Optimised Redundant Cell Collection for Graph Reduction

by

Patrick John Wright

A thesis submitted to the University of London
for the Degree of Doctor of Philosophy

Department of Electronic and Electrical Engineering
University College London

October 1993
Abstract.

Parallel processing offers a route to more powerful computation, but many problems have arisen on the path to the realisation of practical implementations of such architectures. One of the most important stumbling blocks in preventing parallel processors reaching widespread commercial fruition, is the problem of programming. Standard procedural languages, even when modified, offer an inadequate programming paradigm for such machines. A promising technique, combining the power of parallel processing with the software advantages of functional programming, is graph reduction.

In the development of reduction architectures designed for functional programming, one of the most important implementation issues, affecting operational efficiency, is that of garbage collection. Of the three main methods of collection available, the most promising appears to be reference counting. This, however, suffers from a number of drawbacks: it involves a large memory overhead to support the scheme, it will not intrinsically deal with self-referencing (cyclic) structures and it is not real time in nature.

The thesis starts with a short historical perspective on parallel processing, briefly presenting some of the major milestones in parallel architecture research. The discussion continues by introducing functional languages and the problems raised by garbage collection. The major garbage collection techniques are discussed and the reasons for exploring reference counting are explained.

The first aspect of reference counting collection to be addressed is the memory overhead. An architecture for alleviating this problem is presented, based upon limited-width reference count fields. The chapter concludes with analysis of the new techniques and contrasts the worst case performance against the standard methods.

A short chapter then introduces real time systems, with particular reference to functional programming and the attempts to make standard garbage collection techniques suitable for real time systems. The discussion continues with a novel technique for the collection of large redundant data structures in real time. A system is then discussed for implementing the collection of redundant cyclic structures, within the graph, in real time. As each aspect of the technique is explored, algorithms are presented.

The thesis continues by presenting a worse case temporal analysis of the major services offered by the collector to a graph mutator.

A formal verification of the basic real time collector, using the MALvern Program Analysis Suite (MALPAS) is presented.

A simulator for the system has been written, which was used to test the techniques developed and to ascertain if the theoretical predictions are valid. This software implementation of the collector is described and some of the results obtained are presented and discussed. The performance of the new reference counting techniques are compared with an implementation of Baker's algorithm.

The thesis concludes with a critical appraisal of the work carried out, and a discussion of how the research could progress, to realise a practical collector in a viable commercial parallel graph reduction machine.
Acknowledgements.

The author wishes to express his gratitude to Prof. R.J. Offen for all his support and guidance during the course of this research. Thanks are also due to Dr. O.J. Davies, for his help in the early stages of the project.

The financial support of Racal Research Ltd. is gratefully acknowledged, as is the help and guidance from Mr. Eric Harper and Dr. Tim Burnett of Racal.

Thanks are due to the Defence Research Agency, Malvern (CAD1 Division) and also in particular to Dr. Brian Bramson for his invaluable expert advice in the use of MALPAS.

Finally, the author would like to thank his colleagues, Dr. Paul Houselander and Vish Patel for many long and useful discussions.
To those I love
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract.</td>
<td>3</td>
</tr>
<tr>
<td>Acknowledgements.</td>
<td>5</td>
</tr>
<tr>
<td>List of Figures.</td>
<td>12</td>
</tr>
<tr>
<td>List of Tables.</td>
<td>14</td>
</tr>
<tr>
<td><strong>Chapter 1. Introduction to Parallel Processing.</strong></td>
<td>15</td>
</tr>
<tr>
<td>1.1 Background.</td>
<td>15</td>
</tr>
<tr>
<td>1.2 Parallel Architectures.</td>
<td>15</td>
</tr>
<tr>
<td>1.3 The Software Problem.</td>
<td>16</td>
</tr>
<tr>
<td>1.3.1 Imperative Languages.</td>
<td>17</td>
</tr>
<tr>
<td>1.3.2 Alternatives to Imperative Languages.</td>
<td>18</td>
</tr>
<tr>
<td>1.3.3 Logic Languages.</td>
<td>18</td>
</tr>
<tr>
<td>1.3.4 Data Flow Languages.</td>
<td>19</td>
</tr>
<tr>
<td>1.3.5 Functional Languages.</td>
<td>19</td>
</tr>
<tr>
<td>1.4 Conclusions.</td>
<td>19</td>
</tr>
<tr>
<td><strong>Chapter 2. Functional Languages.</strong></td>
<td>20</td>
</tr>
<tr>
<td>2.1 Some Major Features of a Functional Style.</td>
<td>20</td>
</tr>
<tr>
<td>2.1.1 Referential Transparency and Parallel Processing.</td>
<td>20</td>
</tr>
<tr>
<td>2.2 Architectures for Supporting Functional Languages.</td>
<td>21</td>
</tr>
<tr>
<td>2.2.1 Graph Reduction.</td>
<td>21</td>
</tr>
<tr>
<td>2.2.2 Alternative Approaches to the Implementation of Functional Languages.</td>
<td>23</td>
</tr>
<tr>
<td>2.3 Applications for Functional Languages.</td>
<td>24</td>
</tr>
<tr>
<td>2.4 Conclusions.</td>
<td>25</td>
</tr>
<tr>
<td><strong>Chapter 3. Introduction to Garbage Collection.</strong></td>
<td>26</td>
</tr>
<tr>
<td>3.1 Concrete Representation of the Graph.</td>
<td>26</td>
</tr>
<tr>
<td>3.2 The Significance of Garbage Collection.</td>
<td>27</td>
</tr>
<tr>
<td>3.3 Mark-Scan Garbage Collection.</td>
<td>28</td>
</tr>
<tr>
<td>3.4 Copying Garbage Collection.</td>
<td>28</td>
</tr>
<tr>
<td>3.5 Reference Counting Garbage Collection.</td>
<td>29</td>
</tr>
<tr>
<td>3.6 Other Methods of Dealing with Garbage.</td>
<td>30</td>
</tr>
<tr>
<td>3.7 An Appraisal of the Various Garbage Collection Strategies.</td>
<td>31</td>
</tr>
<tr>
<td>3.7.1 Mark-Scan Collection.</td>
<td>31</td>
</tr>
<tr>
<td>3.7.2 Copying Collection.</td>
<td>32</td>
</tr>
<tr>
<td>3.7.3 Reference Counting.</td>
<td>33</td>
</tr>
<tr>
<td>3.8 Choice of Garbage Collector.</td>
<td>33</td>
</tr>
<tr>
<td>3.9 Developments and Variations of Reference Counting Garbage Collection.</td>
<td>34</td>
</tr>
<tr>
<td>3.9.1 Distributed Systems and Weighted Reference Counting.</td>
<td>34</td>
</tr>
<tr>
<td>3.9.2 The Collection of Cycles by a Hybrid Technique.</td>
<td>35</td>
</tr>
<tr>
<td>3.9.3 The Single-bit Reference Count.</td>
<td>36</td>
</tr>
<tr>
<td>3.9.4 Collection of Cycles in Reference Counting by Other Techniques.</td>
<td>37</td>
</tr>
<tr>
<td>3.10 Hughes' Method of Reference Counting.</td>
<td>37</td>
</tr>
<tr>
<td>3.10.1 The Static Graph.</td>
<td>39</td>
</tr>
<tr>
<td>3.10.2 The Dynamic Graph.</td>
<td>40</td>
</tr>
<tr>
<td>3.11 Optimisations of Hughes' Basic Technique.</td>
<td>45</td>
</tr>
<tr>
<td>3.12 Conclusions.</td>
<td>46</td>
</tr>
</tbody>
</table>
Chapter 4. Alleviating the Memory Overhead Problem.

4.1 Introduction. 47
4.2 Local Count Overflow. 47
4.3 Group Count Storage and Overflow.
   4.3.1 Storing the Group Handle. 50
   4.3.2 Group Count Overflow. 52
4.4 Dynamic Use of the Collector's Memory Space. 54
4.5 Use of Caching Techniques. 54
4.6 Quantifying the Memory Overhead.
   4.6.1 The Standard Technique. 54
   4.6.2 The Reduced Memory Overhead Technique. 56
4.7 Optimisation Arising from Pre-reduction Analysis of the Graph. 54
4.8 Conclusions. 65

Chapter 5. Introduction to Real Time Systems.

5.1 An Introduction to Real Time Functional Systems. 66
5.2 A Taxonomy of Real Time Systems. 66
5.3 Real Time Functional Systems. 69
5.4 Mark-scan Garbage Collection in Real Time. 70
5.5 Copying Garbage Collection in Real Time. 72
5.6 Reference Counting Garbage Collection in Real Time. 77
5.7 Real Time Applications for Functional Languages. 81
5.8 Specification of Real Time Systems. 82
5.9 Conclusions. 83


6.1 A High Level Abstract Architecture of the Collector and Mutator. 84
6.2 A Basis for the Collection of Large Redundant Acyclic Structures. 84
6.3 The Collection of Cyclic Structures in Real Time.
   6.3.1 Incremental Real Time Version of Tarjan’s Algorithm 90
   6.3.2 The Deletion of Pointers and Nodes. 93
   6.3.3 The Addition of Pointers and Nodes. 102
   6.3.4 The Main Controlling Element of the Collector. 117
6.4 A Model of the Parallel Behaviour of the Collector and Mutator.
   6.4.1 Scheduling within the Collector. 121
6.5 The Main Data Structures of the Collector. 122
6.6 An Enhanced Set of Collector Services. 124
6.7 Extension of the Acyclic Collection Technique to Cope with Cycles. 125
6.8 Conclusions. 125


7.1 Introduction to the Temporal Analysis of the Collector. 127
7.2 A Performance Overview of the Important Collector Algorithms.
   7.2.1 Specific Simplifications of the Temporal Equations. 127
7.3 Introduction to the Integrated System. 129
   7.3.1 Limited-width Reference Counting in Real Time. 129
7.4 The Spatial Overhead of Real Time Garbage Collection. 134
7.5 Conclusions. 135
## Contents

<table>
<thead>
<tr>
<th>Chapter 8. The Formal Verification of the Collector by use of MALPAS.</th>
<th>137</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>8.1</strong> Introduction.</td>
<td>137</td>
</tr>
<tr>
<td><strong>8.2</strong> An Outline of the MALPAS System.</td>
<td>137</td>
</tr>
<tr>
<td><strong>8.2.1</strong> Establishing the Proof of Correctness of a Program.</td>
<td>144</td>
</tr>
<tr>
<td><strong>8.3</strong> An Overview of the MALPAS IL Model of the Collector.</td>
<td>145</td>
</tr>
<tr>
<td><strong>8.3.1</strong> The newapp Procedure.</td>
<td>145</td>
</tr>
<tr>
<td><strong>8.3.2</strong> The visit Procedure.</td>
<td>148</td>
</tr>
<tr>
<td><strong>8.3.3</strong> The emitgroup Procedure.</td>
<td>150</td>
</tr>
<tr>
<td><strong>8.3.4</strong> The MAIN Procedure.</td>
<td>153</td>
</tr>
<tr>
<td><strong>8.4</strong> Conclusions.</td>
<td>156</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 9. The Simulation.</th>
<th>157</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>9.1</strong> Introduction to the Simulation.</td>
<td>157</td>
</tr>
<tr>
<td><strong>9.2</strong> Control of the Simulation.</td>
<td>158</td>
</tr>
<tr>
<td><strong>9.2.1</strong> Time-step Mode.</td>
<td>159</td>
</tr>
<tr>
<td><strong>9.2.2</strong> Event-driven Mode.</td>
<td>159</td>
</tr>
<tr>
<td><strong>9.2.3</strong> Control of the Collector's Cache.</td>
<td>160</td>
</tr>
<tr>
<td><strong>9.2.4</strong> Results Obtainable from the Simulator.</td>
<td>160</td>
</tr>
<tr>
<td><strong>9.3</strong> Results from the Simulator.</td>
<td>161</td>
</tr>
<tr>
<td><strong>9.4</strong> Comparison with Baker's Algorithm.</td>
<td>173</td>
</tr>
<tr>
<td><strong>9.5</strong> Conclusions.</td>
<td>176</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 10. Conclusions.</th>
<th>177</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>10.1</strong> Summary of the Thesis.</td>
<td>177</td>
</tr>
<tr>
<td><strong>10.2</strong> Appraisal of the Research.</td>
<td>178</td>
</tr>
<tr>
<td><strong>10.3</strong> Future Work.</td>
<td>179</td>
</tr>
</tbody>
</table>

| Appendix A. Some Functional Programs. | 182 |
| Appendix B. The MALPAS Model of the Collector. | 187 |
| **B.1** IL Source. | 187 |
| **B.2** The MALPAS Log File. | 207 |
| **B.3** Output of the Compliance Analyser. | 220 |

| References. | 242 |
| Index. | 254 |
List of Figures.

Figure 2.1 An example functional program 20
Figure 2.2 Simplified representation of the graph reduction process 23
Figure 3.1 Abstract syntax tree 26
Figure 3.2 General representation of a node in the reduction graph 27
Figure 3.3 Possible concrete representation of a reduction graph 27
Figure 3.4 Part of the reduction graph, showing reference counts 30
Figure 3.5 Weighted reference counting collection 35
Figure 3.6 A cycle within the reduction graph 36
Figure 3.7 Formation of the derived graph 38
Figure 3.8 An S.C.C. with group reference count 39
Figure 3.9 The Tarjan search algorithm 40
Figure 3.10 Splitting of a cyclic structure 45
Figure 4.1 Schematic representation of the collector memory structure 48
Figure 4.2 Detail of local count overflow into the main heap 49
Figure 4.3 Detail of group data internal to the collector 51
Figure 4.4 Detail of accommodating group overflow data in the main heap 53
Figure 4.5 Overall reference count overhead versus heap size 55
Figure 4.6 A standard node 55
Figure 4.7 Overhead ratio, versus δ, for a variety of c 56
Figure 4.8 Overhead ratio for fixed collector memory versus δ 59
Figure 4.9 Overhead ratio versus S.C.C. size 61
Figure 4.10 Comparison of the standard and limited-width techniques of reference counting 62
Figure 4.11 Comparison of worst case memory overhead between limited width technique and standard technique for a variety of c 63
Figure 5.1 A deadline 66
Figure 5.2 A hard deadline in a safety critical system 67
Figure 5.3 A soft deadline 68
Figure 5.4 A hybrid real time system 68
Figure 5.5 A taxonomy of real time systems 69
Figure 5.6 Relocating a cell in the standard copying collector 73
Figure 5.7a Relocating a cell in Baker's copying collector 75
Figure 5.7b Relocating a cell in Baker's copying collector (after copy) 75
Figure 5.8 The deletion of an arbitrarily large acyclic data structure 78
Figure 5.9 A T.B.D. stack with buried garbage 79
Figure 5.10 A SLIP structure 80
Figure 6.1 Abstract high level view of the reduction architecture 84
Figure 6.2 An example of an acyclic data structure 85
Figure 6.3 Deletion of an acyclic structure 87
Figure 6.4 Incremental traversal of an acyclic structure 87
Figure 6.5 A large cyclic structure splitting into smaller S.C.C.s 88
Figure 6.6 Tarjan's original algorithm 89
Figure 6.7 The original emitgroup procedure 90
Figure 6.8 Modified real time Tarjan algorithm 91
Figure 6.9 The real time emitgroup procedure 93
Figure 6.10 Deletion of a pointer using deleteref 94
Figure 6.11 Deletion of a pointer in a cycle waiting to be split 95
Figure 6.12 Deletion of a pointer from a partly scanned cycle 97
Figure 6.13 Deletion of a pointer from a fully scanned cycle 97
Figure 6.14 The real time deleteref procedure 101
List of Figures

Figure 6.15 Addition of a new application node 102
Figure 6.16 Sieve of Eratosthenes EASE program 103
Figure 6.17 Part of the directed graph produced by the Sieve of Eratosthenes 103
Figure 6.18 Order of visitation of nodes 104
Figure 6.19 Progress of a Tarjan search on a single S.C.C 104
Figure 6.20 An example EASE program 104
Figure 6.21 The addition of a new application node 109
Figure 6.22 The addition of a new application node during group emission 110
Figure 6.23 The newapp real time procedure 114
Figure 6.24 Addition of a pointer 114
Figure 6.25 The addref real time procedure 115
Figure 6.26 The addref_ext real time procedure 116
Figure 6.27 The addition of a pointer in the addref_ext procedure 117
Figure 6.28 A simplified view of the collector's MAIN routine 118
Figure 6.29 A high level view of the collector's architecture 120
Figure 6.30 Collector and mutator operating in parallel 122
Figure 6.31 Pointer deletion stack element 124
Figure 6.32 A cyclic structure, with intra- and inter-cyclic pointer deletions 125
Figure 7.1 The integrated collector architecture 134
Figure 7.2 Compliance analysis 139
Figure 8.1 Example of (MALPAS) graph reduction 138
Figure 8.2 Output of the MALPAS reader for compliance analysis 143
Figure 8.3 The newapp procedure specification 148
Figure 8.4 Output of the MALPAS compliance analysis 150
Figure 8.5 The emitgroup procedure specification 152
Figure 8.6 The MAIN procedure of the MALPAS collector model 154
Figure 8.7 Comparison of empirical and theoretical memory overhead values 161
Figure 8.8 No of searches in the main heap against max. numerical-width of group descriptor number, for Sieve of Eratosthenes 162
Figure 9.1 Effect of cache on the number of heap searches, for Sieve of Eratosthenes 163
Figure 9.2 Number of collector heap read/writes versus cumulative total, \((\alpha = \beta = 2)\) 164
Figure 9.3 Number of collector heap read/writes versus cumulative total, \((\alpha = \beta = 3)\) 164
Figure 9.4 Number of collector local read/writes versus cumulative total, \((\alpha = \beta = 2)\) 166
Figure 9.5 Number of collector local read/writes versus cumulative total, \((\alpha = \beta = 3)\) 166
Figure 9.6 Number of collector operations versus cumulative total, \((\alpha = \beta = 2)\) 167
Figure 9.7 Number of collector operations versus cumulative total, \((\alpha = \beta = 3)\) 167
Figure 9.8 Number of mutator operations versus cumulative total 168
Figure 9.9 Number of mutator operations versus cumulative total 169
Figure 9.10 Number of mutator read/writes versus cumulative total 169
Figure 9.11 Number of mutator read/writes versus cumulative total 170
Figure 9.12 Number of uncollected cells versus collector time allocation 170
Figure 9.13 Number of uncollected cells versus collector time allocation 171
Figure 9.14 Number of collector local read/writes versus cumulative total 171
Figure 9.15 Number of collector operations for Baker's algorithm (for newapp) 174
Figure 9.16 Number of collector operations for Baker's algorithm (for deleteref) 174
Figure 9.17 Effect of changing \(\tau\) on the memory overhead ratio 173
Figure 9.18 Number of collector operations for the reference counting garbage collector 174
Figure 9.19 Number of collector operations for the reference counting garbage collector 175

-13-
List of Tables

List of Tables.

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 6.1</td>
<td>The scan condition of a node</td>
<td>108</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>Compensation ofemitgroup count totals for newapp</td>
<td>112</td>
</tr>
<tr>
<td>Table 6.3</td>
<td>Compensation ofemitgroup count totals for addref_ext</td>
<td>117</td>
</tr>
<tr>
<td>Table 6.4</td>
<td>The main data structures of the collector</td>
<td>124</td>
</tr>
<tr>
<td>Table 7.1</td>
<td>Summary of the worst case temporal behaviour of the major classes of collector routines</td>
<td>129</td>
</tr>
<tr>
<td>Table 7.2</td>
<td>Table of the worst case temporal behaviour of the major classes of collector routines, in the integrated system</td>
<td>133</td>
</tr>
<tr>
<td>Table 8.1</td>
<td>Types of node</td>
<td>138</td>
</tr>
</tbody>
</table>
Chapter 1. Introduction to Parallel Processing.

1.1 Background.

In the early days of computer design there was a need to minimise the quantity of hardware used in the construction of machines. This was primarily due to purely pragmatic reasons stemming from the high cost and low reliability of hardware components then available. The von Neumann architecture [NEU-45] is ideally suited to these requirements and forms the familiar basis for the majority of general purpose digital computers in use today [TRE-82a].

With the advent of VLSI technology, vast improvements in machine performance have been realised, but it is now clear that diminishing returns are being made on further developments in component technology. This is largely due to, what has been dubbed by Backus [BAC-78], as the von Neumann bottleneck. This is the communications path that links the central processing unit (CPU) of a computer with its memory. The contents of the memory are modified a word at a time, over this communications link, until computation ceases. The bottleneck is both a physical barrier to speed, as well as an intellectual restraint, constraining the programmer to a word-at-a-time methodology of programming. This restraint is apparent in the form of conventional imperative languages, that are based upon the underlying von Neumann architecture.

Until recently much effort has gone into improving the speed of individual components, in order to enhance the overall machine performance. There have also been many architectures produced that are designed to increase the parallelism within the conventional von Neumann framework: vector pipelines, bit-slice, instruction-prefetch etc. [HES-87], have all been used. All these architectures are still based on the von Neumann model, i.e they are more or less enhanced SISD (Single Instruction Single Data stream) machines [HES-87].

1.2 Parallel Architectures.

Alongside the largely commercial development of von Neumann enhancements, there has been a steady stream of academic projects, aimed at achieving a higher degree of parallelism using multiple processing elements. These projects have all shared the common goal of increasing computational through-put by using several processing elements, each of which cooperate on a single universal (programmed) problem.

The main thrust of parallel processing research started in the late fifties with a paper by Unger [UNG-58]. This described a machine with many processing elements, each stepping through the same sequence of instructions but handling different data. This is a so called SIMD (Single Instruction Multiple Data stream) architecture. The idea was developed with the SOLOMON project [SLO-62] in the sixties and reached fruition with Illiac IV [BOU-72]. The Illiac IV project was a failure with a view to building a general purpose parallel supercomputer. This was because real applications could only exploit the processor array for a small percentage of the time it took to run them, the remaining code had to be executed on a relatively slow Burroughs scalar processor. Later projects, targeted at specific tasks, such as the ICL DAP [HOC-88] aimed at numerical processing, were much more successful. This machine was the first SIMD architecture to support active memory, both at system and software levels. Active memory uses the concept of distributing processing power into the memory of the machine. Another notable project was the Massively Parallel Processor (MPP), that contained 16,384 processors. It was designed for image processing and also met with a greater degree of success then the Illiac IV [DUF-78, BAT-80].
Chapter 1. Introduction to Parallel Processing

A development of the SIMD architecture is the systolic array [KUN-82]. This contains processing elements that are more specialised in their application. It displays interconnectivity which is more limited between individual processing elements. These machines are best suited to highly specialised mathematically intensive rôles, such as matrix multiplication, solution of linear systems and discrete Fourier transforms.

Another development of the SIMD concept is that of the Connection Machine [HIL-85]. This machine is designed to mimic directly the operation of the human brain. Each machine can have up to 64K processors. Such machines have achieved peak performance rates of 1,000 million instructions per second.

A milestone in the history of parallel processing was reached with the C.mmp project at Carnegie Mellon University [WUL-72]. A totally different approach to parallel processing was undertaken, each separate processing element executed a different set of instructions and communicated with each other over a bus network, based on a cross bar switch topology. This is a MIMD (Multiple Instruction Multiple Data steam) architecture. Several applications were developed to run on this machine, these varied in their success, depending on how effectively the applications could be partitioned onto the processing elements. The partitioning task was left to the responsibility of the programmer. Carnegie Mellon went on to develop a more versatile multiprocessor (Cm*) based on a system of clustered LSI 11 microcomputers [SWA-77]. The system was run under an operating system that organised the processing across the bus connected clusters. Provided that the applications required few remote references, good performance figures were obtained.

MIMD machines are in principle much more flexible than SIMD machines. This has lead to many projects based on this methodology of computer organisation. A notable class of such machines are those based on the hypercube interconnexion topology, conceived at the California Institute of Technology [SEI-85].

The data flow model of computation was developed as a means of controlling massively parallel hardware and efficiently developing naturally parallel code [DEN-80]. The principles of data flow were developed in the 1960's by compiler writers, who used data flow graphs to do performance optimisations on standard serial computers. A data flow graph is a directed graph in which the nodes represent primitive functions. It was realised in the 1970's that if data flow graphs were executed directly, then the architectures that execute them would be massively parallel. These ideas have lead to the development of data flow machines, such as Manchester University's prototype data flow machine [WAT-79], as well as reduction based functional machines such as ALICE [DAR-81], GRIP [PEY-88] and many others [TRE-82b, VEG-84].

There are many surveys on the topic of parallel processing, and supercomputers, these include [BAE-73, KUC-77, FUL-77, DON-86, PAR-87, STA-87, BEL-89].

1.3 The Software Problem.

A major problem with multiprocessor MIMD machines, arose through the method of programming. The task of dividing the program down into parallel streams was left to the programmer. If this division was not correctly performed, or if the algorithm was not amenable to such partitioning, then the time taken to synchronise processing between each processing element was found to be excessive. The problem was further compounded by the fact that the algorithms were first manipulated in a sequential regime, before being processed to expose parallel control paths. Often standard sequential imperative languages, with special
constructs, were used to program these machines (e.g. parallel FORTRAN and later parallel C were used [KOV-83]). In complex applications the task of coordinating parallel streams of computation, with primitive programming constructs, such as FORK and JOIN, becomes untenably difficult. Recently concurrent languages have been developed, such as Modula-2 [WIR-83] and Ada* [BRAa-80]. These have better constructs that let the programmer initiate and coordinate multiple concurrent tasks. However, despite the apparent improvements made due to stronger programming constructs, these languages still depend upon the programmer uncovering parallelism within the algorithm that is under consideration.

Even advanced languages such as OCCAM [ELI-87], developed for use with the INMOS transputer [MAY-87, HOM-88], involve the programmer with the explicit use of such constructs as PAR and SEQ, controlling PARallel and SEQuential tasks. The language has the advantage that it is specifically targeted to the transputer. This is a development of the von Neumann architecture, that is designed to be run with multiple processing elements. However, OCCAM still suffers from many programming difficulties associated with extended procedural languages within a parallel domain.

It is desirable therefore, that any language that is used to program a parallel machine uncovers any latent parallelism in the algorithm being used, without the programmer having explicitly to specify parallel program control.

Another issue to be considered, in the context of programming in a parallel environment, is programming productivity. It has been found that the productivity of a particular programmer is largely independent of the language in which he is working [EIS-85]. In large applications, comprising 200,000 or more instructions, productivity can be as low as three correct instructions per day [JON-78]. It is therefore desirable to make the language as powerful as possible, that is, to encompass more computing in each construct, rather than just providing more constructs to the programmer. It has been observed that modified imperative languages, with their additional constructs to handle concurrency, lower programming efficiency still further [KOV-83].

Up to 50% of the time spent by a programming team may be used in "program maintenance". This comprises time spent sorting out unforeseen "bugs", that are places where the program diverges from its required specifications. Without a formal form of program proof it is very difficult to produce bug free software. As a result of this, maintenance costs are extremely high in the case of large complex pieces of software. The difficulty of writing error free software for parallel architectures is compounded when modified conventional languages are used.

In order to examine some alternatives to standard imperative languages, it is necessary to clarify what is meant by an imperative language.

1.3.1 Imperative Languages.

These are the familiar "traditional" languages, such as C [KER-88] and Algol [MCG-78] etc. They were a pragmatic development from the underlying von Neumann machine architecture and can be considered as a high-level abstracted model of this machine concept. These languages can be characterised by a number of attributes [VEG-84]:

* Ada is a registered trademark of the U.S. Department of Defence.
Chapter 1. Introduction to Parallel Processing

(a) Program variables imitate words in the machine. They are considered as locations where values can be saved. This is at the root of the intellectual bottleneck of word-at-a-time thinking.

(b) Control statements imitate jumps, for example the if-then-else construction, common in imperative languages.

(c) The assignment statement imitates the fetch and store instructions present in the underlying von Neumann architecture. This is analogous to the von Neumann bottleneck in an imperative language.

(d) Two, or more, algorithmically separated pieces of code may modify the same “global” variable. This interaction between two pieces of code can be undesirable, leading to unforeseen side-effects. Assignment statements thus interact via a global machine state.

(e) Much of the programming task is concerned with data manipulation, rather than basic algorithmic issues.

(f) It is difficult to achieve efficient parallel execution of programs, when several different asynchronous processes can have side-effects on each other, (i.e. they may affect one another in a way that was not specified when the system was originally designed).

(g) The imperative model is not founded on a sound mathematical base. This means that program proofs are difficult.

(h) Imperative languages are inherently sequential in nature.

1.3.2 Alternatives to Imperative Languages.

The difficulty of programming parallel machines with imperative languages, and their lack of useful mathematical properties that could be used in formal program proofs, has lead to research taking place to look at different approaches to programming. Some of the options are discussed briefly below.

1.3.3 Logic Languages.

These languages stem from work carried out in automated theorem proving. The basis for the languages is predicate logic [KOW-83, KOW-85]. Predicate logic allows relationships between objects to be stated and stored in a data base. The user can then ask questions about the objects and the logic language interpreter provides the necessary search algorithms to establish the truth of the proposition. Practical logic based languages, such as Prolog [CLO-81], depend upon features that are not strictly part of the logical framework. These are necessary to provide a reasonable programming environment.

These languages appear to be ideally suited to artificial intelligence applications. They are not efficient, however, in handling numerically intensive work necessary in such areas as mathematical modelling, pattern recognition, digital signal processing etc.
1.3.4 Data Flow Languages.

In this model the program is represented as a directed graph. Each instruction is data driven, i.e. the instruction is carried out when its operand arrives. New languages have been developed to exploit this architecture, such as the University of California’s ID, M.I.T.’s VAL [HAY-82] and Manchester University’s SISAL [WAT-79].

Even with such software developments, the underlying architecture makes the manipulation of data structures relatively difficult. This is because separate words in memory cannot be updated independently. This results in a problem when a data structure is changed, even if it is only a minor modification, the whole structure has to be copied and rewritten back to memory. This appears to be a major flaw in data driven architectures that hinders their use in numerically intensive applications [GAJ-82]. However, some data flow languages can update structures without copying, by use of I-structures.

1.3.5 Functional Languages.

In the 1930’s Alonso Church proposed the Lambda calculus as a means of formally reasoning about equations [CHU-36]. This lead to the possibility of mechanically evaluating expressions. Rosser developed this work [ROS-82], but little was achieved practically until the 1960’s when McCarthy described the first programming language, LISP [MCA-60, WIN-81], that was loosely based on this mathematical framework.

Since then, other languages have emerged that are based on the Lambda calculus, or a similar algebra of programs. Such languages are know as functional languages (which are a subset of declarative languages). There are a number of properties (discussed in chapter 2) that make functional languages appear well suited to solve many of the problems raised by parallel processing projects. Functional languages form the basis for the discussion in chapter 2.

1.4 Conclusions.

In order to increase computational power of computers, much effort has been expended on the research and development of parallel machine architectures. In this chapter a very brief outline of some of the major classes of such machines has been presented.

The problems of programming parallel computers, in a conventional procedural language, were highlighted. Some alternative language types were mentioned and functional languages were introduced as a well suited candidate to solve some of the programming problems of a parallel machine regime.
Chapter 2. Functional Languages.

A program, in the context of a functional programming language, is built up from a set of function definitions. An example of a program, that performs a bubble sort on a list of numbers, is given below in figure 2.1. The language used is EASE, that is employed in the functional language simulator, described in later parts of the thesis.

```
{ 
  (program (sort list)) 
  ((sort l) (if (= l nil) nil 
    (insert (head l) (sort (tail l))))) 
  ((insert x l) (if (= l nil) (cons x nil) 
    (if (<= x (head l)) (cons x l) 
    (cons (head l) (insert x (tail l)))))) 
  (list (cons 99 (cons 23 (cons 1 (cons 102 (cons 4 (cons 3 
    (cons 9 (cons 1 (cons 7 (cons 2 (cons 5 nil)))))))))))))
}
```

Figure 2.1 An example functional program.

2.1 Some Major Features of a Functional Style.

The main building block of a program produced in a functional style, is the function. A functional program allows higher order functions, i.e. function valued parameters; so functions may be passed as parameters to other functions and may be returned as results from other functions. This concept is sometimes described as functions being treated as "first class citizens".

 Obviously this sort, given above, could have been coded in a conventional imperative language, but there are several important differences. The "variables" cannot be updated, since in functional programming there is no notion of the assignment statement; once they have been given a value they cannot be changed. The order of the function definitions is irrelevant, "insert" could just as well have been defined before "sort". There is no explicit order of execution implied in the program, unlike the situation found with imperative code, (though in functional languages data dependencies force execution order, and so there is a degree of implicit ordering).

2.1.1 Referential Transparency and Parallel Processing.

The lack of the assignment statement and no explicit ordering of program execution, are the two important features of functional programming that lead to the notion of referential transparency [FIE-88]. This is the property that the value of an expression denotes a single value, not affected by its computational history. Evaluation of the expression may change its form, but not its value. This contrasts with imperative languages, which display referential opacity (i.e. the value of an expression may depend upon the history of the computation up to the point at which the expression is evaluated). Referential transparency serves to stabilise variables so that finite mathematical techniques can be employed to analyse code fragments, so facilitating the implementation of formal program proofs.
Section 2.2 Architectures for Supporting Functional Languages

A most important feature of referential transparency is, perhaps, the removal of all side-effects. This makes programs less likely to suffer from "bugs" since there is no mechanism in which algorithmically separated sections of code interact with each other, via a global machine state. This is commonly seen where bugs (places where the behaviour of the program departs from its specification) arise in imperative code due to the unforeseen interaction of two variables. These effects are limited by modular programming, but the root cause of the problem, namely the assignment statement, is not eliminated by these techniques. Functional languages offer such a complete solution to this problem.

The ramifications of referential transparency go further than facilitating the formal verification of code; it enables a programmer to be freed from having to control program flow explicitly, along many different paths in a parallel machine. This is because any function can be evaluated at any time, or in any order. There is now an assurance that without side-effects, once the operands of a function become available, the function is guaranteed to evaluate to the same answer at any stage in the computational history. It is obvious that this feature, combined with the Church Rosser property [CHU-36] (that informally stated says that all reduction sequences, of the same code fragment, that terminate do so with the same result), facilitates parallel execution of programs. This is because separate processing elements can work on different parts of the algorithm without interfering with one another. Further more, since functional programs are amenable to formal mathematical manipulation and transformation, a program can be written in a clear and concise fashion and then manipulated, with confidence, to maximise execution efficiency.

Another interesting feature is lazy implementations [PEY-87a] of functional languages, that have the ability to handle some infinite, or non-terminating data structures. Lazy evaluation implies that arguments are only evaluated as they are required. An example, taken from [FIE-88], is given below of a function that returns its first argument if that is less than ten, otherwise it returns the second.

\[
\text{f}(x, y) \leftarrow \text{if } x < 10 \text{ then } x \text{ else } y;
\]

If we now evaluate the expression;

\[
\text{f}(4, \text{<non-terminating expression>})
\]

since only the first argument, 4, is required to produce the result, no attempt is made to evaluate the non-terminating expression, so the value 10 is returned as the correct result. In an eager implementation an attempt would be made to evaluate the expression, resulting in a non-terminating result.

2.2 Architectures for Supporting Functional Languages.

The most common implementation technique, which is specifically designed to support functional languages, is graph reduction. This is briefly introduced below.

2.2.1 Graph Reduction.

As has been seen, functional programs are built from a set of function definitions. The program is run by applying the topmost function to its arguments. In all but the most trivial of examples, this invocation causes other functions to be evaluated. In this way a graph is
spawned, with functions at the nodes and data paths at the edges. As an example, a recursive functional program to evaluate the factorial function is given below, along with a simplified representation of the graph that is created and evaluated throughout the computation of factorial (3). This graph is presented in figure 2.2.

\[
(fact \ n) \ (if \ (= \ n \ 0) \ 1 \ (* \ n \ (fact \ (- \ n \ 1))))
\]

in (fact 3)

This program for factorial (3) is written in a general functional style. In pseudo-English we might describe the function thus:

defining the factorial of \( n \); if \( n \) is nought then return unity else

multiply \( n \) by the factorial of \( n \) minus one.

The graph can be seen to grow and then shrink to a final printable result as the reduction steps occur. The basis of graph reduction is that arguments of functions are shared whenever possible. There are a number of advantages in this:

(a) the argument may be massive, so copying would waste heap space;

(b) the argument may contain redexes (reducible expressions), and so copying could lead to wasted work since we would be duplicating the number of reductions performed, if both arguments were subsequently required.

