time-cost information can be used (i) at compile-time only, and (ii) to slightly better effect at run-time, to reduce the overheads of parallel
computation. Instability due to sharing was also illustrated. In this section we improve the preceding scheduling techniques to overcome the problem of workloads whose mean time-costs are biased by instabilities.

6.5.1 Risk Aversion and the Bumble-Bee

In the search for novel methods of improving dynamic scheduling an interesting relationship was encountered between parallel processing with demand scheduling and the activities of foraging insects (specifically the bumble-bee, which forages without communicating with other bumble-bees [Rea91]). The pictograms of Figure 6.11 illustrate the following analogies between demand scheduling and foraging as performed by the bumble-bee:

\[
\begin{align*}
\text{work} & \leftrightarrow \text{nectar} \\
\text{processing element} & \leftrightarrow \text{bumble-bee} \\
\text{task} & \leftrightarrow \text{flower} \\
\text{task pool} & \leftrightarrow \text{flower bed}
\end{align*}
\]

The \text{work} \leftrightarrow \text{nectar} analogy is straightforward. Bumble-bees need \text{nectar} to keep them alive, and processing elements (PEs) need \text{work} to keep them busy. In the second analogy, the idle PE sends a work request which is effectively foraging for work in a similar manner to the bee. This leads to the third analogy. Bumble-bees select flowers according to their past utility [Rea91], using recognisable attributes such as colour to guide the selection. In previous sections we have attempted to express the expected utility of a task instance by making reference to the template and the \texttt{spawn} instruction used to create the instance. The final mapping, \text{task pool} \leftrightarrow \text{flower bed}, is explained as follows. The bumble-bee forages within a particular flower bed because it believes that food is available there; a PE sends its work request to a specific remote task pool because it believes that work is to be found there.

The analogy is distorted slightly because, in the parallel processing domain, the intelligence for choosing a task does not come from the foraging agent, but from a task pool manager. Therefore, the \text{PE} \leftrightarrow \text{bumble-bee} analogy can be expressed by a total of three separate relationships:

\[
\text{bee's desire for food} \leftrightarrow \text{PE's desire for work}
\]
Figure 6.11: The equivalence between foraging insects and PEs looking for work.

\[
\text{foraging excursion} \leftrightarrow \text{issue of work request}
\]

\[
\text{bee's intelligence} \leftrightarrow \text{remote task pool manager}
\]

Having established a link between bumble-bees and parallel processing, we can observe the algorithms employed by the bees when foraging for food and attempt to discover whether they can be applied to problems in the parallel processing domain. Importantly, the results of [Rea91] show that bumble-bees are ready to forsake high mean/high risk returns in favour of lower, but more consistent returns. This is known as risk aversion. If a certain type of flower returns a high average yield of nectar, but the mean is biased by a small number of excessively high yields, then the bee will ignore the flower. A biased mean implies a number of low yield flowers, visits to which may lead to a short-term net reduction in energy.

We have already seen that there is a potential problem with mean yields in the parallel processing domain. If the time-costs of two or more workloads are compared on the basis of their mean, expected yields then there is a chance that means biased by outlying values will result in unreliable comparisons (see the simulation in Section 6.5.4). We shall therefore attempt to develop a risk aversion technique for time-cost schedulers.
6.5.2 Risk Aversion by Modal Analysis

In Section 6.3.2, advantages were shown for exporting tasks in the order determined by their time-costs. Up until now, time-costs have been calculated as the mean time expected for a task instance to be executed. In Section 6.4 the presence of instability in the time-cost of tasks was illustrated. Given that instability might occur frequently in real workloads which consume run-time input and which have many shared expressions, we suspect that the mean time-cost is too unreliable to be used as a metric for scheduling tasks. It would, perhaps, be better to take the mode of the frequency distribution of time-costs. Unfortunately, there is no reason to assume a continuous, unimodal frequency distribution for time-costs.

Furthermore, consider the example Paragon segment in Figure 6.12 which spawns tasks using three task templates, A, B, and C. Using \( cT \) to denote the observed time-cost of a task instance, \( T \), it is quite possible for \( (cA_1 > cB_1), (cB_2 > cC_1), \) and \( (cC_2 > cA_2) \) in the same program run. This suggests a relationship between templates A, B, and C that is not transitive. To express this, we can define a comparison operator, \( \theta \), such...
that

\[(A \otimes B) \land (B \otimes C) \not\Rightarrow (A \otimes C).\]

\(A \otimes B\) indicates that an instance of template \(A\) takes longer to execute than an instance of template \(B\), with respect to some pre-defined context. For workloads that have already been executed, the \(\otimes\) relation can be defined as follows:

\[A \otimes B \equiv \#(c_A > c_B) > \#(c_B > c_A) \quad (6.1)\]

In definition (6.1), the \(\overline{cT}\) syntax is extended to task templates. It refers to the observed time-cost of a task instance created using template \(T\). The expression \(\#(aRb)\) denotes the number of times that the relationship \(aRb\) was observed to be true.

Clearly, definition (6.1) is meaningful only when the expression \((\overline{cA} > \overline{cB})\) has meaning; and it is not sensible to compare \(\overline{cA}\) with \(\overline{cB}\) if no operational relationship exists between \(A\) and \(B\). As the comparison will be used to determine an ordering for use with dynamic scheduling algorithms, the run-time comparisons made between task instances by the scheduler is a good basis for determining the validity of comparing \(\overline{cA}\) with \(\overline{cB}\).

In our current implementation, comparisons are made when a new task instance is spawned and inserted into the task pool. For example, if the task pool contains tasks \(X\) and \(Y\) when task \(A\) is spawned then \(A\) is compared with the two tasks already in the pool; hence, in this example, the expressions \(\overline{cA} > \overline{cX}\) and \(\overline{cA} > \overline{cY}\) are meaningful. Applying this interpretation, definition (6.1) retrospectively states whether or not one task instance ought to have been considered more expensive than another for scheduling purposes.

In Section 5.5.2 it was shown that predicted time-costs can be attached either to the nodes of the static call-graph for Paragon programs or to the arcs. In the former case, the \(\otimes\) relation is defined over task templates. In the latter case the \(\otimes\) relation requires extra contextual information in the form of the \texttt{spawn} instruction used to create the child task instance.

In the next section we show how the \(\otimes\) relation can be used to improve the basic dynamic scheduling algorithms that use predicted time-costs to determine the order in which tasks are executed.
6.5.3 Stochastic Learning Automata

A stochastic learning automaton (SLA) is a device which uses previously recorded behaviour to make adjustments to future behaviour [NT74, MS86]. For example, in [Kun91] Kunz uses an SLA to guide forced task scheduling. Whenever a processor places a task with another processor, the recipient responds with a score. The score indicates the recipient's level of agreement with the scheduling decision, based on factors such as the number of tasks already awaiting execution, the size of its free memory, the current rate at which the CPU is context switching between tasks, the rate at which operating system calls are being made, and the processor's one-minute load average. When a processor needs to place a task, it generates a random number to select a recipient according to a locally maintained probability vector. Scores received from recipients are then used to adjust the probability vector.

In [Mah90], Maheshwari discusses the idea of obtaining time-cost information by profiling previous executions of a program. This is a form of SLA although the granularity of information update is far coarser than the example given above because time-costs are calculated after the program has completed its execution. Subsequent runs of the program can then use the old information to predict mean time-costs.

In the previous section we defined a new operator, ©, for comparing expected time-costs based on past knowledge. The knowledge is gathered as the program executes and might be applied to the current program if the analysis can be performed on-the-fly. Alternatively, the data can be post-processed and applied to subsequent executions of the program in the manner suggested by Maheshwari. The information required by the © comparison operator is different to that required for a mean time-cost profile. A matrix of counts is maintained either for the nodes or arcs of the static program graph:

\[
C = \begin{bmatrix}
  c_{11} & c_{12} & \cdots & c_{1\tau} \\
  c_{21} & c_{22} & \cdots & c_{2\tau} \\
  \vdots & \vdots & \ddots & \vdots \\
  c_{\tau1} & c_{\tau2} & \cdots & c_{\tau\tau}
\end{bmatrix}
\]

where \( \tau \) denotes the number of arcs or nodes in the graph. Assuming that we are annotating the time-cost of the nodes of the graph, the entry \( c_{ij} \) indicates the number of times that node \( i \) resulted in a more expensive task than node \( j \). Therefore, definition (6.1)
can be restated as:

\[ T_i \otimes T_j \equiv c_{ij} > c_{ji}. \]  

(6.2)

When the arcs of the graph are annotated, the definition is similar but Equation (6.2) is expressed for arcs (spawn instructions) instead of nodes (task templates).

Equation (6.2) can be represented simply by a matrix of boolean values:

\[
B = \begin{bmatrix}
    \text{False} & (c_{12} > c_{21}) & \cdots & (c_{1r} > c_{r1}) \\
    (c_{21} > c_{12}) & \text{False} & \cdots & (c_{2r} > c_{r2}) \\
    \vdots & \vdots & \ddots & \vdots \\
    (c_{r1} > c_{1r}) & (c_{r2} > c_{2r}) & \cdots & \text{False}
\end{bmatrix}.
\]

Since every entry in \( B \) is defined by the \( > \) operator, it does not always follow that \( b_{ij} = \neg b_{ji} \) (where \( \neg \) is the boolean not operator). If \( c_{ij} = c_{ji} \) then \( b_{ij} = b_{ji} = \text{False} \).

The above definition of \( B \) does not take account of any (compile-time) time-cost analysis whatsoever. It merely reports observations from previous program executions and implements a risk aversion strategy. Mean time-cost analysis is not wholly redundant however. When \( c_{ij} = c_{ji} \), we can use a prediction of the mean time-costs to make the decision (in fact, this is similar to the mechanism employed by the bumble-bee for whom mean utility has some part to play in the decision making process when risks are perceived to be small [Rea91]). Therefore, the boolean decision matrix for scheduling becomes:

\[
D = \begin{bmatrix}
    d_{11} & d_{12} & \cdots & d_{1r} \\
    d_{21} & d_{22} & \cdots & d_{2r} \\
    \vdots & \vdots & \ddots & \vdots \\
    d_{r1} & d_{r2} & \cdots & d_{rr}
\end{bmatrix}
\]

where

\[
d_{ij} = \begin{cases}
    \text{True}, & \text{if } c_{ij} > c_{ji} \lor ((i \neq j) \land (c_{ij} = c_{ji}) \land (\overline{c_{Ti}} > \overline{c_{Tj}})) \\
    \text{False}, & \text{otherwise}
\end{cases}
\]

and \( \overline{c_{Ti}} \) is the mean expected time-cost of the template \( T_i \) (again, if arcs rather than nodes are annotated with time-costs, an equivalent expression is used for costs due to spawn instructions rather than task templates).
6.5.4 Scheduling with Risk Aversion

To demonstrate the advantage of scheduling with a risk aversion strategy rather than relying on mean time-cost information, a simulation is constructed using Paragon Program A.7 from Appendix A. The workload consists of a tail-recursive task \( A \) which spawns a number of children. Synthetic closures are shared between \( A \) and its children and the mean time-cost of each child is influenced by the \texttt{EVAL} instruction within task \( A \). The relevant segment of task \( A \) is defined as follows:

\begin{verbatim}
  : Spawn c -> C <closure4, closure1>  // closure4 costs 500, closure1 costs 4500
  Spawn d -> D <closure3>  // closure3 costs 2000
  Spawn e -> E <closure5, closure2>  // closure5 costs 500, closure2 costs 5000

  Eval closure4 closure2
   | closure5 closure1 (factor%)

  : 
\end{verbatim}

If the first branch of the \texttt{EVAL} is selected then child instance \( c \) is the most expensive, followed by \( d \), and then \( e \). If the second branch is selected then instance \( e \) is the most expensive, followed by \( d \), and then \( c \). The selection is governed by the percentage probability: \texttt{factor}. When \texttt{factor} is fixed at zero, the first branch is always selected. When \texttt{factor} is 100, the second branch is always selected. In between, instability is introduced because a mixture of the two time-cost orders is experienced.