Thus graph reduction proceeds with the substitution of pointers to function arguments and a graph is formed. The example given above, in figure 2.2, is simplified. For an in-depth discussion of reduction see [PEY-87a].
Section 2.2 Architectures for Supporting Functional Languages

A common implementation technique uses combinators [TUR-79]. These are employed in the functional language implementation, described in chapter 9. As an alternative to combinators, a more sophisticated technique, supercombinators, may be used [HUG-82a].

2.2.2 Alternative Approaches to the Implementation of Functional Languages.

There are a number of different approaches that have been explored for the implementation of functional languages. One is the so called S.E.C.D. machine, that employs an environment-based computational model, based around four stacks, eponymously, S, E, C and D. This environment provides direct access to the free variables in any lambda abstraction, solving the free variable problem [FIE-88]. This machine was built to run McCarthy's LISP language [MCA-60] and was designed to run LISP on a serial von Neumann machine with the addition of the S.E.C.D. environment stack to facilitate compilation of lambda expressions containing free variables [LAN-82, HEN-80].

Data flow is a method in which the model of computation is to view the data as an active entity, that flows through a collection of static transformers or functions [FIE-88]. The most common implementation of data flow is the model in which the operators wait until the required data flows are passed and then operate on that data, this is termed data-driven data flow.

Figure 2.2 Simplified representation of the graph reduction process.
Chapter 2. Functional Languages

An alternative to graph reduction is string reduction. Graph reduction relies on sharing arguments, the alternative of copying arguments leads to a tree structure, and is known as tree (or more commonly string) reduction. The overheads of string reduction are normally considered prohibitively expensive, though Mago's machine [MAG-80] implements string reduction and relies on massive parallelism to overcome the inherent inefficiency of this type of implementation.

2.3 Applications for Functional Languages.

Functional languages offer possible solutions to many problems that have afflicted procedurally programmed systems. They offer an alternative that may prove viable as a general purpose programming methodology. There are, however, a number of applications to which functional languages seem particularly suited.

An obvious application for functionally programmed systems are for safety critical situations, such as “fly-by-wire” aeroplane control [MEL-82], or safety equipment in the oil industry. The ability rigorously to prove programs are correct is extremely important in such applications.

As has been mentioned, the property of functional languages possessing referential transparency offers the possibility of parallel architectures that are both more easily programmed and more efficient than their procedurally programmed counterparts. This is one of the reasons why there is a growing effort to develop these languages for fifth generation (parallel) computers [SAR-88].

Functional programmed systems are also of interest in both real time and artificial intelligence environments. This is particularly true for those that are supported by a parallel architecture, where a uni-processor approach would not provide sufficient computational power to meet the needs of the application. Despite the difficulties associated with garbage collection (see chapter 3) functional languages have been used for intelligent robotics programs (a real time application) and algebraic manipulation systems (an interactive application) as early as the mid 1970's [STE-75, WAD-76].

Functional languages are also used in other applications with real time constraints. An interesting recently developed functional language, is a one named Fugue [DAN-91]. This has been developed for sound synthesis purposes. It uses the properties of functional languages to manage complex musical structures, and employs a temporal notation to handle sound duration specifications. Nilsen also suggests that applications such as computerised medical instrumentation, robotics and voice recognition, that are currently programmed principally in lower-level languages, such as C and assembler, would be enhanced by the use of higher-level functional languages, if these languages could meet the real time responses necessary.

Digital signal processing is an area where very high processing rates are demanded. It has been pointed out by Burnett [BUR-87] that, though signal processing covers a wide range of problems, there are a number of common algorithms that dictate the principal requirements of a signal processor. These are namely: the discrete Fourier transform, convolution and recursive filtering. Burnett believes, along with others, that reduction architectures are "almost ideal in principle" for digital signal processing applications, if the inefficiencies traditionally associated with functional language implementations (specifically, the garbage collection problem) can be eliminated.
Sanders [SAN-89] has made an interesting study of comparing the use of three functional languages (Hope, Miranda and ML) for the implementation of a real problem in an industrial environment. The solutions obtained in each case are compared to ones obtained for systems written in Ada and C. The results showed that compared to Miranda, the development time (for 1503 lines of Ada) was about 5 times greater for the Ada system. The code for Miranda was at least three times as compact, in terms of number of lines, compared to Ada. This demonstrates the greater expressive power of a functional, as compared to procedural language. Execution speed for Hope and ML were comparable to the times for C (but Miranda was much slower). He concludes that functional languages offer many advantages for the commercial environment, provided that the systems are developed further. Other studies between declarative languages are made in [HAI-89, FLE-90].

2.4 Conclusions.

This chapter introduced the major facets of the functional style of programming. The features highlighted were shown to facilitate programming machines based upon parallel architectures.

Some of the major machine architectures, that are designed to support functional languages, were introduced. The most common architecture, graph reduction, was briefly outlined.

The chapter concluded with a summary of some of the important application areas for functional programming.
Chapter 3. Introduction to Garbage Collection.

The example of the reduction of a factorial function, given in chapter 2, figure 2.2, shows a typical way in which a graph changes during reduction. The building of new parts of the graph demands that new cells are allocated from memory. Cells are obtained from a memory that is organised as a heap. The term heap implies that from a graph reduction point of view, in this type of storage, actual absolute cell addresses are unimportant. The salient feature of this memory organisation is the pointer structure in the reduction graph.

During the reduction process pointers to cells will be discarded. When a cell no longer has any pointers referencing it then it is said to be garbage, i.e., it plays no further rôle in the graph reduction process. When new cells are allocated to the graph, these can be sourced either from the pool of unused cells in the heap, or taken from a list of reclaimed garbage cells, that is termed the free list. If no reclamation of garbage cells occurred then, with large applications, the heap would be eventually exhausted and computation prematurely cease.

Garbage cell identification and recovery is complicated by the fact that there may be many pointers to a particular cell and pointers may form circular structures. It is the job of the garbage collector to perform the task of identification and recovery of all redundant cells.

Before exploring the different options available for garbage collection, the next short section will give a simple example of how the graph may actually be represented within the machine.

3.1 Concrete Representation of the Graph.

As an example, consider the simple expression (+ 4 2). This is simply the application of the function ‘+’ to the two arguments ‘4’ and ‘2’. This expression can be represented as an abstract syntax tree, as shown in figure 3.1.

```
      @
     /\  
    @  2
   /  /
  +  4
```

Figure 3.1 Abstract syntax tree.

The symbol ‘@’ is used to denote that this is an application node. This is an example instance of the tag of a node.

There are a number of different ways in which abstract syntax trees can be represented within the machine, in essence they are similar to the scheme given below.

Each node in the tree is represented by a small contiguous area of store, called a cell. The cell comprises a tag that identifies its particular type, (e.g., application, number, built-in operator, CONS cell etc.), and zero or more fields. The most common implementations have
fixed-size cells with two fields, but variable-sized cells with an arbitrary number of fields are possible.

For a cell containing two fields, a cell may be represented thus:

\[
\text{Tag} \mid \text{Field1} \mid \text{Field2}
\]

**Figure 3.2** General representation of a node in the reduction graph.

The fields may contain either the address of another cell, in which case it is a *pointer*, or alternatively it contains an *atomic* (non-pointer) data value. A possible concrete representation of the expression \((+ 2 4)\) is given below (this is a *boxed* representation, see section 3.5 and [PEY-87a]).

![Concrete representation of a reduction graph](image)

**Figure 3.3** Possible concrete representation of a reduction graph.

The N tag signifies a number and P a built-in function. There are a number of variations based upon this theme [CLA-80, RIC-85], but essentially the architecture is that of the fixed length cell with tag nodes.

The functional language interpreter used in the simulator, described later in the thesis, uses fixed length cells.

### 3.2 The Significance of Garbage Collection

It is becoming increasingly clear that garbage collection is one of the most important efficiency issues that must be addressed in any functional language implementation. Studies with large implementations in LISP have shown that 10 to 40 per cent of processor time is actually spent in garbage collection [STE-75, WAD-76]. Unless the question of garbage collection is resolved, then functional language implementations will always be prohibitively inefficient, as compared to traditional imperative languages.

Morris [MOR-83] has stated, after much practical experience with functionally programmed systems, that the cost of garbage collection is the most important problem, particularly for systems with real time constraints. More modern techniques in garbage collection have ameliorated this problem to some extent, nevertheless it remains a major obstacle to the efficient implementation of functional languages, particularly in real time systems.

The various methods of processing garbage within the heap are examined below.
3.3 Mark-Scan Garbage Collection.

This is the commonest form of garbage collection. A mark-scan algorithm operates in two phases. The first phase is that of marking all cells in the entire accessible structure of the heap. A mark bit is set on each cell that is reachable, i.e., is not garbage. This can be achieved by tree-traversal algorithms [KNU-68]. The standard methods use a stack, that can grow to considerable size, during the marking phase, though the stack depth can be limited using a fixed length stack, at the cost of increased processing [KNU-68]. A development of the marking algorithm is due to both Deutsch and the pair Schorr and Waite (independently of each other) [SCH-67]. This uses a link-reversal technique to eliminate the memory overhead of the stack with only the inclusion of a single mark bit for each cell.

In the standard tree-traversal method each cell is visited twice during the marking phase, with the Deutsch-Schorr-Waite method, this is increased to thrice. This obviously leads to increased processor overhead. Schorr and Waite proposed a hybrid scheme that uses a fixed depth stack wherever possible, but employs the link-reversal technique when this overflows [KNU-68].

The next phase of the garbage collection is the reclamation of garbage cells, or scan phase. The simplest method for achieving this is to sweep linearly the entire heap. Any cells that have not been marked as accessible during the mark phase are reclaimed and appended to the free list, administered by some form of storage allocator.

Compaction of the memory can be achieved at this stage, rather than simply appending the newly reclaimed cells to a free list. This is done by scanning the memory twice. In the first scan two pointers are used, one starting at the bottom of the heap and the other at the top. The bottom marker is incremented until it points to an unmarked cell; the top pointer is then decremented until it points to a marked cell. The contents of the marked cell are then swapped with the unmarked one, a pointer to the new cell is placed in the old and the mark bits are turned off. When the two pointers finally meet, all the cells have been compacted in the lower section of the heap. The second sweep occurs in the compacted area to readjust pointers to cells that have been moved.

It should be noted that with this method of compaction, cells that point to one another will not necessarily occupy contiguous positions in the heap after compaction, i.e., the compaction is arbitrary in nature. Other, more sophisticated (and much more processor intensive) algorithms can be employed to place nodes which reference one another in contiguous positions. This achieves linearising compaction.

3.4 Copying Garbage Collection.

Copying collection was first described by Minsky [MIN-63], according to Baker [BAK-78]. This method of collection works by copying the entire accessible structure in the heap from one portion of the heap (from-space) into another area of memory (to-space). In principle, as the entire accessible graph structure in the from-space is traversed, each referenced cell is moved to the to-space. The original reference to the cell is adjusted so that it points to the moved cell, in to-space. In addition, the new address of the cell, that has been moved, is then left behind in the from-space, this is termed a forwarding address or an invisible pointer. This address is used to update cells that hold pointers to this moved cell, that may be encountered later in the collection process. At the end of the collection cycle, the cells remaining in the from-space must be garbage and so can be discarded.
As the copying occurs, the cells being moved to the to-space are placed adjacent to one another. As a result, once copying is complete, there is a contiguous area of new cells that can be allocated as reduction proceeds. There is therefore no need to maintain a free list of cells. When the to-space is filled up, the two spaces are then flipped, (to-space becomes from-space and vice versa). The process is then repeated.

A refinement of this scheme is due to Lieberman and Hewitt [LIE-83]. The scheme involves splitting the heap into \( n \) regions, \( n-1 \) of which are active. Each region is then collected by swapping it into the one, spare inactive region. Each region is also given a generation number, that indicates when the space came into use. Lieberman and Hewitt made and used the observation that, the newer the region the more likely it is to contain garbage. This is utilised so that newer regions are collected more often than older ones. Another empirical observation is made in this technique, that is most references across region boundaries tend to be to older regions. This is made use of by keeping an entry table of pointers to a particular section from older regions. When the region is garbage collected, only younger regions have to be searched, for pointers into this region.

Baker [BAK-78] has developed a real time version of the standard two space copying collector. This is described in greater depth in chapter 5

Morris [MOR-78], describes a collector that does not require a spare region, into which cells are swapped. The algorithm he has developed also ensures that cells remain in the same order as in the free-space. Unfortunately, the technique is not suitable for real time collection, because the mutator cannot claim new cells until the whole collection process has ceased.

3.5 Reference Counting Garbage Collection.

This method relies on an extra field associated with every cell in the heap. This is the reference count field. This field is maintained with the number of references to that particular cell, i.e., the number of pointers to the cell. Whenever a pointer is added to the graph, which points to the cell in question, then the reference count field is incremented. When a pointer is discarded to the cell then the reference count is decremented. If the reference count falls to zero, then the cell must be garbage since there are no references to it [COL-60].

Figure 3.4 shows diagrammatically reference counts in a typical part of a graph, the last field in each cell holds the reference count. By way of contrast to section 3.1, an unboxed representation of the graph is used here, where data objects are completely described by a single field. This is unlike the boxed representation of figure 3.3. In this representation each field carries an extra marker flag, that is the pointer-bit. This distinguishes the field as a pointer, or data object. This field is shown greyed in figure 3.4.
Chapter 3. Introduction to Garbage Collection

The end fields are reference counts

This node is garbage, it has a zero reference count.

Figure 3.4 Part of the reduction graph, showing reference counts.

The garbage collector maintains the reference counts for each cell as pointers are added and deleted. Once the reference count of a cell has fallen to zero then the cell can be added to the free list and reused. The bottom-most cell shown is garbage, since its reference count is zero.

3.6 Other Methods of Dealing with Garbage.

Burnett [BUR-87] uses a different form of garbage collection in his Mkl architecture. This machine was developed for the graph reduction of the functional language FP [BAC-78]. He uses two spaces, termed scratch space and save space. At compile time, the compiler determines which results are to be saved or placed in scratch space. The user inserts a mark next to a function to indicate that the result should be saved. When so marked, the computed result is placed in save space at run time. If the result is only required as an intermediate result, then scratch space is used and the result is collected as garbage after use by the next function. The two logical areas, nominated as scratch space and save space, are swapped for nested calls to functions, so that intermediate results are saved correctly.

The method is very limited in its application, since it would be difficult to implement in languages other than FP [BUR-89]. It also suffers from the disadvantage that the programmer has explicitly to control garbage collection.

Wadler has suggested a method of avoiding garbage collection altogether, based on a language founded on linear types [WAD-90]. The technique effectively performs memory allocation in advance. Such techniques are referred to as compile time garbage collection. More recently a paper by Wakeling and Runciman [WAK-91] gives some very disappointing results for compile time garbage collection, based on Wadler's methods. They point out some inherent failings in the method, due to lost optimisation at compile-time and extra data structures that have to be created at run-time.
3.7 An Appraisal of the Various Garbage Collection Strategies.

Peyton Jones [PEY-87a] lists the most important criteria when assessing a particular garbage collection technique. These are summarised below:

(a) What are the space and time overheads?
(b) Is compaction supported? An issue that is important in variable-length cell systems.
(c) Is the method efficient in sparsely used heap and virtual memory?
(d) Can it operate in parallel with graph reduction? (This is sometimes termed noisy heap operation).
(e) Can it respond within bounded time? This is vital for real time systems.
(f) What is the effect of heap occupancy?
(g) Can it recover garbage cyclic structures?
(h) How does it perform in a multiple processor element environment?

The next three sections compare the principal types of garbage collector, against the criteria set out above.

3.7.1 Mark-Scan Collection.

The space overheads are a mark bit in each cell and a stack, used to guide the recursive tree-walk in the mark phase. This stack is bounded in size only by the number of cells in the heap, but, as has been explained, this can be limited in size or eliminated altogether by using the Deutsch-Schorr-Waite pointer-reversal algorithm (at the expense of a further mark bit per cell).

The CPU overheads are relatively light, although they increase with pointer-reversal techniques.

Compaction can be achieved in the scan phase, but again there is a processing overhead to pay.

This technique gives very poor performance in situations where the heap is sparsely used, or virtual memory is employed. This is because all accessible cells are visited, not just those in immediate use. Without compaction, in a sparsely used environment, poor paging behaviour becomes a severe problem. The arbitrary compaction, that is usually supported in mark-scan systems, does little to alleviate this problem.

Algorithms have been presented that allow collection to occur at the same time as the useful job of graph reduction. With the standard technique garbage collection is carried out when the heap is almost full, then there is an “embarrassing” pause whilst the collection occurs. On large systems this can take minutes and obviously is both inconvenient for the
user, and rules out any form of real time operation. It has also proved very difficult to forecast when collection will occur. The parallel algorithms, [STE-75, KNU-68, DIJ-78], are considerably more complicated than the standard mark-scan techniques and none have been shown to be guaranteed to respond in bound time. They obviously demand much higher processor overheads (due to processor-collector synchronisation etc) than their "stop-the-world" counterparts. These are discussed in greater depth in chapter 5.

As the heap fills up, the performance of mark-scan techniques degrades sharply. This is because all accessible cells have to be visited in order to recover garbage cells, even though this may yield only a few cells.

Cyclic structures offer no special problems to mark-scan techniques.

Mark-scan collection within a parallel processing environment is difficult to achieve, especially when noisy heap collection is called for. This is because the action of the collector is essentially global, and so a large degree of "choreography" between processors (processor synchronisation) has to occur when collection takes place. This inevitably leads to some pause in the graph reduction, again making this technique unsuitable for real time applications. There have been algorithms developed that are designed to implement forms of mark-scan collection within distributed systems. These include ones by Hudak [HUD-83], Ali [ALI-84] and Hughes [HUG-85], among others.

3.7.2 Copying Collection.

In systems where virtual memory is not used then there is a 100 per cent space overhead, for simple two space collectors. This is greatly reduced in virtual memory systems, where parts of the heap space that are not being currently used are paged out. The processor overhead appears to be less than that of the mark-scan technique (when basic copying collectors are used).

Compaction is inherent in this method whilst the from-space to-space copying occurs. It can be easily arranged that cells that point to one another are placed in contiguous areas of the heap, so that linearising compaction occurs. Because of this, the performance in virtual memory systems is greatly improved, due to data structure locality facilitating paging operation.

The behaviour of this technique in sparsely used heaps is much better than mark-scan, since there is no sweep phase throughout the whole heap. However, in heavily used heaps the performance degrades, as with mark-scan, since all accessible cells have to be visited.

Copying, in its basic form, demands that the reduction process halts, causing the same problems as with mark-scan. However, noisy heap operation can occur with an incremental copying scheme [BAK-78]. This modified scheme greatly increases both CPU time and heap space overheads.

This method, like mark-scan, is also fundamentally global throughout the heap, this leads to the same objections for employing it in parallel architectures, namely a high degree of processor-processor interlock synchronisation is required as collection occurs.

Also, like mark-scan, cyclic structures present no special problems to a copying garbage collector.
3.7.3 Reference Counting.

This method carries both a large space and processor time overhead. The space overhead is taken up with the extra reference count field, associated with every cell. This, in theory, has to be long enough to hold a number that is as large as the maximum numerical total of cells in the system. This problem can be alleviated, as will be demonstrated later. The processor overhead is large, and is due to the maintenance of the reference count field.

In both sparsely and heavily used heaps, the technique is considerably more efficient than the other two methods. This is because it is inherently local in its operation, only visiting a cell to maintain a reference count when there is a pointer operation. Reference counting does not require the whole available cell structure to be visited at one time.

The locality of operation has other considerable benefits, these include an ability to work easily in an active heap environment, as reduction proceeds. This requires no special adaptation of the technique.

The basic reference counting scheme is also essentially inherently distributed in time, (see chapter 5 onwards, concerning real time garbage collection) so this is obviously well suited to real time applications.

No inherent compaction occurs with this scheme, but this causes fewer problems than would be the case with mark-scan or copying, due to the fact that reference counting acts locally within the graph. If a virtual memory system is used, then the use of a separate linearising compactor is not precluded by the use of a reference counting garbage collector.

Reference counting exhibits some difficulty in garbage collecting vector nodes (aggregate structures that have only a single reference count). This is because the constituent members of such a structure will not be collected as they become garbage, but will remain uncollected until the overall reference count falls to zero.

A major draw-back of reference counting is that it cannot recover cyclic structures within the graph, when these have become garbage. Methods have been developed to circumvent this problem. These will be discussed below.

3.8 Choice of Garbage Collector.

The global nature, of the basic methods of both mark-scan and copying collectors, make them inherently unsuitable candidates for garbage collection within parallel architectures. They can be heavily modified in order to circumvent some of the problems raised by the need for excessive processor synchronisation at garbage collection time. This, however, tends greatly to increase the amount of processing time that has to be spent on collection.

Reference counting seems ideal for any parallel architecture, due to its inherently distributed nature, if it were not for two major shortcomings: it has a large processor and space overhead, and it cannot recover cyclic structures. Until recently, these two drawbacks have ruled out reference counting, in all but a few specialised cases. Advances have now been made in the basic reference counting technique. These will be discussed in the next section.
Chapter 3. Introduction to Garbage Collection

3.9 Developments and Variations of Reference Counting Garbage Collection.

There are a number of developments that have been made in the basic reference counting garbage collection technique. Some of the most important ones are explored below.

3.9.1 Distributed Systems and Weighted Reference Counting.

One of the major drawbacks of reference counting is the fact that every time a pointer is added or deleted from the graph a reference count has to be amended in the target node of the pointer operation. In distributed, multi-processor systems, this leads to an undesirable increase in the required communications bandwidth, between the collector and a heap (that may be remotely sited). In such cases this requires expensive bus time to perform the count modification.

Hughes has introduced a mechanism of non-local pointers that makes distributed memory garbage collection more efficient [HUG-85]. This technique maintains all pointers, within a local heap, as ordinary local pointers. Inter heap pointers are implemented via special remote pointer cells, which are never duplicated. This simplifies reference counting in a distributed system. A system similar to this is proposed by Ichisugi and Yonezama [ICH-91]. This also employs the notion of tracking strongly connected components (S.C.C.s) to recover redundant cyclic structures (see section 3.10).

Another approach has been taken by Thomas [THO-81] and Bevan [BEV-85], who independently developed a method for placing the reference count within the pointer to a cell, with the cell itself holding a standard count. The scheme works by associating a weight with each pointer. The method ensures that the count held within the cell reflects the sum of the weights of the pointers pointing to it.

Figure 3.5 shows the method in operation. Initially a node is created with a given reference count, 1024 in this case. The first pointer to it is assigned this weight. When the pointer is duplicated, there is no need to access the cell, the weight of the original pointer is simply divided between itself and the new pointers. When a pointer is deleted, the cell count must be decremented by the corresponding weight of the deleted pointer. This ensures that the sum of the weights of the pointers to a cell, is always reflected by the reference count of that cell.

In this way when a pointer is duplicated, the reference count of the target node does not have to be accessed. This cuts the required bandwidth between the collector and a remote heap.

The technique has the added advantage that it avoids race conditions in a distributed system. These arise when a pointer deletion request to a cell, arrives before a remote pointer addition to the same cell. This may cause erroneous deletion of the cell, before the pointer addition can occur.
Section 3.9 Developments and Variations of Reference Counting Garbage Collection

3.9.2 The Collection of Cycles by a Hybrid Technique.

A problem arises when the graph incorporates cycles. These arise primarily from the use of recursive functions, a particularly important feature in the functional style. Dennis argues that cyclic structures should be banned [DEN-81], however, the efficiency of any interpreter for a functional language is lowered if cycles are not permitted to represent recursion. It has also been shown that multi-pass programs can be transformed into more efficient programs by the use of cyclic structures [BIR-84]. The penalties of possible loss of efficiency and elegance of programming style, preclude the banning of cyclic structures, in most functional language implementations.

The actual problem is illustrated by the simple example given in figure 3.6. Four cells are illustrated, that have a pointer structure in such a way that they reference one another. The reference count is shown in the final field of each cell. If the external reference, marked *, is deleted, the entire cyclic structure of four cells becomes unreachable from any cell external to this cycle. Because the structure is now not externally referenced, all the cells that are part of the cycle must be garbage. However, each of the cells remains with a reference count of unity, and hence are not recognised as garbage. This results in a failure in appending them to the free list of cells. This implies that these cells will never be reused, and so, in the pathological case, heap overflow could occur whilst there are still a considerable number of cells that are actually free for reuse, but have not been recognised as such.
Chapter 3. Introduction to Garbage Collection

Figure 3.6 A cycle within the reduction graph.

As has been mentioned above, both the programming style and compiler techniques can be altered to eliminate cycles from the graph, but the penalties appear to be prohibitively expensive to implement this solution.

A possible solution, to the problem of collecting cycles, involves the use of hybrid collectors. These employ a simple reference counting collector, that performs most of the garbage collection. When memory is nearly exhausted, a mark-scan, or copying collector, is called on to recover any lost cycles in the graph. This method has two main advantages. The first is that for most of the time reference counting (with all its inherent distributed time-space properties) is employed. Also an elegant form of memory optimisation, within reference counting, is simple to include in this scheme. This is the use of limited width reference count fields, in which the count field is only a few bits wide. If the count exceeds the width of the field, then it is given some (reserved) value, interpreted as infinity. This cell must then be collected by standard means, of mark-scan or copying. The reference count field is not updated until collection occurs, at which time it is reset to zero. This overflow of the count field is not very common (if the width is chosen carefully), because most cells in the graph tend not to be shared [STO-84].

The disadvantage of the hybrid approach is that when the mark-scan or copying phase of collection occurs, all the problems associated with these techniques still remain. This results in the hybrid method appearing unsuitable for real time, or multi-processor, applications.

It is apparent that another adaptation has to be employed if reference counting can be contemplated for use in real time systems.

3.9.3 The Single-bit Reference Count.

An extreme form of the limited-width reference count is the single-bit reference count, where overflow occurs if the count exceeds one [WIS-77]. This technique elegantly combines the rôle of mark bit, and single bit reference count. The scheme also employs a small cache that contains the reference counts (greater than unity) of some structures within the graph.

A further optimisation of the one-bit scheme is to place the reference count in the pointer to the cell, rather than storing it in the cell body, in a similar fashion to weighted reference counts (see section 3.9.1). When a cell is first allocated, it has a pointer to it that identifies it as being uniquely referenced (i.e., having an effective reference count of one). When a unique
reference is duplicated both copies of it are then marked as shared. On discarding a pointer, if it is marked as being a unique pointer, then the cell is recovered as garbage, otherwise it is left for the mark-scan collection phase. This method eliminates the need to amend reference counts in the heap when a pointer is duplicated. This scheme does not address the problems associated with the mark-scan phase.

### 3.9.4 Collection of Cycles in Reference Counting by Other Techniques.

Work has been carried out on modifications to the basic reference counting technique, so that it can handle circular structures. Friedman and Wise developed a method that could handle static cyclic structures, such as those created by recursion, but cannot deal with dynamically created structures [FRI-79]. Their method is also limited in its application, being designed for pure LISP type systems.

Bobrow gives another method that can handle all cyclic structures, but needs the support of the programmer to identify the structures [BOB-80]. Programmer participation in cycle identification is clearly a drawback.

Brownbridge [BRO-85] has also developed a method that recovers all cyclic structures, but, according to Hughes [HUG-85], the work published to date contains an error, so its status is unclear.

Axford has developed a method for dealing with cycles [AXF-90], that is based on Friedman and Wise’s, but extended to deal with more general circumstances. The system is designed to work directly on the graphical representation of the source functional program, rather than on the program representation once it has been translated into combinator code. Axford defines two types of pointer: weak ones, which if removed from the graph will leave it acyclic; and strong ones, via which all nodes are reachable from the root (by tracing strong pointers only). He also states that there must be exactly one entry node per strongly connected component (S.C.C.), that is the root of the graph if it is a member of that S.C.C. The reduction graph is then transformed to comply with these requirements. Reference counting is then used to trace strong pointer operation only, that deals with the acyclic graph, so eliminating the problem of collecting cycles.

Unfortunately, whilst the method offers a simple approach to using reference counting to collect cycles, it is not suitable (for efficiency reasons) for use with combinators [PEY-87a], which is a common and useful way to implement graph reduction.

The most promising development is by Hughes [HUG-82b], whose method is based on that by Bobrow [BOB-80]. Work carried out before this by Aerts [AER-81] also uses S.C.C.s for the recovery of redundant cycles. The method is essentially the same as that of Hughes.

### 3.10 Hughes’ Method of Reference Counting.

Hughes’ method relies both on graph theory, and the characterisation of the behaviour of cycles within the graph as reduction proceeds. The key idea is to divide the graph into its strongly connected components (S.C.C.s).
Chapter 3. Introduction to Garbage Collection

A graph is said to be strongly connected if, for any two nodes A and B, there is a path from A to B, and vice versa. A strongly connected component of a graph is a maximal strongly connected subgraph. It is apparent that if one node in the strongly connected component is accessible, then all the nodes within the structure are.

If we consider the reduction graph as an unlabelled directed graph, then we may denote the path from a node a in the graph, to a node b by a $\rightarrow b$. The relation $\rightarrow$ is a reflexive, transitive relation. We may now define an S.C.C. more formally for two nodes within the graph m and n.

$$\text{SCC}(n) = \{m \mid m \rightarrow n \land n \rightarrow m\}$$

and

$$m \in \text{SCC}(n) \iff n \in \text{SCC}(m)$$

Hughes coalesces the nodes in each strongly connected component to form a derived graph, that is acyclic, and is therefore amenable to reference counting techniques of garbage collection. An example of a graph, containing cycles, and its derived acyclic form (whose nodes comprise strongly connected components), is given in figure 3.7. The S.C.C.s are outlined by dotted lines in the original graph.

![Original graph and Derived graph](image)

Figure 3.7 Formation of the derived graph.

The derived graph is always acyclic. This follows from the definition of a strongly connected component. Let us consider two strongly connected components, X and Y, in the derived graph, such that there is a path both from X to Y and vice versa, (i.e., the derived graph contains a cycle). This implies that, in the original graph, there was a node in X connected to a node in Y, and vice versa. Thus both X and Y are in the same strongly connected component, which violates the previous assumption that the groups X and Y are disjoint.

Each group is given a separate group reference count. This contains a count of the number of pointers that are external to the S.C.C. in question. Every member of the S.C.C. contains a pointer to this count, that also serves to identify it as a member of that particular group. When the group count falls to zero, then all the members of the group can be collected.
as garbage. Figure 3.8 shows an S.C.C. with only one external pointer. Once this is deleted, the four members of the cycle can be safely collected, despite the fact that the individual *local* counts for the constituent nodes remain non-zero.

There are two phases in Hughes’ technique. The static phase deals with the graph before reduction commences, and the dynamic phase concerned with the graph during the process of reduction.

![Figure 3.8 An S.C.C. with group reference count.](image)

### 3.10.1 The Static Graph.

The strongly connected components of the graph can be identified using a Tarjan search [TAR-72]. The whole graph is searched by this method, before reduction commences. This allows the derived acyclic graph to be formed.

The basic algorithm (simplified from [TAR-72]) is presented in figure 3.9. (Algorithms are presented in a pseudo-code, which is Modula-2 like). Informally stated, the algorithm works by visiting each node in a depth-first manner. As each node is visited it is placed on a stack, and assigned a depth first search number \( (dfs) \), and a *lowlink* value (which is initially set to the value of \( dfs \)). The value of *lowlink* is reduced if one of node’s descendants is already on the Tarjan stack, when a node is first visited by the procedure *visit*. If the value of *lowlink* is not so reduced, then the procedure *emitgroup* is called. This emits all nodes on the Tarjan stack with *lowlink* greater, or equal to, the value of \( dfs \) for which it was called \( (emitterhresh), \) to form a separate S.C.C.

The search makes only one traversal of the graph, it is therefore linear in complexity in the number of nodes and edges in the graph. See [TAR-72] for a full explanation of the action of the algorithm.
Chapter 3. Introduction to Garbage Collection

The `emitgroup` routine calculates the value of the group reference count, for each group it emits. This is achieved by setting this to the value of the sum of the local reference counts of the constituent nodes of the S.C.C., minus the number of internal pointers in the group. It is for this purpose that local reference counts, as well as group counts, are maintained in the graph. For each node emitted into the new S.C.C., the `emitgroup` procedure ensures that the necessary pointer to the group count is set up correctly (see chapter 6 for a fuller explanation of `emitgroup`).

```
PROCEDURE visit (n);
BEGIN
  IF n.onstack THEN
    RETURN;
  ELSE
    INC (dfsn); (* dsfn is the depth-first search number *)
    d := dfsn;
    n.lowlink := dfsn;
    n.number := dfsn;
    spush (n); (* Places the node on the Tarjan stack and sets the onstack flag *)
    IF the node n is an application node THEN
      REPEAT
        child ∈ children of node n;
        IF child.onstack THEN
          IF child.number < n.number THEN
            n.lowlink := minimum_of (n.lowlink, child.number);
          ENDIF;
        ELSE
          IF samegroup (child, n) THEN
            visit (child);
            n.lowlink := minimum_of (n.lowlink, child.lowlink);
          END (* IF *);
        END (* IF *);
        UNTIL all children have been visited;
      END (* IF *);
    END (* IF *);
    IF n.lowlink = n.number THEN
      emit_thresh := d;
      emitgroup;
      END (* IF *);
  END visit;
```

Figure 3.9 The Tarjan search algorithm.

3.10.2 The Dynamic Graph.

Hughes' collection scheme [HUG-82b] makes use of the various special properties of a graph that is being mutated by reduction, these attributes are examined below.

The process of reduction of a root node, r, can be thought of as comprising several disparate processes; the allocation and initialisation of new nodes, adding pointers from r to its new subnodes, and deleting pointers from r to its old subnodes. The processes of addition and deletion of pointers actually happen simultaneously in a real machine, since pointers are deleted by overwriting them with new pointers. The reason for considering an abstraction from such a concrete machine, is to ensure that non-garbage nodes are not incorrectly collected, so notionally all new pointers are written before old ones are deleted. This avoids the necessity to consider race conditions within the collector.
Section 3.10 Hughes' Method of Reference Counting

It is possible to characterise the behaviour of the graph during some of the separate reduction processes. We include a number of theorems and proofs, developed from the outlines in [HUG-82b, 87], these are given below alongside an informal discussion of each stage of the reduction process. We shall use the convention that symbols before reduction, where appropriate to identify such, are primed, whilst those afterwards are not.

(i) Addition of New Nodes.

Let $G$ be the set of pre-existing nodes and $N$ be the set new nodes that are to be added. Now $G$ is closed, that is:

$$n \in G \land n \rightarrow m \Rightarrow m \in G$$

First, we state that the existing nodes are not modified during this stage:

$$n \in G \Rightarrow (n \rightarrow m \Leftrightarrow n \rightarrow m')$$

It follows that the S.C.C. structure of the old nodes is not affected by this stage:

**Theorem 1**

$$n \in G \Rightarrow SCC(n) = SCC'(n)$$

**Corollary:** the strongly connected component structure of the new nodes is independent of the old nodes.

$$n \in N \Rightarrow SCC(n) \subseteq N$$

**Proof of corollary**

It is clear that if this were not the case then there would be some element $g$ of $G$ in $SCC(n)$ and so $n$ would be in $SCC(g)$ which contradicts the theorem.

The theorem tells us that the S.C.C. structure of the pre-existing graph is not modified by adding new nodes. The corollary tells us that it is only necessary to examine new nodes in order to find the S.C.C.s of the new nodes.

(ii) Addition of Pointers.

In this stage pointers are added from a node $r$ to the old or new nodes in $G \cup N$. we can state three conditions that are maintained during this stage of the reduction.

First, all new nodes become accessible from $r$.

$$n \in N \Rightarrow r \rightarrow n$$

(a)

This condition arises from the fact that all new nodes added are accessible from $r$. 
Chapter 3. Introduction to Garbage Collection

Secondly, no paths are deleted, so the only new paths go through r.

\[ m \rightarrow n \iff m \rightarrow n \lor (m \rightarrow r \land r \rightarrow n) \]  

(b)

Thirdly, the only nodes newly accessible from r are new nodes.

\[ n \in G \Rightarrow (r \rightarrow n \iff r \rightarrow n) \]  

(c)

This final condition implies that the only newly accessible cells for the entire computation are the newly added cells. This property guarantees that unreachable nodes remain unreachable. This ensures that garbage collection of a node that is unreachable is always safe, since it cannot become reachable again.

Before presenting three theorems concerned with the behaviour of the graph during pointer addition, we present two lemmata. The first generalises (c) and states that the only nodes newly accessible from any node are new.

**Lemma 1**

\[ n \in G \Rightarrow (m \rightarrow n \iff m \rightarrow n) \]

**Proof**

\[ m \rightarrow n \iff m \rightarrow n \lor (m \rightarrow r \land r \rightarrow n) \]  

by (b)

\[ \iff m \rightarrow n \lor (m \rightarrow r \land r \rightarrow n) \]  

by (c)

\[ \iff m \rightarrow n \]

since

\[ m \rightarrow r \land r \rightarrow n \Rightarrow m \rightarrow n \]

The second lemma states that the S.C.C. of an old node n is enlarged with those nodes m, newly accessible from r, such that an old path from m to r passes through n.

**Lemma 2**

\[ n \in G \Rightarrow \text{SCC}(n) = \text{SCC}'(n) \cup \{m \mid m \rightarrow n \land n \rightarrow r \land r \rightarrow n\} \]

**Proof**

\[ \text{SCC}(n) = \{m \mid m \rightarrow n \land n \rightarrow m\} \]

= \{m \mid m \rightarrow n \land n \rightarrow m\} 

= \{m \mid m \rightarrow n \land (n \rightarrow m \lor (n \rightarrow r \land r \rightarrow n))\} \]  

by **Lemma 1**

by (b)
Section 3.10 Hughes' Method of Reference Counting

\[ \{ m \mid m \rightarrow^* n \land n \rightarrow^* m \} \cup \{ m \mid m \rightarrow^* n \land n \rightarrow^* r \land r \rightarrow^* n \} \]

\[ = \text{SCC}(n) \cup \{ m \mid m \rightarrow^* n \land n \rightarrow^* r \land r \rightarrow^* n \} \]

The next theorem states that most of the graph is left unchanged by the addition of a pointer.

**Theorem 2**

\[ n \in G - \text{SCC}'(r) \Rightarrow \text{SCC}(n) = \text{SCC}'(n) \]

**Proof**

\[ \text{SCC}(n) = \text{SCC}'(n) \cup \{ m \mid m \rightarrow^* n \land n \rightarrow^* r \land r \rightarrow^* n \} \]

by Lemma 2

Let \( m \) be a member of the latter set, then

\[
\begin{align*}
    m &\rightarrow^* n \land n \rightarrow^* r \land r \rightarrow^* n \\
    \Rightarrow &\quad n \rightarrow^* r \land r \rightarrow^* n & \text{by (b)} \\
    \Rightarrow &\quad n \rightarrow^* r \land r \rightarrow^* n & \text{by (c)} \\
    \Rightarrow &\quad n \in \text{SCC}'(r)
\end{align*}
\]

which contradicts the assumption that \( n \in G - \text{SCC}'(r) \). Therefore the set is empty and

\[ \text{SCC}(n) = \text{SCC}'(n) \]

i.e. only the S.C.C. containing the parent node of the pointer addition (see later) is affected.

Theorem 3 states how the S.C.C. of \( r \) is enlarged with new nodes that form part of this S.C.C.

**Theorem 3**

\[ \text{SCC}(r) = \text{SCC}'(r) \cup \{ n \in N \mid n \rightarrow^* r \} \]

The proof for theorem 3 and those for the following theorems are omitted for brevity.

Theorem 4 tells us that the S.C.C.s of other new nodes (from which \( r \) is not accessible) are not changed.

**Theorem 4**

\[ \neg (n \rightarrow^* r) \Rightarrow \text{SCC}(n) = \text{SCC}'(n) \]
To summarise the above results; when a node and pointer is added to the graph, if the parent of the new node is in a different group to all the children of the new node, then, because the node is not on a path from one member of an S.C.C. to another, it follows that the new node is not part of the S.C.C. In this case the new node forms singular, trivial node by itself.

Alternatively, if the new node has any child that is in the same group as its parent (i.e. it must be on a path from one node in an S.C.C. to another), it follows that this new node must be part of the parent node's S.C.C. As a result it must be incorporated into this cycle.

When a new pointer is added to the graph, if it is between two nodes in the same S.C.C., then the only action required is to increment the target node's local reference count. If, however, the pointer is external to the target node's S.C.C., then the group counter, for the target node, must be incremented, along with the local count of the target node.

The addition of nodes and pointers to the graph can only cause groups to grow. New nodes either form part of an existing S.C.C., or form a singular node by themselves. In particular, addition of nodes and pointers to the graph cannot cause S.C.C.s to split into smaller cycles, or cause them to merge, forming larger strongly connect components.

(iii) Deletion of Pointers.

We consider the deletion of a pointer from a node $r$ to a node $s$. There are three conditions that are satisfied in this operation.

First, $r$ and $s$ are connected before the pointer deletion.

\[ \overset{\rightarrow}{r} \rightarrow s \]

Secondly, no paths are added by the deletion of a pointer from $r$.

\[ m \rightarrow n \Rightarrow m \rightarrow n \]

Thirdly, only paths through $r$ and $s$ are affected.

\[ m \rightarrow n \land (\rightarrow m \rightarrow r \lor \neg s \rightarrow) \Rightarrow m \rightarrow n \]

Theorem 5 states that S.C.C.s can only break up, they may not grow due to a pointer deletion.

Theorem 5

\[ SCC(n) \subseteq SCC'(n) \]

The next theorem states that only the S.C.C. containing $r$ is affected.

Theorem 6

\[ n \notin SCC'(r) \Rightarrow SCC(n) = SCC'(n) \]
Finally, theorem 7 states that if \( r \) and \( s \) are initially in different S.C.C.s then the S.C.C. structure is not affected.

**Theorem 7**

\[
s \not\in \text{SCC}'(r) \Rightarrow \text{SCC}(r) = \text{SCC}'(r)
\]

Again, to summarise the above theorems; if a pointer is deleted between two nodes that are in different groups, then this does not affect the S.C.C. structure. In this case all that is required is the target local count is decremented (and group count if it is a member of another S.C.C.).

If, on the other hand, the parent and target nodes of the pointer deletion are in the same S.C.C. then this may cause the cycle to split. The deletion of such a pointer will affect only the S.C.C. that it belongs to. Figure 3.10 shows the deletion of such a pointer within a large S.C.C. Each cycle is shown by a dotted line. The pointer deletion results in the original cycle splitting to form a number of *trivial* nodes and smaller S.C.C.s. The process of splitting the group and identifying the new S.C.C.s is performed by use of a Tarjan search.

When such an *intra-cyclic* pointer is deleted, the target node's local reference count is decremented. No change is necessary to the group count. The `emitgroup` procedure deals with the group counts of the new S.C.C. structure, in the manner explained in section 3.10.1.

It should also be noted that the process of deleting a pointer cannot cause an S.C.C. to grow, or merge with another.

![Figure 3.10 Splitting of a cyclic structure.](image)

### 3.11 Optimisations of Hughes' Basic Technique

Optimisations of the scheme concentrate on the main area of memory usage required to carry out the collection, since the basic technique involves a considerable memory overhead. Each node has a local reference count field, a pointer to the root of its S.C.C. and a marker bit to identify the latter field as a group reference count, if it is the root of a S.C.C. In addition to these two fields, the Tarjan search also uses two fields and a stack [TAR-72]. It is obviously necessary to minimise the amount of memory used by this scheme to make it practicable. Hughes [HUG-82b] has suggested a number of optimisations to the basic scheme given above.
First, one of the fields used in the Tarjan search can be eliminated. The \textit{lowlink} field can be incorporated into the group-pointer field, since all stacked nodes can be identified by their \textit{lowlink} values as to which group they belong to.

Secondly, the problem of minimising the amount of memory occupied by the reference count field can be simply addressed by incorporating a simple flag bit (\textit{trivial} flag), that is set if the node is part of an S.C.C., i.e., there is more than one node in the S.C.C. In this case the group reference count is redundant and the node simply incorporates the local reference count field. In general, single nodes, that are not part of an S.C.C., (which is normally the majority of the graph) have no special group reference count field. This optimisation does not, however, address the problem of large reference count fields being necessary when vast cyclic structures are formed. Part of the project looks at this aspect of memory optimisation. This will be discussed further in chapter 4.

\textbf{3.12 Conclusions.}

This chapter has outlined the importance of the subject of garbage collection, in a functionally programmed graph reduction computer. The basic methods of garbage collection were introduced, along with some variations designed to improve various aspects of their performance.

Reference counting collection was highlighted as the most promising technique, with particular reference to real time multi-processor applications. The traditional drawbacks of this type of collection were introduced. The problem of collecting redundant cycles was discussed, and several solutions were mooted. The method of tracking strongly collected components, within a derived reduction graph, was considered to be the most appropriate technique. The problems of excessive memory overhead, necessary to support this scheme, were also discussed.
Chapter 4. Alleviating the Memory Overhead Problem.

4.1 Introduction.

In chapter 3 the overhead of reference counting was shown to be substantial (this is quantified in this chapter). This is because each node of the main heap has a local reference count associated with it. This has to be long enough to hold a count, that can theoretically be as large as the total number of nodes in the heap space. The problem is compounded because Hughes’ technique requires that there should be an additional field, with each node in an S.C.C. This holds either a group reference count, or a pointer to the group count. This also has to be as long as the local count. The ensuing chapter deals with reducing these memory overheads that are traditionally associated with reference counting. A new design for a garbage collector is developed to encompass these ideas. The techniques assume that a node in the main heap has two fields associated with it, each wide enough for a full heap pointer.

In principal the method employs distinct areas of memory, which are allocated for the collector’s exclusive use, i.e., separate from the mutator’s main graph reduction heap space. In a practical realisation of the scheme, this memory would physically be built into a separate hardware collector. This arrangement has a number of advantages. It allows a much more flexible configuration of memory, which is important in this technique. It also divorces the heap memory from the “housekeeping” memory of the collector, an idea expounded by Hudak [HUD-84]. Such an arrangement provides performance advantages over a scheme where the main mutator memory is used for the use of the collector as well. This is because the collector can access local and group reference counts in local memory, a process that can be engineered to be much faster than an access to the heap. It also cuts down interference with the mutator and collector when these run asynchronously in parallel. Part of the collector memory structure is shown schematically in figure 4.1.

The majority of the memory within the collector comprises a “long-thin” block, which has the same logical address structure as the main heap memory, (i.e., each node in the main heap has a location in this separate memory associated with it). This collector block is made up of two limited-width fields. The first is a local reference count, and the second is a group descriptor number field. Thus each cell in the main heap is associated with a local reference count and a group descriptor number, both of which are held within the collector’s own memory structure.

4.2 Local Count Overflow.

Local count overflow occurs when the value of the reference count, for any particular node, exceeds that which can be represented in the limited-width field. Such an occurrence is indicated by an overflow bit being set in the local count field. The overflow data is then normally accommodated in a full-width, linear, limited-height block of memory within the collector (shown in figure 4.1). This indirection is achieved by copying the stored reference count to a free location in the overflow block of memory. The address of the location of the full-width count, within the overflow block, is then placed in the limited-width count field (overwriting the original reference count).

The reference counts in the overflow memory are then maintained as normal. When the count falls below the overflow value, the appropriate cell in the overflow block is released for further use. The count is then replaced, and updated normally, in the main collector memory block containing the limited-width local counts. The bit that marks local count overflow, in
Chapter 4. Alleviating the Memory Overhead Problem

the limited-width local count field, is reset.

The size of the overflow block is obviously restricted by the width of the local count field, since this must contain a pointer to the overflow block of memory when an overflow situation arises. This potential drawback could be eliminated by the use of a hash linking system [KNU-68, BOB-75], but this causes difficulty in predicting machine response time, thus introducing problems in a real time environment. For this reason and for the fact that any hash linking system is slower than direct pointer traversal, hashing was ruled out for local count first stage overflow.

Shaded area shows the memory which is local to the collector

Figure 4.1 Schematic representation of the collector memory structure.

Since the overflow block of memory, for local counts, is limited in size, this cannot be guaranteed to hold all overflow data that may be generated. If this situation arises, such extra data is accommodated in the main heap. This cannot be easily achieved by a hash link to the free list of cells, because such a list is a randomly distributed linked structure. (The free list is a linked list of all the available free cells in the main heap, see chapter 2 for more details). Instead, the data is stored in the cell that is at the top of the main heap free list. This cell is marked with a pointer to the originator of the overflow condition and tagged as holding local reference count data (by the use of an LO tag) and detached from the free list.
The segment of the main heap, in which the overflow cell is to be found, is then stored in the local count field, replacing the original limited-width count. The heap is notionally partitioned into $2^a$ segments (where $a$ is the local count field width). The segment pointer reflects in which area the heap cell (containing overflow data) is situated, this cell having been taken from the free list. Finally the overflow bit of the limited-width field is set and one reserved bit in the count is also set. The rest of the field then contains the segment (area) in which the overflow data can be found. The two set overflow bits indicate that the count is stored in main memory; it also indicates that the information held on the rest of the local count field is actually now a main heap segment pointer.

A cell within the main heap can store all data necessary for the maintenance of a local reference count. There are two full-width fields available, one of which stores a pointer to the originator cell of the overflow, whilst the other handles the local count, which can be fully represented in the cell. The original cell address has to be stored alongside the local count in order to be able to retrieve the count information for that cell, at a later time (when performing a segment search in the main heap). In figure 4.1, collector locations that have overflow data stored in the main heap are shaded in black. Figure 4.2 shows a heap overflow cell in greater detail.

**Figure 4.2** Detail of local count overflow into the main heap.

When information is required for a cell, which has local count data stored in the main heap, it is retrieved by performing a linear search in the appropriate segment of the heap. This main heap segment is indicated by the segment pointer held in the limited-width count field. A linear search in this segment then takes place until a cell marked as holding local reference count data (tagged LO), and containing the correct originator's cell address, is encountered.

If the collector has a local count width of $a$ bits, and the main heap contains $n$ cells, then, since the heap can be split into $2^a$ segments, the time taken to recover overflow data on a particular cell will be $O(n/2^a)$. This guarantees a maximum upper bound time for the collector to perform a pointer operation, a factor vital in the design of real time systems. For analysis of the behaviour of this technique in real time systems see chapter 6.
When the count stored in main memory falls below the overflow value, the count is restored to the collector’s limited-width memory and the two overflow bits are reset. The cell containing the overflow data is replaced on the free list of cells.

There is a trade-off between reducing the memory overhead and keeping the number of searches in the main heap to a minimum, since these are expensive in CPU time. As the width of the local count field increases this will obviously decrease the number of cells that reach the overflow condition, so fewer need have their reference counts stored in the overflow block. The overflow block can also be larger since there is a longer field available to address it. This results in fewer cells spilling reference count information into the main heap. Lastly, as the width increases, the area in which overflow data has to be searched for in the main heap, is decreased. All these factors lead to the conclusion that the greater the memory overhead, then the faster the response time of the collector. This holds true until all counts are held internally to the collector, without overflow. Increasing the available collector memory, for any given application beyond this point, will not yield any further gains in response time. In order to guarantee no overflow into the main heap we would require (for a heap of $2^N$ cells, each local and group descriptor field to be $S$ bits wide). This is obviously the degenerate case where full width fields are employed.

### 4.3 Group Count Storage and Overflow.

Each cyclic group, as it is identified by the Tarjan search, is allocated a group descriptor number, from a stack of available numbers. A tag bit, known as a trivial tag, indicates that the cell is not a member of a larger cyclic structure. Every cell in a particular group is given the same descriptor number. The group descriptor numbers serve as pointers to a block of memory that houses a full-width group reference count, as well as other data held on the group, such as the group handle. (A group handle is a set of nodes through which all members of an S.C.C. are reachable, it is made up of the addresses of destination nodes when intra-cyclic pointers are deleted during the reduction process. For more details see chapter 6, section 6.3.2).

#### 4.3.1 Storing the Group Handle.

Figure 4.3 shows detail of the group descriptor number block, along with the stack used to hold the group handle details (for reasons of clarity, this stack is not shown in figure 4.1). The group number serves as a pointer to the first frame holding information on the handle of that group, as well as a pointer to the group reference count.

The group handle stack organisation is based upon a stack frame system [HEN-90, FAR-85], but with modifications. This provides flexibility to modify the way handles are stored for any given application in which the collector is to be used. If an application tends to produce only a few cycles that are relatively stable with respect to splitting, then a larger frame size may be employed; however if a great number of unstable cycles are produced then a smaller frame size may be more efficient (i.e. results in fewer group handle details being held in the main heap). Typically, for a hard, fast real time system, the frame size would be set by the system designer after empirical studies of the applications behaviour. In practice, the typical size of a handle is found to be only about half the size of its associated S.C.C. (see chapter 9), so the handle stack does not tend to be of any considerable size.

Each new group allocated within the limited-height, group descriptor block, is assigned a frame in the handle stack. The frame size is determined by the system’s designer. Each frame either ends with a pointer to the next frame, containing data on that group’s handle, or the null
Section 4.3 Group Count Storage and Overflow

handle is used to indicate the end of group information at any point in the current frame. A single mark bit is also used per stack location. If the mark bit in the final frame element is set, then that indicates that the pointer is actually pointing to overflow in main heap space, rather than data in the next internal collector frame. This occurs when no free frames are available to hold additional handle data. The field widths of the stack are full-width, so are capable of containing pointers to the main heap.

Figure 4.3 shows four cells that belong to S.C.C.s. Two nodes belong to group number 1. The group numbers are used to point to the group reference count, held on a full-width block. The group number also points to the first frame used by the group to hold its group handle. Thus for group 3, the start of the handle is held in the third frame down, from the top of the stack. In the figure, groups 1 and 2 have their handles fully accommodated within the stack. The last element of the first frame points to the second element of the second frame, holding the handle information for the group. The end of the handle is indicated by zeros. Group 3, however, has only one frame allocated, due to the lack of stack space, and has further handle information in the main heap. This is indicated by the last element of its frame being tagged by a bit set to one, indicating that it contains a pointer to the main heap.

![Diagram of group data internal to the collector.](image)

Figure 4.3 Detail of group data internal to the collector.

When a handle of a group is traversed, the first frame is examined and then any subsequent frames. When this is done incrementally, (for real time purposes), the next location on the handle stack is stored as a return address for the traversing procedure, when it passes control back to its calling routine.
When handle information is held in the main heap, the initial node, which is allocated from the free list, contains the first handle address, along with the address of the next location in the handle chain. It is tagged as holding handle information, (by a group descriptor handle, GDH, tag). This is consistent with full group data overflow, see below and figure 4.4.

There is a compromise to be struck when deciding on the frame size. If it is too small, then an excessive number of stack elements will be taken up pointing to the next frame for the handle; if it is too large, then space will be wasted by having partially filled frames.

4.3.2 Group Count Overflow.

In a similar fashion to local count overflow, when the reserved block of memory used for full-width group reference counts overflows, information is redirected to the main heap. However, the situation is a little more complex for overflow of group data into the main heap (see figure 4.4). When a group is allocated, and there are no more short group numbers available, then the first two cells from the free list are allocated to store the group number and count. These cells are tagged as GDN and GDC (group descriptor number and group descriptor count, respectively). The cells also hold a pointer to a linked list of cells that store the group handle as it is formed. The group handle stack internal to the collector is not used when the main group data is held in the main heap space. As described above, main heap cells that hold group handle data are tagged as GDH (group descriptor handle). The group number assigned to such an S.C.C. is the address of the first cell taken from the free list (which is thus unique for each group designated in this way). The cells containing the group number and reference count are known as the group descriptor cells.

As each member of the group is identified during the Tarjan search it is assigned a cell in the main heap, from the free list. This contains a pointer to the original cell and a pointer to the group descriptor cell holding the group number. This newly assigned cell is tagged as GDP (group descriptor pointer) cell. Each cell within that particular S.C.C. has its own group descriptor pointer cell assigned to it. A segment pointer, for the group descriptor pointer cell, is now written to the collector’s limited-width group number field, in a similar fashion to local count overflow. A single bit is also set in the group number field to indicate overflow has occurred and that the field now holds a main heap segment pointer.
Section 4.3 Group Count Storage and Overflow

The stack of available short group numbers can be easily maintained, using a linked free list of available numbers within the group descriptor number block of the collector. This does not need to be in a separate stack, but can be maintained within the group reference count block itself. This requires just one register, which points to the first free location in the group count block. The free list is then maintained within the count block area. If there is no available group number then this register holds a null pointer, to indicate group count overflow to the main heap must take place.

When a group is discarded, and if its details are stored in the collector (i.e., when overflow has not occurred) then that location in the collector is freed and the free list of short group descriptor numbers is updated.

If a group, which has its details stored in main heap memory, is subsequently accessed after a location in the collector’s group information memory has become free, the group information on that S.C.C. is not moved from main memory back to the collector’s since this would involve tracing every member of that particular cycle.

When group information needs to be recovered for a cell, which has its group information stored in the main heap, then the group descriptor cells are found by searching for the individual group descriptor pointer cell for the node in question, in the appropriate main heap segment. The pointer to the group descriptor is subsequently traced.

When group number overflow has not taken place this method facilitates the testing of cells to ascertain which S.C.C. (if any) they belong to. This is achieved by simply comparing the group descriptor numbers for the two nodes in question. This operation is essential to the tracking of strongly connected components within the reduction graph (see chapter 6).
4.4 Dynamic Use of the Collector’s Memory Space.

Cells that do not belong to a cycle, which typically form the majority of the reduction graph [STO-84], do not need to use the group descriptor number field in the collector memory.

In such cases, where nodes are flagged as being trivial, then the limited-width local count field and the limited-width group descriptor number field can be coalesced within the collector for that particular node. This results in the ability to store a longer form of the local reference count.

This obviously decreases the number of cells that have to store counts in the overflow table or main heap memory and reduces the search space for cells that have overflowed into the heap, since a shorter heap segment can be defined in these cases. If and when these cells are incorporated into a S.C.C. structure, then the effective width of their local count in the collector is diminished by the width of the group number field. This has to be taken into consideration and any new overflow conditions so arising, have to be dealt with. (Analysis of the reduction in overflow space is considered in section 4.6, whilst chapter 7 incorporates this technique in the real time analysis of the collector).

4.5 Use of Caching Techniques.

It has been found, from work carried out with a simulator of the collector system (see chapter 9), that only a relatively few cells have large local reference counts and that in general there are only a small number of cyclic structures within the reduction graph. It has also been observed that for a particular processor, at any given time, reduction work on the graph tends to be clustered in memory space. These observations lead to the technique of using a small cache within the collector that stores pointers to cells in the main heap that hold data on either local or group overflow. If a cell has a pointer to overflow data in the main heap, stored in the cache, then a linear search of the main heap segment can be avoided.

4.6 Quantifying the Memory Overhead.

4.6.1 The Standard Technique.

The standard technique demands that every cell has a local count and a group count or pointer. As Hughes [HUG-82b] points out, it is unnecessary to store a group count for nodes that do not belong to an S.C.C. In these cases a single mark bit is set aside, and the node is flagged as being trivial. It is also important that the collector can readily test to see if nodes are on the Tarjan stack. This is accomplished by setting aside a further bit, for nodes that are non-trivial, to indicate whether or not they are on the Tarjan stack, this is the onstack bit.

Let us consider the memory overhead, for a heap of \(2^6\) nodes, using the standard technique, outlined above. Additional memory, required by the collector for such things as the depth first search number, \(dfs\) (see the explanation of the Tarjan search algorithm in chapter 2), storage of the handle for each group, the recursion stack for Tarjan’s algorithm and other workspace is not considered here.

A trivial, singular node, will require a local reference count field \(\delta\) bits wide, plus an additional tag field to mark it as being trivial. Thus a trivial node requires \(\delta + 1\) bits per node. A node that is a member of an S.C.C. also requires a local count and a group pointer or
reference count, both $\delta$ bits wide. In addition to these two fields, the trivial bit occupies one
bit, and the bit which flags the node as being on the Tarjan stack. Therefore a node that is a
member of an S.C.C. requires $2\delta + 2$ bits.

If the fraction of cells that are members of S.C.C.s in the heap, at any given time, is $c$, then the overall memory overhead required to support reference counting is given by:

$$
\text{overall overhead} \Rightarrow (2^\delta - c2^\delta)(\delta + 1) + c2^\delta(2\delta + 2) \text{ bits}
$$

$$
\Rightarrow 2^\delta (\delta c + \delta + c + 1) \text{ bits}
$$

Equation 4.1

Figure 4.5 gives a graph of the number of bits used to support reference counting versus $\delta$, ($c$ set to a constant value of 0.5 for the sake of simplicity here, though it is typically in the
range 0.05 to 0.1), as determined by equation 4.1. During the course of the following
discussion, a grey area will be used to indicate likely practical operating conditions as found
by simulation, and various other sources [SAL-85, HAR-88].

Figure 4.6 shows a typical standard cell. It assumes a node with two full width fields, each $\delta$ bits wide and a tag bit, taken to be four bits wide in this example, (the width of the tag
bit is implementation dependant). This does not include any fields used by the garbage
collector or memory manager.

---

Defining the overhead ratio, as the total memory overhead in bits to the total _useful_ heap
size (heap size comprising only standard nodes) in bits, we get:

Figure 4.5 Overall reference count overhead versus heap size.

Figure 4.6 A standard node.
overhead ratio \( \Rightarrow 2^\delta (5c + \delta + c + 1) / 2^\delta (2c + 4) \)  

Equation 4.2

Figure 4.7 gives the overhead ratio, against \( \delta \), for a variety of \( c \). It can be seen that this ratio increases rapidly for small \( \delta \), but once \( \delta > 8 \) the ratio is fairly constant for any given value of \( c \).

![Figure 4.7 Overhead ratio, versus \( \delta \), for a variety of \( c \).](image)

### 4.6.2 The Reduced Memory Overhead Technique.

The situation for the reduced memory overhead scheme is explored below. The width of both the local count and the group descriptor numbers is defined by the user of the collector. This also effectively governs the maximum size of the local count overflow block and the group descriptor number block (see figure 4.1). This is because the size of the limited-width field determines the maximum size of address that can be held in that field.

There is also the added complication that arises when the local and group count blocks overflow into the main heap. This type of overflow, in the case of local counts, is affected by two factors that are interrelated. For any given application, as the local count width is increased, in the main block of collector memory, the number of cells that require counts longer than can be accommodated in a local count limited-width field decreases. This results in fewer cells overflowing to the overflow block, which in turn results in fewer local counts spilling into the main heap. The size of the overflow block can also be made larger, since there is a greater width of field to address it. This further reduces the amount of overflow data in the main heap. With group number overflow, the larger the number field, then the larger the count block and so fewer group counts are accommodated in the main heap.

If \( \alpha \) is the length of the local count, in the limited width block, and \( \beta \) is the length of the group number, then assuming, for any given \( \alpha \) and \( \beta \), that the maximum amount of addressable local count overflow and group number memory is available within the collector, then we may simply write:
Section 4.6 Quantifying the Memory Overhead

overhead' \Rightarrow 2^5 (\alpha + \beta + 4) + 2^\alpha \delta + 2^\beta \delta \text{ bits} \quad \text{Equation 4.3}

The + 4 term arises from a status word, which is used for local count purposes to mark the cell as trivial, to indicate overflow into the overflow block and to indicate overflow into the main heap. It is also used by cells that are members of an S.C.C. to indicate the state of group number overflow. This simplified equation ignores the group handle stack and the memory occupied in main memory by local and group counts and handle information.

Equation 4.3 makes too many simplifications to be useful, so we need to extend the ideas upon which it is based. Firstly we consider the question of local reference counts; the actual length available to store any particular local reference count will be greater if the node is not a member of an S.C.C. (see section 4.4 on dynamic use of the collector's memory). We therefore find that there are \((2^8 - c2^5)\) locations in the collector that have \((\alpha + \beta)\) bits available to store the local count and \(c2^8\) locations that only have \(\alpha\) bits.

Consider a set of cells \(p\), that have local reference counts, \(v\), in the range \(\alpha \leq v < (\alpha + \beta)\), and a distinct set of nodes \(p'\), that have local counts in the range \((\alpha + \beta) \leq v \leq \delta\). Each local count, which spills into the main heap memory, occupies one node, in order to store the full-width local count. Since the size of the local count overflow block is \(2^\alpha\), the number of nodes that have their local counts stored in the main heap is given by:

\[
\text{main heap overflow for local counts} \Rightarrow p - (p(2^8 - c2^5) + p' - 2^\alpha) \text{ nodes} \quad \text{Equation 4.4}
\]

The term \((p - \varphi(2^8 - c2^5))\) arises because there are \((2^8 - c2^5)\) locations in the limited width block that are capable of storing a local reference count of up to \((\alpha + \beta)\) bits. However, there is not a one to one correlation between these longer locations and the nodes with local reference counts that are in the range \(\alpha \leq v < (\alpha + \beta)\) (i.e. the set of cells \(p\)). We therefore introduce \(\varphi\), the correlation factor relating the number of locations capable of holding local counts up to \(\alpha + \beta\) bits in length (equal to \(2^8 - c2^5\)) and the set of cells \(p\). \(\varphi\) lies in the range \(0 \leq \varphi \leq 1\).

Next, we must consider group count overflow, which only occurs when the number of groups in the graph exceeds \(2^\beta\), i.e., when \(\chi > 2^\beta\), where \(\chi\) is the number of cycles in the graph at any particular time. When group information is held in the main heap, one cell is occupied by the group number, whilst another is occupied by the group reference count. The group handle will obviously vary in size from one group to another. It is difficult to reason cogently about its size, and a notional average value will thus be assigned to it, \(\eta\). Additionally, each member of the cycle, whose group information is held in the main memory, will also have a main group cell assigned to it, as a group descriptor pointer.

It is obviously impossible to specify the number of nodes in any particular group for all classes of graph reduction problem, we therefore work with an average, assuming that each S.C.C. has \(\psi\) member nodes. This average will necessarily be crude, since the groups may vary greatly in size.

With the method outlined above, there is no predetermined way to predict which group will be accommodated in the collector, and which in the main heap (but see section 4.7 below). We therefore assume that the groups are of the average size, \(\psi\), and occupy a maximum amount of collector space possible before overflow to main memory occurs.
Chapter 4. Alleviating the Memory Overhead Problem

Since the group number block can accommodate $2^b$ groups, if $\chi > 2^b$, then $\chi - 2^b$ groups must have their group data stored in the main heap. This results in $2(\chi - 2^b)$ nodes being occupied in the main heap for group count and group number, $\eta(\chi - 2^b)$ on account of the group handle and $(c2^5 - \psi 2^b)$ nodes to accommodate the group descriptor pointer for each member of an S.C.C.

We know that there are $c2^5$ nodes that are members of S.C.C.s, where $c$ is the proportion of cyclic nodes, as before. Therefore $\chi = c2^5/\psi$. The number of cells occupied in the main heap, due to group number overflow is therefore given by:

$$\text{main heap overflow for group counts} \Rightarrow 2(c2^5/\psi - 2^b) + (c2^5 - \psi 2^b) + \eta(c2^5/\psi - 2^b)$$
$$\Rightarrow 2^b((2c + \eta c)/\psi + c) - 2^b(2 + \eta + \psi) \text{ nodes}$$

Equation 4.5

The overall overflow into the main heap is the sum of the local and group count overflows, i.e.

$$\text{overall main heap overflow} \Rightarrow (p + p' - \varphi(2^5 - c2^5) - 2^\alpha$$
$$+ (2^5 ((2c + \eta c)/\psi + c) - 2^b(2 + \eta + \psi))) \text{ nodes}$$

Equation 4.6

Assuming that a standard node is as that shown in figure 4.2, we can therefore write that the total number of bits taken up by main heap overflow is given by:

$$\text{overall main heap overflow} \Rightarrow (p + p' - \varphi(2^5 - c2^5) - 2^\alpha$$
$$+ (2^5 ((2c + \eta c)/\psi + c) - 2^b(2 + \eta + \psi))) (2^6 + 4) \text{ bits}$$

Equation 4.7

The total overhead of the limited width technique is given by the sum of the heap overflow plus the fixed collector memory overhead, given by equation 4.3.

$$\text{total overflow} \Rightarrow (p + p' - \varphi(2^5 - c2^5) - 2^\alpha$$
$$+ (2^5 ((2c + \eta c)/\psi + c) - 2^b(2 + \eta + \psi))) (2^6 + 4) + 2^b(\alpha + \beta + 4) + 2^\alpha 2^b \text{ bits}$$

Equation 4.8

Using the same definition of overhead ratio as above, we obtain:

$$\text{overhead ratio'} \Rightarrow ((p + p' - \varphi(2^5 - c2^5) - 2^\alpha + (2^5 ((2c + \eta c)/\psi + c)$$
$$- 2^b(2 + \eta + \psi))) (2^6 + 4) + 2^b(\alpha + \beta + 4) + 2^\alpha 2^b + 2^\beta 2^b) / 2^6 (2^6 + 4)$$

Equation 4.9

The term $(2^5(\alpha + \beta + 4) + 2^\alpha 2^b + 2^\beta 2^b) / 2^6 (2^6 + 4)$ is the contribution from the fixed size internal memory of the collector. Figure 4.8 shows the overhead ratio, due to this term only. The assumption (for simplicity) is made that $\alpha = \beta$, which would generally not hold true in a practical system, since setting $\alpha > \beta$ would probably yield improved performance from the collector. This is because the number of S.C.C.s in a graph will usually be less than the number of local counts exceeding $2^\alpha$, for $\alpha < 10$, (this is an empirical observation, see chapter 9). Not withstanding this, figure 4.8 holds good in order to show the general trend, for various values of $\alpha + \beta$, of overhead ratio versus main heap size in bits.
Section 4.6 Quantifying the Memory Overhead

It can be seen that the value of $\alpha$ and $\beta$ have to be chosen carefully with respect to the main heap size. If a relatively large heap is used ($\delta > 20$) then the ratio falls to a low level (0.35) for values of $\alpha + \beta$ as large as 10 bits. In these cases the contribution to the overhead from overflow data held in the main heap is potentially the most important element of the memory overhead ratio. The equations governing figure 4.8 also assume that the maximum possible amount of collector memory is used for local overflow and group data. This could obviously be substantially reduced for large $\alpha$ and $\beta$, at the risk of increasing the overflow data in the main heap.

Figure 4.8 Overhead ratio for fixed collector memory versus $\delta$.

The main heap memory used is given by equation 4.8. The overhead ratio for main heap overflow is therefore given by:

\[
\text{main heap overflow ratio} = ((p + p' - \varphi(2^\delta - c2^\delta) - 2^\alpha \\
+ (2^\delta ((2c + \eta\psi)/\psi + c) - 2^\delta (2 + \eta + \psi))) (2^\delta + 4)) / 2^\delta (2^\delta + 4)
\]

Rearranging gives:

\[
\text{main heap overflow ratio} = (p + p' - 2^\alpha - 2^\delta (2 + \eta + \psi)) / 2^\delta \\
+ (c (2 + \eta + \psi + \varphi\psi) - \varphi\psi) / \psi
\]

Equation 4.10

It is clear, provided that $p$ and $p'$ are not very large, and $\alpha$ and $\beta$ are of the order $\leq 10$, then the dominant term of equation 4.4 is $(c (2 + \eta + \psi + \varphi\psi) - \varphi\psi) / \psi$. For the term $(p + p' - 2^\alpha - 2^\delta (2 + \eta + \psi)) / 2^\delta$ to be significant, $\alpha$ and $\beta$ would have to be very small (probably $< 4$), as would $\delta (< 10)$, in order to ensure that $p$ and $p'$ were significant as compared to $2^\delta$. Generally we find that there are not many highly shared cells in the graph, (cells with large local reference counts) as compared to the total heap size [STO-84, HAR-88], i.e., $p$ and...
p' will tend to be small for any $\alpha + \beta > 6$. If we take $\alpha + \beta > 10$, then the first term of equation 4.10 can be ignored.

We take $\phi = 1$ for the sake of simplicity in this argument. This gives us the approximate relationship:

$$\text{main heap overflow ratio} \Rightarrow \frac{(c(2 + \eta + 2\psi) - \psi)}{\psi}$$

Equation 4.11

We may also make a reasonable assumption about the maximum average handle length, that is $\psi = \eta$. This is probably rather pessimistically large, since it implies that, on average, every member of an S.C.C. will appear once on a group handle. Though the handle may be actually longer than this, since a node may appear twice (or more times) on a group handle, this first order approximation appears to be valid for the purposes of this discussion (empirical evidence from the simulator shows that the maximum handle length that is usually encountered, is about half the size of the (maximum) size of its associated S.C.C.).