Figure 6.13: Mean time-costs calculated for Program A.7.
Figure 6.14: Risk aversion is at least as good and can be better than scheduling with mean time-cost information.

The mean time-cost analysis of Figure 6.13 shows that child instances $c$ and $e$ have approximately equal mean time-costs when $\text{factor}$ is fixed at 29.4. Below this, $c$ is perceived to be more expensive than $e$, and above this $c$ is perceived to be cheaper than $e$. From our knowledge of the \texttt{eval} instruction and its effect on time-cost orderings, however, it would be better to assume that $c$ is the most expensive for all values of $\text{factor} < 50$. The risk aversion (stochastic learning) strategy of Section 6.5.3 does exactly this by observing run-time behaviour. The simulation was executed for $0 \leq \text{factor} \leq 100$ with tasks dynamically scheduled (i) randomly, (ii) according to risk aversion via a stochastic learning process, and (iii) according to mean time-cost information. The performances of the three scheduling strategies are plotted in Figure 6.14 and are summarised as follows:

(i) The \textit{random} schedule exhibits an approximately linear relationship with $\text{factor}$.

Observing the extremes of Figure 6.13, it is clear that this occurs because the
range of time-costs for \( c, d, \) and \( e \) is narrower when the first branch of the \texttt{EVAL} instruction is executed (e.g. when \( \text{factor} = 0 \)) than when the second branch is taken (e.g. when \( \text{factor} = 100 \)). As the difference between the costs is increased, the load becomes less evenly balanced between processors. (This effect also occurs for the risk aversion and mean time-cost schedules, but is less obvious because their run-time overheads are influenced by more dominant factors.)

(ii) The risk aversion schedule is least successful when \( \text{factor} = 50 \). At this point the workload is at its most unstable because both branches of the \texttt{EVAL} instruction are equally likely. Even the most accurate (fixed) prediction of which child task will be the most expensive will be incorrect in at least half of the cases. The risk aversion schedule improves as \( \text{factor} \) becomes smaller or larger than 50. The improvement demonstrates an approximately linear relationship between instability and observed run-time overhead.

(iii) For \( 0 \leq \text{factor} < 30 \) the schedule based on mean time-cost information closely follows the risk aversion schedule. When \( \text{factor} \approx 30 \) a worst case performance is observed for the time-cost schedule. This is just beyond the point at which the mean time-cost analysis starts to report that child instance \( e \) is more expensive than instance \( c \). The tasks are therefore scheduled correctly for approximately 30\% of cases. The performance is worse than that achieved by the random algorithm. Incorrect scheduling continues until \( \text{factor} = 50 \). Beyond \( \text{factor} = 50 \) the performance of the mean time-cost schedule is again similar to that of the risk aversion algorithm.

When \( \text{factor} = 50 \), the risk aversion and mean time-cost algorithms are both better than random scheduling. This is because random scheduling sometimes schedules child instances other than \( c \) or \( e \) first, thus resulting in an incorrect schedule. The risk aversion strategy never predicts this ordering and the time-cost strategy predicts that either \( c \) or \( e \) is the most expensive, unless \( 25 < \text{factor} < 33 \) (see Figure 6.13) when \( d \) is scheduled first.

In the above example, the risk aversion schedule gives similar performance to the mean time-cost schedule over most of the graph, and significantly better performance when \( 30 \leq \text{factor} \leq 50 \).
6.5.5 Conclusions and Further Work with Risk Aversion

We have demonstrated that improvements in dynamic scheduling can be achieved if task pool comparisons are observed at run-time and the outcomes used to influence future scheduling decisions. The technique requires that every comparison is logged and that the time-cost of the tasks so compared is monitored. This is an expensive operation. The technique is readily applicable to the Paragon synthetic workloads because these are always executed with full tracing facilities enabled. For real functional languages, however, tracing is an overhead that must be accounted for.

To overcome the cost of measuring run-time costs and logging comparisons, it might be useful to investigate instability further. If it is possible to predict at compile-time which tasks are most likely to be unstable then we can reduce the number of tasks that are traced and whose task pool comparisons are logged by concentrating on these only. For the other tasks, we can rely on the mean time-cost analysis that was applied in Section 6.3.

6.6 Managing Latency Periods

In [Sar87] Sargeant makes three points about latency periods in parallel processing:

1. Latency causes processors to idle. For example, this can occur under demand scheduling (see Section 6.1.2) when a processing element (PE) completes its current task and has no further tasks in its local task pool. The PE sends a request-for-work message to the most heavily loaded PE, and idles until either a task descriptor or a denial is received.

2. Latency causes a process switching overhead.

3. Latency costs nothing if there is sufficient parallelism because there is always alternative work to perform when a task is blocked or messages are in transit.

At the time of his article, Sargeant believed that the property of latency-tolerance was restricted to fine-grain machines. The purpose of this section is to investigate the possibility of applying latency-tolerance techniques to our coarse-grain architecture. The experiment presented here represents a preliminary investigation into improved latency-tolerance. Completion of the experiment relies on an extension to the current imple-
Figure 6.15: Increasing the number of PEs per physical processor to improve latency-tolerance.

mentation of the DIGRESS architecture [Cla92a] and is reserved for future work. The preliminary results are, however, encouraging.

6.6.1 Improving Latency-Tolerance by Reconfiguring the Architecture

On the DIGRESS architecture, processors are typically Unix workstations on which PEs are executed as virtual processors. It is theoretically possible to execute more than one PE per physical processor as illustrated in Figure 6.15. Notice that only one local system manager (or Ism [GB91]) is present on each physical processor, irrespective of the number of PEs. All of the PEs on one physical processor communicate with the outside world (and with each other) via the same local system manager.

Intuitively, there appears to be a major disadvantage in executing multiple PEs on a single processor because an extra overhead is experienced: the operating system is required to continually switch between PE jobs to enable them to run concurrently. However, the likelihood of a physical processor becoming idle is reduced. If one of its PEs enters an idle state then the remaining PEs are able to consume a correspondingly larger number of CPU cycles during the period of idleness. This has the effect of improving the latency-tolerance per physical processor. An improvement in absolute execution times relies on the following factors:

1. There is sufficient potential parallelism in the workload to warrant the increase in
PEs.

2. The improved latency-tolerance has a more significant effect than the increase in overhead due to extra job scheduling by the operating system (and other factors such the inevitable reduction in locality of reference when more PEs are employed).

The first factor is important. It can be argued that if there is sufficient potential parallelism for more PEs then it would have been better to use more physical processors. However, as the DIGRESS processors are workstations, the limit on the number of processors that can be assigned to the computation is likely to be low. The technique is intended primarily for programs which exhibit more parallelism than can be exploited in the usual manner by the available processors. We make no claims for the technique with respect to massively parallel machines.

In response to the second factor we state the following hypothesis:

**Hypothesis 6.1** *It is expected that (subject to a saturation point for the number of PEs per physical processor) job scheduling overheads incurred by the operating system will be small in comparison to the improvement achieved by improved latency-tolerance on a coarse-grain parallel machine.*

The motivation for this hypothesis is that, as granularity is increased to overcome the expense of message passing [Mah92], the frequency with which new tasks are created will be reduced. This will have the effect of protracting the average idle time experienced by a processor awaiting work.

### 6.6.2 Experiments with Latency-Tolerance

To test Hypothesis 6.1, the workload described by Paragon program A.5 in Appendix A is executed for three different architectural configurations and for a range of grain sizes. The workload contains a combination of tail-recursive tasks, divide and conquer workloads, and simple tasks which perform primitive work only. Shared expressions and delayed evaluation are also present. These induce extra overheads when locality of reference is forsaken by an increase in the number of PEs.

By taking a basic Paragon program and using the granularity index to increase granularity, we effectively generate a larger program. The number of tasks in the program’s dynamic call-tree is independent of the grain size. If we had used a real program, the
effect of increasing grain size would be to reduce the number of tasks. In both cases, a similar reduction in the frequency with which tasks are created is experienced. By fixing the shape of the call-tree, we can experiment with workloads of progressively increased cost, but whose overheads due to task creation and message passing are independent of granularity. It is therefore straightforward to eliminate these overheads when the experimental results are analysed.

The PE/processor configurations are as follows:

1. One PE, one processor. This is used to characterise the fixed overheads experienced due to task creation and executing instructions other than work. No messages are passed and the processor never becomes idle.

2. Two PEs, two processors (one PE per processor). This is the simplest possible message-passing configuration without running multiple PEs on any one processor.

3. Two PEs, one processor. This is the simplest possible multiple-PE per processor configuration. Although the PEs are both on the same physical processor, they continue to communicate by messages passed via their local Ism and will suffer periods of idleness.

We are restricted to these configurations by the current implementation of DIGRESS local system managers. At present, the Ism's do not allow more PEs than two per physical processor, and they limit each physical processor to only one PE when more than one physical processor is employed. On the strength of the results obtained from these preliminary experiments, however, this situation is to be rectified.

Figure 6.16 illustrates the relative overheads measured for the above configurations (overheads are expressed as a ratio of absolute overhead relative to the time spent executing work instructions). When the overheads are fixed (remember that the workload contains the same instructions, irrespective of the granularity) then the relationship of the relative overhead, \( V\% \), to the workload granularity, \( G \), is given by \( V = 100vG^{-1} \), where \( v \) is the absolute cost of the fixed overhead. Taking logarithms,

\[
\log V = \log(100vG^{-1}) = \log 100v - \log G. \tag{6.3}
\]

Examining the log/log plot of Figure 6.16, it is evident that we obtain the behaviour characterised by Equation (6.3) both for the 1 PE/1 processor configuration and for the 2 PEs/1 processor configuration. In the latter case, the intercept on the y-axis is larger,
reflecting the approximately constant increase in overhead due to job scheduling by the operating system.