It will be noticed that equation 4.11 does not involve $\delta$, i.e., it is independent of the size of the main heap. It does, however, depend upon the proportion of cycles in the heap, $c$, and their average size, $\psi$. So, for large $\delta$ there will tend to be a constant average overflow into the main heap, assuming that $\psi$ and $c$ are constant for a given application. Equation 4.11 appears to represent very nearly the worst case situation, in which overflow has occurred. In fact, if $\alpha$ and $\beta$ are chosen carefully, this should actually prevent main heap overflow. Figure 4.9 shows the overhead ratio, as determined by equation 4.11, for a variety of $c$ and $\psi$, taking $\gamma = \eta$.

It may seem a little surprising that the overhead ratio in figure 4.9 decreases for large $\psi$ and small $c$. This rather anti-intuitive result arises originally from the second term in equation 4.5, $(c2^\delta - \psi 2^\beta)$. The rationale behind this term is the fact that there are in total $c2^\delta$ cells in the graph that are members of an S.C.C., and the group count block in the collector can store the group reference counts for $2^\beta$ groups (since the group descriptor width is $\beta$). It therefore follows that the greater the average size of each S.C.C. ($\psi$), then the greater the number of nodes that have information stored internal to the collector, and not in the main heap.

It is, however, unreasonable to suppose that the actual effect of the overhead will dip negative at some points, as shown in figure 4.9. In these regions the first term of equation 4.10 will dominate, (the simplification to the model only holds true with positive overflow to the main heap, i.e., once overflow ceases the simplifications fail). Also the effect of the simplification of setting $\phi = 1$ may play a part in this effect ($\phi$ will actually be less than one). The effect of the first term in equation 4.10 is not shown for a number of reasons: first, it is difficult to make any reasonable assumptions about $p$ and $p'$; secondly their effect will still be small due to the divisor, $2^\beta$; and thirdly, this region of low $c$ represents very small overheads, as compared to higher $c$ regions, and so is of little practical importance for the system designer when engineering a system to respond in the worst case scenarios in a real time environment (see chapter 5 for a discussion of real time systems).

Figure 4.9 shows that the overhead due to main memory occupation by overflow data is small for values of $c \leq 0.4$. The much larger overhead ratios for large $c$ can be accounted for, and largely dismissed, in two ways. Firstly, a large value of $c$ is unlikely; it appears from the
Section 4.6 Quantifying the Memory Overhead

Simulations performed (see chapter 9), that most cells are not members of S.C.C.s, so the situation of more than 40% of the graph being incorporated into cycles occurs very rarely in our experience. Secondly, the model governing the equations portrayed in figure 4.9 assume that overflow of both local and group counts have occurred. As stated earlier, this situation should be rare in most applications, where a judicial choice of \( \alpha \) and \( \beta \) has been made.

![Graph showing overhead ratio versus S.C.C. size.](image)

**Figure 4.9 Overhead ratio versus S.C.C. size.**

If we take a value of \( c \) of 40%, which is an extreme upper limit found empirically, (see chapter 9 for experimental results), and add this to the overhead shown in figure 4.8, the overall overhead ratio for the reduced memory scheme should be approximately as in figure 4.8, except with a constant value of approximately 0.14 added for the overflow to main memory. This is shown in figure 4.10, which compares the limited width method, to the standard method for \( c = 40\% \), \( 2 \leq \delta \leq 30 \), and for varying value of \( \alpha \) and \( \beta \). The plot for the standard technique, given in figure 4.10, has been adjusted to reflect the possible contribution to the overhead ratio from the storage of group handles. This was not taken into consideration in section 4.6.1. and figure 4.7. The overhead for the handle, in this case, was taken to be the same as that for the limited width scheme, (from equation 4.11) namely \( c\eta / \psi \). Since \( \eta = \psi \) was assumed, this is simply the constant \( c \). In figure 4.10, the relative values of overflow ratio, for the purpose of comparison (if not the absolute values), are therefore independent of \( \eta \). Empirically we find that \( \psi \) has a typical value in the range \( 10 \leq \psi \leq 30 \), from the studies run with the simulation of the collector (see chapter 9).
Figure 4.10 Comparison of the standard and limited-width techniques of reference counting.

Figure 4.10 represents a reasonable, if rather pessimistic, comparison of the theoretical maximum overhead of the limited width system (including overflow into the main heap), as compared to the standard technique. It is, however, instructive to try to predict the worst case situation for the new technique. Equation 4.10 governs the overflow ratio for the limited-width technique (for overflow to the main heap). An examination of earlier equations show that this will be the significant contribution to the overhead ratio in a worst case situation. This equation will be at a maximum if we assume that the maximum possible number of cells has a local reference count greater than \( \alpha + \beta \). This occurs when \( p' \) is set equal to \( 2^6 \), which implies \( p = 0 \). This means that no local counts are able to be accommodated in the limited-width local count fields of the collector, and so all local count data will have to be accommodated either in the overflow block, internal to the collector, or else in the main heap.

It is also necessary to take a reasonable value for \( \alpha \) and \( \beta \), the values \( \alpha = 6 \) and \( \beta = 4 \) have been found by simulation to represent a good compromise between the competing factors in equation 4.10, (see chapter 9). The average size of S.C.C. also is important in equation 4.10. The equation 4.10 is at a maximum for minimum \( \psi \). Since we cannot have 0 nodes per S.C.C., we shall set \( \psi = 1 \).

Finally, a value for the average length of group handle has to be devised. Each cell can have \( 2^8 \) references to it, if each of these is intra-cyclic then that cell could appear \( 2^8 \) times on the group handle. This could be true for the \( 2^8 \) cells in the heap, (if the heap comprises a single cycle) so that \( \eta \) can theoretically be of any size up to an extreme of \( 2^{28} \). However, a maximum value of the handle size can be imposed "artificially", by ensuring that before a node is added to a group's handle, the handle is first searched to ensure that that node is not already on the handle, if it is then it is not added. In this way we might enforce a policy that no node is duplicated on the group handle. This expedient allows us to set an upper theoretical bound on \( \eta \). This is of use in the development of this model of the spatial behaviour of the collector, rather than being a practical proposition. This is because in practice \( \eta \) has always been found to be very much smaller than \( 2^{28} \) (typically \( \eta < 20 \) and it
would also be very inefficient to search the entire group handle every time a node is added to it. The value of η chosen is 10% of δ, this represents a value greater than has ever been encountered in the simulation (see chapter 9), and so reflects an empirical upper worst case bound on the average size of group handle.

Substituting for η, p and ψ, in equation 4.10, we get:

$$\text{Maximum overhead ratio} \Rightarrow \frac{(δ - 2^α - 2^β (2 + 0.1δ + 1))}{2^δ + c (2 + 0.1δ + 2) - 1}$$  
Equation 4.12

Substituting for α = 6 and β = 4 in 4.12, we get:

$$\text{Maximum overhead ratio} \Rightarrow \frac{(-0.6δ - 112)}{2^δ + c(4 + 0.1δ) - 1}$$  
Equation 4.13

It is clear that the first term in equation 4.13 will always be negative. This term arises from the fact that the internal collector memory, which is used to accommodate handle information on internally held group data, is not accounted for in the equation. It is assumed that no spill of handle data occurs into the main heap from groups whose data is held within the collector. As in figure 4.10, a correction is added to the plots for the standard techniques to reflect the size of the group handle that also has to be stored when using this method. This enables a comparison to be made, as shown in figure 4.11, between the standard technique and the estimated worst case example of the limited width technique. Again, because each of the plots, for the new and standard techniques, made the same assumptions about the average handle length, the comparative values shown in figure 4.11 will not change with different values of η.

Figure 4.11 Comparison of worst case memory overhead between limited width technique and standard technique for a variety of c.

Figure 4.11 shows that under extreme conditions the limited width scheme shows a greater memory overhead than the standard technique. It should be stressed that these
conditions are never likely to be met in practice. Also, because of the diversity of variables governing the behaviour of the limited width scheme, a number of assumptions has to be made, as specified above. Some of the values given have been from empirical evidence, which, to date, has been derived from a limited experimental basis (see chapter 9). The author has tried to make pessimistic estimates for these variables, so that a "maximal" worst case situation is modelled in figure 4.11, but inevitably until the system is tested on larger scale, real world problems, there remains a question mark about the validity of some of these assumptions.

4.7 Optimisation Arising from Pre-reduction Analysis of the Graph.

In section 4.6.2 it was stated that there was a random choice of which groups had their data stored in the internal memory of the collector, and if overflow had occurred, which ones have their data stored in the main heap. This situation is clearly undesirable from considerations of the efficient occupancy of memory. This is because cyclic groups containing a low number of nodes may have their count accommodated in the internal collector memory, whilst S.C.C.s, containing many more nodes, may have their group data held in the main memory. Since each member of an S.C.C., which has group data stored in the main heap, requires its own node in the heap, it is obviously desirable to store larger S.C.C.'s in the internal collector memory, where no main heap space is consumed for each constituent node of an S.C.C. If group number overflow occurs, then the smaller cycles should be the ones that are chosen by the collector to have their data held in the main heap. This reduces the memory overhead, by cutting down the number of nodes, used in the main heap, as group descriptor pointers.

As described in chapter 3, a Tarjan search is performed before any graph reduction takes place. This serves the task of initially decomposing the reduction graph into its constituent acyclic graph of S.C.C.s and trivial nodes. After each cycle has been traced by the Tarjan search and processed by the procedure that emits the group as an S.C.C., it is possible to determine the size of each S.C.C. If the length of time taken is not critical for this preliminary processing of the graph, (which can be likened to a stage of compilation in a procedural language), then it is conceivable that the process could be performed twice. The first run would be used to ascertain the sizes of each of the S.C.C.s in the graph. On the second, already having identified the sizes of each cycle in the first run, it would actually place each S.C.C.s data, either in the collector, for the largest 2^b groups, and any other smaller ones in the main heap.

An alternate to having two Tarjan searches of the graph would be to process all groups normally, and then perform any swapping as deemed necessary, between groups whose data is held by the collector, and those in the main heap. This process would appear to be prohibitively expensive and complicated at run time, but may be justified before reduction proper is underway, by the main heap space savings so made.

The simpler method of performing two Tarjan searches of the graph, prior to reduction, is used in the software simulation of the collector (see later chapters).

This method gives better initial overhead ratios than simply randomly placing the group data. This overhead benefit remains for classes of problems where few groups are formed, modified or split during reduction, so resulting in the size and structure of the S.C.C.s remaining fairly constant throughout the life of the program execution. If, however, many new groups are formed, and the general cyclic structure of the overall graph changed greatly

-64-
during the reduction process, then the advantages in performing this initial extra analysis of the graph, before run time, appear to be relatively short lived.

4.8 Conclusions.

In this chapter new techniques have been presented to minimise the memory overhead normally associated with reference counting garbage collection. The basic methodology employed is to utilise limited width fields to hold count information within the garbage collector itself. When these fields overflow, this is dealt with in a controlled, and quantifiable, manner.

It has been demonstrated by analytical techniques in chapter 4 that the limited-width reference count scheme offers a saving in memory, for supporting reference counting, as compared to the standard method. This is clearly the case for all but the most extreme operating conditions. The arguments put forward are partially based on empirical evidence (see chapter 9) and may vary when the collector is implemented in a system running larger, more complex examples. The results put forward in this chapter will still hold good, with modifications to some of the constants. This should only affect, to a small degree, the relative comparison between the standard and limited width methods of reference counting.

Notwithstanding these reservations about the above analysis, it has been clearly demonstrated that the new technique substantially reduces the memory overhead in all conditions except, possibly, where the heap size is small, the number of cycles in the heap is very high (c > 50%) and the number of nodes in each S.C.C. is also very low. Figure 4.11 gives an estimated worst case analysis comparison of the new and standard techniques. Even in the most extreme case, the new technique has only twice the overhead ratio of the standard one. Indeed, with a lower percentage of cycles in the graph, the worst case value for the new technique is twice as good as the standard.

The technique of maintaining the collector's memory separate from the main heap divorces the heap and mutator from the "housekeeping" storage necessary to maintain the local and group reference counts. This means that, at least for a single processor/collector combination (but see chapter 10 for a consideration of multi-processor environments), one of the traditional objections to reference counting collection is largely removed. That is the bandwidth overhead used to communicate the amendment of counts every time a pointer operation is issued by the mutator is eliminated. This is because any local or group count alterations are done in the collector's own store, with the obvious exception of counts held in the main heap, without further communication between the collector, mutator and the heap. (c.f. weighted reference counts [THO-81, BEV-85], and indirection nodes [HUG-85], described in chapter 3).

In the latter part of the chapter a simple technique for optimising the handling of group information is presented. This employs information about the cyclic structures present in the reduction graph, prior to the start of the reduction process, to optimise the positioning of group data between the collector's private memory and the main heap space.

The system has been designed to facilitate "tuning" of several parameters (local count width, group descriptor width and stack frame size) in order to allow the engineering of an optimal system for any given application.
Chapter 5. Introduction to Real Time Systems.

5.1 An Introduction to Real Time Functional Systems.

There are many applications where a real time response from a computer system is desirable, or indeed essential. Such applications include control systems, text editors, operating systems, simulators, digital signal processors and many others.

Appel [APP-88] gives the following definition of a real time algorithm, in the context of a garbage collection algorithm running with a graph mutator:

"An algorithm (garbage collector) is real time if the mutator (the program) is never interrupted for longer than a very small constant time. A collector has small latency if the interruptions are short."

Kalinsky [KAL-89] states that a real time system should display the following characteristics:

- **liveliness**, it must always be capable of responding to a given stimulus in a predetermined way;
- **safety**, for a given input, it must always provide the correct response;
- **timeliness**, it must respond within a given time constraint, determined by the system designer.

5.2 A Taxonomy of Real Time Systems.

Many real time system requirements incorporate timing information in the form of deadlines. An acute deadline is shown in Figure 5.1 [CHE-88].

![Figure 5.1 A deadline.](image)
In figure 5.1 the notional “value” of an event occurring is mapped against the time taken to complete the event. Here “value” is loosely defined to mean the contribution this event has to the overall system's objectives. With the event represented in figure 5.1 this is taken to be zero before the start time and returns to zero after the deadline has passed. The mapping of value against time is obviously dependant upon the particular application under consideration. It is common to find that in high performance reactive systems, such as a real time operating systems, the earlier a response is achieved after the start time the higher the value.

In a safety critical system actual damage may occur if the deadline is missed (i.e. “negative value” results). This is common in such systems as “fly-by-wire” control, railway signalling, weapons guidance etc. With the situation depicted in figure 5.1 the failure that arises if the deadline is missed is benign (no negative value results if the deadline is missed). In figure 5.2 the failure becomes more severe as time passes beyond the deadline. A system can be defined as a hard real time system if the damage has the potential to be catastrophic [LAP-89]. For some systems the “damage” may be financial, rather than physical (e.g. automatic on-line financial transaction software, etc.).

A soft real time system is one in which there remains positive value in the computational outcome even though it does not meet the deadline, i.e. missing a deadline does not compromise the integrity of the system. Such soft systems include text editors, CASE tools, CAD packages etc, in which there may be a user response time specified, for a given input stimulus, in a non functional requirements definition of the application. If such requirements are not met exactly (i.e. the package responds slightly slower than required under certain circumstances) then no actual (physical or economic) damage results and some value is retained in a missed deadline. Figure 5.3 depicts an example of a soft deadline.

Figure 5.2 A hard deadline in a safety critical system.

Figure 5.3 A soft deadline in a non functional application.
Many systems display characteristics of both hard and soft real time systems, where some deadlines are hard and others soft. These are termed hybrid real time systems. With a soft real time system the value of a response always remains positive (the curve approaches the x-axis asymptotically in figure 5.3). A hybrid system may be characterised by three deadlines (for any time dependant response). D1 represents the time that the response provides maximum value, D2 represents the last time, after which the contribution falls to zero and lastly D3, at which time the contribution is negative. An example of such a hybrid real time system is automatic track recognition software, used in the production of a tactical air picture.

In [COO-93], Cooling presents a taxonomy of real time systems. In addition to the hard and soft categorisation, real time systems may also be classified as fast and slow. As a loose "rule-of-thumb", systems with a response time of less than ten seconds may be classified as fast. Examples of such systems are presented in figure 5.5.
Section 5.3 Real Time Functional Systems

<table>
<thead>
<tr>
<th>SOFT</th>
<th>FAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>on-line</td>
<td>CASE tool</td>
</tr>
<tr>
<td>terminal</td>
<td>vehicle</td>
</tr>
<tr>
<td>functions</td>
<td>stability</td>
</tr>
<tr>
<td></td>
<td>augmentation</td>
</tr>
</tbody>
</table>

Figure 5.5 A taxonomy of real time systems.

It is not possible to characterise the garbage collector described in this work in terms of the classification outlined in the figure above, in isolation from the final application in which it is to be employed. Since hard, fast real time systems demand the most stringent requirements in terms of performance, the discussion and analysis presented in this work will concentrate on such applications. In such situations the critical performance criteria are based on worst case analysis (since this determines the success or otherwise of meeting critical deadlines under extreme conditions). It is argued by many professional researchers in the field of real time systems [BRU-93, COO-93] that it is impracticable to produce realistic analytical models to predict the average performance of real time systems (other than small systems that may be treated by the application of queueing theory). Because of these considerations worst case analysis will be performed to predict system behaviour. The simulation of the collector (chapter 9) provides observed average behaviour under certain test conditions.

5.3 Real Time Functional Systems.

Functional languages offer several advantages for programming real time systems. The property of referential transparency, that greatly facilitates the proving of programs to be correct, obviously is key in the meeting of Kalinsky’s [KAL-89] safety requirement. Functional languages also offer ease of support for parallel processing. Such parallel architectures allow much greater computational throughput and so help meet the timeliness and liveliness requirements. These, and other properties of functional languages, have been briefly outlined in chapter 2.

However, functional languages have also traditionally presented problems, with respect to real time systems. Morris [MOR-82] has stated, after much practical experience with functionally programmed systems, that the cost of garbage collection is the most important problem, particularly for system with real time constraints. With the advent of newer techniques in garbage collection, this assertion by Morris is probably overstated. Nevertheless, garbage collection still remains a very important issue in the design of real time systems based upon a functional approach.

Chapter 2 outlined the major classes of garbage collector. Each of these will be explored below, with particular reference to real time systems.
5.4 Mark-scan Garbage Collection in Real Time.

Mark-scan garbage collection, first proposed by McCarthy [MCA-60] is probably the most common form of collection in current use. In its simplest form, garbage collection is triggered when the heap is almost full. The graph mutation process is then halted, whilst every accessible cell is traced. The whole heap is then scanned, in order to collect any inaccessible cells. This results in an “embarrassing” pause in program execution, whilst collection takes place, (in this pause all mutator activity halts and the program appears to “hang”). Obviously, the performance of such a collector makes it unsuitable for a real time system.

Algorithms have been presented for mark-scan collection, that allow the collector to run simultaneously, or partly simultaneously, with the graph mutator. These concurrent mark-scan algorithms are more complicated than their simple “stop-the-world” counterparts. This idea was first introduced, according to Knuth, by Minsky [KNU-68, exercise 2.3.5-12]. Steele [STE-75] presents such a real time algorithm, for a single processor heap-space architecture. Like many subsequent algorithms, it relies upon heavy synchronisations between the graph mutator and the collector. Steele relies upon the use of semaphore primitives [DIJ-68].

Wadler [WAD-76] gives the following definition of a standard mark-scan collector, that returns all cells which are inaccessible back to the AVSL (available space list):

**Algorithm C (Collector).**

\[ C1. \text{ For all } i, 1 \leq i \leq R, \text{ if the node pointed at by } \text{ROOT} (i) \text{ is not marked, then mark it.} \]

\[ C2. \text{ Find a marked node. Call this node } A. \text{ If one (or both) of } \text{ALINK} (A) \text{ and } \text{BLINK} (A) \text{ is unmarked, then mark it (or them). Repeat step } C2 \text{ until no more nodes with unmarked descendants are found.} \]

\[ C3. \text{ For all } i, 1 \leq i \leq M, \text{ examine } \text{NODE} (i). \text{ If it is unmarked, then return it to AVSL. If it is marked, reset its status to unmarked.} \]

Where the heap consists of \( M \) nodes, \( R \) is the set of accessible nodes and \( \text{ALINK} \) and \( \text{BLINK} \) are the pointers from a node.

This assumes the standard operation of a mutator, that has a fundamental action, defined (in the context of a LISP based system) by:

**Operations M (Standard Mutator)**

\[ M1. \text{ For any accessible nodes } A \text{ and } B, \text{ and any } i, 1 \leq i \leq R: \]

\[ \text{Call CREATE_NODE} (A,B,i). \]

\[ M2. \text{ For any accessible nodes } A \text{ and } B: \text{ Change } \text{ALINK}(A) \text{ or } \text{BLINK}(A) \text{ to point at } B. \]
Section 5.4 Mark-scan Garbage Collection in Real Time

M3. For any accessible node A and any i, 1 \leq i \leq R: Set 
\text{ROOT}(i) to point at A.

Wadler points out that the above algorithm should continually execute in parallel with the mutator. This cannot be achieved with a standard definition of mutator activity, see [WAD-76], since the collector may, during the course of the graph reduction process, erroneously return a cell to the AVSL, whilst it is still accessible from the mutator. This may occur, for example, if the collector were operating step C3 and a \textit{CREATE NODE} (A, B, i) was called. \textit{CREATE NODE} (A, B, i) is a routine that allows the mutator to gain access to a new node by removing it from the AVSL, setting its ALINK to A and BLINK to B, and returns a pointer to the new node in \text{ROOT}(i). If this new node was placed in a location higher than the collector had yet examined then when the collector finally reaches the new node, it will be returned to AVSL, because it is not marked.

A modified parallel algorithm, given by Wadler, is set out below. It utilises semaphores, like Steel's algorithm, to make access to AVSL exclusive to either the mutator or the collector.

\textbf{Operations M (Parallel Mutator)}

\textbf{M1'.} For any accessible nodes A and B, and any i, 1 \leq i \leq R: 
P(\text{AVSL semaphore}), call \text{CREATE NODE} (A, B, i), set the mark bit of the newly created node pointed at by \text{ROOT}(i), and V(\text{AVSL semaphore}). If one (or both) of A or B is unmarked, then mark it (or them).

\textbf{M2'.} For any accessible nodes A and B: Change ALINK (A) or BLINK (A) to point at B. If B is unmarked then mark it.

\textbf{M3'.} For any accessible node A and any i, 1 \leq i \leq R: Set \text{ROOT}(i) to point at A. If A is unmarked, then mark it.

The final step of the collector algorithm must also be changed to:

\textbf{C3'.} For every i, 1 \leq i \leq M do the following: P(\text{AVSL semaphore}). If NODE (i) is unmarked and not already on AVSL, then return it to AVSL. V(\text{AVSL semaphore}). If NODE (i) is marked, reset its status to unmarked.

The paper by Dijkstra [DIJ-78] presents a more sophisticated parallel collector. This exemplifies the difficulties of the issues raised by mutator/collector synchronisation, in concurrent mark-scan algorithms. He uses two mark bits per cell, to represent one of three cell colours: \textit{white} (unmarked), \textit{black} (marked) and \textit{grey} (in use by the mutator). The grey cells are considered likely to be turned black.

Dijkstra makes the mutator responsible for turning white cells grey, as and when they are requested. It also must issue an interrupt to the collector when only one cell remains on the free list. This interrupt enables collector activity. The collector returns control back to the mutator when at least one cell has been freed. The collector acts by greying the first active
cell (cell accessible from the mutator) and the first cell on the free list. The collector then greys any cells connected to a grey cell, and then blackens the grey parent cell. When tracing ceases any white cells are incorporated into the free list, and black cells are then whitened. Thus any inactive grey cells are firstly blackened by the collector, and then whitened. On the next collector cycle these are then appended to the free list.

Kung and Song [KUN-77] improve on Dijkstra’s scheme. This development employs four colours, and unlike Dijkstra’s method it does not have to trace the cells in the free list. The authors show this method to be more efficient. Another method of mark-scan concurrent collection is given by Ben-Ari [BEN-84] and further proofs of the algorithms presented in Ben-Ari’s paper are given in [SNE-87].

All these algorithms suffer from the need to synchronise mutator and collector operations, to some extent. They also all seem to suffer from the basic failing of mark-scan collection, in that the action of the collector remains essentially global in the graph. This implies that in loosely coupled systems action of the collector outside its local heap space represents an expensive operation. Hughes addresses the problem of a mark-scan collector running on a distributed system [HUG-85], but the algorithm presented is not real time in nature. Lamport [LAM-78] has suggested a method by which Dijkstra et al.’s method is extended to deal with multiple mutators, but this relies on further synchronisation between mutators and collectors and between mutators and mutators.

It is apparent that the problem of producing a real time mark-scan collector, particularly for a loosely coupled distributed system, is a very difficult one [HIC-84]. This is because the action of the collector, in its basic form, demands the cessation of graph mutation and to act globally on the whole heap space.

5.5 Copying Garbage Collection in Real Time.

Copying garbage collection in real time was first suggested in Baker’s classic paper [BAK-78]. Baker bases his method on the original copying technique suggested by Minsky [MIN-63]. The basic method, as mentioned in chapter 2, divides the heap into two semispaces. Since Baker’s algorithm presents a possibly practicable alternative to reference counting collection, it will be explained in some detail here.

The starting point for the description of Bakers scheme is the description of Minsky’s basic two space collector. In this collector the heap area is split into two equally sized semispaces. For the purpose of this discourse these are named fromspace and tospace. During graph mutation all new cells are allocated in only one of the two semispaces; this is labelled the tospace semispace and the other the fromspace semispace.

The current semispace acts rather like a push-only stack; new cells are simply allocated on the top of the contiguous block of currently active cells and are accessed through pointers rather than a pop operation. The address of the current topmost active cell in the current semispace is given by a register B. Throughout this discussion heaps are drawn growing towards the top of the page in the direction of decreasing memory addresses; pushing a cell corresponds to decrementing B.

When there is insufficient space to allocate a new cell at the top of the current work space, the garbage collector is invoked. When the garbage collector is so invoked, the mutator operation is immediately suspended. The collector works by recursively tracing through the current semispace (fromspace), starting from the entry cell of the reduction graph, locating the
active cells. As these are located they are moved to the other semispace (tospace). The register $B$ is maintained to point to the topmost word in tospace. This is initialised to the value of the bottom of tospace before the collector is invoked and is incremented each time a cell is moved to tospace. The original reference to each cell moved is set to point to the new location of the cell in tospace; additionally, in the old location of the moved cell in fromspace, a forwarding address is written, pointing to the new location in tospace. This is used to update any other references to that cell, that may be located subsequently. Any such references encountered are replaced by the forwarding address, so that such cells now contain a reference to the active cell in tospace, rather than its old location in fromspace. This ensures that shared references are preserved during garbage collection. A tag bit is used to distinguish a forwarding address from a normal reference.

![Figure 5.6](image)

Figure 5.6 Relocating a cell in the standard copying collector.

Figure 5.6 shows a cell being moved from fromspace to tospace. Once the cell $x$ has been moved, the collector is recursively called until all cells have been moved in this fashion. At the end of the garbage collection all the active cells present in the fromspace will have been moved to tospace. The area above the newly moved cells in tospace forms the freespace, from which any new cells required by the mutator are taken. When the collector is run again the roles of fromspace and tospace are swapped over.

One of the obvious problems with this type of collector is that it requires twice as much space as for a mark-scan collector. However, this is not a major disadvantage if used in a virtual memory machine. A much more serious objection is that the mutator must halt whilst
Chapter 5. Introduction to Real Time Systems

Garbage collection is performed (when the active cells are traced and copied). This was the issue addressed by Baker [BAK-78] in order to make this technique suitable for real time systems.

Baker's collector employs two semispaces as before. The collector is called when the freespace is exhausted. Unlike the previous simple semispace collector, however, the collector relocates \( k \) cells and then returns control back to the mutator. This contrasts to the simple collector, described above, which runs until completion before allowing the mutator to continue. The next time the mutator requires a new cell to be allocated another \( k \) cells are relocated before the new cell allocation is complete. This process continues until all cells have been moved from fromspace to tospace in this incremental fashion. The mutator may then allocate cells from freespace until this is once again exhausted.

Using this scheme it is possible that the mutator will encounter references to cells in fromspace before they have been relocated. This problem is solved by giving the impression to the mutator that all cells have been copied to tospace instantaneously, (this is sometimes referred to as maintaining a read barrier) when the garbage collector was first called. This illusion is achieved by immediately relocating any cell that are referenced in fromspace, and if the mutator operation results in another reference in fromspace then this cell is also relocated to tospace.

When the collector is started it swaps tospace and fromspace and initialises \( B \) so that all new cells allocated are in tospace. This is the FLIP operation. The active topmost cell in the reduction graph in fromspace is then moved to tospace and a forwarding address is placed in the space left behind in fromspace. Unlike the simple collector the descendants of the relocated cell are not traced immediately. Instead, a second pointer \( S \) is set to scan each cell in tospace, from its the base of tospace to the top, relocating any cells in fromspace that it finds from a reference in tospace. As \( S \) works up tospace towards \( B \) it updates references in tospace that point to forwarding addresses in fromspace. Thus cells below \( S \) have been fully updated, with their references changed from pointing to fromspace to pointing to tospace. Cells above \( S \) and below \( B \) have been moved from fromspace but are awaiting the update of their pointers. The pointer \( S \) makes \( k \) iterations in tospace, as described above, and then control is passed back to the mutator. When a new cell is required by the mutator the collector makes another \( k \) iterations. By associating the copying of \( k \) cells with the allocation of new nodes, the garbage collector can guarantee that tospace will not be exhausted before all objects have been copied from fromspace.
Section 5.5 Copying Garbage Collection in Real Time

Figure 5.7a Relocating a cell in Baker's copying collector (prior to copy).

Figure 5.7b Relocating a cell in Baker's copying collector (after copy).

In order to prevent newly allocated cells being needlessly traversed by $S$ (if these are allocated interleaved with moved cells between $S$ and $B$) another pointer $TOP$ is introduced. New cells are allocated from the top of the heap, $TOP$ therefore grows downwards towards $B$. 
Chapter 5. Introduction to Real Time Systems

Figure 5.7 (adapted from [FIE-88]) summarises the copying of a cell labelled y, from fromspace to tospace. This may be caused by three reasons, depicted in diagrams (a), (b) and (c). First, the mutator may reference the cell x (say by requiring its head pointer, that points to y), and since the collector maintains the illusion that all cells have been moved to tospace when FLIP was called, cell y is copied to tospace and a forwarding address is left behind in fromspace. Secondly, after the FLIP operation a reference by the mutator was made to cell y in fromspace, so causing y to be copied. Thirdly, whilst S was traversing tospace a reference from x to y was encountered, so triggering the copying of y.

The memory overhead of Baker’s algorithm can be quantified at a crude level. Suppose the two semispaces have just been flipped and that there are N accessible cells (in fromspace). The collector will move N cells from fromspace to tospace. Since the transport of cells is performed when new cells are allocated, N/k cells will be newly allocated to the mutator to perform this copying. This requires a total semispace capacity of N + N/k cells. Thus a total heap space of 2N(1 + 1/k) cells is required. The worst case spatial performance of the collector occurs for k = 1, in which case a total heap size of 4N is required to support an initial population of N active cells. An alternative view of this argument is to suppose that each semispace has M cells and that the maximum cell requirement of the application program is N, then k must be at least N/(M - N).

The CPU overhead of the algorithm may be explored in a general way by considering two distinct situations. First, the collector is not active. In this case all mutator operations are performed with no additional collector overhead, since the read barrier is not active. Secondly, when the collector is active there is additional latency introduced in mutator operations since the collector must maintain the read barrier. A worst case situation arises when a cell is referenced by the mutator that is in fromspace requiring the retrieval of a descendant of that cell (which itself is in fromspace). This incurs the cost of copying that cell to tospace, writing a forwarding address in the old location of the cell in fromspace, then performing the same operation for the child cell, finally the reference in the parent cell to the child, is updated to reflect the move of the child. The addition of a new cell for the mutator from freespace carries a heavy penalty. This requires that S performs k incremental steps up fromspace. On each step, the worst case scenario is as follows; the cell S is visiting has a child in fromspace, this cell is moved to tospace, its new address is then written to fromspace as a forwarding address and the pointer to the moved child in the parent is updated. This operation occurs k times before the mutator can resume operation. Baker’s algorithm was implemented and a comparison between it and a reference counting collector is made in chapter 9.

Wong et al. [WON-86, WON-87] have proposed a hardware real time collector based on Baker’s algorithm, whilst Appel [APP-88] has produced another copying collector, based on Baker’s, designed to run on standard multi-processors. Zorn [ZOR-89] reports that there have been no successful implementations of Baker’s algorithm, for real time applications, without dedicated hardware support.

A refinement of this scheme, due to Lieberman and Hewitt [LIE-83], is to split the heap into several spaces, rather than just two. This is to prevent the obvious 50% heap redundancy introduced by the simple to- and from- space copying collectors. The scheme also employs the observation that:

the most recently allocated heap cells almost invariably contain the most garbage.

-76-
Section 5.6 Reference Counting Garbage Collection in Real Time

This means that the collector works in different spaces, and gives more time to spaces with newly allocated nodes, since these are more likely to contain garbage.

Nilsen [NIL-88] has produced a real time collector that can handle variable sized objects, and does not make any assumptions that the target language of the collector will be LISP, as assumed in other research.

Copying collectors suffer from some of the problems of mark-scan collection; the action of the collector, in its fundamental form is not distributed in time or space.

The modified schemes of Baker et al. address these problems. However, like the extended reference counting scheme presented in this work, some elegance is lost as compared to the basic collection techniques, in order to provide garbage collection suitable for a real time multi-processor system.

5.6 Reference Counting Garbage Collection in Real Time.

The basic reference counting technique is inherently distributed in time and space. This is because each time a pointer operation is performed by the mutator, the collector amends any reference counts as necessary and normally collects garbage as soon as it arises.

The process of dealing with a reference count for each cell means that the collector acts locally in the heap, and so this minimises mutator-mutator synchronisation in a multi-processor environment. According to Appel et al. [APP-88], Rovner [ROV-85a, ROV-85b] was the first person to demonstrate that reference counting collectors were “unobtrusive” in real time distributed systems. Rovner used the collector for systems programming, an application with very tight real time constraints. However, the system that he employed could not reclaim redundant cyclic structures.

Wise has developed a hardware architecture for a reference counting garbage collector in a multi-processor environment [WIS-85], and argues that reference counting can be a viable technique if hardware support is provided for the update of reference counts.

Though not directly associated with any of the forms of garbage collection mentioned earlier, Duprat [DUP-90] has designed a hardware heap manager that is designed to run in parallel with a graph mutator. The functionality of the unit is limited at the moment to simple identification of space and deallocation of storage. The manager has been implemented in V.L.S.I. (Very Large Scale Integration). Currently its usefulness is strictly limited, since the maximum heap size it can handle is about 4Kbyte.

Since a reference counting garbage collector is essentially distributed in time, it makes the technique inherently suitable in real time applications. However, unbounded time behaviour may occur in exceptional circumstances, such as when large data structures are being recovered. This can occur when the top most cell in a data structure becomes garbage, thus causing the rest of the cells in that structure to become redundant. Garbage cells within the structure are traced recursively, until either non-application nodes, or nodes that are referenced externally from the structure, are encountered. This situation is shown in figure 5.8. (This is an exceptional case since most cells are not shared [STO-84]).
The arbitrarily large structure may become redundant when this pointer is deleted.