For the configuration of 2 PEs executing on 2 processors the gradient of the plot is shallower than for the other cases. This indicates that the absolute overhead is no longer fixed, but approximately related to the granularity by \( v = v_{2,2} G^m \) for some suitable value of \( m \). The relative overhead thus becomes:

\[
V_{2,2} = 100v_{2,2} G^{m-1}
\]

which gives the logarithmic expression,

\[
\log V_{2,2} = \log 100v_{2,2} - (1 - m) \log G.
\]  

(6.4)

The shallow, negative gradient observed in Figure 6.16 is obtained when \( 0 < m < 1 \) in Equation (6.4).

6.6.3 Conclusions and Further Work on Latency-Tolerance

For each of the above experiments, the average number of messages passed between PEs remains constant as the granularity of the workload increases. A constant-factor change in the relative cost of message passing will thus result in a corresponding change in the
y-intercept of the graph, leaving the gradient unchanged. Therefore, the shallow gradient of the two processor experiment cannot be explained by the expense of passing messages between physical processors.

When there is only one physical processor, the processor will spend very little of its time idle. In the 2 PEs/2 processors case, one of the processors will become idle every time its PE completes the execution of the last task in its task pool. Therefore, the gradients of Figure 6.16 strengthen the conviction that it is intolerance to latency which introduces overheads that are related to the granularity of the workload.

In these experiments, large, coarse-grain workloads benefit from the latency-tolerance afforded by executing multiple PEs on a single physical processor. The increased job scheduling overhead must therefore have been overcome, as predicted by Hypothesis 6.1. However, we have been able to measure only three, somewhat extreme cases. What of more general cases? We must attempt to predict what would happen if there were many physical processors, each running multiple PEs.

It might be argued that if the available parallelism is higher than can be exploited by the physical processors of the machine then it is unlikely that latency periods will occur due to lack of work. Furthermore, if work is not available then increasing the number of PEs can only be detrimental. The latter argument seems to be valid if the knowledge of available work across the whole machine is correct at all times. In real implementations, however, the cost of frequently updating the global knowledge is prohibitive [HG84, GH86] and there is a trade-off between the frequency of update and the effects of using out of date information.

Incorrect loading information can lead to requests being sent to processors which now have empty task pools. Currently, our PEs are programmed to hold on to requests for work received from remote PEs if (i) the local task pool is empty, but (ii) the recipient of the request has potential for generating new tasks. An alternative approach of sending an immediate denial leads to the problem of thrashing as requests and denials continue until the global load information is updated. At times when there really is too little work to keep all of the PEs busy, thrashing is particularly problematic without some method of keeping it in check. Whichever mechanisms are employed to avoid thrashing in the

\[ ^3 \text{Idleness can occur for 2 PEs/1 processor in the rare event that both PEs are executing work, but have empty task pools, and simultaneously have to fetch a remote synthetic closure from the other PE.} \]

\[ ^4 \text{Other mechanisms can also be employed to stop thrashing, such as the use of a back-off timer before a second message is resent.} \]
presence of bad load information, the average latency increases for idle PEs seeking work. In particular, the mechanism of holding on to requests definitely causes latency periods to be related to the rate at which new tasks are spawned and hence to the granularity of the workload.

We predict that in the presence of certain scheduling heuristics (such as those mentioned in the previous paragraph), the latency-tolerance obtained using just two PEs running on a single processor will extend to the more general case of two PEs running on each of many physical processors. Less optimism is expressed for more than two PEs per processor, but final conclusions are delayed until further experiments are possible.

It is felt that, when only a small number of workstations are available for computing the value of a large program, this technique for improving latency-tolerance will prove to be beneficial. The technique will be especially applicable to programs which exhibit little algorithmic parallelism, but rely instead on the implicit parallelism. For these programs there is greater potential for idle periods between limited bouts of work.

Finally, the technique of using multiple PEs per physical processor appears to be comparable with using a single PE per physical processor and giving each PE a number of separate task pools, as illustrated in Figure 6.17. Instead of relying on the operating system to schedule several PE jobs, each PE is responsible for managing its task pools independently. When one of the task pools becomes empty, the PE sends a message to another PE to obtain further work. The empty task pool cannot supply work to the PE but the remaining task pools can. The PE is therefore tolerant of the latency of sending the work-request message. This insight also removes the operating system's job scheduling overhead and improves the locality of reference because the same address space is used by the tasks extracted from all of the task pools on the PE.

On first inspection, maintaining multiple task pools might appear to reduce the ability of the scheduler to impose a total order on tasks because multiple pools imply a partial ordering. However, the scheduler has access to all of the task pools at the same time so can choose the most or least expensive of all tasks available to satisfy remote and local requests for work, respectively. The operation of merging the contents of task pools increases the cost of extracting a task, but this is offset by a reduction in cost of task insertion because there are fewer tasks per pool.

There is clearly much scope for further work in this area, requiring the re-design and re-implementation of the DIGRESS local system managers and the Paragon processing
6.7 Summary and Conclusions about Dynamic Task Management

Many experiments have been reported in this chapter exploring attempts to reduce run-time overheads. The central theme of the research was inspired by Maheshwari's hypothesis that tasks for remote execution ought to be spawned in a definite order [Mah90, Mah92]. It has been clear from the experiments that this hypothesis is justified. However, it is equally clear that the intention to schedule workloads in a definite order will be thwarted when there is a low correlation between predictions about time-costs and actual time-costs observed at run-time. If the correlation cannot be maintained at a reasonable level then there is no advantage in using time-cost information in preference to scheduling tasks randomly. Two main causes were identified for limiting the accuracy of time-cost predictions:

1. lazy evaluation,
2. run-time input.

Lazy evaluation renders asymptotic time-complexity measures difficult to use because the value of input parameters cannot always be determined in advance of spawning a task. If
absolute time-costs must be predicted correctly, this presents a serious limitation. Using Paragon workloads we avoid the problem by characterising costs with random variables. This is similar to using a profiler to make a study of the typical inputs to functions during previous program runs, and using those to guide predictions for future executions.

Instability in workload costs was demonstrated with the aid of simulated run-time input. Even when time-cost analysis is improved to take account of sharing, we showed that instability in observed time-costs can reduce the effectiveness of scheduling based on mean time-cost analysis. The problem with the unsupported use of mean time-costs is that the risk of an incorrect prediction is not taken into account when a decision is made. Wegbreit's early attempts at time-complexity analysis [Weg75] acknowledged the characterisation of a time-cost by the 4-tuple \((\min, \max, \text{mean}, \text{variance})\). Three of these components give a measure of risk. However, it is not necessary to make such complex characterisations because our concern is simply to identify the likelihood of correct comparisons. We can measure this directly, by logging comparisons made by task pool managers and profiling the cost of the associated tasks. As we already consider profiling as a pre-requisite for characterising time-costs then this seems to be a sensible progression. The results obtained using risk analysis have been encouraging.

It is possible for the input data of a program to change so dramatically that earlier profiles have no relevance to the current execution. If this happens, any scheduling algorithm which requires profile data from previous executions is liable to fail to reduce run-time overheads. The use of profiled information for scheduling algorithms should therefore be restricted to programs which follow similar trends on different sets of data. For example, one of the reasons for developing DIGRESS was the existence of a complex data processing problem related to geological surveys. An analysis program will be executed on survey data which describes deep core samples taken off-shore around the coasts of the world. When many core samples are analysed from the same area, it is reasonable to expect that the data will exhibit common trends. Therefore, it may be possible to reduce program overheads for the majority of the analyses by profiling the initial executions. When the computations are expensive this may well reduce overheads sufficiently to make the meta-analyses worthwhile. Paragon has proved to be a good vehicle for rapidly prototyping test-beds for new scheduling schemes. Initial hypotheses can be strengthened or weakened using Paragon workloads, and areas of contention can be quickly identified. Using the information derived from Paragon experiments as a
guide, we can start to plan further experiments based on large, real functional programs.

Finally, in Section 6.6 a new technique was described for improving latency-tolerance in the DIGRESS architecture. The initial experiments are promising and it is felt that, of the work presented in this chapter, the latency-tolerance results are the most generally applicable.
Chapter 7

Conclusions and Further Work

In this chapter the contributions of the thesis are summarised and conclusions are drawn from the work presented. Relationships between this and work by others are described and opportunities for further work are discussed.

7.1 Summary and Conclusions

The work presented in this thesis makes five main contributions to functional programming research:

1. A novel approach to constructing experimental workloads has been developed by synthesising the run-time behaviour of functional programs.

2. A time-cost analysis has been devised for the synthetic workloads. The analysis takes shared computations into account and computes the expected mean time required to execute individual tasks.

3. Extensions have been provided to existing methods for dynamically managing lazy workloads on loosely-coupled architectures.

4. Experimental data is now available detailing experience with different workload granularities. From this, it is possible to deduce suitable workload granularities for executing real functional programs in parallel on the DIGRESS architecture.

5. A tool for profiling higher-order, lazy functional programs has been developed. This improves the quality of information that is available to analyse the run-time behaviour of functional programs.
These areas of research are summarised below and conclusions are drawn.

7.1.1 Modelling Functional Workloads

Traditionally, programs are partitioned at compile-time into a number of sequential threads of execution, called tasks. Task granularity is a crucial factor governing the efficiency of the program when it is executed in parallel. At run-time the program is distributed among the parallel processors of the reduction machine by dynamic scheduling and load balancing algorithms, either embedded in the code for the tasks or supplied externally by a task management package.

There is a potential conflict when attempting to measure the behaviour of real programs because it may be difficult to determine whether an inefficiency is due to bad partitioning or is caused by an inefficient task management scheme. The problem is compounded by the large degree of transformation that functional programs typically undergo when they are compiled. The final run-time program may bear little or no resemblance to the original source code and the resulting stimulus can be difficult to predict. This obfuscates the construction of test workloads and we have commented on the profound shortage of well-structured, well-understood, standard benchmarks.

In an attempt to alleviate these problems we have designed an experimental language which facilitates the rapid construction of synthetic workloads. The run-time stimulus and a granularity index are specified explicitly by the experimenter and the compiler performs no transformations on the workload whatsoever.

One of the most useful features of synthesising functional workloads is that the results are scalable. The same experiment can be repeated for a single workload, over a range of granularity coefficients. This helps to identify the degree to which a result is dependent on the average size of the tasks. An important attribute of scalable workloads is that the amount of communication and task manipulation overhead remains constant when the number of processing elements is fixed. It is therefore straightforward to account for these overheads when the results of experiments are analysed. The overheads can be varied, of course, by reconstructing the workload description. This corresponds to re-writing a real functional program or redesigning its compiler. The synthetic workloads therefore provide a more malleable description of experimental stimuli than those afforded by real functional programs.