Glaser and Thompson [GLA-87] have done work in the area of the collection of large data structures. They suggest the use of a so-called to-be-decremented stack (T.B.D. stack). This stack holds the addresses of cells that are due to have their reference counts decremented. Thus the operation of decrementing a reference count comprises the addition of the cell address to the T.B.D. stack. When a new cell is to be allocated, the top cell on the T.B.D. stack is examined. If its reference count is one, then that cell must be garbage (since its count was to be decremented to zero). This cell can therefore be reallocated, with the addresses of any cells that it points to added to the T.B.D. stack. If, however, the top element of the T.B.D. stack has a reference count greater than unity, then its reference count is decremented, in the usual way, and it is popped from the stack. The process is then repeated with the next cell address on the stack, and so on, until either a cell with a reference count of unity is encountered, or the stack is exhausted, in which case a new cell is allocated from the free list of cells, in the conventional manner. This process of using a T.B.D. stack achieves a form of lazy garbage collection. It overcomes the problem of dealing with large data structures that become redundant, since it eliminates the need to traverse the whole structure at one time. It does, however, introduce a number of other problems, which potentially spoil its real time behaviour:

the height of the stack used could reach $O(2^n)$ addresses high, for a heap size of $n$ cells (assuming a graph comprising binary nodes);

it cannot be guaranteed to respond within bound time for a request to allocate a new node. This is because the top of the stack may be occupied by cells that have reference counts greater than unity. An arbitrary number of cells has to be processed from the T.B.D. stack, before either a cell with a unity count, or the bottom of the stack, is encountered. Thus a supply of garbage cells may be buried under a great number of cells with reference counts greater than one. This situation is shown in figure 5.9.
Section 5.6 Reference Counting Garbage Collection in Real Time

All these cells have to be processed until a garbage cell is encountered, (a cell with a reference count of one).

This garbage cell is "buried" under six non-garbage cells.

Figure 5.9 A T.B.D. stack with buried garbage.

An incremental approach to the collection of large acyclic structures is given in chapter 6. This uses a stack that has a lower upper bound on its size than the T.B.D. stack. It does not, however, solve the problem of buried garbage within the stack.

Another inherent problem of reference counting collection is its inability to collect cyclic structures. As explained in detail in chapter 3, this is overcome by decomposing the graph into its strongly connected components (S.C.C.s) [HUG-82b]. Each S.C.C. has its own reference count. The cycle is collected when this falls to zero. An S.C.C. is identified by performing a Tarjan search [TAR-72], which requires that every node in a cycle is visited once. This operation occupies a time that is bounded by the size of the cycle, which in its turn is bounded only by the size of the heap. This operation may be called during graph reduction, and the mutator is required to suspend operation until the search is over. This obviously negates the real time constraints of timeliness and liveliness.

The solution to this problem is to scan the cycles incrementally, so that the mutator is able to proceed after a maximum bound time. This technique is developed in chapter 6.

Reference counting appears to be an attractive technique for garbage collection, in real time systems. This is with the proviso that the problems of the collection of large structures, and the scanning of S.C.C.s can be overcome. Detailed solutions to these problems are presented in later chapters.

The communications overhead, associated with updating the reference count in a cell each time a pointer operation occurs, can be alleviated by employing weighted reference counts [THO-81, BEV-85]; or by other techniques, such as external pointer indirection nodes [HUG-85]. This makes reference counting even more suitable for distributed, real time systems. For more details refer to chapter 3.

Weizenbaum [WEI-63] developed an approach (called SLIP) to reference counting collection for list structures that could be adapted for use in a real time system. His system was originally developed for list processing applications embedded within a Fortran
environment. The SLIP system maintains a complex structure intended to implement a symmetric list structure. A cell is either a READER, HEADER or a member of a list. All cells contain two separate word pairs. If the cell is a member of a list then the second word pair is either a datum or a pointer to a sub-list HEADER (this is called a list NAME).

A SLIP structure is shown in figure 5.10 (from [WEI-63]).

![Figure 5.10 A SLIP structure.](image)

A HEADER cell is annotated with a "2", a list member with a "0" and a cell containing a NAME with a "1". The READER stack is used to aid list indexing, (providing a history for facilitating the traversal of list structures) constituent cells of this stack are marked with a "3". This stack does not concern us in the discussion of SLIP.
Section 5.7 Real Time Applications for Functional Languages

It can be seen from the figure that the HEADER cell (one of which is associated with each list) maintains a pointer to both the top and bottom of the list to which it is associated. A reference count is maintained in the HEADER for its associated list. When the list becomes garbage it is appended to the bottom of the LAVS (List of Available Space). This corresponds to the free list maintained in most garbage collectors for graph reduction. As cells are required these are taken from the LAVS one at a time.

When a cell is encountered annotated with a "1", indicating it contains the NAME of a sub-list (i.e. a pointer to a sub-list's HEADER) then the reference count for the sub-list named by it is decremented. If this list has become garbage then it too is appended to the bottom of LAVS.

SLIP displays the advantage that entire list may be moved en masse to the LAVS when it becomes redundant; this makes SLIP a candidate for the collection of redundant acyclic structures. However there are a number of serious drawbacks to the system as described by Weizenbaum when applied to a graph reduction architecture. Every "cell" is actually a cell pair, this imposes a massive memory overhead (of the order of a factor of two) over the more usual "boxed" or "unboxed" representation (see chapter 3 and [PEY-87a]). Another objection to the use of this system in a real time system is the processing overhead associated with the addition of every new cell to the graph. When a cell is claimed from LAVS it is either appended to an existing list or forms a list on its own. If the newly claimed cell does not form a part of a acyclic structure (this seems to be the most common situation by far, as reported in [HAR-88]) then SLIP claims a second cell for the HEADER to form a new sub-list (every list, including ones with only a single member have a HEADER). This HEADER must be initialised with a pointer pointing to the first cell in the new cell-pair (as the top of the new sub-list) and another pointer pointing to the second cell in the new cell-pair (as the bottom of the sub-list). The HEADER is also initialised with a reference count of 1. The parent list must have the cell containing the reference to the new cell changed from a data holding cell-pair to one containing a pointer to the new sub-list's HEADER. This involves setting its flag bit from "0" to "1". This procedure compares to the much simpler node addition procedure (as used in the collector described in this work) applied to cells not participating in cyclic structures (which comprise the majority of cells; approximately ninety five per cent as found by simulation, see chapter 9), in which a new cell is claimed from the free list and its reference count is initialised to one. The behaviour of SLIP is similarly processor intensive for the addition of a cell to an existing list. The HEADER cell must be modified to point to the new bottom of the list. This is achieved by traversing the HEADER pointer found in the bottom cell of the list, modifying the pointer and appending the new cell. The new cell must then have a pointer set to the HEADER as well as the cell above it modified from pointing to the HEADER to the newly appended cell.

Whilst SLIP displays advantages for collecting entire acyclic structures in a single operation, there is an excessive price to be paid in terms of processing for the addition of nodes to the graph during "normal" mutator operation. For this reason it is deemed unsuitable in the application of real time systems.

5.7 Real Time Applications for Functional Languages.

Functional programmed systems are of interest in both real time and artificial intelligence environments. This is particularly true in those applications that are supported by a parallel architecture, where a uni-processor approach would not provide sufficient computational
power to meet the needs of the application. Despite the difficulties associated with garbage collection (see above and chapter 3) functional languages have been used for intelligent robotics programs (a real time application) and algebraic manipulation systems (an interactive application) as early as the mid 1970's [STE-75, WAD-76]. Functional languages are also used in other applications with real time constraints. Recently, in Arizona, the language Icon has been used to write interactive text editors and CommonSpeak is used for handling computer communications [NIL-88], whilst the functional language Erlang is used in parallel real time telecommunications systems [ARM-92]. Nilsen also suggests that such applications as computerised medical instrumentation, robotics and voice recognition, that are currently programmed principally in lower-level languages, such as C and assembler, would be enhanced by the use of higher-level functional languages, if these languages could meet the real time responses necessary. Dannenberg [DAN-91] et al. have developed a functional language, named Fugue, designed for computer music applications. He states that in order to achieve real time performance the issue of garbage collection will have to be addressed.

Digital signal processing is an area where very high processing rates and real time response characteristics are demanded. It has been pointed out by Burnett [BUR-87] that, though signal processing covers a wide range of problems, there are a number of common algorithms that dictate the principal requirements of a signal processor. These are namely: the discrete Fourier transform, convolution and recursive filtering. Burnett believes that reduction architectures are almost ideal in principle for digital signal processing applications. This is because the fundamental algorithms, at the centre of digital signal processing techniques, map efficiently onto reduction based architectures [BUR-87]. To exemplify how reference counting is suited to these classes of problems, consider the example of a filter, of the general form:

\[
\text{ipdata} \Rightarrow \text{filter}_1 \Rightarrow \text{filter}_2 \Rightarrow \ldots \Rightarrow \text{filter}_n \Rightarrow \text{opdata}
\]

The input data will typically be a list. Many intermediate lists will be built and garbaged incrementally (due to laziness). In this case reference counting garbage collection is well suited to this problem, since it can cope with the constant flow of new garbage.

Burnett makes the proviso that the inefficiencies traditionally associated with functional language implementations have to be eliminated. Conventional forms of garbage collection are probably the most important source of these inefficiencies.

Winter et al. [BOU-88, WIN-90sa, WIN-90b] have developed a real time functional, multi-processor architecture, CTDNet2. This uses a combination of both lazy and eager evaluation techniques [PEY-87a]. The garbage collection method used is that of a modified form of weighted reference counting. Winter argues [WIN-90b] that reference counting in inherently the most suitable garbage collection technique for real time systems. CTDNet2 is designed to be implemented on a transputer [HOM-88] array.

5.8 Specification of Real Time Systems.

In Górski's paper [GÓR-88] several different formal methods for specifying real time systems are described. The approach taken in the latter part of chapter 6 follows his general guidelines, in using specified system primitives and quantifying system response time in terms of these primitives. The application of deadlines to the correctness of a system may be applied by such techniques as the duration calculus [HAN-93]. This technique was designed for the analysis of highly trusted systems. However, the research is still in a seminal form and not yet suitable for the analysis of the requirements for a large systems. The methods used in
this work to reason about temporal aspects are simple and based upon a simplified execution model of a basic machine (see chapter 6).

5.9 Conclusions.

The early part of this chapter covered the three principal classes of garbage collector and how they could be utilised in a real time environment. Mark-scan collection was considered to be an inappropriate techniques for real time systems. Baker's real time copying collector was also analysed in some detail. This appears to offer a viable collector at the expense of quite a considerable processing overhead.

Reference counting was identified as the most suitable technique for real time functional programming. This was by virtue of the fact that it tends to be inherently distributed in both time and space.

The drawbacks associated with the basic reference counting garbage collection technique were highlighted. Some possible solutions to these were given. In particular the T.B.D. stack and Weizenbaum's methods were introduced and some of their possible shortcomings were given.

The final part of the chapter gave some examples of real time applications for which functional languages are especially suited.

The following chapter explores real time reference counting in greater depth, and develops a number of new techniques to solve the long standing problems of reference counting in real time.

6.1 A High Level Abstract Architecture of the Collector and Mutator.

The highest level view of the entire reduction system is given by diagram 6.1. At this level of abstraction we can consider the system as comprising two major architectural elements, the mutator and the garbage collector (with associated storage manager). There is a communications path between the collector and mutator, this is used to pass mutator interrupts to the collector and signals and data back to the mutator. These interrupts comprise requests for services from the collector, such as delete a pointer, allocate a new node, add a pointer etc. In general the mutator requests primitive operations from the collector and after the collector has performed the operation, or left the graph in a safe state (a state that will not cause non-deterministic behaviour, this concept will be explored in the next section), a signal (handshake) is passed back to the mutator to continue with the graph mutation. It is vital, in a real time system, that the collector responds to a mutator interrupt in a time with a fixed, known upper bound.

![Diagram 6.1: Abstract high level view of the reduction architecture.](image)

6.2 A Basis for the Collection of Large Redundant Acyclic Structures.

As has been seen in chapter 5, the collection of redundant acyclic structures can cause problems because the deletion of a single pointer can cause many other cells to become garbage. Figure 6.2 shows a simple example of an acyclic structure that becomes redundant through the deletion of a single pointer. Normally each cell is traced recursively, (each one becoming garbage in turn, causing the deletion of the pointers to its siblings) until either a non-application node, or a node that is referenced externally from this structure, is encountered. This recursive operation is obviously arbitrarily expensive in time and only terminates after the whole redundant structure has been traversed; such an operation would violate the ability of the collector to respond to a mutator interrupt in a bound time.

Although Weizenbaum [WEI-63] has presented a possible solution to this problem, it was seen in chapter 5 that this scheme is not be generally suitable for real time applications. The solution adopted in this work is to use an incremental technique for collecting such redundant structures. This approach eliminates the problems inherent in the SLIP method, whilst not introducing a massive space and processor overhead, apparent in Weizenbaum's collector. The method appears to offer similar advantages to the T.B.D. stack technique [GLA-87]. The technique of the incremental method is described below with the aid of an example.
Section 6.2 A Basis for the Collection of Large Redundant Acyclic Structures

Figure 6.2 An example of an acyclic data structure.

Figure 6.3 shows an arbitrary data structure, all of which becomes redundant by the deletion of the pointer to node 1. In the case of a standard reference count collector, all fourteen nodes of the structure would be traversed and appended to the free list at once. This might possibly cause an unacceptable delay before the graph mutator proceeds with reduction.

The incremental scheme scans a set number of nodes ($r$), decrementing reference counts and collecting nodes as necessary. During the structure traversal, a pointer deletion stack is used to hold pointers to cells that are not going to be immediately visited. A simple graph traversal rule is employed:

traverse the head pointer, placing the node pointed to by the tail on the pointer deletion stack.

One call to the incremental collector takes the starting node (node 1 in the example given figure 6.3) and places the node pointed to by the tail pointer (node 8) on the pointer deletion stack (see figure 6.4). The node pointed to by the head pointer (node 2) then has its reference count decremented. If the reference count of this node (node 2) has now been decremented to zero then the process is repeated for this node. If, on the other hand, the reference count of the head node (node 2) is not zero then the tail node is visited (node 8) in the same way. The call continues in this fashion until $r$ nodes have been visited ($r$ is set to four in figure 6.3). When the $r^{th}$ node is visited both head and tail nodes (if they exist) are written to the pointer deletion stack. When a non-application node is encountered (one without children), then the call to the collector continues with the node on top of the pointer deletion stack. The call may terminate before $r$ cells have been visited if the entire redundant structure has been traversed and collected. The next call to the incremental collector starts the process with the node on top of the pointer deletion stack (node 6 in the example).

Whilst the data structure is being traversed in this manner, the mutator is running in parallel with this activity; as a consequence we now define a critical section in this technique (a section of the routine that may not be interrupted by the mutator) as one call to the incremental collector (i.e. $r$ nodes are visited without interruption by the mutator). Once a critical section is complete, then the collector is free to service a mutator interrupt (a request from the mutator for a pointer or cell operation). If such an interrupt is waiting, then this is then dealt with immediately, after which the collector continues with the next incremental call to the collector, (which is again an entry into a critical section).
Chapter 6. Real Time Systems

Critical sections are defined in such a way as to leave the graph in a safe state. This is a state that is well characterised so that the mutator interrupt may be serviced and the collector then resume correct operation. If the graph were not left in a safe state, then erroneous collector activity may result.

After the initial call to the collector has taken place and the mutator has progressed the graph reduction process, the collector may have to service a new mutator interrupt. If a pointer addition is required then this is guaranteed to terminate in bound time, since, in this case, only one pointer is added and a reference count updated. If, however, a pointer deletion is requested by the mutator, then this may lead to the destruction of another similarly large data structure. When this situation arises then the structure is scanned, as before, with its nodes being added to the pointer deletion stack as required. If, however, the target node of the deleted pointer is not shared (i.e. has only one reference to it, which comprise the majority of cells), and is not part of a larger redundant structure (i.e. all the children of this cell have reference counts greater than one), then this cell is immediately freed by placing it on the free list of cells, and another partial scan of the data structure can commence. Once this secondary structure has been traversed then the original entity can be dealt with in a similar fashion.

The maximum stack size, for the pointer deletion stack, is theoretically less than that for the T.B.D. stack. For a system with a heap comprising n nodes, the T.B.D. stack could reach O(2n) nodes high, assuming a binary pointer structure. This compares to the incremental system, with, on average, r nodes examined at each call, the maximum height is O(2n)/r. So if r is reasonably large, this theoretical maximum is appreciably less than the T.B.D. stack size. This technique displays an ability to recover cells as they become redundant, comparable with the T.B.D. scheme.

The system could conceivably encounter the pathological case, where the free list has been exhausted, but there are still garbage cells present in a data structure that has yet to be fully scanned. If we consider a cell becoming garbage, this, at most, requires two pointers to be deleted (assuming this is an application node in a binary graph). Thus we can, on average, balance the rate at which such cells become garbage to the rate at which pointers queue up on the pointer deletion stack. This is achieved if we insist that at least two pointers are deleted at each call to the incremental collector (noting that cells cannot be allocated or deallocated whilst the routine is running).

Figure 6.4 shows the contents of the pointer deletion stack along with the progress of examination and collection of nodes, for the redundant structure shown in figure 6.3.
Section 6.3 The Collection of Cyclic Structures in Real Time

In order to cope with cycles within the reduction graph, the technique of identifying the strongly connected components (S.C.C.s) within the graph, using a Tarjan search and then maintaining a separate group reference count for every S.C.C. was presented in chapter 3.
Chapter 6. Real Time Systems

However, recovery of cyclic structures also causes problems in the ability of a garbage collector to respond to mutator interrupts in bound time. It has been shown, in earlier parts of the thesis, that the deletion of a pointer between two nodes in the same cyclic structure, may cause the whole, arbitrarily large, structure to be traversed by a Tarjan search (in order to ascertain the new S.C.C. structure that may result from such an intra-cyclic pointer deletion). This operation is bounded in time only by the size of the cyclic structure being traversed. Such a class of operation, in an unmodified form, is thus inherently unsuitable for a real time environment.

Figure 6.5 illustrates how the deletion of a single pointer can cause the original large cyclic structure to decompose into smaller elements. These smaller elements may themselves be strongly connected components. Such components in figure 6.5 are outlined by a dotted line.

![Figure 6.5 A large cyclic structure splitting into smaller S.C.C.s.](image)

The general idea behind a real time version of the cyclic collection scheme is to make Tarjan’s algorithm incremental. Between each incremental step the collector is free to service an awaiting mutator interrupt. When the collector is performing an incremental step then the mutator is free to run in parallel until it requires a collector service. This situation is analogous to that in section 6.2, where the collector incrementally traverses acyclic structures, in between servicing mutator interrupts.

The Tarjan algorithm has two main elements, each has to be run incrementally, the visit procedure and the emitgroup procedure. Figure 6.6 outlines the original Tarjan algorithm [TAR-72], whilst figure 6.7 gives the emitgroup procedure. The procedure visit basically identifies S.C.C.’s within the graph and places nodes on a stack, whilst emitgroup is called when an S.C.C. has been identified. It pops nodes off the Tarjan stack that make them part of the new S.C.C. The other task of emitgroup is to calculate the group reference count for the S.C.C., this is achieved by adding the local count of each cell to the group count, as it is popped off the stack; if, however, an intra-cyclic pointer is identified from that node then the group count is decremented (so once the S.C.C. is fully emitted the group count equals the...
sum of the local counts of its constituent nodes minus the number of intra-cyclic pointers from these nodes). The mechanism of the Tarjan search is discussed in greater depth in chapters 3 and 8.

PROCEDURE visit (n);
BEGIN
  IF n.onstack THEN
    RETURN;
  ELSE
    INC (dfsn); (* dfsn is the depth-first search number *)
    d := dfsn;
    n.lowlink := dfsn;
    n.number := dfsn;
    spush (n); (* places the node on the Tarjan stack and sets the onstack flag *)
    IF the node n is an application node THEN
      REPEAT
        child ∈ children of node n;
        IF child.onstack THEN
          IF child.number < n.number THEN
            n.lowlink := minimum of (n.lowlink, child.number);
          ENDIF;
        ELSE
          IF same_group (child, n) THEN
            visit (child);
            n.lowlink := minimum of (n.lowlink, child.lowlink);
          END (* IF *);
        END (* IF *);
      UNTIL all children have been visited;
    END (* IF *);
    END (* IF *);
    IF n.lowlink = n.number THEN
      emit_thresh := d;
      emitgroup;
    END (* IF *);
  END (* IF *);
END visit;

Figure 6.6 Tarjan’s original algorithm.
PROCEDURE emitgroup;
BEGIN
  ng := new_group; (* new_group returns a new group descriptor *)
  WHILE NOT EMPTY stack
    LOOP
      pop (stack, y);
      IF (FIRST stack).number < emit_thresh THEN
        temp_stack := (L {y} & temp_stack); (* y is NOT in this SCC *)
      ELSE
        temp_stack := (L {y} & temp_stack); (* y is NOT in this SCC *)
        ng.group_count := ng.group_count + y.local_count;
        IF y.head ∈ spl_grp OR new_group THEN
          ng.group_count := ng.group_count - 1;
        END (* IF *);
        IF y.tail ∈ spl_grp OR new_group THEN
          ng.group_count := ng.group_count - 1;
        END (* IF *);
        Place y in the new S.C.C.;
      END (* IF *);
    END (* LOOP *);
    stack := temp_stack;
  END emitgroup;

Figure 6.7 The original emitgroup procedure.

The next section explains how these algorithms may be made incremental, so allowing then to be suspended whilst a mutator interrupt is serviced by the collector.

6.3.1 Incremental Real Time Version of Tarjan’s Algorithm

Similarly to the case in section 6.2, we define a critical section (a section that cannot be interrupted by the mutator) for both the Tarjan search (visit) and the emitgroup procedures. With emitgroup this is simply one iteration of the loop (so mutator requests can be serviced between calls to the loop). In the procedure visit we define the critical section as one complete call to the procedure (i.e. the section starts when the procedure is called and ends just before it is recursively called again).

In general, a mutator interrupt may be serviced between each call to a critical section, in both visit and emitgroup. The way that such an interrupt (requesting a new node, pointer addition or deletion etc) interferes with the Tarjan algorithm will be explained in the sections dealing with these services. However, modifications have to be made to both visit and emitgroup to allow for this interference.

A number of minor changes have to be implemented in the basic Tarjan algorithm, as presented in figure 6.6, in order to allow incremental action. Figure 6.8 gives a simplified overview of the modified Tarjan search (the modifications are presented in lighter text in figure 6.8).
PROCEDURE visit (n, dfsn);
BEGIN
  IF n.onstack THEN
    RETURN;
  ELSE
    INC (dfsn); (* dfsn is the depth-first search number *)
    d := dfsn;
    n.lowlink := dfsn;
    n.number := dfsn;
    n.visited := TRUE;
    n.scan := FALSE;
    spush (n); (* Places the node on the Tarjan stack and sets the onstack flag *)
    IF the node n is an application node THEN
      REPEAT
        child ∈ children of node n;
        IF child.onstack THEN
          IF child.number < n.number THEN
            n.lowlink := minimum of (n.lowlink, child.number);
          ENDIF;
        ELSE
          IF same_group (child, n) THEN
            visit (child, dfsn);
            n.lowlink := minimum of (n.lowlink, child.lowlink);
          END (* IF *);
        END (* IF *);
        UNTIL all children have been visited;
      END (* IF *);
    END (* IF *
    n.scan := TRUE;
    IF n.lowlink = n.number AND NOT emit_stop THEN
      emit_thresh := d;
      emitgroup;
    END (* IF *);
  END (* IF *
END visit;

Figure 6.8 Modified real time Tarjan algorithm.

In outline, the real time version of the algorithm is very similar to its non-incremental counterpart. The Tarjan search now has to mark nodes, so that the progress of the search, with respect to any particular node, is known by the collector. For this reason a node has two extra flags associated with it, visited and scanned. Prior to being visited by visit for the first time, both flags are set to false. The Tarjan search sets visited true, on encountering a node for the first time and sets scanned true when the recursive loop in the search is complete.

The procedure visit also publishes the value of the depth first search number that triggered the emission of a group. This is held by the global variable emit_thresh, (previously this needed only to be passed to emitgroup). This again is used for determining the state of a node. Additionally, dfsn is made a parameter of visit, since the procedure will be called several times to split a single S.C.C.

The emitgroup procedure is called in the normal way, however a further predicate is introduced on the guard on the call to emitgroup; this is the test on the flag emit_stop. This flag is introduced as a globally visible entity (to the collector) that is set in other parts of the
collector. It signifies that the current group must be split again before the new group identified is emitted. It is reset after the search of the present group is complete and the group has been placed on the queue of cycles waiting to be split.

Some change is necessary to the emitgroup procedure (figure 6.9). The procedure publishes the group descriptor number (emit_group_id) of the new group that it is emitting. An integer count (emit_interior_count) is maintained of the number of intra-cyclic pointers so far encountered within the S.C.C. being emitted, this is used to calculate the final group reference count of the new S.C.C. The group reference count, of the group being emitted, is also published and maintained as a separate variable (group_count). The procedure sets a global flag (emit_flag) on entry to indicate that emitgroup is active, this is reset as the procedure finishes.

The global, emit_interior_count, keeps track of the number of intra-cyclic pointers encountered so far by emitgroup. This count is incremented each time such a pointer is encountered. A pointer is deemed to be between nodes in the S.C.C. that is currently being emitted if the node being dealt with by emitgroup has its child in the group being emitted (emit_group_id) or the child is in the group being split (splgrp) and its number (the depth first search number, set in visit) is greater or equal to the threshold value (emit_thresh) published by visit. This latter case identifies a pointer from the new group, to a node in the group being split, that will migrate from the old group to the new group before emitgroup terminates.

Finally, because other elements of the collector may modify the group count of the new S.C.C. as it is being emitted, (see the next sections explaining how the mutator may interfere with emitgroup), the final group count for the new S.C.C. (emit_group_g_count) must reflect this interference. Also, it is possible, due to this interference, that this S.C.C. has become garbage during its emission; emitgroup tests for this and if the group is now garbage it is collected. As mentioned above one iteration of the loop within the procedure is defined as a critical section; emitgroup can be suspended to service a mutator request at the bottom of each iteration of the loop. The process of collecting a group as garbage also has to be made incremental. This is discussed towards the end of this chapter.
PROCEDURE emitgroup;
BEGIN
  group_count := 0;
  emit_interior_count := 0;
  emit_flag := TRUE;
  emit_group_id := new_group(); (* new_group returns a new group descriptor *)
  WHILE NOT EMPTY stack LOOP
    pop (stack, y);
    IF NUMBER FIRST stack < emit_thresh THEN
      temp_stack := (L (y) & stack); (* y is NOT in this SCC *)
    ELSE
      group_count := group_count + y.local_count;
      IF y.head = fg OR ee THEN
        emit_interior_count := emit_interior_count + 1;
      END (* IF *);
      IF y.tail = fg OR ee THEN
        emit_interior_count := emit_interior_count + 1;
      END (* IF *);
      Place y in emit_group;
    END (* IF *);
  END (* LOOP *);
  stack := temp_stack;
  emit_group.g_count := emit_group.g_count + group_count - emit_interior_count;
  emit_flag := FALSE;
  IF emit_group.g_count <= 0 THEN
    append all members of the S.C.C. to the free list and destroy the group;
  ENDIF;
END emitgroup;

Figure 6.9 The real time emitgroup procedure.

Having presented the modifications necessary to Tarjan's algorithm, in order to allow incremental collection and servicing of mutator interrupts between critical section calls, the ensuing sections present the services provided by the collector to the mutator. The process of graph reduction, performed by the mutator, can be considered in two distinct phases. The first is the addition of new nodes and pointers in the graph, and the second is the overwriting of the root node of the reduction (which involves the deletion of pointers).

We start the discussion of the basic collector services, with the process of pointer deletion, since it transpires that this is the simplest to comprehend and so provides an introduction to the real time collector design.

6.3.2 The Deletion of Pointers and Nodes.

This service provided by the collector deletes a pointer from a parent node to a child node. It maintains both the local and group reference counts, where necessary, and initiates the collection of any garbage resulting from the pointer deletion.

The deletion of a pointer always causes the local reference count of a child node to be decremented. Recall, from chapter 3, that it may also cause an S.C.C. to split (if it is between two nodes in the same S.C.C.), or if it is between nodes in different S.C.C.s then the target S.C.C.'s group reference count is decremented. The action of pointer deletion can neither cause an S.C.C. to grow or merge with another.
Figure 6.10 shows the non-incremental version of the deleteref algorithm. In essence it is simple (for simplicity of presentation only application nodes are considered here). The child of a pointer deletion always has its local reference count decremented. If the child and parent are in different groups then the child's group reference count is decremented. The group is collected as garbage if the group count has fallen to zero, otherwise if the child has a zero local reference count then the head and tail grandchildren have their reference counts decremented. (This is actually done in an incremental way, as described in section 6.2; however for the sake of clarity the incremental collector will not be shown, its interaction with the collector service routines is discussed in section 6.7). The cell is then pushed onto the free list.

If the child and parent are in the same group then if the child's local count is non-zero the group must be split (using visit and emitgroup) since an intra-cyclic pointer has been deleted. This is performed immediately (with the non-incremental version of this algorithm). If, however, the child's local count has fallen to zero, then its children have their reference counts decremented by deleteref and the child is pushed onto the freelist as garbage; since an intra-cyclic pointer has been deleted the S.C.C. must be split, but the child cannot be used as the starting point since it has been collected. The answer is to collect the group, starting with the grandchild, that is in the S.C.C. to be split.

PROCEDURE deleteref (parent, child);
  decrement_local_referencecount (child);
  IF parent AND child are NOT in the same group THEN
    decrement_group_referencecount (child.group);
    IF the child's group_count = 0 THEN
      append all members of the S.C.C. to the free list and
      destroy the group;
    ELSIF child's local_count = 0 THEN
      deleteref (child, head);
      deleteref (child, tail);
      freecell (child);
      END (* IF *);
    ELSIF parent AND child are in the same group THEN
      IF child's local_count = 0 THEN
        deleteref (child, head);
        deleteref (child, tail);
        freecell (child);
        split the parent and child's group, starting from the grandchild that
        is in the same group as the child and parent;
        END (* IF *);
        split the parent and child's group, starting from the child node;
        END (* IF *);
      END (* IF *);
  END deleteref;

Figure 6.10 Deletion of a pointer using deleteref.

The first change that has to be considered when implementing an incremental version of the collector is that many groups may be split at one time. For instance, suppose the collector was splitting an S.C.C. using the incremental version of the Tarjan search. At the end of a critical section a mutator interrupt is serviced to delete an intra-cyclic pointer in a different S.C.C. This would require the collector to start splitting a separate S.C.C. before it had finished the first one. In order to simplify the nature of the interference between the mutator and collector we insist that only one group should be split at a time. If there were more than one group being split at a time (as outlined above) then this interaction would be complicated to characterise. When the collector is busy splitting an S.C.C., if another should require splitting this is postponed, on a queue of S.C.C.s waiting to be split, until the current split has
completed. The requirement to maintain at most one group being split at a time, necessitates
the introduction of three additional data structures.

The first is a group handle, this was introduced in chapter 3 and is discussed in greater
depth below. A group handle is a set of destination nodes when an intra-cyclic pointer deletion
has occurred. It allows all members of an S.C.C. to be traced, via its handle, even when
pointers within the cycle have been deleted.

The second is a queue of S.C.C.s waiting to be split, this is the split_queue. It is required
since we do not now split groups as soon as an intra-cyclic pointer deletion has occurred.

The third arises from the consideration of a node's local count falling to zero when it is
part of the S.C.C. currently being split. If the node were simply collected by appending it to
the free list the context of the node (its original children) would be lost to visit or emitgroup
(since the free list is actually efficiently maintained as a linked chain through the left hand
pointers of each garbage cell on the list). This leads to the introduction of the cyclic_free_list,
that holds nodes that can be safely deleted after the current scan of an S.C.C. has completed.
This means that the freecell procedure tests the cell about to be freed. If this node is in the
group currently being split, or in the group being emitted by emitgroup then the node is
placed on the cyclic_free_list. This stack is incrementally traversed once the current call to
the Tarjan search is over and its constituent nodes can be safely appended to the free list. The
stack maintains the integrity of the pointer structure, for the constituent nodes of the list. This
problem does not arise for S.C.C.s on the split_queue, since the handle for the group copes
with such node deletions and because the Tarjan search is not actively searching these groups
the loss of the deleted nodes context is unimportant.

The concept of the group handle is now explored in greater depth. As explained above,
with the simple scheme, as soon as an intra-cyclic pointer is deleted, a call is made to visit to
split the group, once this call has finished then the mutator may proceed. In the real time
scheme only one Tarjan search is permitted to be active at any one time (and hence a
maximum of one call to emitgroup). This is to maintain the simple model of mutator
interference, that is being built up in this chapter. As a result of this imposed restriction
groups have to be queued on the split_queue, to await splitting.

There is a problem with this scheme of maintaining a split_queue. An example of this
problem is given in figure 6.11.

![Figure 6.11 Deletion of a pointer in a cycle waiting to be split.](image-url)
Figure 6.11 shows a cycle waiting to be split, while waiting an intra-cyclic pointer is deleted (due to the collector servicing a mutator interrupt at the end of a critical section in Tarjan’s algorithm). This would ordinarily prevent node 1 being accessed when the Tarjan search is finally called. The answer to this problem is to maintain a group handle. This is a set of nodes that are the destinations of deleted pointers for any particular S.C.C. In the case given above, node 1 would be placed on the handle for this S.C.C. When a group is first identified for splitting, by the deletion of an intra-cyclic pointer, the initial destination node forms the first member of the group handle. This serves as a starting node for the Tarjan search.

If we formally define a handle, $H$, of an S.C.C., $G$, (with edges $E$ and a constant set of nodes, $N$) to be a set of nodes through which all members of the original S.C.C. may be traversed and initially there is an edge from nodes $a$ to $b$ in $G$, that is deleted i.e.

$$(\exists a : a \in G) \land (\exists b : b \in G) \land (\exists a \rightarrow b : a \rightarrow b \in E') \land \lnot(\exists a \rightarrow b : a \rightarrow b \in E)$$

Where primed objects refer to initial states (i.e. $E'$ is the set of edges before the pointer deletion and $E$ is the set after deletion. This convention is used throughout this thesis).

A handle for $G$, after the pointer deletion is:

$$H = H' \cup \{b\}$$

Proof

Consider nodes $c$ and $h$ such that:

$$(\exists c : c \in G) \land (\exists h : h \in H') \land (\exists h \rightarrow c : h \rightarrow c \in E')$$

:. for any accessible path $P$

$$(\forall P' h \rightarrow c : (a \rightarrow b) \notin (h \rightarrow c)) \Rightarrow (\exists P h \rightarrow c) \lor (\forall P' h \rightarrow c : (a \rightarrow b) \in (h \rightarrow c)) \Rightarrow (\exists P h \rightarrow c)$$

Therefore, a group handle is maintained for every S.C.C. awaiting splitting, comprising the destination nodes of all intra-cyclic pointer deletions, affecting the S.C.C.

We now address the problem of recovering the nodes in an S.C.C. that has become garbage. Since an S.C.C. may be of indeterminate size, the process of appending all its members to the free list takes an indeterminate time, since each node must be traced in turn. Because of this, when a group count falls to zero, the group is not immediately collected, but rather the group descriptor pointer is placed on a stack; so a fourth data structure is introduced, the group_dispose_stack, that allows groups to be traversed incrementally, collecting the members of that group onto the free list. The process of traversing the group employs the group handle to facilitate collecting all the nodes of each garbage S.C.C.

Due to the incremental scheme, the mechanisms for pointer deletion need to be modified to accommodate the interference so caused with the Tarjan search. There are distinct situations that have to be explored; the first is if the search is not operating on either the parent or child node. This case is identical to the non-incremental scheme and no special action need be taken.
Section 6.3 The Collection of Cyclic Structures in Real Time

The second case is that the search has not reached the source and destination nodes, but they are in the group currently being split. Again, no special action need be taken since any change in the cyclic structure will be taken into account by the search, when the new nodes are finally visited (accepting the fact that freecell will append the child node to the cyclic_free_list if it is deleted rather than the free list).

The third case involves the situation where the child node has been visited, but not the parent. Figure 6.12 shows part of a cyclic structure, the shaded nodes have been visited by the Tarjan search. This situation requires only slight modification from the ordinary scheme. The destination node does not have to be added to the group handle, since it has already been visited.

![Figure 6.12 Deletion of a pointer from a partly scanned cycle.](image)

The fourth case, illustrated in figure 6.13, concerns the situation where both source and target nodes have been visited and the target node is not in a newly emitted group. In this situation we cannot “recover” from the pointer deletion, so the S.C.C. is put on the split_queue, in order to be resplit.