The main alternatives to modelling functional stimuli are using traces of real program
executions [Des89] and using real programs in their own right. From experience gained with model workloads, it is often necessary to construct very specific stimuli to test an hypothesis. Therefore, for real functional programs to prove useful as stimuli for systems under test, it will be necessary to construct a large library of standard benchmarks whose resultant workloads are well understood. Whenever a compiler is updated, the workloads will change and must be calibrated afresh. Currently, the nofib benchmark suite [Par93] is under active development at the University of Glasgow. It is hoped that, eventually, it will be as straightforward to construct test workloads by combining programs from benchmark suites such as this, as it is to use artificial stimuli. In the interim, the synthetic functional workloads provided by Paragon have proved to be a useful source of predictable, experimental stimuli.

7.1.2 Time-Cost Analysis

A great deal of work has taken place in the field of time-complexity analysis (e.g. [Le 85, Le 88, Weg75, Ros86, Ros89, BH89, DJG92, San90, Wad88]) but few serious attempts (with the notable exception of [Mah90]) have been made to apply the results to practical problems such as dynamic task management. For practical use we really need to predict the physical length of time that a task will take to execute. For this reason, we adopt the term time-cost in preference to the more conventional term, time-complexity. Measures of time-complexity typically do not require knowledge of the input to a program, but merely report an asymptotic time-complexity function. We ought to be more precise than this but, given that much of the input to a program is presented at run-time only, a compile-time analysis will often fail to provide the correct results. Nevertheless, compile-time analysis can at least give us a basis for making scheduling decisions.

The key contribution of the time-cost analysis presented in Chapter 5 is that laziness is dealt with in a sensible way. Previous analyses have made little or no attempt to account for shared computations. For example, the work of Sands [San90] deals with call-by-name semantics but does not detect where shared expressions are evaluated. When a shared expression can be evaluated by any one of a number of different tasks, our analysis makes a statistical approximation and shares the cost of executing the shared expression between the tasks. This gives an estimate of the mean time required to execute individual tasks, making the analysis directly applicable to dynamic scheduling algorithms. The Paragon time-cost analysis adopts an approach similar to [San90] in that programs undergo cost-
preserving transformations, resulting in new programs that are simpler to analyse.

The time-cost analysis is defined over the syntax of the Paragon synthetic workload language and is therefore simpler than an equivalent analysis for a full, functional programming language. Recursive functional programs lead to recursive complexity equations, the solution of which usually require fixed-point iteration [AH87, HH91, PJC87, MH87]. To describe non-flat data objects such as lists, potentially infinite domains are constructed [Wad87b]. These have to be approximated by finite domains to ensure that fixed-point solutions terminate. The advantage of using the synthetic workload language is that the above complications are avoided. Consequently, the effort to determine the utility of the sharing analysis was minimised.

The information provided by time-cost analysis with sharing taken into account has resulted in better dynamic scheduling than that obtained when sharing was not taken into account. However, this relies on relative costs within workloads remaining stable with respect to changes in run-time input. As workload instability increases, the utility of all time-cost analyses is reduced. If a time-cost analysis is to be used for scheduling purposes then the improved results demonstrated in this thesis suggest that similar improvements can be achieved by augmenting the analyses for real functional languages with a sharing analysis. Further experiments will be required to determine the extent to which instability causes problems with real functional programs.

7.1.3 Dynamic Task Management

Synthetic workloads have been used to experiment with techniques for managing the execution of tasks dynamically. We have been able to identify the conditions in which various compile-time scheduling decisions are likely to yield an inefficient distribution of program tasks and have shown methods of improving existing techniques.

The work was based on an idea presented in Maheshwari's thesis [Mah90]. Maheshwari states that if tasks are spawned in a definite order then lower overheads can be achieved. We have verified this proposition but draw slightly different conclusions about the underlying reason for the result. Maheshwari believes that the improvement is mainly due to the careful management of the timing graph shown in Figure 6.4 of Chapter 6, and the subsequent reduction in the time spent by a parent task awaiting the results of its children. He also assumes that there will always be sufficient processing capacity in the parallel machine to immediately absorb all of the work made available for parallel eval-
uation. We have found that for demand scheduled, coarse grained computations, there will almost always be some idle time either by the parent, or by child tasks because it is extremely difficult to ensure that the difference in costs between successively spawned tasks is exactly the amount required to overcome idleness. The problem is further compounded by instabilities in the workloads. The result is therefore more likely due to the fact that exporting the most expensive tasks makes more efficient use of communications. Also, when there is insufficient processing capacity for all tasks to be exported then it is better for several small tasks to be evaluated sequentially by the processing element which spawned them than for the small tasks to be exported leaving larger tasks behind. The former case is more likely to result in more evenly balanced processing elements.

In general, the ability of processing elements to context switch between tasks when task-suspension occurs means that it is insufficient to rely on compile-time decisions. By improving the scheme so that scheduling decisions are taken at run-time, lower overheads are observed in the presence of context switching. Despite this improvement, there remains a fundamental problem when time-cost predictions are used to schedule tasks: if ever the observed costs are such that the ordering of tasks differs from that predicted, then the reduction in overheads will be attenuated.

The use of profilers to characterise workloads and obtain estimates for the relative costs of tasks can also be improved. In addition to determining mean time-costs, the comparisons made by the scheduling algorithm are logged and the observed time-costs used retrospectively to determine the correct outcome of each comparison. In this way, a stochastic learning automaton can be constructed to guide future comparisons according to past results. Mean time-costs alone can be poor predictors when the mean is biased by unstable run-time input or lazy evaluation.

In conclusion, making predictions completely immune to instabilities in the time-cost of individual tasks appears to be an insurmountable problem. However, successive improvements in the heuristics used to predict time-costs can take us some way to achieving Maheshwari’s original goal of exporting tasks in an order which results in lower run-time overheads. For coarse-grain architectures such as DIGRESS, even the smallest reductions in overheads are welcome.
7.1.4 Experiments with Granularity

Many of the experiments presented in Chapter 6 were conducted over a range of granularities. The granularity index multiplies the work units specified in Paragon WORK instructions. We can define granularity to be the ratio of primitive processing time to messages passed. The average granularity of a workload is therefore determined by $g \bar{w}/m$ where $g$ is the granularity index, $\bar{w}$ expresses the mean work units per WORK instruction in the workload description, and $m$ specifies the average number of messages passed per work instruction. Using this simple formula, the minimum acceptable granularity for workloads on DIGRESS can be calculated by inspecting the experimental results. Throughout Chapter 6, overheads are plotted relative to the primitive work performed. The curves therefore approximate a $1/x$ relationship if the overheads are fixed and the workload granularity is variable. When the grain size is small the relative overhead rises quickly with small reductions in granularity. As the grain size is increased the relationship begins to approach a straight line. The point at which this starts will be taken to be the minimum acceptable granularity.

From the experimental data gathered, we can conclude that the minimum granularity is achieved when there are approximately 38 work units per message passed. All of the experiments were executed on DECstation 3100 workstations which require $\approx 1.73 \times 10^{-4}$ cpu-seconds to execute a single work unit. Therefore, for this hardware, we have found that an average of about 6.6 cpu-milliseconds of work is required between messages.

7.1.5 A Profiling Tool for Higher-Order, Lazy Functional Programming Languages

Functional languages are still largely undiscovered as serious tools for the construction of real programs. One reason for this is the lack of development utilities such as debuggers and profilers [Cla93]. In this thesis there is a genuine need to analyse the run-time behaviour of real functional programs to provide rough guidelines for constructing synthetic workloads.

The profiling tool whose implementation is described in Appendix C is a new method of tracing the behaviour of higher-order, lazy functional programs. Clayman suggests in [Cla93] that the results of a profiled program execution should be reported in such a way that they correspond closely to the text of the original source program. This may
seem obvious, but many functional language implementors are used to thinking in terms of underlying implementation issues rather than in terms of the high-level source code. The difference is seen mainly when lazy evaluation causes an expression to be evaluated by a function other than that in which it was defined.

The profiler is able to record results using both implementor-friendly and programmer-friendly styles. This provides a much greater insight into what really happens during the execution of a lazy functional program and is particularly helpful for constructing sensible synthetic programs when the effects of laziness on scheduling is being studied.

7.2 Summary of Related Work

Our work has focused on two major research areas: modelling lazy functional workloads, and using the resulting workloads as experimental stimuli for experimenting with dynamic scheduling techniques. Although we could find no previous attempts to model functional language workloads in the manner described in Chapter 3, the idea of synthesising imperative workloads is not new. In Section 2.2.4 we described the work of Kunz [Kun91] who uses executable synthetic workloads to experiment with job scheduling in the Unix operating system. We also discussed the work of Phillips, Skilling, Candlin, et al at the University of Edinburgh whose research uses synthetic workloads to characterise the run-time behaviour of parallel imperative programs [CFPS92, CPS92, CGS89, PS91].

Within the functional language implementation community, it has been common to use real functional programs as workloads for experimentation [HPJ91, LV91, RW92, BH92, CC91, AHPJT91]. However, some work on the use of trace-based workloads [HL84, Sve90] has been carried out at the University of Glasgow by Deschner [Des89] (see Section 2.2.2). Deschner's work resembles our own in that experiments are conducted using abstractions of real functional programs. Deschner uses the workload abstraction to drive a simulated graph reduction system. Perturbations are made to the way that the reduction system reacts with the workload, but the workload remains fixed. Using Paragon workloads, we drive real reduction hardware and can make perturbations to both the reduction system and to the workloads themselves without having to alter any functional language source code.

The closest work to the dynamic scheduling research reported here is that of Maheshwari [Mah90, Mah92]. Maheshwari explored the use of time-complexity analysis for
deriving information that would be useful in scheduling parallel functional programs. He restricted the scheduling activities to compile-time and used a measure of asymptotic time-complexity. In Chapter 6 we demonstrated limitations in the Maheshwari's original ideas and showed how the limitations could be overcome in many cases by improving time-cost analysis and using the information at run-time. Related scheduling techniques resulting from the work of Hammond and Peyton Jones [HPJ92, HPJ91], Peyton Jones and Clack [PJCS89], Sargeant [RS87, Sar87], and Hudak and Goldberg [Gol88, Hud84, HG84, HG85], among others, are discussed in Section 6.1. Work related to the time-cost analysis performed in Chapter 5 is discussed in Section 7.1.2.

Finally, the functional language profiling tool whose implementation details are presented in Appendix C is one of a number of tools currently under development. The related work of Runciman and Wakeling [RW90, RW92], Sansom and Peyton Jones [SPJ92], and of Appel, Duba, and MacQueen [ADM88] is described in Section 1.1.4.