If the target node is in the group currently being emitted (but the source is not), then this is treated as a normal inter-cyclic pointer deletion. The same is true if the child is in the group being emitted and the parent is in the group being split. If both parent and child are in the group being emitted then this too must be added to the split_queue in the normal fashion, to be resplit.

![Figure 6.13 Deletion of a pointer from a fully scanned cycle.](image)

When the current group is being split and emitgroup is not active, then if a resplit of the S.C.C. currently being scanned is necessary (as in figure 6.13), the emit_stop flag is set, so that
the group will be resplit before the new S.C.C. is emitted. If the group emission had proceeded in the normal fashion then this would constitute an erroneous action by the collector, since the whole of the original S.C.C. must be resplit.

When a node is deleted, then it is removed from the appropriate group handle (if it was originally a member of the handle). This is so the collector does not try to trace an S.C.C. via garbage nodes.

As outlined earlier, in section 6.3.1, it may be determined if a node will form part of the S.C.C. being emitted, if emitgroup is active and node.number ≥ emit_thresh. With this insight the rest of the action of this algorithm is straightforward and is presented in figure 6.14.
PROCEDURE deleteref (parent, child);
  decrement_local_referencecount (child);
  IF child is not in group being split or emitted THEN
    push_split (child's group_id);
    IF child’s local_count = 0 THEN
      remove_from_handle (to, child’s group);
      deleteref (child, head);
      deleteref (child, tail);
      freecell (to);
    ELSIF child’s local_count ≠ 0 THEN
      (* push node onto handle to guide later splitting *)
      push_handle (to, child’s group_id);
    END (* IF *);
  ELSEIF parent AND child are NOT in the same group THEN
    decrement_group_referencecount (parent.group);
    IF the parent’s group_count = 0 THEN
      append all members of the S.C.C. to the free list and
      destroy the group;
    ELSIF child’s local_count = 0 THEN
      remove_from_handle (to, child’s group);
      deleteref (child, head);
      deleteref (child, tail);
      freecell (to);
    END (* IF *);
  ELSEIF child is in the group being split, but not yet visited THEN
    (* i.e. we are already splitting the group the child belongs to *)
    IF parent AND child are in the same group THEN
      (* no need to put the SCC on the split_queue since it is already being split *)
      IF child’s local_count = 0 THEN
        (* no need to remove it from the handle, this will be done by
          process_cyclic_fee_list at the end of this group split *)
        deleteref (child, head);
        deleteref (child, tail);
        freecell (to);
      END (* IF *);
    ELSE
      (* push node onto old group’s handle to guide current Tarjan search,
        no need to add current group to the split_queue *)
      push_handle (to, spl_grp);
    END (* IF *);
  ELSEIF parent AND child are NOT in the same group THEN
    (*do not decrement group count of old group this will be recalculated by emit*)
    IF child’s local_count = 0 THEN
      deleteref (child, head);
      deleteref (child, tail);
      freecell (to);
    END (* IF *);
  ELSEIF child has been visited, emitgroup has been called and the child
    will NOT form part of the SCC being emitted THEN
    (* child will NOT form part of the SCC currently being emitted, but
      has been visited by the Tarjan search *)
    IF parent is not in the group being split OR parent is in the group
    being emitted OR parent will be in the SCC being emitted THEN
    (* an external pointer from the new group, or other group,
      to the old one, when the parent is emitted, no need to decrement old group’s
      grc since this will be recalculated by emit *)
    IF child’s local_count = 0 THEN
      deleteref (child, head);
      deleteref (child, tail);
      freecell (to);
    END (* IF *);
ELSIF parent is in the group being split, but has not yet been visited THEN
(* no need to add to handle, since we have already visited the child *)
IF child’s local_count = 0 THEN
  deleteref (child, head);
  deleteref (child, tail);
  freecell (to);
END (* IF *);
ELSIF parent is in the group being split, emitgroup is active and parent will NOT form part of the SCC being emitted THEN
(* have to resplit the OLD group after emission has occurred *)
push_split (spl_grp);
IF child’s local_count = 0 THEN
  deleteref (child, head);
  deleteref (child, tail);
  freecell (to);
ELSE
  push_handle (to, spl_grp);
END (* IF *);
END (* IF *);
ELSIF child is in the group being split, emitgroup is active and it will form part of the SCC being emitted THEN
IF parent is not or will not be in the SCC being emitted THEN
(* we know that this is an external pointer to the new group, emit will calculate the new group’s grc *)
IF child’s local_count = 0 THEN
  deleteref (child, head);
  deleteref (child, tail);
  freecell (to);
END (* IF *);
ELSIF parent is or will be in the SCC being emitted THEN
(* an internal pointer in the new group, split the new group after emission *)
push_split (emit_group_id);
IF child’s local_count = 0 THEN
  remove_from_handle (to, emit_group_id);
  deleteref (child, head);
  deleteref (child, tail);
  freecell (to);
ELSE
  push_handle (to, emit_group_id);
END (* IF *);
END (* IF *);
ELSIF child has been visited and emitgroup is NOT active THEN
IF parent is NOT in the group being split THEN
(* this is an external pointer, since we know that the child is in the group being split *)
IF child’s local_count = 0 THEN
  deleteref (child, head);
  deleteref (child, tail);
  freecell (to);
END (* IF *);
ELSIF parent is in the group being split and has not been visited THEN
(* skip - no need to add to handle *)
IF child’s local_count = 0 THEN
  deleteref (child, head);
  deleteref (child, tail);
  freecell (to);
END (* IF *);
ELSIF parent is in the group being split and has been visited THEN
(* an internal pointer in the old group, split the old group before emission *)
push_split (spl_grp);
Section 6.3 The Collection of Cyclic Structures in Real Time

IF child's local_count = 0 THEN
    deleteref (child, head);
    deleteref (child, tail);
    freecell (to);
ELSE
    push_handle (to, spl_grp);
END (* IF *);
emite_stop := true;
END (* IF *);
ELSIF child is in the SCC being emitted THEN
    IF parent will not be in the SCC being emitted THEN
        (* this is an external pointer, since we know that
        the child is in the new group that was emitted,
        decrement the new group's grc, do not collect it if it is
        garbage, emit will do that *)
        decrement_group_referencecount (parent.group);
        IF child's local_count = 0 THEN
            deleteref (child, head);
            deleteref (child, tail);
            freecell (to);
        END (* IF *);
    ELSIF parent is or will be in the SCC being emitted THEN
        (* an internal pointer in the new group, split the new group after emission *)
        push_split (child's group_id);
        IF child's local_count = 0 THEN
            remove_from_handle (to, child's group);
            deleteref (child, head);
            deleteref (child, tail);
            freecell (to);
        ELSIF child's local_count != 0 THEN
            (* push node onto handle to guide later splitting *)
            push_handle (to, child's group_id);
        END (* IF *);
    END (* IF *);
ELSIF child is in the SCC being emitted THEN
    (* this is an external pointer, since we know that
    the child is in the new group that was emitted,
    decrement the new group's grc, do not collect it if it is
    garbage, emit will do that *)
    decrement_group_referencecount (parent.group);
    IF child's local_count = 0 THEN
        deleteref (child, head);
        deleteref (child, tail);
        freecell (to);
    ELSE
        push_handle (to, spl_grp);
        END (* IF *);
    ELSIF parent is or will be in the SCC being emitted THEN
        (* an internal pointer in the new group, split the new group after emission *)
        push_split (child's group_id);
        IF child's local_count = 0 THEN
            remove_from_handle (to, child's group);
            deleteref (child, head);
            deleteref (child, tail);
            freecell (to);
        ELSIF child's local_count != 0 THEN
            (* push node onto handle to guide later splitting *)
            push_handle (to, child's group_id);
        END (* IF *);
    END (* IF *);
END deleteref;

Figure 6.14 The real time deleteref procedure.
6.3.3 The Addition of Pointers and Nodes.

Recall from chapter 3 that the addition of a node and pointers to a graph can either cause an existing S.C.C. to grow by incorporating the new node into the S.C.C. (if it is on a path from one node in an S.C.C. to another in the same S.C.C.), or the new node forms an S.C.C. by itself. The target node (child) of a pointer addition always has its local reference count incremented. The addition of nodes and pointers can only cause S.C.C.s to grow or stay the same size, never merge, shrink or split.

First, consider the addition of an application node to the graph, by a procedure newapp. Figure 6.15 outlines the basic, non-incremental version of newapp. This takes two parameters, the first is the parent node of the new application node, the second is the new node itself, already loaded with its pointers to its children, the head and tail pointers of newnode.

PROCEDURE newapp (parent, newnode);
BEGIN
  IF newnode.head OR newnode.tail e parent's group THEN
    incorporate newnode in the parent's group;
  ELSE
    newnode is a singular node;
  ENDIF;
  addref (newnode, newnode.head);
  addref (newnode, newnode.tail);
END newapp;

Figure 6.15 Addition of a new application node.

This procedure simply tests if the newnode is on a path from the parent to another node in the same S.C.C. as the parent. The discussion in chapter 3 shows that such a node must be part of the parent’s group, otherwise it forms a singular node on its own. The action of addref (the addition of a pointer) will be discussed later in this section.

The action of the incremental Tarjan search complicates this simple situation. If the parent node is not a member of the S.C.C. being emitted then the situation is unchanged, since the Tarjan search will not affect this node.

However, if the parent node is a member of the S.C.C. currently being scanned by the Tarjan search and has not been fully scanned by visit (i.e. the node has not been marked as scanned by visit) then this situation too is unchanged, since the Tarjan search will take the new structure into account when the parent node is revisited by visit (note that the Tarjan search visits all children of a parent node).

Finally, if the parent node is fully scanned (visit has set the scanned flag for the parent node), then any changes to the graph, affecting the parent node, will not be taken into account by the Tarjan search (since this node will not be visited again by the search). In this eventuality special action must be taken to try to recover the cyclic structure of the S.C.C. being split. If this cannot be achieved then the S.C.C. must be resplit.

As an aside it may be thought that once a node has been fully scanned then it will be guaranteed to form part of the next S.C.C. to be emitted, this is the assumption made by Hughes [HUG-87]. This is not necessarily the case [HUG-92]. Figure 6.17 gives an example of part of a stylised form of the graph structure initially produced (before the reduction...
The Collection of Cyclic Structures in Real Time

Section 6.3 The Collection of Cyclic Structures in Real Time

process starts) from a functional program used to compute prime numbers using the Sieve of
Eratosthenes. In figure 6.16 the functional program, that produced the graph is presented. The
language in which it is written is EASE, which is a simple S-expression language and is used
in the simulator. This is discussed in greater depth in chapter 9.

```
((program primes)
 (primes (sieve (from 2)))
 ((sieve s) (if (= (head s) 31) 31 (cons (head s) (sieve (filter (relprime
 (head s)) (tail s))))))
 ((relprime p n) (if (= p n) false (if (<= p n) (relprime p (- n p))
 true))
 ((filter p 1) (if (p (head 1)) (cons (head 1) (filter p (tail 1)))
 (filter p (tail l))))
 ((from n) (cons n (from (+ 1 n))))
 (false 0)
 (true l))
```

Figure 6.16 Sieve of Eratosthenes EASE program.

Figure 6.17 Part of the directed graph produced by the Sieve of Eratosthenes.

The nodes are actually visited in the order, given in figure 6.18. The nodes, which are
numbered in italic in figure 6.18, are emitted as singular nodes, separate from any cyclic
structure. The nodes marked in bold, are emitted as part of strongly connected components
within the graph. Nodes 42 and 43 are marked as being fully scanned, i.e., all their children
have been visited, but the group that incorporates nodes 157, 113 and 131 is emitted before
group comprising nodes 42, 87, 89 and 92. This demonstrates that even though a node may be
marked as being fully scanned, it does not necessarily form part of the next group to be
emitted.
Chapter 6. Real Time Systems

Nodes 92, 89, 87 and 42 form a complete S.C.C.

Nodes 79 ... 43 form singular, non-cyclic components of the graph.

Nodes 157, 113, 131... form part of a separate S.C.C.

Figure 6.18 Order of visitation of nodes.

Instances occur where one node is fully scanned, but a relative of it, in the same S.C.C., may not yet have been visited by the search, i.e., the descendant node is not on the Tarjan stack. Figure 6.19 shows a stylised part of a graph produced by a simple, if rather esoteric, EASE script, given in figure 6.20. All the nodes shown form a single S.C.C.

The node marked as 1, in figure 6.19, reached a state of being fully scanned before node 2 is visited for the first time. Therefore no inference with regard to nodes occupying the same S.C.C., can be drawn from the fact that a node has not yet been visited, even when some nodes have reached a fully scanned status within a particular cycle, (if emitgroup is not active).

Figure 6.19 Progress of a Tarjan search on a single S.C.C.

Figure 6.20 An example EASE program.

```
((program (cycle 1 2 4))
 ((cycle p q r) (if (= p 1) (cons q (cycle 2 99 4))
          (if (= p 2) (cons (cycle 3 99 4) (cycle 3 99 4))) q)))
```
Section 6.3 The Collection of Cyclic Structures in Real Time

The above discourse gives us the insight to discuss when a node, that has been marked as fully scanned by visit (scan flag set), can be characterised with respect to membership of the S.C.C. currently being emitted by emitgroup (or, if emitgroup is not active, the next S.C.C. to be emitted). If emitgroup is active (emit flag is set), then if the node has a depth first search number (node.number) greater or equal to the emit threshold (emit_thresh), then we know that it will eventually be moved by emitgroup from the old S.C.C. to the one currently being emitted. On the other hand, if emit_flag is not set then we cannot predict if the node will migrate to the next S.C.C. to be emitted (since we cannot predict the value of emit_threshold), or remain in the old S.C.C. (to be emitted in another S.C.C. at a later time).

With this knowledge we can now modify newapp to deal with the situation when the parent node has been marked as fully scanned by visit.

If the parent has been fully scanned by visit, emitgroup is active and the dfsn of the parent is lower than the threshold value (so it will not be in the next S.C.C. to be emitted), then if one of the children of the new node is in the same group as the parent (and is also fully scanned and has a dfsn lower than the threshold), this implies that the new node must also form part of the group to which the parent belongs; in this case this is the group currently being split by the Tarjan search. As already discussed, it is too late for the Tarjan search to take such a modification in the reduction graph into account, since the parent is fully scanned and will not be revisited. As a result we do not know if this is an inter or intra pointer operation with regards to the new S.C.C. structure that will exist after the current Tarjan search has completed. Without this knowledge we cannot place the child in an S.C.C. relevant to the cyclic structure after the Tarjan search has split the current S.C.C. As a consequence of this, the current group being scanned by the Tarjan search must be resplit after the current group has been emitted by emitgroup. (We can safely let emitgroup proceed since we know that the new node would not have been placed in the group emitgroup is processing). This resplit is achieved by placing the group currently being split (spl_grp) on the queue of groups waiting to be split, the split_queue. The new node is incorporated into spl_grp so that it will be correctly visited by visit when the group is resplit. The parent is also placed onto the handle of spl_grp, to guarantee that there is at least one starting point to resplit the old group.

If, instead, we consider the situation, where the parent node is fully scanned, emitgroup is active and the dfsn of the parent is lower than the threshold value (as above) but either the head or tail nodes are also fully scanned and have a dfsn above the threshold (and the other child is not a member of the group being split) then we know that the new node is external to the group currently being emitted, (since the child of the new node will be in the S.C.C. currently being emitted, whilst the parent is guaranteed not to be). Because of this the new node will form a singular trivial (non-cyclic) S.C.C. by itself.

Finally, again considering the parent in the same state as above, if both children are in any state other than those identified above then this is a normal exterior pointer to an S.C.C., so the new node forms a singular S.C.C. as before.

The situation for the parent being in a state where it is marked as fully scanned, but emitgroup is not active, is rather simpler than above. If either head or tail nodes are in the same group as the parent then the new node forms part of the group being scanned; however, since the Tarjan search will not revisit the parent node we must resplit the current S.C.C. This
must be done before emitgroup proceeds, since the structure of the S.C.C. next to be emitted may be affected by this pointer addition. This is achieved by setting the emit_stop flag and pushing spl_grp onto the split_queue. There is no need in this case to push the parent node onto the group's handle, since the original group handle will be restored before the old group is resplit.

Now we consider the case of the parent node being in a state where it is marked as fully scanned, emitgroup is active and the parent node's dfsn is set above the threshold value. In this situation we know that the parent node will form part of the S.C.C. being emitted. Because of this consideration, if either the head or tail nodes of the new node are in the S.C.C. being emitted, then the new node is on a path from a node that will be in the new S.C.C. to one that is already in it, therefore it is incorporated into the group being emitted. The group reference count of the new S.C.C. is incremented in the normal way. The situation is similar if either the head or tail nodes are marked as fully scanned, emitgroup is active and the node's dfsn is set above the threshold value, which implies they too will form part of the S.C.C. being emitted, so the new node is in a path between two nodes that will be in the S.C.C. being emitted. The only difference is that we do not increment the new group's group reference count since this will be done by emitgroup as the parent is emitted.

The final case to consider is the parent has already been emitted as part of the group currently being emitted by emitgroup. If the head or tail nodes are marked as fully scanned and have a dfsn greater or equal to the threshold value, then the new node is incorporated into the new group. The same is true if either of the children is already a member of the new S.C.C.

The above discussion has covered a number of cases, with parent and child nodes in different states. This motivates us to introduce the notion of scan_state of a node. It allows a concise discussion of the states that a node may take, so enabling a precise characterisation of the behaviour of the addition of nodes and pointers with respect to these states. It should be noted that not all of the states identified below are necessary. The states nn, pp and qq could be amalgamated, but we keep them separate for clarity and for use by MALPAS in the proof of the techniques (see chapter 8).
The scan-state of a node is given by the following:

\[ xx: \] Not in the group currently being split or emitted;
\[ nn: \] In the group being searched \((spl\_grp)\) but the search has not yet reached it;
\[ pp: \] In the group being searched, the Tarjan search has visited it, but the loop in the Tarjan search is not yet complete, \texttt{emitgroup} is not active;
\[ qq: \] In the group being searched, the Tarjan search has visited it, but the loop in the Tarjan search is not yet complete, \texttt{emitgroup} is active;
\[ f_i: \] In the group being searched, the loop in the Tarjan search is complete, but the \texttt{emitgroup} procedure is not active yet;
\[ f_x: \] In the group being searched, the loop in the Tarjan search is complete, the \texttt{emitgroup} procedure is active, and \texttt{node.number} < \texttt{emit\_threshold} \((n.num < et)\);
\[ f_e: \] In the group being searched, the loop in the Tarjan search is complete, the \texttt{emitgroup} procedure is active, and \texttt{node.number} ≥ \texttt{emit\_threshold};
\[ ee: \] In the group being currently emitted \((grp\_em)\) by the \texttt{emitgroup} procedure.

The above is perhaps best presented in a table (table 6.1). The visited column represents a node as having first been visited by the Tarjan search, the scanned column represents the loop in the Tarjan search having finished for that node. The heading \texttt{in spl\_grp} indicates that the node is in the group currently being split by the Tarjan search and \texttt{in grp\_em} implies the node is in the S.C.C. currently being emitted.
Table 6.1 The scan condition of a node.

In the above table a question mark indicates either a “don't care” or an impossible state. It should be noted that not all combinations of states are possible, for instance it is not possible to have a node in the $f_x$ state and for emitgroup to be active.

We can now summarise the above discussion, concerning newapp and the way in which the new node is incorporated into the existing S.C.C. structure in the graph, in the figures below. These do not cover every case but seek to clarify the major departures from the standard scheme. It should also be noted that the symmetric case exists for the tail node of the new node.

Figure 6.21a
Section 6.3 The Collection of Cyclic Structures in Real Time

It is also necessary to consider the maintenance of the group reference count and \textit{emit\_interior\_count} (the total number of intra-cyclic pointers encountered whilst emission occurs) by \texttt{emitgroup} as new nodes are added in this way. The procedure \texttt{emitgroup} must correctly maintain the difference between the number of intra and inter cyclic pointers as nodes are examined. At the end of the emission of the new group this difference becomes the S.C.C.'s group reference count. This should not change for a node addition into the new S.C.C. as described above, because the new node to be incorporated into the S.C.C. has no external references to it (the pointer to it is from another node, its parent, in the same S.C.C.).
So this invariant must be maintained under interference by the mutator (the mutator is interfering with the emission of nodes, in this case, by adding a new node). Only with parent nodes in the states $f_e$ or $ee$ does any special action have to be taken, in all other states these totals will be maintained in the normal fashion when the nodes are processed by emitgroup. First, consider diagram 6.22, with parent node in a $f_e$ state and one or more of the head and tail nodes in the $ee$ state. The state $y$ indicates a state other than $ee$ or $f_e$. The S.C.C. currently being emitted must have its group reference count incremented, this is because as the parent is emitted the emit_interior_count will be incremented (since the new node is now part of the S.C.C. being emitted), but the new node will not be processed by emitgroup, so incrementing the group reference count by the new node's local count. As a result in all cases the group reference count must be incremented by one, to compensate for this.

In case a) neither the head or tail nodes will be processed by emitgroup since the addition of the new node, because they have already been emitted into the new group. As a result we must further increase the group count by two. As the parent is emitted emit_interior_count will be incremented, however, it must be augmented by a further two, to reflect the two internal pointers to the head and tail nodes from the new node. So, in total (outside emitgroup) we increment the internal count by two and the group count by three.

In case b) the head and tail nodes will be considered by emitgroup, so the group count need only be increased by one and emit_interior_count by two, to compensate for the new node.

Cases c), d) and e) can be considered in a similar manner, to derive the necessary corrections to emitgroup's counts.
Section 6.3 The Collection of Cyclic Structures in Real Time

Figures 6.22 The addition of a new application node during group emission.

It is also necessary to consider the corresponding cases for the parent node in the ee state. Since the parent will not be considered by emitgroup after the addition of the node, it is necessary to increment emit_internal_count and the group reference count in all cases (a to e). It is now possible to summarise the results in figure 6.22, symmetric cases are not presented in the table (table 6.2).

The real time implementation of the newapp algorithm is given in figure 6.23.
### Table 6.2 Compensation of `emitgroup` count totals for `newapp`.

<table>
<thead>
<tr>
<th>par_sc</th>
<th>hd_sc</th>
<th>tl_sc</th>
<th>grp_count</th>
<th>em_in_cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f_e)</td>
<td>(ee)</td>
<td>(ee)</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>(f_e)</td>
<td>(f_e)</td>
<td>(f_e)</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(f_e)</td>
<td>(f_e)</td>
<td>(ee)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(f_e)</td>
<td>(y)</td>
<td>(ee)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(f_e)</td>
<td>(y)</td>
<td>(f_e)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(ee)</td>
<td>(ee)</td>
<td>(ee)</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>(ee)</td>
<td>(f_e)</td>
<td>(f_e)</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>(ee)</td>
<td>(f_e)</td>
<td>(ee)</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>(ee)</td>
<td>(y)</td>
<td>(ee)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(ee)</td>
<td>(y)</td>
<td>(f_e)</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
PROCEDURE newapp (parent, newnode);
BEGIN
  IF parent scan_con = xx OR nn OR pp OR qq THEN
    (* the new structure will be taken into account by the Tarjan search
      when it returns to the parent node *)
    IF newnode.head OR newnode.tail ∈ parent's group THEN
      (* cannot have a node spanning two groups, so incorporate it into
        the parent's group *)
      incorporate newnode in the parent's group;
    ELSE (* must be a singular node *)
      newnode is a singular, trivial node;
    END (* IF *);
  ELSIF parent scan_con = fx THEN
    IF newnode.head = nn OR qq OR fx OR
      newnode.tail = nn OR qq OR fx THEN
      (* old group is to be resplit, after emission. The child and
        parent are both in the old group, cannot rectify
        SCC structure *)
      push_split (spl_grp); (* only push if NOT already on stack *)
      (* push the parent onto the handle to guarantee that there is a
        starting point for the Tarjan search *)
      push_handle (par, spl_grp);
      incorporate newnode in the old group;
    ELSE (* must be an external node, including child scan_con = fe *)
      newnode is a singular, trivial node;
    END (* IF *);
  ELSIF parent scan_con = fz THEN
    IF newnode.head OR newnode.tail ∈ parent's group THEN
      (* old group is to be resplit, before emission *)
      push_split (spl_grp);
      (* no need to push the parent onto the handle to guarantee that there is a
        starting point for the Tarjan search, since the old handle will be
        restored before re-scanning. Place child in the old group since it will be
        visited by Tarjan search later *)
      incorporate newnode in the parent's group;
      (* do not allow emit to proceed by setting the emit_stop flag *)
      emit_stop := true;
    ELSE (* must be a singular node *)
      newnode is a singular, trivial node;
    END (* IF *);
  ELSIF parent scan_con = fe THEN
    IF newnode.head = ee OR newnode.tail = ee OR
      newnode.head = fe OR newnode.tail = fe THEN
      increment the new S.C.C.'s group reference count;
      incorporate newnode in emit_group;
    IF newnode.head = ee THEN
      increment the new S.C.C.'s group reference count;
      emit_interior_count := emit_interior_count + 1;
    END (* IF *);
    IF newnode.tail = ee THEN
      increment the new S.C.C.'s group reference count;
      emit_interior_count := emit_interior_count + 1;
    END (* IF *);
    IF newnode.head = fe THEN
      emit_interior_count := emit_interior_count + 1;
    END (* IF *);
    IF newnode.tail = fe THEN
      emit_interior_count := emit_interior_count + 1;
    END (* IF *);
  ELSE (* must be an external node, including child scan_con = fe *)
    newnode is a singular, trivial node;
  END (* IF *);
Chapter 6. Real Time Systems

ELSIF parent scan_con = ee THEN
    IF newnode.head = ee OR newnode.tail = ee OR
        newnode.head = fe OR newnode.tail = fe THEN
        emit_interior_count := emit_interior_count + 1;
        increment the new S.C.C.'s group reference count;
        incorporate newnode in emit_group;
        IF newnode.head = fe THEN
            emit_interior_count := emit_interior_count + 1;
        END (* IF *);
        IF newnode.tail = fe THEN
            emit_interior_count := emit_interior_count + 1;
        END (* IF *);
        ELSE (* must be an external node, including child scan_con = fe *)
            newnode is a singular, trivial node;
        END (* IF *);
    END (* IF *);
    addref (newnode, newnode.head);
    addref (newnode, newnode.tail);
END newapp;

Figure 6.23 The newapp real time procedure.

Pointer addition is performed by procedure addref; this is subordinate to newapp and
performs the normal maintenance of local and group reference counts when a pointer is added
from a parent to child. This makes no assumption about the originator and target nodes cyclic
states. It thus must perform tests to see if the originator and target nodes are in the same
cyclic structure. If this is the case, then a simple increment of the target node’s reference
count is all that is required. If, however, the originator and target nodes are in different
S.C.C.s then the target node’s group count has also to be incremented. An outline of the non-
incremental implementation of the pointer addition algorithm is given in figure 6.24.

PROCEDURE addref (parent, child);
BEGIN
    increment local referencecount (child);
    IF parent AND child are NOT in the same group THEN
        increment group referencecount (child);
    END (* IF *);
    END addref;

Figure 6.24 Addition of a pointer.

The procedure, in essence, is very simple. Given the destination node, child, the local
reference count for this node is incremented. A test is then carried out to find if the target and
the parent are in the same group. If this is not the case then the group count associated with
the target is incremented. The algorithm must be modified to deal with the interference
between it and the incremental application of Tarjan’s algorithm.

When the new node is added by newapp it either forms a new, singular node (so is in
scan_state = xx), forms part of the group currently being split (scan_state = nn), is in the
Section 6.3 The Collection of Cyclic Structures in Real Time

group being emitted \((\text{scan\_state} = \text{ee})\) or forms part of another S.C.C. not currently being acted upon by the incremental Tarjan search \((\text{scan\_state} = \text{xx})\). The new node (the parent in the \text{addref} procedure) does not take any other \text{scan\_state}.

The modified \text{addref} algorithm is presented in figure 6.25. The situation is unchanged from the simple case presented above, when the parent node has not been fully scanned, since in this situation the modified graph structure will be taken into account by the Tarjan search when it revisits the node.

With a parent in the \text{ee} state, if the child is in the state \(f_e\) then this will be an internal pointer in the group being emitted, when the child node is emitted. If the child is in state \(nn\) or \(qq\) or \(f_e\) then the pointer is external to the group currently being split. However, the group being split does not have its group reference count incremented since this will be recalculated when the group is finally emitted. If the child has a \text{scan\_state} = \text{xx} then the target group reference count is incremented in the normal way.

```latex
PROCEDURE \text{addref}(\text{parent, child});
BEGIN
increment_local_referencecount(\text{child});
IF \text{parent} = \text{xx} OR \text{nn} THEN
  IF \text{parent} AND \text{child} are NOT in the same group THEN
    increment_group_referencecount(\text{child});
  END (* IF *);
ELSIF \text{par} = \text{ee} THEN
  IF \text{child} = \text{fe} OR \text{child} = \text{ee} THEN
    (* Skip *)
  ELSIF \text{child} = \text{fx} OR \text{child} = \text{qq} OR \text{child} = \text{nn} THEN
    (* this is an external pointer when the group is emitted, do not increment
      the old group's count since this will be done by \text{emitgroup} *)
  ELSIF \text{child} = \text{XX} THEN
    increment_group_referencecount(\text{child});
  END (* IF *);
END (* IF *);
END \text{addref};
```

Figure 6.25 The \text{addref} real time procedure.

The algorithm presented in figure 6.25 is suitable as a subordinate function to \text{newapp}; however, we may wish to implement an \textit{explicit pointer addition}. This is provided as a service by the collector to the mutator. A typical situation where this may be required is when the root of the current reduction is overwritten. This specifies an existing node within the reduction graph (that may be in any \text{scan\_state}) and the new children of that node. The extended algorithm, that covers all \text{scan\_states} of the parent node, is presented in figure 6.26. The consideration of the action of this procedure is very similar to \text{newapp} so it will not be repeated here. It should be noted here that this type of pointer addition may cause an S.C.C.s to be resplit, as with \text{newapp}, when we cannot repair the group structure.
PROCEDURE addref_ext (parent, child);
BEGIN
  increment_local_referencecount (child);
  IF parent = xx OR nn OR pp OR qq THEN
    IF parent AND child are NOT in the same group THEN
      increment_group_referencecount (child);
      END (* IF *);
    ELSIF parent = fx THEN
      IF child = nn OR qq OR fx THEN
        (* old group is to be re-split, after emission. The child and parent are both
in the old group, cannot rectify SCC structure *)
        push_split (spl_grp); (* only push if NOT already on stack *)
        (* push the parent onto the handle to guarantee that there is a
starting point for the Tarjan search *)
        push_handle (parent, spl_grp);
        ELSIF child = fe
        (* SKIP - emitgroup will calculate the new group counts *)
        ELSIF child = ee
        increment the new S.C.C.'s group reference count;
        ELSE
        increment_group_referencecount (child);
        END (* IF *);
    ELSIF parent = fz THEN
      IF child ∈ parent's group THEN
        (* old group is to be re-split, before emission *)
        push_split (spl_grp);
        (* no need to push the parent onto the handle to guarantee that there is a
starting point for the Tarjan search, since the old handle will be
restored before re-scanning. Place child in the old group since it will be
visited by Tarjan search later *)
        (* do not allow emit to proceed by setting the emit_stop flag *)
        emit_stop := true;
        ELSE
        increment_group_referencecount (child);
        END (* IF *);
    ELSIF parent = fe THEN
      IF child = ee
        increment the new S.C.C.'s group reference count;
        ELSIF child = fe
        (* SKIP *)
        ELSIF child = fx OR child = qq OR child = nn THEN
        (* this is an external pointer when the group is emitted, do not increment
the old group’s count since this will be done by emitgroup *)
        ELSE
        increment_group_referencecount (child);
        END (* IF *);
    ELSIF parent = ee THEN
      IF child = ee
        emit_interior_count := emit_interior_count + 1;
        ELSE
        emit_interior_count := emit_interior_count + 1;
        ELSIF child = fx OR child = qq OR child = nn THEN
        (* this is an external pointer when the group is emitted, do not increment
the old group’s count since this will be done by emitgroup *)
        ELSE
        increment_group_referencecount (child);
        END (* IF *);
    END (* IF *);
ELSIF parent = ee THEN
  increment the new S.C.C.'s group reference count;
  emit_interior_count := emit_interior_count + 1;
ELSIF child = fe THEN
  emit_interior_count := emit_interior_count + 1;
ELSIF child = fx OR child = qq OR child = nn THEN
  (* this is an external pointer when the group is emitted, do not increment
the old group’s count since this will be done by emitgroup *)
ELSE
  increment_group_referencecount (child);
END (* IF *);
END (* IF *);
END addref_ext;

Figure 6.26 The addref_ext real time procedure.
In a similar fashion to newapp the addref_ext procedure must maintain the invariants of emitgroup. Table 6.3 shows the compensation of these totals for the addition of a pointer.

<table>
<thead>
<tr>
<th>par_sc</th>
<th>child_sc</th>
<th>grp_count</th>
<th>em_in_cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_e )</td>
<td>( ee )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( f_e )</td>
<td>( f_e )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( f_e )</td>
<td>( y )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( ee )</td>
<td>( ee )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( ee )</td>
<td>( f_e )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( ee )</td>
<td>( y )</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.3 Compensation of emitgroup count totals for addref_ext.

The derivation of these compensation values may be obtained by reference to figure 6.27. With the parent in the \( f_e \) state and the child in the \( ee \) state, since the child will not be visited by emitgroup again, the group count for the new S.C.C. has to be incremented by one. If the parent and child are both in the \( ee \) state then both the emit_interior_count and the group count must be incremented. Finally, if the parent is in the \( ee \) state and so will not be visited again by emitgroup and the child is \( f_e \) then only the emit_interior_count is incremented.

![Figure 6.27](image)

**Figure 6.27** The addition of a pointer in the addref_ext procedure.

### 6.3.4 The Main Controlling Element of the Collector.

The final element of the collector is the main controlling loop. An overview of this is given in figure 6.28. In essence, this comprises three nested loops, the first is an infinite loop (never terminating) that cycles the action of the collector as long as the mutator is active. The
next loop takes group descriptors from the \textit{split\_queue}, so that each group waiting to be split is dealt with in turn. The inner most loop takes groups and splits them using \textit{visit} and employing the group handle to guide the search. Once this inner loop has been completed, if \textit{emit\_stop} was set then the original group handle, for the group being split is restored, since this group has to be resplit. All nodes on the \textit{cyclic\_free\_list} are now released to the free list by a call to \textit{process\_cyclic\_free\_list}. These nodes can be safely returned to the free list at this stage, since the incremental Tarjan search has terminated so the context information of the nodes on this stack is no longer required by the collector.

The loop structure, described above, notionally runs in parallel with two other routines, \textit{process\_group\_dispose\_stack} and \textit{process\_pointer\_deletion\_stack}. The first of these incrementally collects garbage cells for S.C.C.s that have become redundant. This must be done incrementally since we need to visit each node in turn, by traversing the S.C.C., in order to return the constituent nodes to the free list. The \textit{process\_pointer\_deletion\_stack} incrementally deletes acyclic structures, as described in section 6.2. These three tasks form the collector's \textit{background tasks}.

Figure 6.29 gives a high level architectural overview of the elements of the collector introduced so far. The mutator requesting services (the collector's external functions) is depicted as a thick black line, from the mutator to the main module of the collector. This provides an interrupt service handler for the mutator. At the end of a critical section (which ever is running at the time) the mutator request is serviced, by calling one of the collector's external functions (such as \texttt{newapp}, \texttt{deleteref} etc); this again is depicted as a thick black line. The diagram shows data structures as grey boxes, functions as black boxes, data flow by grey lines and control flow by black lines. The control of the three main parallel processes, within the collector (the incremental Tarjan search and associated routines, \textit{process\_group\_dispose\_stack} and \textit{process\_pointer\_deletion\_stack}), is shown by a dotted line from the main module.
Section 6.3 The Collection of Cyclic Structures in Real Time

MAIN
emit_stop := false;
spl_grp := 0;
grp_id := 1;
init_sg := 0;
WHILE true (* loop forever *)
LOOP
  init_sg := 0;
  WHILE NOT EMPTY split_queue (* loop whilst there are nodes on the split queue *)
  LOOP
    (* spl_grp is set here *)
    pop_split (split_queue, spl_grp) ASSUME POST;
    temp_group_handle := group_handle!spl_grp;
    WHILE NOT EMPTY (group_handle!spl_grp)
    LOOP
      pop_handle (z, spl_grp) ASSUME POST;
      visit (z, init_sg) ASSUME POST;
    ENDLOOP;
    IF emit_stop THEN
      (* this group will be resplit, so restore its old group handle,
        minus any nodes that have been deleted *)
      group_handle := temp_group_handle;
    END (* IF *);
    emit_stop := false;
    spl_grp := 0;
    (* can now safely reclaim nodes that were on the cyclic_free_list *)
    process_cyclic_free_list;
  ENDLOOP;
ENDLOOP;

ENDMAIN

//
process_pointer_deletion_stack;
//
process_groupDispose_stack;

Figure 6.28 A simplified view of the collector's MAIN routine.
Figure 6.29 A high level view of the collector's architecture.
6.4 A Model of the Parallel Behaviour of the Collector and Mutator.