7.3 Further Work

The work presented herein forms one part of a larger, ongoing project dealing with the parallel reduction of functional programs using loosely-coupled workstations. Much of the work reported in this thesis is still to be incorporated into the full system, thereby providing the foundation for a significant amount of further work. Earlier chapters have already alluded to some of the investigations that are still to be completed. Below, the previous discussions are consolidated and expanded to give an overall picture of the direction in which the research is leading.

7.3.1 Improvements to Paragon

During this period of research, we have found that the Paragon workload description language has proved to be a convenient vehicle for experimenting with new scheduling schemes. In its present form, however, Paragon is lacking in some minor respects and we believe that further development of the language will bring increased benefits.

A number of possible extensions were mentioned in Section 3.4 such as an extended instruction set to allow the specification of hardware constraints. Another extension is that of modelling structured data objects. In Section 4.6.3, an approximation for a data-parallel computation was presented. A more accurate representation was not possible...
because Paragon does not currently possess any method for representing data structures and their corresponding properties of synchronisation. Further research is needed to develop the extra Paragon instructions to model both data-dependent synchronisation and to characterise the associated memory usage. The work of Runciman and Wakeling [RW92] has shown the significance of studying memory activity at run-time, and Paragon remains incomplete while it is unable to model this aspect of workloads.

Many of our experiments have been concerned with overcoming instabilities due to run-time inputs and lazy evaluation. It has been possible to model this by generating random variables that affect the locality in which shared expressions are evaluated. Future work on scheduling problems would benefit from the ability for Paragon workloads to accept real run-time inputs. The ability of the time-cost analyser to predict the run-time stimulus is therefore restricted, thus bringing Paragon further in line with real functional programs. The exercise would be a good stepping stone towards the adaptation of the analyses and scheduling algorithms for real functional workloads.

7.3.2 Time-Cost Analysis

The time-cost analysis developed for Paragon has brought rewards when used to make dynamic scheduling decisions. However, the rewards are restricted to certain classes of workloads. The success of the technique is most acutely affected by the instability in actual time-costs with respect to those predicted at compile-time.

Before a decision is made to develop an equivalent analysis for real functional programming languages, further information is required. Primarily, it is necessary to determine for real workloads, how often, and to what extent, instability will attenuate the utility of the compile-time predictions. Such a study will only become meaningful, however, when it is conducted over a comprehensive library of large functional programs. When this is completed, it is hoped that at least for some types of program, instability will prove to be of sufficiently small effect to suggest that improved time-cost analyses will yield measurable gains in dynamic scheduling behaviour.

The analysis presented in Chapter 5 is defined for the subset of Paragon programs in which synthetic closures are not returned as results of tasks, and for which the evaluation of one synthetic closure does not imply to the evaluation of a second. Work has begun on removing these restrictions and further work is to be completed before a full analysis is achieved. The proposed amendments to the language are also likely to precipitate the
need for further enhancements to the analysis.

7.3.3 Lexical Profiling Tool

The lexical profiling implementation of Appendix C is given in terms of a straightforward interpretive graph reduction engine. However, the techniques are relatively general and will require a minimal amount of modification for use with compiled graph reduction.

Whenever graph cells are constructed by compiled code, they can be marked with the profiler's 'colouring' information in much the same way that we mark the interpreted graph. The biggest difference between compiled and interpreted graph reduction is that compiled code attempts to avoid constructing graph whenever possible, relying instead on the computation of intermediate results on the stack [FW87, BPJR88, PJS89, AJ89b].

The colouring information must therefore be placed into the object code and copied onto the stack whenever data is held there. If it proves to be common for consecutive entries on the stack to be marked with the same information then it may be possible to optimise the storage of profile data on the stack by the judicious placement of markers such as those employed by [FW87] to trigger graph updates.

It is proposed that the compiled-code version of the profiler is developed for the $\langle \nu, G \rangle$-machine [AJ89b], for which we already have an in-house compiler [Par90]. It will be a relatively straightforward exercise to update this compiler to cope with profiling information.

7.3.4 Extending Scheduling Techniques to DIGRESS

The experiments conducted during this research have taken place on an early prototype of the DIGRESS communications subsystem [GB91]. The next logical step is to extend the scheduling techniques developed for Paragon workloads to the implementation of the parallel Four Stroke Reduction Engine [CPJ86] currently being built on DIGRESS and later to the proposed implementation of the $\langle \nu, G \rangle$-machine [AJ89b]. A parallel version of the lexical profiling tool is also being constructed for DIGRESS. It is expected that the extensions to a full parallel implementation of the Haskell language and the subsequent experimentation with dynamic scheduling based on the work of this thesis, will constitute a major research project.

The preliminary latency-tolerance experiments in Section 6.6 have indicated that latency-tolerant computations may be possible by increasing the number of processing
elements executing per physical processor. The current implementation of the DIGRESS local system managers (Ism's) [GB91] does not allow this in the general case. This facility is to be added to the Ism's and the experiments extended for more general PE/processor configurations. The Paragon processing elements are also to be redesigned to incorporate multiple task pools per PE. It is hoped that, on completion of more comprehensive experiments, the new technique for achieving latency-tolerance can be carried forward to the full functional language implementations on the DIGRESS architecture.
Appendix A

Experimental Paragon Programs

This appendix lists the Paragon programs used by the experiments described in this thesis. All Paragon workloads are pre-processed by cpp, the C language pre-processor, thus allowing macros to be defined with #define.

A.1 Program with task suspensions

```c
main { // most expensive
    Declare shared_a {
        Work 30000
    }
    Declare delayed_b {
        Work 100
    }
    Spawn a -> A_Task <shared_a>
    Spawn b -> B_Task <delayed_b>
    Work 500
    Eval shared_a
    Work 1000
    Block a b
}

A_Task { // 2nd most expensive
    Work 5000
    Eval $1
    Spawn a1 -> A_Child
    Work 5000
    Block a1
}

A_Child { // 4th most expensive
    Work 45000
}

B_Task { // 3rd most expensive
    Spawn b1 -> B_Child
    ...
    Spawn b20 -> B_Child
    Eval $1
    Work 10000
    Block b1 ... b20
}

B_Child { // least expensive
    Work 2500
}

A.2 Program with unshared, non-strict arguments

#define child(w) {
    Eval $1 $2 
    Spawn ex -> expensive_sub 
    Spawn ch -> cheap_sub 
    Work w 
    Block ex ch 
}

main {
    Declare arg1 { Work 750 }
    Declare arg2 { Work 750 }
    Declare arg3 { Work 750 }
}
A.3 Program with shared, non-strict arguments

#define child(w) { \
    Eval $1 $2 \ 
    Spawn ex -> expensive_sub \ 
    Spawn ch -> cheap_sub \ 
    Work w \ 
    Block ex ch \ 
}
number of sub-tasks are spawned. The declared sequences of the iteration template are shared operations performed on the state items and each of these may be required by the sub-processes according to the influence of some run-time input. This choice is represented by the multiple options for spawning sub-processes.

```c
#ifndef x
#define x 10
#endif

#define tailcalls 15
#define light 1000
#define middl 10000
#define heavy 10000
#define local 10000
#define results 1000

iteration {
    Declare sh1lt { Work light }
    Declare sh1md { Work middl }
    Declare sh1hv { Work heavy }
    Declare sh2lt { Work light }
    Declare sh2md { Work middl }
    Declare sh3lt { Work light }
    Declare sh3md { Work middl }
    Declare sh3hv { Work heavy }

    Spawn child1
        -> processX <sh1lt,sh1md>
        | processY <sh1lt,sh1md>
        | processY <sh2lt,sh2md>
        | processX <sh3lt,sh3md>

    Spawn child2
        -> processY <sh1lt,sh1hv> (x%)
        | processY <sh1lt,sh1md>
        | processY <sh2lt,sh2md>
        | processY <sh3lt,sh3hv> (x%)
        | processY <sh3lt,sh3md>

    Spawn child3 -> processZ

    Work local

    Block child1 child2 child3
    Work results

    // These are the tasks to monitor
    Spawn test1
        -> process1 <sh1lt,sh1md,sh1hv>
    Spawn test2
        -> process2 <sh2lt,sh2md>
    Spawn test3
        -> process3 <sh3lt,sh3md,sh3hv>

    Work local

    Block test1 test2 test3
    Work results

    Tail tailcalls
}

#define process { \
    Eval $1 $2 $3 \ 
    Work local \ 
}

processX process
processY process
processZ process
process1 process
process2 process
process3 process
```

A.5 A large, general purpose, workload

```c
#ifndef tailcalls
#define tailcalls 10
#endif

main {
    Declare exp1 {
        Spawn a -> subproc1
        Spawn b -> subproc1
        Work 20
        Block a b
    }

    Declare exp2 { Work [0..30] }
    Declare exp3 { Work [10..40] }
    Declare exp4 { Work [20..50] }
    Declare exp5 { Work [30..45] }
    Spawn a -> process1 <exp2>
        | process1b <exp3>
    Spawn b -> process2
    Spawn c -> process3
    Spawn d -> process4a <exp1>
```
div_conq_1 {
    Spawn l_arg1 -> process2
    Spawn l_arg2 -> process9
    Spawn l_arg3 -> process11
    Block l_arg1 l_arg2 l_arg3
    Work 50
    Spawn left -> div_conq_1 (45%)
        | leaf1
    Spawn r_arg1 -> process4a (20%)
        | process4b (20%)
        | process4c
    Spawn r_arg2 -> process12
    Block r_arg1 r_arg2
    Work 50
    Spawn right -> div_conq_1 (45%)
        | leaf1
    Block left right
    Work 100
}

leaf1 {
    Work 75
}

div_conq_2 {
    Work 50
    Spawn left -> div_conq_2 (40%)
        | leaf2
    Spawn r_arg -> process6
    Work 50
}

process1a {
    Spawn a -> subproc1
    Spawn b -> subproc2
    Spawn c -> subproc3
    Block a b c (50%)
    Spawn d -> subproc4
    Eval $1 (50%)
    Block a b c d
    Work 50
}

process1b {
    Eval $1
    Work 50
}

process2 { 
    Work 100
}

process3 {
    Declare exp1 { Work 50 }
    Declare exp2 { Work 50 }
    Declare exp3 { Work [10..100] }
    Spawn a -> subproc2 <exp1,exp2>
        | subproc3 <exp1,exp3>
    Work 60
    Eval exp2 exp3 | exp2
    Block a
    Spawn r -> process3 (30%)
        | terminate
    Block r
    Work [10..20]
}

process4a {
    Eval $1 (50%)
    Work 20
}
process4b {
    Work 50
    Eval $1 (80%)
}

process4c {
    Eval $1
    Work 30
}

process5 {
    Work 100
}

process6 {
    Work 20
    Eval $1 (80%)
}

process7a {
    Eval $1
    Work 20
}

process7b {
    Work [20..40]
    Eval $1 (20%)
    Work 10
}

process7c {
    Work [30..50]
    Eval $1
}

process8 {
    Work 250
}

process9 {
    Declare exp1 { Work 20 }
    Declare exp2 { Work 30 }
    Spawn a -> subproc2 <exp1,exp2>
    Spawn b -> subproc2 <exp1,exp2>
    Spawn c -> subproc2 <exp1,exp2>
    Spawn d -> subproc2 <exp1,exp2>
    Work 30
    Block a b c d
}

process10 {
    Work [10..100]
}

process11 {
    Declare exp1 { Work 5 }
    Declare exp2 { Work 30 }
    Work 10
    Spawn a -> subproc2 <exp1>
      | subproc3 <exp1,exp2>
    Spawn b -> subproc2 <exp2>
      | subproc4
    Work 30
    Block a b (50%)
    Spawn r -> process11 (45%)
      | terminate
    Block a b r
    Work 5
}

process12 {
    Work [10..50]
    Work [20..30]
}

subproc1 {
    Work 30
}

subproc2 {
    Work 30
    Eval $1 (10%) | $2 (10%)
}

subproc3 {
    Work 10
    Eval $1 (10%) | $2 (10%)
}

subproc4 {
    Work 30
    terminate {
      Work [10..50]
    }

}
A.6 Program to test scheduling based on time-cost analysis