As explained above, the graph mutator runs in parallel with the collector. There will be a latency in servicing any mutator request that involve the graph topology, since the currently active critical section must terminate before the request can be dealt with.

There are two basic refinements that we make to this scheme. First, we observe that since requests for new nodes do not affect the graph structure (until pointer addition to the new node is performed) these can be asynchronously taken from the free-list at any time, without having to wait for the completion of a collector's critical section. This service is provided by a `newnode` procedure. Secondly, if we arrange for a handshake protocol to be implemented with the mutator, we can actually interrupt a critical section, issue a signal telling the mutator that the requested service will be provided. The mutator then resumes graph reduction; meanwhile, the collector queues this request, finishes the critical section and then services the queued request. If the mutator requests another service before this cycle has completed, then it must wait for the process of completing the last critical section and service its last request, before it receives the collector's signal to continue. In this way we avoid race conditions between the collector and mutator. This scheme improves the average latency of the collector, but not its worst case performance.

Figure 6.30 shows a simplified schematic of the interaction between the collector and mutator. A mutator request (add or delete a pointer etc) is queued, then the current critical section is completed. The request is then serviced. The mutator resumes operation as soon as the request has been queued.

As stated above, this mode of operation only allows requests to be queued one deep, so the mutator may have to halt (t1) before it can gain access to the collector.

The collector has two basic modes of operation. The first is simply servicing requests from the collector. The second is performing background tasks, comprising the incremental search and emission of nodes, to maintain the acyclic S.C.C. structure of the reduction graph, as well as servicing the pointer deletion stack (`process_pointer_deletion_stack`) and the disposal of groups that have become garbage (`process_group_dispose_stack`).

6.4.1 Scheduling within the Collector.

The collector can be thought of as operating in a hard real time environment, where it has to respond to a mutator request before a set upper bound of time. No assumption can be made about the arrival rate of mutator requests. The collector latency is given by the worst case analysis of the critical sections of the code, whilst ignoring request queueing (a prediction of worst case behaviour is given in chapter 7).

Background tasks are scheduled using an online scheduler. The scheduler monitors the length of the free-list, as this shortens more time is spent on `process_pointer_deletion_stack` and `process_group_dispose_stack`, since these yield garbage cells for the free-list more quickly than the other incremental tasks. Once the free-list has grown to a pre-determined size, background task resource utilisation is once again equally shared between the three main collector background tasks.
6.5 The Main Data Structures of the Collector.

Many different data structures have been introduced and described in this chapter. We are now in a position to provide a simple overview of all of these, in the form of a table (table 6.4).
### Section 6.5 The Main Data Structures of the Collector

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cyclic_free_flag</td>
<td>A boolean associated with each node, it is set when the node is eligible to become garbage, but cannot yet be appended to the free_list since it forms part of the S.C.C. currently being split.</td>
</tr>
<tr>
<td>cyclic_free_list</td>
<td>A list of nodes identified as garbage, that will be appended to the free list of cells when the current Tarjan search has completed.</td>
</tr>
<tr>
<td>emit_flag</td>
<td>A boolean exported from emitgroup to signify that the procedure is active.</td>
</tr>
<tr>
<td>emit_group_id</td>
<td>The group identifier of the S.C.C. currently being emitted by emitgroup.</td>
</tr>
<tr>
<td>emit_interior_count</td>
<td>A count of the number of intra-cyclic pointers encountered by emitgroup during the emission of a group.</td>
</tr>
<tr>
<td>emit_stop</td>
<td>A boolean: when this flag is set emitgroup is NOT called.</td>
</tr>
<tr>
<td>emit_thresh</td>
<td>The depth first search number at which the current call of emitgroup was called.</td>
</tr>
<tr>
<td>free_list</td>
<td>A list of nodes available for reuse.</td>
</tr>
<tr>
<td>group_dispose_stack</td>
<td>A stack of groups waiting to be deleted.</td>
</tr>
<tr>
<td>group_handle</td>
<td>A set of nodes, associated with each S.C.C., through which the entire group can be traced when the Tarjan search is called.</td>
</tr>
<tr>
<td>pointer deletion stack</td>
<td>A stack of pointers, comprising source and destination nodes, that are awaiting deletion; the pointer deletion type is also stored with each element on the list</td>
</tr>
<tr>
<td>scanned</td>
<td>A boolean associated with each node, it is set when the node is initially encountered by the Tarjan search.</td>
</tr>
<tr>
<td>spl_grp</td>
<td>The group identifier of the S.C.C. currently being split.</td>
</tr>
<tr>
<td>split_queue</td>
<td>A queue of groups (comprising group identifiers), waiting to be split.</td>
</tr>
<tr>
<td>stack</td>
<td>The Tarjan stack.</td>
</tr>
</tbody>
</table>
Chapter 6. Real Time Systems

### Table 6.4 The main data structures of the collector.

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>visited</td>
<td>A boolean associated with each node, it is set when the node is finally processed by the Tarjan search.</td>
</tr>
</tbody>
</table>

6.6 An Enhanced Set of Collector Services.

There have been a number of collector services mentioned so far in this discussion. These are `addref_ext`, `deleteref`, `newapp` and `newnode`. The pointer operations, can be refined. In certain circumstances the mutator may be aware of the S.C.C. structure of the graph. As a result we introduce different *flavours* of pointer operation. The motivation for introducing variants in the form of pointer operations is purely for performance reasons. The collector implements three different kinds of pointer deletion. These deletions are of the type *simple*, *external* and *full*. The deletion types are characterised by the following description. The simple type (`deleteref_simple`) takes no account of cyclic structure, and straightforwardly decrements the target node's reference count. The external deletion (`deleteref_external`) assumes that the pointer deletion is *inter*-cyclic, and so both the target node's local and cyclic reference counts must be decremented. The *full* (`deleteref_full`) type of pointer deletion makes no such assumptions and tests to see if a pointer is *inter-* or *intra-* cyclic in nature. This final form of operation is obviously more expensive in collector time, because of the tests that have to be performed in order to ascertain which (if any) groups the target and parent nodes of the pointer belong to. This corresponds exactly to the description of the `deleteref` operation, described above. A similar set of operations exist for `addref_ext`.

With this description in mind, we must amend the structure of the pointer deletion stack. It is no longer sufficient to keep a simple stack of destination nodes, as described in section 6.2. The stack must also contain the pointer deletion type. Also, in order to perform incremental collection we store the parent node of the pointer deletion operation. An actual element in the pointer deletion stack is shown in figure 6.31.

<table>
<thead>
<tr>
<th>Node No.</th>
<th>Deletion Type</th>
<th>Parent Node</th>
</tr>
</thead>
</table>

**Figure 6.31** Pointer deletion stack element.

When the stack is processed the appropriate pointer deletion routine is called for each node in turn.
6.7 Extension of the Acyclic Collection Technique to Cope with Cycles.

There are a number of complications that arise when the naïve scheme given in section 6.2, for the collection of acyclic data structures, is implemented. This is due to the interaction between this scheme and the incremental collection of cyclic structures, as described in section 6.3.

Figure 6.32 shows a typical cyclic structure, that has already been corrupted by the previous deletion of one pointer (from nodes 5 to 1). The diagram illustrates the problem that occurs when an external pointer to node 1 is subsequently deleted. This operation is performed by a call to an external pointer deletion procedure. Node 1 is collected as garbage, since its local reference count has fallen to zero (the group count is also decremented).

Node 1 would have been entered in the group handle. When node 1 is deleted, it is necessarily removed from the group’s handle. If the deletion of the pointer from node 1 to 2 is delayed, due to incremental collection, then node 2 is not placed on the group handle. A problem now occurs with this situation if the group is traversed using a Tarjan search, before the pointer from node 1 to 2 is deleted. This deletion would have ordinarily resulted in node 2 being added to the group handle.

This external pointer is deleted

Previously deleted pointer

Another external pointer

Figure 6.32 A cyclic structure, with intra- and inter-cyclic pointer deletions.

This problem is overcome by making sure that when deleting a node in the incremental scheme, if it belongs to an S.C.C., a test is carried out to ascertain if the collection of the node triggers the deletion of an intra-cyclic pointer. If this is the case, then the destination node is added to the handle.

6.8 Conclusions.

This chapter has introduced two important techniques for the operation of a reference counting garbage collector in real time. The first is an incremental technique for the collection of large redundant acyclic structures. This displays similar advantages to the T.B.D. scheme [GLA-87], allowing such structures to be collected piecemeal. The second technique allows cycles to be recovered by the use of reference counts for each maximal cyclic structure identified in the reduction graph. It performs all operations in an incremental way, so facilitating its use in a real time system.
Chapter 6. Real Time Systems

A model for the parallel execution of the collector and mutator has been introduced. This relies upon the definition of critical sections within the collector. These are sections that may not be interrupted by the mutator, but between calls to these sections the collector is free to service mutator requests.

Towards the end of the chapter, an overall abstract architecture was presented for the collector. This forms the basis for the implementation of the simulator of the system (presented in chapter 9).

In the ensuing chapter the temporal behaviour of the new techniques are explored, along with the ramifications of introducing the limited width scheme (described in chapter 4).

7.1 Introduction to the Temporal Analysis of the Collector.

The next section is concerned with the detail of the collector, as described in chapter 6, with respect to its real time performance. Initially, no account is taken of the memory overhead reduction techniques, discussed in chapter 4. The effects of how these interact with the real time system are discussed later in the chapter. The following discussion outlines how the collector will behave, when the mutator requests a service offered by the collector.

7.2 A Performance Overview of the Important Collector Algorithms.

Software graph execution models [SMI-90] have been derived for all collector procedures. Using path accession techniques the worst case response times for each of the procedures was obtained, in terms of collector operations. The graph execution models are not presented here for brevity, but the main results are presented in table 7.1 and the equations that follow. (Pointer addition and deletion is assumed to be the full type, see section 6.6).

\begin{align*}
\text{addref} & \Rightarrow Li + 7Lr/w + 7Lm + 4Sg + En \\
\text{deleteref} & \Rightarrow Li + 16Lr/w + 12Lm + 2Hr/w + FC + G_i + GCO \\
\text{newapp} & \Rightarrow 28Lr/w + 3Hr/w + 2Li + 16Lm + 10Sg + 3En \\
\text{visit} & \Rightarrow 54Lr/w + 33Lm + 14Hr/w + 2Sg \\
\text{emitgroup} & \Rightarrow 41Lr/w + Ng + 27Lm + En + 10Hr/w + Ac + Li + 3G_i + Sg
\end{align*}

Equation 7.1
Equation 7.2
Equation 7.3
Equation 7.4
Equation 7.5

Where \text{Ac} is the time taken to allocate a cell to an S.C.C., \text{En} is the time to emit a single node, \text{Li} represents the time to increment a local reference count, \text{G} to increment a group reference count, \text{GCO} is the time taken to test if an S.C.C.'s group count is zero, \text{FC} characterises the time taken to append a cell to the free list. \text{Lr/w} represents a read or write to the collector's local memory, \text{Lm} represents a collector's service time unit (primitive CPU operation) and \text{Hr/w} represents a read or write to the main heap memory. \text{Sg} is the time taken to test if two nodes are in the same group and \text{Ng} is the allocation of new group descriptor.

7.2.1 Specific Simplifications of the Temporal Equations.

The final aim of the temporal specification of the collector routines, is to use only the three primitives: \text{Lr/w}, \text{Lm} and \text{Hr/w}. However, in order to make further simplifications in the
relationships given above it is necessary to make some assumptions that only hold good for a 
system that does not employ limited-width reference counts (see chapter 4). The extended 
system, employing such techniques, will be examined, on the basis of the above equations 
later in the chapter. In this section specialised simplifications will be explored. This will 
allow the extra overhead of employing the limited-width techniques to be contrasted with the 
results found in this chapter.

The main assumption made here is that all data held on both local and group counts is 
local to the collector, and that no such information has to be recovered from the main heap. It 
also makes the simplification that the local count overflow block is never required. This 
corresponds to a situation where full-width reference counts are employed and sufficient work 
space is also always allocated to the collector for such tasks as stack allocation, group handle 
storage etc.

The first temporal primitive to address in this manner is \( L_i \), which represents the time 
taken to increment or decrement, a local reference count. This may be characterised by \( 2L_r/w + L_m \), which simply arises in the fact that firstly the local count is read, then incremented, and 
then written back to the collector local-count block. The second, analogous group-count 
primitive, is \( G_i \), that represents the time taken to increment, or decrement, a group reference 
count. The process involves tracing a pointer to the group count table, and then modifying the 
count as required. This may be represented by the approximation: \( 4L_r/w + L_m \).

The time taken to test if a group count is zero, is given by the term \( GC_0 \). This can be 
taken to be simply \( 2L_r/w + 2L_m \). \( FC \) represents the process of freeing a cell (i.e., appending it 
to the free list). If no overflow has taken place then this is a simple process, involving \( 3L_r/w + L_m + H_r/w \). The term \( H_r/w \) arises because a pointer field in the freed cell is used to hold a 
pointer to the next free cell, thus maintaining the linked list of free cells.

The time taken to test if two cells are in the same group is \( S_g \). This can simply be written 
as \( 4L_r/w + 2L_m \). \( En \) can be simplified to: \( 6L_r/w + 4L_m + 2H_r/w \). The allocation of new group 
descriptors, denoted by \( Ng \), takes \( 4L_r/w + 2L_m \). Finally, \( Ac \) can be simply written as \( L_r/w \), 
since in the case of no overflow it simply involves writing a group pointer to a location in the 
group number block of the collector.

Having made these further simplifications, it is now possible to write equations 7.1 - 7.5 
in terms of \( L_r/w, L_m \) and \( H_r/w \) only. These results are presented in table 7.1. The table also 
classifies the operations, by ascribing a function type to each operation. External functions are 
those made available to the mutator as an interrupt service, internal functions form part of the 
collector’s background tasks. A function is deemed to be compound if it involves the calling 
of one or more of the simple collector functions.
Section 7.3 Introduction to the Integrated System

<table>
<thead>
<tr>
<th>Collector Function</th>
<th>Function Type</th>
<th>( L_{r/w} )</th>
<th>( L_m )</th>
<th>( H_{r/w} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>addref_ext</td>
<td>simple, ext.</td>
<td>31</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>deleteref</td>
<td>simple, ext.</td>
<td>26</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>newapp</td>
<td>compound, ext.</td>
<td>90</td>
<td>50</td>
<td>7</td>
</tr>
<tr>
<td>visit</td>
<td>compound, inter.</td>
<td>58</td>
<td>35</td>
<td>14</td>
</tr>
<tr>
<td>emitgroup</td>
<td>compound, inter.</td>
<td>68</td>
<td>39</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 7.1 Summary of the worst case temporal behaviour of the major classes of collector routines.

Where \( L_{r/w} \) represents a read or write to the collector's local memory, \( L_m \) represents a service time unit (primitive CPU operation) and \( H_{r/w} \) represents a read or write to the main heap memory. Obviously \( H_{r/w} \) is a more costly operation, in general, than \( L_{r/w} \).

7.3 Introduction to the Integrated System.

Earlier in the chapter the real time behaviour of the incremental collector system was outlined (table 7.1). This was done in isolation with regards to the limited-width scheme, as detailed in chapter 4. Also in chapter 4, an analysis was presented of the spatial requirements of the limited-width scheme. In this section and the next, the temporal and spatial behaviour of the combined system is explored. Comparisons are draw between the overheads in processor time, necessary to support simple reference counting garbage collection, limited-width collection, and incremental real time collection.

7.3.1 Limited-width Reference Counting in Real Time.

In section 7.2, equations 7.1 to 7.5 were presented in general terms, using a number of different system primitives to describe the behaviour of each collector operation. We now simplify these equations to comprise only the primitives \( L_{r/w} \), \( L_m \) and \( H_{r/w} \) whilst taking into account the limited-width reference counting scheme, as described in chapter 4. However, the only assumptions now made, for the sake of simplicity, will be that group handles, for groups whose data is held in the collector area, will be completely held in the main collector memory and that they will not partially, or wholly, spill into the main heap space (see chapter 4 for an explanation of this mechanism). This is a reasonable assumption to make, since in practice group handles are found to be fairly small (see chapter 9). Also, no account will be made of the caching technique (chapter 4). Caching, in general, will lower the response time for a routine dealing with a local count or group details, that has data held in main memory. By ignoring this, the worst case analysis of the system response should not be affected. In chapter 9, a brief description of error checking mechanisms, which are incorporated into the collector, is given. This is not accounted for in these temporal equations, since such operations are not fundamental to the operation of the algorithms being described.
Chapter 7. Real Time Analysis of the Integrated System

The simplification of the primitives will be very briefly outlined here. The final values were actually gleaned from and checked by the instrumented software implementation of the collector (see chapter 9). The temporal primitives will be dealt with in the same order as section 7.2.

In dealing with local reference counts, three distinct possibilities arise when considering the critical paths of any given function, which processes such counts. The first is that no overflow has occurred and that all the local count data is held in the limited-width field of the collector. This situation is essentially analogous to the one assumed above, where no overflow has occurred. In general, very little processing overhead is incurred in the limited-width scheme for this situation, save the testing of one bit which flags local-count overflow. For this reason this state will not be considered further, since it is so similar to the case explored in section 7.2. The next situation arises when local data is held in the overflow area, which is internal to the collector. The third possibility is when local counts are held in the main heap. Each of the last two situations will be dealt with separately. This is in order to contrast the varying time overhead, associated with overflow that is either internal, or external to the collector.

In both local and group count overflow situations, the case considered is when the cell, which is being searched for by the collector in the relevant segment of main memory (see chapter 4 for an explanation of this mechanism), is held in the last location of that segment. This is obviously the worst case scenario, since this involves the greatest number of heap searches. The average performance of such a linear search will obviously be twice as fast as this.

The process of incrementing, or decrementing, a local reference count is represented by \( Li \). This simplifies to \( 16 Lr/w + 10 Lm \) for internal overflow, and \( (2n/2^a + 17)Lr/w + (6n/2^a + 4)Lm + (2n/2^a + 2)Hr/w + Fc \), for the case of overflow to the main heap. The term \( n/2^a \), (where \( a \) is the width, in bits, of the local count field in the collector main block of memory and \( n \) is the size of the main heap in number of cells), arises from the fact that this is the maximum number of cells in the main heap that have to be searched, in order to locate the local count overflow cell. The \( FC \) term arises because the collector may have to return one cell to the main heap, which contained the local count overflow data, if the local count has fallen below the level at which overflow occurs. \( Li \) will be further simplified once \( FC \) has been quantified.

In the case of group counts, there are only two possibilities to consider for the analysis of the critical path for functions that deal with group data. The first is simply that the count is stored wholly in the memory of the collector. This situation is again similar to that described above and so will not be reiterated here. The second case is where group information has spilled into the main heap. This will necessarily result in a slower response time from the collector, since it has to find the information in the heap, as well as imposing the further penalty of the bandwidth overhead of main heap communication. Because of this, the cases concerning group counts will consider the situation where the group information is held in the main memory. Such situations represent the worst case, from a point of view of response time.

In the following cases, it will be assumed that group count overflow has taken place to the main heap, and where relevant, both types of local count overflow will be investigated.

Incrementing or decrementing the group count, represented by \( Gi \), can be simplified to \( (2n/2^a + 8)Lr/w + (6n/2^a + 3)Lm + (2n/2^a + 1)Hr/w \). Corresponding to the case of local
reference count overflow to the main heap. The term \( n/2^b \) arises because this is the maximum number of nodes that have to be searched in order to locate the required main heap node, containing the correct group descriptor pointer.

\[ GCO \text{, which is the time taken to test if a group count has fallen to zero, can be simplified to: } (n/2^b + 1) LR/w + (6n/2^b + 6) LM + 2n/2^b HR/w. \]

\[ FC \text{, the time taken to place a cell on the free list takes: } (2n/2^a + 2n/2^b + 14) LR/w + (5n/2^a + 6n/2^b + 15) LM + (2n/2^a + 2n/2^b + 5) HR/w, \text{ for both group count and local count overflow to the main heap. The } n/2^a \text{ and } n/2^b \text{ terms arise, since the cell may have a group descriptor pointer cell, and a local count overflow cell, associated with it. In which case, each has to be located and then freed from the main heap. If local count overflow is in the memory of the collector, then } FC \text{ simplifies to } (2n/2^b + 17) LR/w + (6n/2^b + 16) LM + (2n/2^b + 5) HR/w. \]

Since \( FC \) has been characterised, it is now possible to rewrite \( Li \) as: \((4n/2^a + 2n/2^b + 31) LR/w + (11n/2^a + 6n/2^b + 19) LM + (4n/2^a + 2n/2^b + 7) HR/w, \text{ for local count overflow to the main heap. There is no change to its value for the case of internal local count overflow, namely: } 16 LR/w + 10 LM. \]

Next, the operation \( Sg \) is simplified. This is the time taken to test if two nodes are in the same group. The worst case situation for this operation is when both nodes have their group information held in main heap memory. In this eventuality, \( Sg \) simplifies to: \((4n/2^b + 12) LR/w + (12n/2^b + 14) LM + (4n/2^b + 2) HR/w. \) The time taken to emit a single node, \( Ze \), can be written, for internal local count overflow, as: \((2n/2^b + 29) LR/w + (6n/2^b + 25) LM + (2n/2^b + 10) HR/w, \text{ or as: } (2n/2^a + 2n/2^b + 33) LR/w + (5n/2^a + 6n/2^b + 30) LM + (2n/2^a + 2n/2^b + 10) HR/w, \text{ for the case of local overflow to the main heap.} \]

\( Ng \), which is the allocation of new group descriptor, can be simplified to: \(16 LR/w + 8 LM, \text{ for internal overflow, and } 17 LR/w + 12 LM + 9 HR/w \text{ for external. Finally, the time to allocate a cell to a group, } Ac, \text{ can be written as: } (2n/2^b + 8) LR/w + (6n/2^b + 15) LM + (2n/2^b + 8) HR/w. \]

Having made the above simplifications, it is now possible to rewrite equations 7.1 to 7.5. The equations marked with \( (a) \) are for local count overflow internal to the collector, and \( (b) \) for overflow in the main heap.
Chapter 7. Real Time Analysis of the Integrated System

addref_ext \rightarrow (18n/2^\alpha + 100) Lr/w + (54n/2^\beta + 98) Lm +
(18n/2^\beta + 18) Hr/w \quad (a)

\Rightarrow (6n/2^\alpha + 20n/2^\beta + 119) Lr/w + (16n/2^\alpha + 
60n/2^\beta + 112) Lm + (18n/2^\alpha + 20n/2^\beta + 25) Hr/w \quad (b)

Equation 7.6

deleteref \rightarrow (4n/2^\beta + 57) Lr/w + (12n/2^\beta + 41) Lm +
(12n/2^\beta + 14) Hr/w + (n/2^\beta + 1) Lr/w \quad (a)

\Rightarrow (6n/2^\alpha + 7n/2^\beta + 75) Lr/w + (16n/2^\alpha + 
24n/2^\beta + 55) Lm + (6n/2^\alpha + 8n/2^\beta + 15) Hr/w \quad (b)

Equation 7.7

newapp \rightarrow (46n/2^\beta + 267) Lr/w + (138n/2^\beta + 251) Lm 
+ (46n/2^\beta + 51) Hr/w \quad (a)

\Rightarrow (14n/2^\alpha + 50n/2^\beta + 309) Lr/w + (37n/2^\alpha 
+ 150n/2^\beta + 284) Lm +
(14n/2^\alpha + 50n/2^\beta + 65) Hr/w \quad (b)

Equation 7.8

visit \Rightarrow (151 + 12n/2^\beta) Lr/w + (111 + 22n/2^\beta) Lm 
+ (30 + 14n/2^\beta) Hr/w \quad (a)

\Rightarrow (162 + 16n/2^\beta + 6n/2^\alpha) Lr/w + (120 + 
23n/2^\beta + 10n/2^\alpha) Lm +
(33 + 16n/2^\beta + 9n/2^\alpha) Hr/w \quad (b)

Equation 7.9

emitgroup \Rightarrow (126 + 14n/2^\beta) Lr/w + (108 + 42n/2^\beta) Lm 
+ (33 + 14n/2^\beta) Hr/w \quad (a)

\Rightarrow (166 + 6n/2^\alpha + 16n/2^\beta) Lr/w + (126 + 
16n/2^\alpha + 48n/2^\beta) Lm + (49 + 6n/2^\alpha 
+ 16n/2^\beta) Hr/w \quad (b)

Equation 7.10

Table 7.2, at the end of the chapter, presents a summary of the equations 7.6 to 7.10. Both cases of local count overflow, internal and external to the collector, are presented.

It should be noted that, by substituting for values of \( \alpha, \beta \) and \( n \), in table 7.2, the number of operations, in terms of the primitives \( Lr/w, Lm, Hr/w \) may be derived, for each of the collector functions. These may then be expressed in absolute times, for a given target execution environment, if the typical value of each primitive operation is substituted. If, for example, we take an ARM 3 processor, operating at a main CPU speed of 36MHz, a subsystem speed of 12MHz, \( \alpha \) set to 8 bits, \( \beta \) set to 6 bits and \( n \) set to 4 MByte, we derive the following approximate worst case times, for addref_ext:

\[ Lr/w = 33 \text{ ms}, \quad Lm = 98 \text{ ms}, \quad Hr/w = 98 \text{ ms} \quad \text{(internal overflow)} \]
\[ Lr/w = 39 \text{ ms}, \quad Lm = 117 \text{ ms}, \quad Hr/w = 133 \text{ ms} \quad \text{(heap overflow)} \]
Table 7.2 Table of the worst case temporal behaviour of the major classes of collector routines, in the integrated system.

The response times given in table 7.2, assume that a reference count for a cell has overflowed to the main heap then that count is located and amended before the mutator interrupt is serviced. This is in fact an unduly conservative approach; when the mutator issues a service request and a main heap search has been initiated then the search is suspended and the request is serviced, after which the main heap search is continued. Such an approach produced a collector latency very nearly degenerate with the results outlined in the first table (table 7.1) above. However, this approach does store a backlog of work for the collector to perform and may prevent two pointers or more from being processed on the pointer deletion stack, on average, so leading to saturation of the heap and premature termination of the collector, before all possible garbage cells have been recovered. This pathological situation may always be avoided if the collector is allowed to enter a panic mode, in which the operation of the mutator is suspended and the collector runs until sufficient garbage cells are recovered to allow the resumption of the mutator. This form of operation should only be employed as a last resort, since it destroys the real time properties of the collector.

Allowing the collector to suspend a search in the main heap, to service a mutator interrupt, adds another task to the background task pool of the collector, the main_heap_search. The revised final high level architecture for the collector is given in figure 7.1. The interaction of the main_heap_search with other functions and data structures within the collector and with the heap, is not shown for the sake of clarity.
7.4 The Spatial Overhead of Real Time Garbage Collection.

This section explores the extra memory overhead, which is necessarily introduced in order to support the real time collection of garbage. No attempt is made to provide a quantitative analysis, since this would require many assumptions to be made to facilitate the development of a practical analytical memory model.
The conventional limited-width technique requires a status word of four bits per node, to support garbage collection (see section 4.6.2, equation 4.3). The real time system would appear to require a longer status word per node, to support the flags \textit{scanned}, \textit{visited} and \textit{cyclic\_free\_flag} (the use of these flags is described in chapter 6).

As an optimisation to the standard method, the value of \textit{lowlink} can be stored in the group pointer field, since once a node is on the Tarjan stack, the value of the group pointer is no longer required, since in these cases we know that the nodes are in the group being split. The flags, \textit{scanned}, \textit{visited} and \textit{cyclic\_free}, are also only of interest after the group descriptor number is no longer required, when the group is being split. So \textit{lowlink} and these flags can also be stored in the group descriptor field, at the cost of only one further flag, indicating that this field does not contain the group descriptor number, but rather lowlink and four flags. Theoretically, \textit{lowlink} could reach a maximum, equal to the total number of cells in the heap. In this case it could not be stored in the group descriptor number field, since this also contains four flags. In such an unlikely event, a form of indirection to the main heap, to store the value of \textit{lowlink}, is arranged. A limited width field has to be made available for each node that is in the group being split.

Additional to the overhead per cell, discussed above, there is also a number of stacks that has to be maintained to support the technique, the Tarjan stack, the group handle stack, the \textit{pointer\_deletion\_stack}, \textit{split\_queue}, \textit{group\_dispose\_stack} and the \textit{cyclic\_free\_stack}. The size of these stacks will be a complex function of the relative amount of CPU resources the scheduler allocates to the four main areas in its background task pool, the rate at which the mutator calls intra-cyclic pointer deletions, the number and size of cycles within the graph and the number of nodes in large acyclic structures. An implicit stack is also maintained to control the recursion in Tarjan's algorithm. The maximum size of the stack will be in the order of the largest cycle contained in the graph.

\textbf{7.5 Conclusions.}

This chapter has presented an estimate of the temporal aspects of the behaviour of the collector. The theoretical predictions made will be contrasted with actual measurements made with a simulation of the system (presented in chapter 9).

Table 7.2 presented below, shows how much more expensive is the situation when local reference count data is held in the main heap, both in terms of collector activity and collector-heap communication bandwidth. This affects the efficiency of numerous collector activities, such as the simple update of local reference counts. As explained, the worst case scenario, outlined in this table, is never actually observed, since the collector may suspend a search for overflow data in the main heap, in order to service a mutator interrupt.

The scheme of implementing the parallel collector compounds the difficulty of assessing the memory required, within the collector, to deal with the incrementally formed stacks. The production of a general analytical model of the memory required for the collector, under all operating conditions, appears to be a difficult (if not intractable) task and is not attempted here.
Chapter 7. Real Time Analysis of the Integrated System

The theoretical performance predictions have been made at a very fine level of granularity, involving low-level CPU and sub-system primitives. The results have ignored certain factors that may play a rôle in the production of final performance figures. These factors may be characterised by constant overheads for any given execution environment. Such costs include procedure call overhead and context switches etc. In a typical standard execution environment (e.g. DEC VAX) these costs range from 5 to 25 percent [ZOR-89]; a specialised real time execution environment would display much lower overheads. Since these overheads are fixed and represent only a small cost, with respect to the execution speed of the collector functions, the results obtained from the theoretical predictions made in this chapter and those given for the simulator in chapter 9, provide general guidance to the execution profile for the major collector functions.

An overall architecture has been presented for the collector. This is the culmination of the design process described in earlier parts of the thesis. A simulation of the complete architecture has been written and is presented in chapter 9.
Chapter 8. The Formal Verification of the Collector by use of MALPAS.

8.1 Introduction.

The proof of correctness for parallel garbage collectors has been attempted by a few authors, typical of these are [GRI-77, DIJ-78]. These “on-the-fly” collectors deal with relatively small systems as compared to the one presented in this work. Dijkstra is quoted as saying [GRI-77] that the small collector problem, that he and his four colleges addressed, was “one of the most challenging in parallel programming”.

MALPAS (the MALvern Program Analysis Suite) was used to facilitate the proof of the collector system presented in this work.

8.2 An Outline of the MALPAS System.

MALPAS [RTP-92b] has been developed over a number of years at the Royal Signal and Radar Establishment, Malvern, Worcestershire (latterly the Defence Research Agency, Malvern) and is based upon extensive work carried out at the Establishment and Southampton University. It is designed for the proof of highly trusted systems, such as weapons control software, nuclear power station primary cooling circuit control software and military aircraft “fly-by-wire” systems.

To produce a proof of a system, the code to be considered must first be written, or automatically translated, into MALPAS intermediate language (IL) [RTP-91]. The collector simulator was written using Modula-2 and unfortunately at this time no Modula-2 to IL translator was available. As a result all IL for this work was hand coded and manually verified. Hand coding of IL represents a risk, in terms of the integrity of the proof of correctness of the system, however both IL and Modula-2 are Pascal like languages, so the translation between the two is relatively straightforward and easily verified.

MALPAS itself comprises many software evaluation tools. The one of prime interest in this work is the compliance analyser in conjunction with the IL reader.

The IL reader produces a directed graph for each program section. The nodes in this graph represent points in the program section, typically the beginning of statements. The arcs between the nodes represent the flow of execution of the statements in the program section. Associated with each arc is information about what the program does when the part of the program represented by that arc is executed.

In order to analyse the program section, the graph produced by the IL reader requires transforming; this is achieved by a reducer. Nodes are eliminated from the graph by combining arcs together to form a new graph structure. This process of graph reduction continues until there are no nodes remaining that may be eliminated. MALPAS uses ONE-ONE, HECHT, KASAI, HECHT-KASAI and TOTAL reduction [BRAb-80], (ONE-ONE is a subset of HECHT and KASAI). These are classes of node; a node is defined as possessing a type \([p, q]\), if it has \(p\) immediate predecessors and \(q\) immediate successors (not including itself). Table 8.1 gives an overview of the types of node.
Chapter 8. The Formal Verification of the Collector by use of MALPAS

<table>
<thead>
<tr>
<th>Type of Node</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>$p=0$</td>
</tr>
<tr>
<td>END</td>
<td>$q=0$</td>
</tr>
<tr>
<td>ONE-ONE</td>
<td>$p=1 &amp; q=1$</td>
</tr>
<tr>
<td>HECHT</td>
<td>$p=1 &amp; q&gt;0$</td>
</tr>
<tr>
<td>KASAI</td>
<td>$p&gt;0 &amp; q=1$</td>
</tr>
<tr>
<td>TOTAL</td>
<td>$p&gt;0 &amp; q&gt;0$</td>
</tr>
</tbody>
</table>

Table 8.1 Types of node.

Figure 8.1 [RTP-92b] shows the graph reduction for the program section consisting of one conditional statement (IF .. THEN .. ELSE .. ENDIF) and several assignment statements. The graph is fully reduced.

Figure 8.1 Example of (MALPAS) graph reduction.
Once the graph reduction phase is complete the compliance analyser may be invoked. This verifies the program against its specification, which is embedded in the program text. The specification is written in terms of pre and post conditions for each procedure. Optionally, a DERIVES relation list may be used to show how INOUT and OUT parameters are derived in terms of the procedure’s parameters. Extensive use may be used of functional replacement rules for both post and derives specifications.

The compliance analyser determines if the program diverges at any point from its specification, this is termed a threat to the program. Figure 8.2 shows how the threat to a program is derived in terms of its inputs, outputs and values mapped to the out parameters produced by the program (the line marked as PROC on the diagram, defining the mapping from IN values to OUT) [BRA-85].

\[
RISK = \text{PROC} - \text{POST}
\]
\[
\text{THREAT} = \text{PRE} \cap \text{domain}(\text{RISK})
\]

**Figure 8.2** Compliance analysis.

The program is considered to receive inputs, in the set \(IN\) and produce outputs in the set \(OUT\). Compliance analysis seeks to compare the actual behaviour of the program with its intended behaviour.
For the intended behaviour, only certain inputs are deemed to be legal. This is denoted by the set $\text{PRE} \subseteq \text{IN}$. Furthermore, for each $\text{in} \in \text{IN}$, the program is intended to produce an element $\text{out} \in \text{OUT}$, such that the pair $(\text{in}, \text{out})$ is also legal. The set of legal $(\text{in}, \text{out})$ pairs is denoted by $\text{POST} \subseteq \text{IN} \times \text{OUT}$ (this is denoted by the light grey area in figure 8.2). Thus, the program, with inputs in $\text{PRE}$ should produce outputs in $\text{POST}$.