```c
#ifndef tailcalls
#define tailcalls 20
#endif

main {
    Spawn a -> A
    Spawn b -> B
    Spawn c -> C
    Spawn d -> D
    Spawn e -> E
    Spawn f -> F
    Work 50
    Block a b c d e f
    Work 50
    Tail tailcalls
}

// define a generic template:
//
#define generic(name,shared,fixed) \\
    name {
        Declare shared { Work shared } \\
        Declare share2 { Work shared } \\
        Work [125..375] \\
        Eval share1 (40%) | share2 (30%)\ \\
        Work [250..750] \\
        Eval share1 (30%) | share2 (25%)\ \\
        Work [375..625] \\
        Eval share1 (20%) | share2 (75%)\ \\
        Work [250..750] \\
        Eval share1 (40%) | share2 \ \\
        Work [125..375] \\
        Eval share1 | share2 (30%) \ \\
        Work fixed \ 
        Eval share1 | share2 \\
    }

// use the generic template to build 6 separate task templates: //
//
generic (A, 250, 5500) 
generic (B, 250, 5000) 
generic (C, 250, 4500) 
generic (D, 250, 4000) 
generic (E, 1250, 1000) 
generic (F, 1250, 500) 
```

A.7 Program to simulate instability for exploring risk aversion

```c
#ifndef tailcalls
#define tailcalls 20
#endif

A {
    Declare closure1 { Work 4500 }
    Declare closure2 { Work 5000 }
    Declare closure3 { Work 2000 }
    Declare closure4 { Work 500 }
    Declare closure5 { Work 500 }
    Spawn a -> B
    Spawn b -> B
    Spawn c -> C <closure4, closure1>
    Spawn d -> D <closure3>
    Spawn e -> E <closure5, closure2>
    Eval closure4 closure2
    Eval closure5 closure1 (factor%)
    Block a b
    Work 100
    Block c d e
    Work 100
    Tail tailcalls
}

C {
    Work 500
    Eval $1 $2
    Work 1000
}

D {
    Work 2500
    Eval $1
    Work 500
}
```
A.8 Program to demonstrate loss of parallelism due to throttling

```c
#define children(a,b,c,d) \
  Work 100 \ 
  Spawn a -> child \ 
  Spawn b -> child \ 
  Spawn c -> child \ 
  Spawn d -> child \ 
  Block a b c d (20%) \ 
  Work [100..1000]

#ifdef DIVIDE_AND_CONQUER

  child { Work [1000..10000] }

  # define recursion \ 
  Work [100..1000] \ 
  Spawn l -> producer (48%) \ 
  Spawn r -> producer (48%) \ 
  Block l r \ 
  Work 4000

#endif

#ifdef TAIL_RECURSIVE

  child { Work [5000..20000] }

  # define recursion \ 
  Work 10000 \ 
  Tail 6

#endif

producer {
  children ( c1, c2, c3, c4)
  children ( c5, c6, c7, c8)
  children ( c9, c10, c11, c12)
  children (c13, c14, c15, c16)
  children (c17, c18, c19, c20)
```
Appendix B

Experimental Results

The experiments and results presented in this appendix illustrate points made in the main chapters of this thesis.

B.1 Excessive Throttling can be Counter-productive

In [PJCS89] it is mentioned that throttling can lead to a loss of parallelism which cannot be recovered once the work is committed to sequential evaluation. Here we demonstrate this effect using Paragon Program A.8 from Appendix A. By conditional compilation, the program produces two distinct workloads, one tail-recursive and the other following a divide and conquer pattern. The results of throttling the workloads are shown in Figure B.1. It is clear that when parallelism is throttled too severely, even the divide and conquer algorithm can become inefficient.

![Figure B.1: Loss of parallelism when the throttling threshold is too low.](image)

Relative run-time overhead (%)
Table B.1: Profile results for quicksort.

<table>
<thead>
<tr>
<th>Function:</th>
<th>qsort</th>
<th>qsort</th>
<th>qsort</th>
<th>qsort</th>
<th>qsort</th>
<th>++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Called by:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expt.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1.06</td>
<td>200</td>
<td>1.84</td>
<td>346</td>
<td>2.28</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1.08</td>
<td>200</td>
<td>1.98</td>
<td>425</td>
<td>4.18</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.74</td>
<td>200</td>
<td>1.68</td>
<td>388</td>
<td>2.54</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1.00</td>
<td>200</td>
<td>2.16</td>
<td>418</td>
<td>3.88</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0.84</td>
<td>200</td>
<td>1.36</td>
<td>372</td>
<td>2.52</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0.74</td>
<td>200</td>
<td>1.56</td>
<td>372</td>
<td>2.50</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0.88</td>
<td>200</td>
<td>1.82</td>
<td>332</td>
<td>1.96</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0.98</td>
<td>200</td>
<td>2.26</td>
<td>472</td>
<td>4.56</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1.00</td>
<td>200</td>
<td>1.56</td>
<td>399</td>
<td>2.84</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1.14</td>
<td>200</td>
<td>2.00</td>
<td>449</td>
<td>4.56</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0.86</td>
<td>200</td>
<td>1.58</td>
<td>341</td>
<td>2.34</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1.04</td>
<td>200</td>
<td>1.58</td>
<td>341</td>
<td>2.34</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1.02</td>
<td>200</td>
<td>2.10</td>
<td>412</td>
<td>4.12</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0.92</td>
<td>200</td>
<td>2.10</td>
<td>444</td>
<td>4.42</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1.22</td>
<td>200</td>
<td>2.58</td>
<td>447</td>
<td>4.52</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0.84</td>
<td>200</td>
<td>1.50</td>
<td>408</td>
<td>3.06</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>0.92</td>
<td>200</td>
<td>2.28</td>
<td>476</td>
<td>4.78</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>0.98</td>
<td>200</td>
<td>2.28</td>
<td>476</td>
<td>4.78</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>0.84</td>
<td>200</td>
<td>2.08</td>
<td>383</td>
<td>3.70</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>1.28</td>
<td>200</td>
<td>2.36</td>
<td>379</td>
<td>3.24</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>0.92</td>
<td>200</td>
<td>2.08</td>
<td>407</td>
<td>4.88</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>0.54</td>
<td>200</td>
<td>1.65</td>
<td>434</td>
<td>2.88</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>1.04</td>
<td>200</td>
<td>2.06</td>
<td>368</td>
<td>3.84</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>1.04</td>
<td>200</td>
<td>2.10</td>
<td>401</td>
<td>3.70</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>0.84</td>
<td>200</td>
<td>2.14</td>
<td>342</td>
<td>2.96</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>1.18</td>
<td>200</td>
<td>1.54</td>
<td>377</td>
<td>2.66</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>1.28</td>
<td>200</td>
<td>2.26</td>
<td>492</td>
<td>4.94</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>1.28</td>
<td>200</td>
<td>2.08</td>
<td>383</td>
<td>3.40</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>0.74</td>
<td>200</td>
<td>1.60</td>
<td>468</td>
<td>3.38</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>0.74</td>
<td>200</td>
<td>1.70</td>
<td>455</td>
<td>3.30</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>0.74</td>
<td>200</td>
<td>1.70</td>
<td>518</td>
<td>3.96</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>0.84</td>
<td>200</td>
<td>1.66</td>
<td>336</td>
<td>2.18</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0.80</td>
<td>200</td>
<td>1.52</td>
<td>392</td>
<td>2.78</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>0.76</td>
<td>200</td>
<td>1.78</td>
<td>510</td>
<td>3.72</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>0.92</td>
<td>200</td>
<td>1.78</td>
<td>510</td>
<td>3.72</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>0.74</td>
<td>200</td>
<td>1.46</td>
<td>510</td>
<td>4.02</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>0.90</td>
<td>200</td>
<td>1.72</td>
<td>399</td>
<td>2.74</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>0.70</td>
<td>200</td>
<td>1.74</td>
<td>347</td>
<td>2.32</td>
</tr>
<tr>
<td>38</td>
<td>1</td>
<td>0.60</td>
<td>200</td>
<td>1.42</td>
<td>308</td>
<td>1.90</td>
</tr>
<tr>
<td>39</td>
<td>1</td>
<td>1.14</td>
<td>200</td>
<td>2.16</td>
<td>368</td>
<td>3.30</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>0.94</td>
<td>200</td>
<td>2.20</td>
<td>351</td>
<td>2.88</td>
</tr>
<tr>
<td>41</td>
<td>1</td>
<td>0.92</td>
<td>200</td>
<td>2.00</td>
<td>428</td>
<td>4.02</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
<td>1.12</td>
<td>200</td>
<td>2.22</td>
<td>422</td>
<td>4.00</td>
</tr>
<tr>
<td>43</td>
<td>1</td>
<td>0.74</td>
<td>200</td>
<td>1.46</td>
<td>347</td>
<td>2.30</td>
</tr>
<tr>
<td>44</td>
<td>1</td>
<td>1.02</td>
<td>200</td>
<td>2.18</td>
<td>473</td>
<td>4.68</td>
</tr>
<tr>
<td>45</td>
<td>1</td>
<td>0.64</td>
<td>200</td>
<td>1.60</td>
<td>476</td>
<td>3.62</td>
</tr>
<tr>
<td>46</td>
<td>1</td>
<td>0.78</td>
<td>200</td>
<td>1.60</td>
<td>356</td>
<td>2.28</td>
</tr>
<tr>
<td>47</td>
<td>1</td>
<td>1.04</td>
<td>200</td>
<td>1.80</td>
<td>426</td>
<td>4.30</td>
</tr>
<tr>
<td>48</td>
<td>1</td>
<td>0.88</td>
<td>200</td>
<td>1.54</td>
<td>350</td>
<td>2.26</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
<td>0.88</td>
<td>200</td>
<td>1.82</td>
<td>371</td>
<td>2.58</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>1.10</td>
<td>200</td>
<td>2.14</td>
<td>431</td>
<td>4.44</td>
</tr>
<tr>
<td>51</td>
<td>1</td>
<td>0.84</td>
<td>200</td>
<td>2.04</td>
<td>356</td>
<td>3.60</td>
</tr>
<tr>
<td>52</td>
<td>1</td>
<td>0.84</td>
<td>200</td>
<td>1.68</td>
<td>509</td>
<td>4.02</td>
</tr>
<tr>
<td>53</td>
<td>1</td>
<td>1.00</td>
<td>200</td>
<td>1.80</td>
<td>375</td>
<td>3.50</td>
</tr>
<tr>
<td>54</td>
<td>1</td>
<td>1.06</td>
<td>200</td>
<td>2.30</td>
<td>372</td>
<td>3.28</td>
</tr>
<tr>
<td>55</td>
<td>1</td>
<td>0.96</td>
<td>200</td>
<td>2.20</td>
<td>373</td>
<td>3.36</td>
</tr>
<tr>
<td>56</td>
<td>1</td>
<td>1.02</td>
<td>200</td>
<td>2.08</td>
<td>353</td>
<td>3.16</td>
</tr>
<tr>
<td>57</td>
<td>1</td>
<td>0.90</td>
<td>200</td>
<td>1.58</td>
<td>439</td>
<td>3.20</td>
</tr>
<tr>
<td>58</td>
<td>1</td>
<td>0.88</td>
<td>200</td>
<td>2.32</td>
<td>473</td>
<td>4.66</td>
</tr>
</tbody>
</table>
Appendix C

Implementation Details for Lexical Profiling

Implementation details of the lexical profiling scheme for higher-order, lazy functional programming languages are presented here. The mechanisms discussed relate directly to an interpreted graph reduction engine such as the Four-Stroke Reduction Engine [CPJ86]. However, they may also be included into a fully compiled abstract machine by suitably amending the instruction set to incorporate extra profiling parameters and operations. It is assumed that the reader is familiar with the terminology and operations of interpretive graph reduction. Standard texts such as [PJ87b] and [FH88] provide a thorough introduction to the subject.

A brief synopsis of lexical profiling is given in Section C.1. Sections C.2 and C.3 describe the operations performed at compile-time and run-time, respectively, in order to effect lexical profiling. Finally, Section C.4 demonstrates the correct behaviour of the lexical profiling implementation using an example which contains both higher-order functions and lazily evaluated arguments.

C.1 Synopsis of Lexical Profiling

Lexical profiling as originally suggested by Clayman in [CPC91, Cla93] is designed to report:

- the number of calls made to and from specified functions,
- space-usage, and
• the time taken to perform computations,

for higher-order, lazy functional programs, in terms of to the lexical affinities present in the source code. The idiosyncrasies of profiling lazy, higher-order programs are best described by example. Consider the following short program:

```haskell
let map f [] = []
    map f (x:xs) = f x : map f xs
increment x = x + 1
  g = map increment [1..1000000]
  h = map increment [1..10]
  in (g, h).
```

When this program is executed, increment will be invoked many times from within map via map's higher-order parameter. The explicit references to increment, however, are lexically enclosed within the definitions of g and h. From the programmer's point of view it makes sense to talk about the function increment which originates from either g or h in preference to that which was invoked by map because this is more readily related back to the source code. This is the essence of lexical profiling. In larger examples where increment is passed as a higher-order argument by many functions, the information provided by the lexical profiling style is far simpler for the programmer to assimilate than the non-lexical alternative.

The mechanism which keeps track of the origins of higher-order functions can be extended to overcome the problems of profiling in the presence of lazy evaluation. The following function definition is used to illustrate the problem:

```haskell
h = f expA expB.
```

Here, the expressions expA and expB originate from the definition of function h. However, lazy evaluation semantics demands that the expressions will not be evaluated until their normal forms are definitely required. We must therefore keep track of the origin of all argument expressions so that when the expressions are eventually evaluated the profiler can assign the cost of the evaluation to the correct function.

Figure C.1 illustrates the general framework of the profiler. The programmer specifies which functions are to be profiled and in which manner (see Section C.2) when the compiler is invoked. The profiling results are written to shared memory as the program executes. These can therefore be read by a separate report process which can display the results as they are generated. The high-level function names are not present in the
executable program, therefore an auxiliary file is generated by the compiler so that the report process can map the encoded profile data back to the original functions.

C.2 Changes to the Compiler

For correct lexical profiling, the compiler examines the source program prior to any transformations that it may wish to perform. The lexical structure of the original program can then be recorded. After this, the compiler is free to transform the program with the provision that the lexical profiling information must be manipulated sensibly.

The following discussion assumes that the functional program is represented internally by the compiler as a parse-graph (see e.g. [PC91]) and that the nodes of the parse-graph are represented by cells which have been extended to include profiling information. To keep track of the relationships between the parse-graph and the source program we assign a unique colour for each function to be profiled. Typically, colours will be represented by integers and will be assigned to every cell in the parse-graph. Each cell also contains a 1-bit flag to indicate whether it represents the root of the definition of a profiled function. When the flag is set, the cell is said to contain a root-marker.

Colour assignment is performed in two stages. The first stage identifies the roots of
the function definitions within the graph:
for each function to be profiled:
   (i) locate the root of the subgraph which represents the function definition.
   (ii) set the root-marker for this cell.
   (iii) assign a unique colour to the cell.

The second stage propagates the colours to the other cells in the graph:
for each cell whose root marker is set:
   (i) recursively propagate the colour of the cell to all of its descendants, terminating each branch of the recursion on encountering a cell whose colour is already determined.
   (ii) if the recursion terminates on a cell whose root-marker is not set and whose colour is different from that being propagated then the unprofiled child is shared by two (or more) profiled parents. Two methods for dealing with this problem are described below.

Figure C.2 illustrates the problem of an unprofiled function which is shared by two or more profiled functions. The graph represents the case when profiles are requested for \( f \) and \( g \), but not for \( h \), where \( f \), \( g \), and \( h \) are defined as follows:

\[
\begin{align*}
  f &= \exp_A h \\
  g &= h \exp_C \\
  h &= \exp_B
\end{align*}
\]

Two alternative solutions are provided:

1. Duplicate the cells which represent the shared child, as demonstrated in Figure C.3.

   The result of the program is unchanged but a loss of sharing may occur, thereby distorting the outcome of the profile.
2. Profile the child separately in its own right (see Figure C.4). Although this results in a loss of accuracy when propagating the costs of child functions backwards to their parents, the sharing properties of the source program are retained.

It is intended that both of these options are available to the programmer and the choice of which to use for a specific profiling exercise is governed by the requirements of the resulting information.

C.3 The Run-time Mechanisms

For straightforward interpretive graph reduction, a program is represented by a graph of binary cells. Each cell consists of left-hand and right-hand fields whose contents are determined by the abstract machine. (Other fields may also be required by the reduction engine to store status information but these have no effect on the profiling
A naïve attempt at profiling is shown in Figure C.5. The graph represents the following program segment:

\[
\begin{align*}
\text{let } & g \ x = x \ \text{exp}_A \ \text{exp}_B \\
& h = g \ f\\
\text{in } & h
\end{align*}
\]

and each graph cell is augmented with the colour of the function which constructed it in the same manner as the compiler's parse-graph shown in Figures C.2, C.3, and C.4. We shall henceforth use the terms constructor-function and its associated constructor-colour.

The graph segment shown in part (ii) of Figure C.5 is the result of instantiating \( g \) with the argument field, \( @ \), using template instantiation [PJL92, Ch.2]. The redex has been overwritten by the result and the movement of key fields from graph segment (i) to graph segment (ii) is illustrated by the labels \( @, @, \) and \( @ \).

There are several points of interest in this example. Firstly, note that the constructor-colour of the overwritten redex has not been updated. This is explained in more detail in Section C.3.4. Secondly, notice that in part (i) of the figure, prior to instantiating \( g \), the reference to function \( f \) was contained within a cell whose constructor-colour was given as \( h \) but in part (ii) of the figure, after the instantiation, the reference is contained within a cell whose constructor-colour is given by \( g \). This clearly contravenes the rules of lexical profiling because it now appears as though the reference to \( f \) occurred lexically within \( g \).
The problem is due to the unboxed argument, $\odot$, and can be solved by ensuring that all argument values are boxed. Figure C.6 demonstrates this principle. In part (i) of the figure, prior to the instantiation of $g$, the argument containing the reference to $f$ is boxed and thus occupies an extra cell. After the instantiation (part (ii) of the figure), the cell is unchanged, hence maintaining a constant constructor-colour for the reference to $f$.

The method of boxed arguments provides correct lexical profiling information but requires the reduction engine to represent all arguments as boxed values. This will need changes to be made to the way that most abstract machines perform graph reduction and is very inefficient. Ideally, the profiling mechanism should only require changes that relate directly to profiling and not to the reduction strategy.

Fortunately the problem can be overcome by assigning additional colouring information to the left-hand field, $L$, and to the right-hand field, $R$, of each cell as depicted in Figure C.7. Since $L$ and $R$ each have their own profiling information, values can be safely unboxed without losing vital information. In total, six colours are used, plus a 1-bit marker (root) which is set in the top cell of profiled functions (see Section C.2) and
reset in all other cells.

The six colours are arranged into three pairs: \( c \leftarrow o \) (defined for the whole cell), \( c_L \leftarrow o_L \) (defined for the left-hand field), and \( c_R \leftarrow o_R \) (defined for the right-hand field). The \( \leftarrow \) operator is used simply to combine the two colours of each pair into a single, compound symbol. Table C.1 provides a summary of the colouring information.

The \( c, c_L, \) and \( c_R \) colours indicate the constructor-function for the whole cell, the item in the left-hand field, and the item in the right-hand field, respectively. For the cells of supercombinator templates, \( c = c_L = c_R \). These are assigned statically when the program is loaded, according to the colouring information supplied by the compiler. When supercombinator templates are instantiated (see Sections C.3.1 and C.3.2), a mutable copy of the template graph is constructed. The \( \textit{fields} \) which contained formal parameters in the template are instantiated with both the value and the profiling colours of the actual parameters. Therefore, in the mutable part of the program graph, the \( c, c_L, \) and \( c_R \) colours within a single cell may be different.

The colours \( o, o_L, \) and \( o_R \) are \textit{origin-colours} and are used to record which functions lexically contain references to the corresponding constructor-functions in the source program. Profiling information can then be reported back to the programmer in terms of the lexical function origins present in the source code. Consider the example program given on page 227. Here, the origin-colours of cells constructed by \texttt{map} will be set to the colour of function \( g \) for the first call to \texttt{map} and to the colour of function \( h \) for the second call. Again, the \( o, o_L, \) and \( o_R \) colours may differ from each other when either or both of the left- or right-hand fields are instantiated by actual parameters. Since origin-colours are calculated during instantiation, they are not defined for the cells of supercombinator templates.

C.3.1 Instantiating Profiled Supercombinators

If the root marker in the top cell of the supercombinator's template is set, then the template represents a profiled function. Figure C.8 illustrates the procedure for instantiating profiled functions, using the supercombinators \( f_1 \) and \( f_2 \) which are defined as follows:

\[
\begin{align*}
f_1 &= f_2 \exp_A \\
f_2 x &= \exp_B x.
\end{align*}
\]

Consider part (i) of the figure which represents the state of the reduction immediately prior to the instantiation. The \( o, o_L, \) and \( o_R \) colours of every cell in \( f_2 \)'s template are as
yet undefined. In this case, the origin of the reference to $f_2$ is given by the constructor-colour, $e_L = s$, in the left-hand field of the redex. That is therefore the value assigned to the origin-colours of the instantiated graph shown in part (ii) of the figure. Notice that, as before, the colours $c \leftarrow o$ of the overwritten redex remain unchanged and that the actual parameter, $\otimes$, retains its colours, $u \leftarrow v$.

### C.3.2 Instantiating Unprofiled Supercombinators

Figure C.9 illustrates the procedure for instantiating a supercombinator whose root marker is *not* set. This uses the same supercombinator definitions as Figure C.8 but represents a call to a function, $f_2$, that is not being profiled in its own right. For the purposes of profiling, $f_2$ has been subsumed into its origin-function, $f_1$. The instantiation therefore differs from that of profiled supercombinators in two ways. Firstly, the constructor-colour in the left-hand field of the redex will *always* be identical to that of the subsumed supercombinator. Secondly, the origin-function for the subsumed supercombinator is the same as the origin-function of the profiled function into which it was subsumed. Therefore, the origin-colour is determined by the *origin-colour*, $o_L = t$, in the left-hand field of the redex. All other aspects of the instantiation are identical to the profiled case.
C.3.3 Call-count Profiling

The number of calls to a profiled function is determined by the number of times its associated supercombinator is instantiated. Sections C.3.1 and C.3.2 described two types of instantiation, the first of which deals with profiled supercombinators. Call-counts are incremented only for profiled supercombinators which must, of course, include CAFs. CAFs are not necessarily instantiated in the normal way because they have no arguments. Instead a profiled CAF is detected when the unwind operation [PJ87b] passes through a cell whose root marker is set. The call-count for the associated CAF is then incremented and the root-marker of the CAF is then reset (it would be wrong to count many invocations of a shared CAF separately because the CAF is evaluated only once).

A separate call-count register is maintained for every \( c \leftarrow o \) combination, where \( c \) is statically bound to the cells of the supercombinator template and \( o \) is determined by the method described in Section C.3.1.

C.3.4 Space Profiling

Space profiling uses the profiling colours \( c \leftarrow o \) attached to the cells in the program graph to determine the total number of cells allocated for a function and the maximum number of cells allocated at any one time. Space profiling information is recorded separately for each \( c \leftarrow o \) combination and requires only small changes to be made to the cell allocation.

\(^1\)Constant applicative forms [PJ87b].
and garbage collection code so that the relevant registers are updated each time a cell is allocated or deallocated. To monitor continuously the number of heap cells that are currently active we are assuming reference-count garbage collection [GRW88, Hug87b, Axf90, Hud86, LM86, Shu88]. If a different style of garbage collection were used, such as two-space copying [Bak78, Rud86] or mark/sweep [Coh81, Hug85] then the number of active heap cells could only be approximated by an upper bound. This would only be accurate immediately after a garbage collection has taken place and would steadily lose accuracy as cells become inactive, up to the next collection.

In classical graph reduction the redex is overwritten with the result after each reduction has taken place to ensure that shared values are not recomputed (there are many and varied discussions about this in e.g. [Joh84], [Aug84], [FW87], [PJ87a], [PJ87b], [BPJR88], [PJS89], [AJ89a], [AJ89b]). For space profiling purposes we must take care when overwriting a redex to leave the profiling colours of the cell intact (the profiling colours for the fields may change of course), otherwise the de-allocation of the cell by the garbage collector will decrement a different space-usage register to that which was incremented when the cell was allocated. The colours attached to the fields of the updated redex ensure that the higher-order lexical profiling continues to function correctly.

C.3.5 Time Profiling

The implementation of time profiling is closely tied to the unwind operation of graph reduction. The profiler maintains a separate timing for each $c_L \leftarrow o_L$ colour combination and makes use of a current $c_L \leftarrow o_L$ register (we use the colours of the left-hand field because the spine of the graph is encoded there and it is the pointers in the spine which determine function calls). At the start of the reduction all timers are zeroed, the current $c_L \leftarrow o_L$ register is set to the $c_L \leftarrow o_L$ of the initial redex, and the system time, $t$, is read. Reduction then proceeds as normal by unwinding the spine.

When unwind encounters a cell whose $c'_L \leftarrow o'_L$ colour differ from the current $c_L \leftarrow o_L$:

1. The current system time $t'$ is read.
2. The accrued time for $c_L \leftarrow o_L$ is incremented by the elapsed time $t' - t$.
3. The timer $t$ is updated with $t'$.
4. The current $c_L \leftarrow o_L$ register is set to $c'_L \leftarrow o'_L$. 

236
The unwind operation is performed each time a redex is overwritten. As long as the unwind always starts at the overwritten redex, no further action is required to profile time costs. This mechanism provides the required time, space, and call-count profiling information even when lazy evaluation causes the graph to become fragmented and results in many context switches.

C.4 A Single-Stepped Example of Lexical Profiling

In this section the steps of the reduction shown in Figure C.10 are explained with specific reference to lexical profiling activities. The figure is based on the following function definitions:

\[
\begin{align*}
    h &= g \ f \\
    g \ x &= x \ \text{exp}_A \\
    f \ y &= + \ \text{exp}_B \ y
\end{align*}
\]

and the explanation relies on the profiling terms described in the previous sections; the reader should be familiar with these before continuing.

Part (i):

1. The current \( c_L \leftarrow o_L \) register is set to \( h \leftarrow \bullet \) and the spine is unwound (the time for \( h \leftarrow \bullet \) is not yet updated).

2. The profiled function, \( g \), is encountered and the origin-colour for the function is determined from the left-hand field of \( h \) where \( c_L = h \).

Part (ii):

1. The redex is overwritten with the instantiated copy of \( g \) (the cell allocator increments the space usage for \( g \) as the new cells are allocated). The formal parameter, \( z \), has been overwritten by the argument field, \( \odot \), complete with its profile colours, \( h \leftarrow \bullet \). The origin-colour of this reference to \( g \) is set to \( h \) as determined in part (i). It is written into \( o_R \) in field \( \bigcirc \) and into \( o, o_R, \) and \( o_L \) of the cells in the subgraph, \( A \).

2. The call-count for \( g \leftarrow h \) is incremented.

3. The spine is unwound a second time (again, the timers are not yet updated).
4. The profiled function, \( f \), is encountered and the origin-colour for the function is determined from the field \( \circ \) where \( c_L = h \). This is correct because the reference to \( f \) originated from \( h \) in part (i).

Part (iii):

1. The instantiation proceeds as before. The redex is overwritten with the instantiated copy of \( f \) (the cell allocator adjusts the space-usage for \( f \) as the cells are allocated). The argument field, \( \circ \), overwrites the formal parameter and the origin-colours in and below field \( \circ \) are set to \( h \).
2. The call-count for $f \leftarrow h$ is incremented.

3. The spine is unwound for a third time and, since the $f \leftarrow h$ colours of the left-hand field differ from the current $c_L \leftarrow o_L$ register value of $h \leftarrow \bullet$, the elapse timing is adjusted for $h \leftarrow \bullet$. The current $c_L \leftarrow o_L$ register is set to $f \leftarrow h$ and the unwind continues until the primitive function $+$ is reached.

4. At this point the values of $A$ and $B$ are required. Each is reduced in turn before the $+$ function can be executed. Notice that the profile colours of $A$ and $B$ correctly identify their constructor- and origin-functions, hence the technique has behaved correctly in the presence of lazy evaluation.
<table>
<thead>
<tr>
<th>Colour</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>Identifies the function responsible for creating the current graph cell.</td>
</tr>
<tr>
<td>$o$</td>
<td>Identifies the origin of the function responsible for creating the current graph cell. From this colour we can determine where the function reference was made, lexically within the source code.</td>
</tr>
<tr>
<td>$c_L$</td>
<td>Indicates which function originally created the left-hand field of the current cell. This is needed to keep track of the constructor colour of functions that have been passed as arguments. If the field contains a higher-order argument then $c_L \neq c$. This colour is used in Section C.3.1 to determine the origin colour of a profiled supercombinator when the supercombinator is instantiated.</td>
</tr>
<tr>
<td>$o_L$</td>
<td>The origin colour of the left-hand field of the current cell. As for $c_L$, if the field contains a higher-order argument then $o_L \neq o$. This colour is used in Section C.3.2 to determine the origin colour of a non-profiled supercombinator when the supercombinator is instantiated.</td>
</tr>
<tr>
<td>$c_R$</td>
<td>The constructor colour of the right-hand field of the current cell. Again, if the field contains a higher-order argument then $c_R \neq c$. This colour is needed when the value of the field is passed as an actual parameter to another supercombinator where it is subsequently applied as a function to some arguments.</td>
</tr>
<tr>
<td>$o_R$</td>
<td>The origin colour of the right-hand field of the current cell. As for $c_R$, if the field contains a higher-order argument then $o_R \neq o$ and, again, the colour is needed when the value of the field is passed as an actual parameter to another supercombinator.</td>
</tr>
</tbody>
</table>

Table C.1: A summary of colouring information.
Bibliography


K. Davis and P. Wadler. Strictness analysis in 4D. In S.L. Peyton Jones, G. Hutton, and C.K. Holst, editors, Functional Programming Workshop,


[Par90] D.J. Parrott. Compiling C$\mathcal{G}$F into $(\nu, G)$-machine code. Internal Note IN/93/2, University College London, Department of Computer Science, July 1990.


256


[SY90] Sun Yong Qiang and Yuan Wei. Exploitation and control of parallelism in FP system. Dept Computer Science, Shanghai Jiao Tong University, Shanghai 200030, P.R. China. Submitted to PARLE '91., December 1990.


260