The actual behaviour of the program is denoted by some subset $\text{PROC} \subseteq \text{IN} \times \text{OUT}$ (denoted by the entire shaded area in figure 8.2). The THREAT to the program, expressed as a predicate on $\text{IN}$, comprises those inputs that satisfy the pre-condition but cause the post condition to be violated. When the program meets its specification then the THREAT evaluates to false.

A simple example is given below in figure 8.3, of a procedure, written in MALPAS IL and its specification. This is a procedure scan_con (lines 18 - 53) that given a node, $x$, returns its scan_condition, $\text{sc}$. The first part of the IL is the procedure specification (lines 18 - 28), the second part (lines 30 - 53) is the IL procedure body. The function $s_c$ is also given in the example (lines 1 - 15).
FUNCTION s_c (vertex, node_details-array, boolean, integer, integer, integer): scancondition;
REPLACE (X: vertex; n_a: node_details-array; ef: boolean; et: integer; eg: integer; s_g: integer)
s_c (X, n_a, ef, et, eg, s_g)
BY nn IF SCAN (n_a!ID x) = a AND GROUP_ID(n_a!ID x) = s_g,
BY pp IF SCAN (n_a!ID x) = b AND NOT ef,
BY qq IF SCAN (n_a!ID x) = b AND ef,
BY fz IF SCAN (n_a!ID x) = c AND NOT ef,
BY fx IF SCAN (n_a!ID x) = c AND ef AND NUMBER x < et
AND GROUP_ID(n_a!ID x) = s_g,
BY fe IF SCAN (n_a!ID x) = c AND ef AND NUMBER x >= et
AND GROUP_ID(n_a!ID x) = s_g,
BY ee IF GROUP_ID(n_a!ID x) = eg,
BY xx;

PROCSPEC scan_con (IN x: vertex OUT sc: scan_condition)
(OUT sc: scan_condition)
(IN node_array: node_details-array
IN emit_flag: boolean
IN emit_group_id: integer
IN spl_grp: integer
IN emit_thresh: integer)
DERIVES sc AS s_c (X, node_array, emit_flag, emit_thresh,
emit_group_id, spl_grp)
PRE NOT GARBAGE (node_array!ID x)
POST sc = s_c ('x, 'node_array, 'emit_flag, 'emit_thresh,
'emit_group_id, 'spl_grp);
PROC scan_con;
IF SCAN (node_array!ID x) = a AND
GROUP_ID(node_array!ID x) = spl_grp THEN
sc := nn;
ELSIF SCAN (node_array!ID x) = b AND NOT emit_flag THEN
sc := pp;
ELSIF SCAN (node_array!ID x) = b AND emit_flag THEN
sc := qq;
ELSIF SCAN (node_array!ID x) = c AND NOT emit_flag THEN
sc := fz;
ELSIF SCAN (node_array!ID x) = c AND emit_flag AND
NUMBER x >= emit_thresh AND
GROUP_ID(node_array!ID x) = spl_grp THEN
sc := fe;
ELSIF SCAN (node_array!ID x) = c AND emit_flag AND
NUMBER x < emit_thresh AND
GROUP_ID(node_array!ID x) = spl_grp THEN
sc := fx;
ELSIF GROUP_ID(node_array!ID x) = emit_group_id THEN
sc := ee;
ELSE
sc := xx;
ENDIF;
ENDPROC

Figure 8.3 The formal model of a procedure using MALPAS IL.

There are a number of points worthy of note in this procedure specification and formal model. The out parameter of scan_con (sc) is modelled by the function s_c, in the derives relation (lines 24 - 25) which details a number of replacement rules to derive its value, in terms of the procedure’s inputs [SMI-89]. In this case the post condition simply mimics the
Chapter 8. The Formal Verification of the Collector by use of MALPAS

derives relation, typically the post condition may also have further predicates not present in the derives list. When scan_con is called MALPAS is capable of modelling the call in two ways:

the first and simplest is to substitute the derives relations for all INOUT and OUT parameters of the called function. If the derives list is not complete MALPAS invents abstract functions for the missing parameter definitions (after giving a warning that it is doing this);

the second method of modelling a procedure call is to annotate the call, where it occurs in the body of the IL text with the postfix ASSUME POST; this declaration includes the post condition in the threat assessment to the correctness of the calling procedure (see below for a discussion of threat assessment).

The MALPAS IL reader produces the following for the procedure (when called as a front end to compliance analysis), the partial output from the reader for compliance analysis is given in figure 8.4. The reader constructs assignments to built in variables _assume, _threat, _as_threat. At the end of the procedure MALPAS evaluates the following:

\[
\text{MAP}
\]
\[
\quad \_\text{as\_threat} \ := \ _\text{assume} \land _\text{predicate} \land \neg<\text{procedure AS relations}>
\]
\[
\quad \_\text{threat} \ := \ _\text{assume} \land _\text{predicate} \land \neg<\text{procedure POST expression}>
\]
\[
\text{ENDMAP}
\]

where MAP indicates the start of a parallel assignment block. The built in variable _assume is initially set:

\[
\quad _\text{assume} \ := \ <\text{procedure PRE expression}>
\]

and _predicate is the total path accessor predicate (the predicate that is evaluated to arrive at the point where the threat evaluation takes place). Finally, all threats are ORed together to form a total threat for the entire procedure.

The compliance analysis within MALPAS evaluates these threats for every semantically possible path through the IL code. It reports if there is a breach in veracity of the compliance of the implementation to its specification (rather than its correctness). If the procedure fully meets its specification both threats evaluate to false.

When a procedure is called, from the body of another procedure, then MALPAS checks that the preconditions for the called procedure cannot be violated. It ascribes a threat to the precondition of the called procedure. This evaluated to FALSE when there is no threat to the precondition.

For other features of the IL language the reader is referred to the language reference manual [RTP-92a] and the MALPAS user guide [RTP-92b]. The output of the compliance analysis phase and the MALPAS log file produced is given in appendix B.
Section 8.2 An Outline of the MALPAS System

PROC SPEC scan_con (IN x: vertex
OUT sc: scan_condition)
IMPLICIT (IN node_array: node_details.array
IN emit_flag: boolean
IN emit_group_id: integer
IN spl_grp: integer
IN emit_thresh: integer)
DERIVES sc AS s_c (x, node_array, emit_flag, emit_thresh,
emit_group_id, spl_grp)
PRE NOT GARBAGE (node_array !ID x)
POST sc = s_c ('x, 'node_array, 'emit_flag, 'emit_thresh,
'emit_group_id, 'spl_grp);

PROC scan_con;
#1: _assume := <procedure scan_con PRE expression>
#2: IF SCAN (node_array!ID x) = a AND
GROUP_ID(node_array!ID x) = spl_grp THEN
#4: sc := nn;
#5: [SKIP]
#6: ELSIF SCAN (node_array!ID x) = b AND NOT emit_flag THEN
#7: sc := pp;
#8: [SKIP]
#9: ELSIF SCAN (node_array!ID x) = b AND emit_flag THEN
#10: sc := qq;
#11: [SKIP]
#12: ELSIF SCAN (node_array!ID x) = c AND NOT emit_flag THEN
#13: sc := fz;
#14: [SKIP]
#15: ELSIF SCAN (node_array!ID x) = c AND emit_flag AND
NUMBER X = emit_thresh AND
GROUP_ID(node_array!ID x) = spl_grp THEN
#16: sc := fe;
#17: [SKIP]
#18: ELSIF SCAN (node_array!ID x) = c AND emit_flag AND
NUMBER X < emit_thresh AND
GROUP_ID(node_array!ID x) = spl_grp THEN
#19: sc := fx;
#20: [SKIP]
#21: ELSIF GROUP_ID(node_array!ID x) = emit_group_id THEN
#22: sc := ee;
#23: [SKIP] ELSE
#24: sc := xx;
#25: [SKIP]
#3: ENDF;
#26: [SKIP]
#STOP MAP
_as_threat := _assume AND _predicate AND NOT <procedure
scan_con AS relations>
_threat := _assume AND _predicate AND NOT <procedure
scan_con POST expression>
ENDMAP

#END: ENDPROC

Figure 8.4 Output of the MALPAS reader for compliance analysis.
8.2.1 Establishing the Proof of Correctness of a Program.

The above discussion has shown how the use of PRE and POST conditions, along with DERIVES relations can be used to establish the correctness of a program. It is important to be precise concerning the meaning of "correctness"; in this context it signifies that the program does not diverge from its specification.

The "art" of writing a complete, consistent and unambiguous software specification is a matter of current research (for example see [ALS-93]). However, once this has been derived and an implementation modelled in IL, then, for each procedure in the IL, the INOUT and OUT parameters may be modelled (using DERIVES relations and POST conditions) to provide a complete model of how that procedure maps its inputs to its outputs. No side-effects are allowed in MALPAS, all variables are included in the parameter list for a procedure. The IMPLICIT declaration (see figure 8.3 above, as an example (lines 19 - 23)), efficiently models "global" variables that are in scope within the procedure. The procedure may not modify any other variables not declared in either the explicit or implicit parameter lists. This means that a necessary set of conditions is present to model a procedure [WAR-87], i.e. we must model all parameters that are modified by a procedure. There is an exception to this rule, MALPAS will invent an abstract function (after issuing a warning) for any parameters not modelled. (The collector flag emit_group is modelled in this way, see appendix B). This may be useful where the requirement to model such parameters is not as stringent as others. An example of this may be a procedure that sets a boolean, if we do not model how this is set, in terms of the procedure's parameters, then MALPAS will invent an abstract function that will return true or false whenever it is encountered. This may be sufficient to prove the correctness of another procedure, (i.e. establishes that the threat to the calling procedure is always false, no matter what value the boolean takes).

The modelling of all INOUT and OUT parameters provides a necessary constraint on the MALPAS modelling technique (with the caveat concerning abstract functions). However, if the specification for the procedure is presented in terms of the procedure's inputs and outputs this also provides a sufficient set of constraints. Unfortunately, this form of specification is unlikely to be encountered in a practical system. The specification is usually presented at a higher level of abstraction (unless a formal notation, such as Z [SPI-89] is used). As a result, the POST condition of a procedure usually has to be made "richer" to provide a complete specification that meets the procedure's requirements [WAR-87]. However, this method of modelling gives a much better framework for the development of proofs than the "ad hoc" methods used in the past. As an example of this problem, one collector [DIJ-78] was presented and then found susceptible to a mode of failure concocted by Woodger (Woodger's scenario [BEN-84]). Since MALPAS checks every path through the program, no "surprises" like this can be lurking in the logical undergrowth of the algorithms.
8.3 An Overview of the MALPAS IL Model of the Collector

The basic real time collector was formally modelled using MALPAS IL. The model comprised the real time incremental scheme, for the collection of redundant cyclic structures, as presented in chapter 6. The complete source for this is presented in appendix B.1. The purpose of this section is to highlight some of the more important aspects of the proof of correctness of the collector. The reader is referred to the IL text and MALPAS output (appendix B) for a complete description of the proof.

The reduction graph is represented as an abstract data type, tree, with a single entry point ROOT. A node in the graph is represented as a vertex that has an id, lowlink, number, visited (flag) and stacked (a flag indicating that the node is on the Tarjan stack). The local count, l_count and group identifier, group_id, garbage and cyclic_free flags and the scan condition of the node, scan, are stored in a node_details-array. The group count is stored in another array, the g_array. Each vertex in tree is associated with a corresponding element in the node_details-array and g_array arrays. The limited width count scheme and the incremental collection of acyclic data structures is not modelled in order to reduce the demands on the MALPAS simplifier. No optimisation of the storage of lowlink or dfsn (number) is made.

Each node in the reduction graph is associated with an enumerated type, scan. This takes the values a, b or c, where a indicated that strong_connect has not visited that node, b indicated that the visited flag is set, but not scanned and finally c indicated that both flags are set. This device is used purely for convenience in the IL model.

8.3.1 The newapp Procedure.

The action of this procedure is described in detail in chapter 6 (section 6.3.3). In essence it takes as its input a parent node and a two children. It then takes a node from the free list, the parent becomes this new node's parent and the two children become its children. The new node is either incorporated into the existing S.C.C. structure or forms a singular S.C.C. by itself. The procedure must also maintain the invariants of the emitgroup procedure (see section 8.3.3 below).

The specification for this procedure is given in figure 8.5. Some simplifications are made. The new node, taken from the free list, is pre-loaded to point to its children, these being taken as non-garbage accessible cells in the reduction graph. (Note, a free list is not maintained in this model, but all cells are marked by a flag, garbage and placed on an explicit stack when they become free. This model of the free list is applied for reasons of simplicity).

The only precondition applied is that the free list should not be empty (since the procedure will take the new cell from this list).

In the case of this procedure the derives relation mirrors the postconditions, so only the postconditions will be discussed here. First, the new node (new_app) is taken as the first node from the free list. Secondly, the function newapp_na ensures the new node is placed in the correct group (in accordance with the policy outlined in section 6.3.3). The variable grp_id is maintained as the next available group descriptor number, this is incremented by newapp_grp_id when the new node is a singular node by itself (i.e. not incorporated into another S.C.C.). The flag emit_stop is set when the the parent is in the f2 state and one of the children is also in the group being split. The functions newapp_ga and newapp_iec ensure that the emitgroup totals are maintained correctly. Finally the last two parts of the
postcondition shows that the *split_queue* and the appropriate group handles are maintained correctly. This models the complete action of the procedure, ensuring that post conditions applied are sufficient to prove the correct operation of the algorithm.

This establishes partial correctness of the procedure. Total correctness is trivially established by MALPAS by checking that there are no loop structures in the procedure and that the END of it is unique and reachable by all paths of the procedure. This guarantees the liveliness of the procedure.
Section 8.3 An Overview of the MALPAS IL Model of the Collector

PROCSPEC newapp (IN par: vertex
INOUT g_array: integer-array
INOUT node_array: node_details-array
OUT new_app: vertex)

IMPLICIT (IN spl_grp: integer
INOUT emit_stop: boolean
INOUT split_queue: integer-list
INOUT group_handle: vertex-list-array
INOUT grp_id: integer
IN free_list: vertex-list
IN emit_thresh : integer
IN emit_flag: boolean
IN graph: vertex-tree
INOUT emit_interior_count: integer
IN emit_group_id: integer)

DERIVES new_app AS FIRST free_list,
node_array FROM node_array & free_list & par & graph & emit_flag & emit_thresh & emit_group_id & spl grp & g_array
AS newapp_na (node_array, FIRST free_list, par, graph, emit_flag, emit_thresh, emit_group_id, spl_grp, g_array),
grp_id FROM node_array & free_list & par & graph & emit_flag & emit_thresh & emit_group_id & spl_grp & g_array
AS newapp_grp_id (node_array, FIRST free_list, par, graph, grp_id, emit_flag, emit_thresh, emit_group_id, spl_grp, g_array),
emit_stop FROM par & node_array & emit_flag & emit_thresh & fz & emit_group_id & spl_grp & free_list & g_array & graph
AS (s_c (par, node_array, emit_flag, emit_thresh, emit_group_id, spl_grp) = fz) AND (
  sg (par, ROOT LEFT ret_graph (ID FIRST free_list, graph), g_array, node_array) OR
  sg (par, ROOT RIGHT ret_graph (ID FIRST free_list, graph), g_array, node_array)),
g_array AS newapp_ga (node_array, FIRST free_list, par, graph, emit_flag, emit_thresh, emit_group_id, spl_grp),
emit_interior_count AS newapp_iec (node_array, FIRST free_list, par, graph, emit_flag, emit_thresh, emit_group_id, spl_grp),
split_queue FROM node_array & free_list & par & graph & emit_flag & emit_thresh & emit_group_id & spl_grp & g_array
AS newapp_ps (node_array, FIRST free_list, par, graph, emit_flag, emit_thresh, emit_group_id, spl_grp, g_array, split_queue),
group_handle FROM node_array & free_list & par & graph & emit_flag & emit_thresh & emit_group_id & spl_grp & g_array & group_handle
AS newapp_ph (node_array, FIRST free_list, par, graph, emit_flag, emit_thresh, emit_group_id, spl_grp, g_array, group_handle)

PRE NOT EMPTY free_list
Figure 8.5 The newapp procedure specification.

The procedure freecell checks that all cells that become garbage have a zero reference count and that those cells that should be placed upon the cyclic_free_stack are correctly appended to this stack.

This form of specification (given for newapp) is typical of the rest of the procedures in the MALPAS specification (apart from visit and emitgroup which are detailed below) and so will not be presented here. They follow the straightforward outlines given in chapter 6. The reader is referred to the complete IL model in appendix B for details of the other major procedures.

8.3.2 The visit Procedure.

The primary requirement for the procedure visit, as described in chapter 6, is to identify S.C.C.s within the reduction graph. It does not change any facets of the graph or nodes dealing with local or group reference counts.

The visit procedure is modelled as the strong_connect procedure in MALPAS. The preconditions for the call of strong_connect are straightforward. The graph should not be empty, the root of the graph (the point at which the Tarjan algorithm is called) has not been visited by the procedure yet and that the free list is not empty. The post conditions state that the graph should not be garbage from the root (i.e. traced from the root, all accessible cells should not be garbage), the root node has been marked as visited or scanned and finally the
Section 8.3 An Overview of the MALPAS IL Model of the Collector

The function visit gives a simple set of replacement rules that maintains the invariant, for the part of the reduction graph to be acted upon by the Tarjan search, this is based upon the algorithmic proof offered by Tarjan [TAR-72].

LOWLINK (v) = \min \left( \{v\} \cup \{wl(v, w) \land \exists u(u, w, v) \land \exists w(u, w, v) \land (u, w, v) \in S_G)\} \right)

That is LOWLINK (v) is the smallest vertex (has the lowest depth first search number) which is in the same S.C.C. as v (S_G) of the graph G and is reachable by traversing zero or more tree arcs followed by at most one frond or cross links (denoted by \(\rightarrow\)). Tarjan has shown that when LOWLINK (v) = v then this is the root of an S.C.C. (and so triggers emitgroup). MALPAS verifies that this holds after a call to the Tarjan search. Correctly calculating the value of lowlink for a node ensures that the root of the S.C.C.s are identified correctly, which in turn implies that the S.C.C.s have been identified [TAR-72].

The interference of the mutator with the collector is modelled in strong_connect as non-deterministic guarded command loop (by use of the procedure comm) between calls to critical sections of the code. The choice of continuing the loop or choosing execution of newapp, deleteref or addref_ext is non-deterministically made. The action of the mutator on the graph is simulated by choosing accessible nodes at random from the graph, using the ra_no function (in order to provide parameters to newapp, deleteref and addref_ext where necessary).

The MALPAS log file is given in appendix B.2. It can be seen that there is a threat to the Tarjan search identified for those paths where the mutator interferes with the procedure. This threat (supplied in the MALPAS output file, this is not given for the sake of brevity) arises from the fact that under certain circumstances (described in chapter 6) newapp, deleteref and addref_ext can cause the S.C.C. currently being acted upon to be resplit (the S.C.C. is pushed onto the split_queue again under these circumstances). MALPAS verifies that it is only under these specific circumstances that the graph is not as described by visit (all other post conditions hold). Figure 8.6 shows part of the output from the compliance analysis. This illustrates a threat to strong_connect. It starts by stating that no accessible nodes are garbage, from the root, the graph is not empty etc. It shows that the condition implemented by the function visit does not hold (text emboldened). This occurs when the parent and child nodes of deleteref are in the f_s state, as would be expected (see chapter 6), so causing the S.C.C. to be resplit. The IL specification for deleteref proves that under these circumstances the S.C.C. is indeed returned to the split_queue correctly.
Chapter 8. The Formal Verification of the Collector by use of MALPAS

\[ \text{false_pre_deleteref}_{11} := \]
\[ \text{not_garbage}(\text{ID ROOT}(\text{graph}), \text{graph}, \text{node_array}) \]
\[ \text{AND GARBAGE} (\text{node_array} ! \text{ID ra_no}(\text{REST}(\text{hidden}), \text{graph}, \text{ii}, \text{node_array})) \text{ AND} \]
\[ \text{NOT}(\text{emit_flag}) \text{ AND NOT}(\text{TEMPTY}(\text{graph})) \]
\[ \text{AND NOT}(\text{EMPTY}(\text{hidden})) \text{ AND NOT}(\text{EMPTY}(\text{free_list})) \text{ AND NOT}(\text{EMPTY}(\text{REST}(\text{hidden}))) \]
\[ \text{AND SCAN} (\text{node_array} ! \text{ID ROOT}(\text{graph})) = a \text{ AND FIRST}(\text{hidden}) = \text{two AND grp_id} > 0 \]

\[ \text{false_pre_addref_ext}_{16} := \]
\[ \text{not_garbage}(\text{ID ROOT}(\text{graph}), \text{graph}, \text{node_array}) \]
\[ \text{AND GARBAGE} (\text{node_array} ! \text{ID ra_no}(\text{REST}(\text{hidden}), \text{graph}, \text{ii}, \text{node_array})) \text{ AND} \]
\[ \text{NOT}(\text{emit_flag}) \text{ AND NOT}(\text{TEMPTY}(\text{graph})) \]
\[ \text{AND NOT}(\text{EMPTY}(\text{hidden})) \text{ AND NOT}(\text{EMPTY}(\text{free_list})) \text{ AND NOT}(\text{EMPTY}(\text{REST}(\text{hidden}))) \]
\[ \text{AND SCAN} (\text{node_array} ! \text{ID ROOT}(\text{graph})) = a \text{ AND FIRST}(\text{hidden}) = \text{three AND grp_id} > 0 \]

\[ \text{threat} := \text{not_garbage}(\text{ID ROOT}(\text{graph}), \text{graph}, \text{node_array}) \text{ AND NOT}(\text{emit_flag}) \text{ AND} \]
\[ \text{NOT}(\text{TEMPTY}(\text{graph})) \]
\[ \text{AND NOT}(\text{EMPTY}(\text{hidden})) \text{ AND NOT}(\text{EMPTY}(\text{free_list})) \text{ AND} \]
\[ \text{NOT}(\text{EMPTY}(\text{deleteref}_18(\text{free_list}, \text{node_array}))) \]
\[ \text{AND graph NE visit}(\text{graph}, i, 0, \text{g_array}, \text{spl_grp}, \text{nodearray}) \]
\[ \text{AND SCAN} (\text{node_array} ! \text{ID ROOT}(\text{graph})) = a \text{ AND FIRST}(\text{hidden}) = \text{two} \]
\[ \text{AND s_c}(\text{ra_no}(\text{REST}(\text{hidden}), \text{graph}, \text{ii}, \text{node_array}), \text{node_array}, \text{false}, \text{emit_thresh}, \text{emit_group_id}, \text{spl_grp}) = fz \]
\[ \text{AND s_c}(\text{ra_no}(\text{REST}(\text{hidden}), \text{graph}, \text{ii}, \text{node_array}), \text{node_array}, \text{false}, \text{emit_thresh}, \text{emit_group_id}, \text{spl_grp}) = fz \]

Figure 8.6 Output of the MALPAS compliance analysis.

Figure 8.6 shows that the preconditions of both \textit{deleteref} and \textit{addref_ext} can be violated (these procedures are described in later sections). This is because the node chosen from the graph is actually garbage. This corresponds to a mutator error so does not concern us here.

It should be noted that all semantically distinct node types are used in the correctness tests.

Apart from the cases where the group must be resplit, due to the action of mutator requesting a service on part of the graph that is currently being split (in which case we resplit this S.C.C.) the procedure \textit{strong_connect} is shown to meet its specification (i.e. correctly identifies S.C.C.s within the reduction graph).

This has established partial correctness for \textit{strong_connect}. Total correctness may not be demonstrated here, because the mutator may add nodes and pointers to the graph, within the cycle that is currently being scanned. This means that these nodes too may be visited. This process is obviously indeterminate with respect to the operation of the collector.

### 8.3.3 The emitgroup Procedure.

The requirement for the procedure \textit{emitgroup} is that once the root of a S.C.C. has been identified by \textit{visit}, it should pop all nodes off the Tarjan stack, that have a \textit{dfsn} greater or equal to the threshold value set by \textit{visit}; these form a new S.C.C. It should also maintain a count of the number of intra-cyclic pointers (\textit{emit_interior_count}) as they are emitted into the new group, and the final group count for the new S.C.C. should equal the sum of all the local reference counts of its members, minus \textit{emit_interior_count}.
The preconditions for *emit* are quite simple (see figure 8.7). They state that the Tarjan stack should not be empty, its first element must have a *dfs* that meets or exceeds the threshold value, the function *element_test* must hold (this checks that at least one member of the Tarjan stack has a *dfs* that meets or exceeds the threshold value).

The procedure *emit* simply increments the current (global) S.C.C. number, *grp_id*, to assign a new, unique number to the new S.C.C. The postconditions state that *element_test* should not hold after *emit* has completed (this ensures that all nodes that are eligible have migrated to the new S.C.C.). Also, *emit_interior_count* is maintained correctly (taking into account the interference from *newapp* and *addref_ext*, provided by *t_iec_inf*). The new S.C.C. should have a group reference count that is calculated as above (again taking into account the interference from *newapp*, *addref_ext* and *deleteref*, provided by *t_g_inf*). Finally, the function *emit_na* ensures that all nodes within the Tarjan stack have correctly migrated to the new S.C.C.

As with *strong_connect*, the interference of the mutator with the collector is modelled by calls to *newapp*, *addref_ext* and *deleteref*. These calls are made at the start of each loop within *emit*, to simulate the critical section, as defined in chapter 6.

It should be noted that an abstract function is used to model the *emit_flag* parameter for *emit*. The implication of this is that when *emit* is called from another procedure, the boolean *emit_flag* is simply modelled as either true, or false. Figure 8.7 also shows other minor pre and post conditions. These will not be discussed here since they are not fundamental to the correctness proof of the procedure (they are used as part of the mechanism to implement non-determinism within MALPAS).
Chapter 8. The Formal Verification of the Collector by use of MALPAS

**Figure 8.7 The emitgroup procedure specification.**

We can summarise the specification for `emit` by considering the following. The specification for this relies on two key replacements rule sets, the function `int_count` that ensures that `emit` maintains the invariant:
Section 8.3 An Overview of the MALPAS IL Model of the Collector

\[ emit_{\text{interior\_count}} = \sum_{n=0}^{N} \text{node}(n)_p \]

where \( \text{node}(n)_p \) is the number of interior pointers from the node that \( emit \) is currently processing, \( N \) is the total number of nodes in the new S.C.C. The procedure \( emit \) also maintains the local invariant for \( group_{\text{count}} \), modelled by the function \( g_{\text{count}} \) as

\[ group_{\text{count}} = \sum_{n=0}^{N} \text{node}(n)_l \]

where \( \text{node}(n)_l \) is the local reference count for the \( n^{th} \) node, currently being processed by \( emit \).

The new group's reference count, \( g_{\text{count}} \) is derived by \( emit \) as:

\[ g_{\text{count}} = group_{\text{count}} - emit_{\text{interior\_count}} + g_{\text{count}} \]

The derivation of \( g_{\text{count}} \) involves the \( g_{\text{count}} \) itself because this is potentially modified by \( \text{newapp} \) or \( \text{deleteref} \) between iterations of the loop in \( emit \).

The maintenance of these invariants establishes partial correctness for \( emit \). We must also consider total correctness. This is established very simply by considering the length of the emit stack. This forms part of the loop assertion; the total correctness is demonstrated by the fact that each iteration of the loop in \( emit \) monotonically decreases the size of the \( emit_{\text{stack}} \). The mutator cannot interfere with this (i.e. the mutator does not insert nodes onto the stack) and so the emit loop is guaranteed to terminate.

It should be noted, as with \( \text{strong\_connect} \) there is a PRE threat ascribed to the call of \( \text{addref\_ext} \) and \( \text{deleteref} \) because of the possibility that the mutator selects a garbage cell to be referenced (again this describes a fault condition in the operation of the mutator).

8.3.4 The \( MAIN \) Procedure.

The main controlling loop of the collector, given in the \( MAIN \) IL procedure, is modelled using an infinite loop (we are not concerned with the termination properties of this since we require the collector to run as long as the mutator is active). Figure 8.8 gives this procedure. No scheduling action is implemented in the IL model of the collector. We are not able to establish the total correctness of the next inner loop, that pops nodes from the split queue. This is because the action of the mutator may cause nodes to be added to the \( split_{\text{queue}} \) between iterations of this loop. As a result of this the termination properties of this loop depend upon the mutator interference. This argument is also applied to the innermost loop that takes nodes from a particular S.C.C.'s handle. This again may be added to in between iterations of this loop due to the action of the mutator.
Chapter 8. The Formal Verification of the Collector by use of MALPAS

MAIN
VAR spl_grp, grp_id, emit_group_id, emit_interior_count: integer;
VAR z, aa, bb, cc, dd: vertex;
VAR temp_group_handle: vertex-list;
VAR init_sg, ii: integer;
VAR pg: vertex-tree;
emit_stop:= false;
spl_grp:= 0;
grp_id := 1;
init_sg := 0;
declare (node_array, lower_heap_limit, upper_heap_limit)  ASSUME POST;
declare (g array, lower_heap_limit, upper_heap_limit)  ASSUME POST;
declare (group_handle, lower_heap_limit, upper_heap_limit)  ASSUME POST;
WHILE
ASSERT NOT EMPTY hidden AND NOT T_EMPTY graph AND
not_garbage (ID ROOT graph, graph, node_array) AND NOT EMPTY free_list;
true [loop forever]
LOOP
init_sg := 0;
WHILE
ASSERT NOT EMPTY hidden AND NOT T_EMPTY graph AND
not_garbage (ID ROOT graph, graph, node_array);
NOT EMPTY split_queue [Loop whilst there are nodes on the split queue]
LOOP
[spl_grp is set here]
pop_split (split_queue, spl_grp) ASSUME POST;
temp_group_handle := group_handle!spl_grp;
WHILE
ASSERT true;
NOT EMPTY (group_handle!spl_grp)
LOOP
pop_handle (z, spl_grp) ASSUME POST;
pg := ret_graph (ID z, graph);
strong_connect (pg, init_sg) ASSUME POST;
ENDLOOP;
IF emit_stop THEN
[This group will be resplit,
so restore its old group handle,
minus any nodes that have been deleted]
store (group_handle, spl_grp, temp_group_handle)  ASSUME POST;
ENDIF;
emit_stop:= false;
spl_grp:= 0;
[Can now safely reclaim node that were on the cyclic_free_list]
process_cyclic_free_list (cyclic_free_list)  ASSUME POST;
ENDLOOP;
ENDMAIN

Figure 8.8 The MAIN procedure of the MALPAS collector model.
Figure 8.9 shows that there is a threat to the preconditions for strong_connect. These reflect the situation where the free list is exhausted (in which case graph reduction must halt), or the reduction graph is empty or the hidden list is empty (this is purely a device in MALPAS to model non-deterministic behaviour and is unimportant to the proof of correctness of the collector). Apart from the cases detailed above the MAIN procedure establishes that all procedures are called correctly and that there are no threats to their preconditions. The loop assertions (see figure 8.8) ensures that all nodes that are accessible from the root of the graph have not (erroneously) become garbage.

The MALPAS IL model of the collector does not implement a separate process for the disposal of garbage groups. It models a simple scheme where once a group becomes redundant it is traced, stacked and collected (rather than being placed upon the group_dispose_stack). For the sake of simplicity, no mutator interrupts are modelled during this process.
8.4 Conclusions.

This chapter has outlined the proof of the collector using MALPAS. It has established total correctness for all parts of the program, excepting the MAIN routine and strong_connect that implements Tarjan's depth first search algorithm. These two procedures are interfered with by the graph mutator, so we may only establish partial correctness for these. MALPAS shows that all procedures meet their specifications, that cells are only collected as garbage when they have a reference count of zero and are inaccessible from the root, and that garbage cells are collected correctly. No other threats to the program's compliance to its specification, other than those detailed in this chapter, are observed (see appendix B.2).

The full IL source, MALPAS log file and part of the compliance output are presented in appendix B.
Chapter 9. The Simulation.

9.1 Introduction to the Simulation.

The simulation of the techniques described above, was based upon a functional language interpreter, implemented in the first instance by Hughes. The entire system was then rewritten in Modula-2, maintaining some of the original structure, but implementing the ideas discussed in earlier chapters.

There were two main factors underlying the motivation to implement a simulation of the system. The first was to enhance the conviction that the collector design was sound. This lead to the inclusion of much code dedicated to detecting any erroneous behaviour in the collector (this is described below). The second factor was to gain some understanding of the expected requirement for resources in the collector (particularly in terms of CPU time) when the system is implemented.

The interpreter uses a simple S-expression functional language, named EASE, (Equations And Simple Expressions). An EASE program comprises a list of definitions, having the form (lhs, rhs). The left hand side is a variable name, or a function name, whilst the right hand side is an expression. Expressions are built from names and numbers by function application. Names without definitions are taken as symbolic constants. The following standard functions are defined in the language: S, K, I, B, C, Y, +, -, *, /, <=, =, atom, if, head, tail, cons, nil. Undefined symbols are not reduced. A ‘cons’ cell is simply left as an unreduced application, which is the form that both the ‘head’ and ‘tail’ functions expect to find such a cell. A program is executed by printing the value of the name “program”, by a suitable function printing routine. Presented below is a simple EASE script, by way of an example, that computes the factorial of 6.

\[
\text{(program (fact 6))
\text{(\text{\text{fact n} (\text{if} (=} \text{n} \text{0}) 1 \\text{(* n (fact \text{(- n 1)})}}))})
\]

When the simulator is run, the program is firstly compiled into combinator form [TUR-79, JOY-85], after which reduction proceeds. This is terminated when the value of “program” is returned to the user.

Reduction is carried out via a reduction stack, that holds pointers to objects. Objects can be numbers, combinators, symbols or applications. Evaluation is driven by print routines, that push a value onto the stack, reduces it and finally prints its normal form, that is an expression containing no reducible expressions (redexes) [PEY-87a].

Hartel has observed [HAR-88] that combinator reduction provides a much greater rate of storage accesses than supercombinator implementations [HUG-83]. For this reason it provides a worst case client for the collector; therefore a mutator based on combinator graph reduction is ideal to assess the suitability of a new real time collector technique.

The entire program is separated into a number of modules, the module Storage handles all memory management. The main facets of the functionality of this module were covered in the preceding chapters. Some further detail is given in this chapter. The module was implemented in some 7000 lines of Modula-2.
Chapter 9. The Simulation

The storage manager incorporates safeguards, to ensure that the simulator is operating correctly. When a cell is referenced, a check is made to see that it is not on the free list of cells (if it is, then this must constitute an error condition, and the simulator is halted, providing a suitable error message locating the source of the exception). Additionally, when reference count data is retrieved from the main heap area, if the number of searches exceeds the maximum length of segment for the current collector settings, then once again, an error condition is flagged. When a cell is appended to the free list, a check is made to see if the local reference count is zero. If this is not the case, then the only other mechanism that could legitimately cause such a cell to be freed, is if it is a member of an S.C.C., that is currently being disposed of. In such a case a check is also run to ensure that this group is being disposed of correctly (i.e., the relevant group count must be zero).

At the end of the reduction process, the entire heap spaced is scanned, checking that all nodes have been freed correctly. (This occurs after all cells and groups have been recovered, which have arisen due to the collector lagging the mutator operation). Checks are also made to ensure that there are no active groups, or that any local reference counts are above zero. Such checks would obviously normally not be necessary, in a practical implementation of the system.

Appendix A gives a sample of some of the examples run on the simulator. These were chosen so that the full functionality of the collector was employed. This is useful in terms of strengthening the assertion that the system is functioning correctly.

The simulator of the collector was implemented on a uni-processor architecture, and so true concurrent multitasking of the collector and mutator was not possible. Use was made of separate coroutines within modula-2, to simulate concurrent multitasking between collector and mutator. (The modula-2 tasking structure was not available in the version of the compiler used for this simulation).

9.2 Control of the Simulation.

The user of the simulator, has control over the size of both the local count and group number fields ($\alpha$ and $\beta$). These can be set to minimum and maximum values, with a user defined increment (so that the simulation can be automatically rerun with the new values set on each run). The program can then be run for various values of $\alpha$ and $\beta$ and data collected on each run.

The user can also set the size of the main heap that is to be used. This has a maximum value dependant upon the platform upon which the simulation is being run. (On an IBM PC emulator, this maximum was found to be only about 2500 cells).

The simulator has two primary modes of operation, time-step and event-driven.
## 9.2 Control of the Simulation

### 9.2.1 Time-step Mode.

In time-step mode, a number of user-defined parameters are available: \( \tau \), \( \theta \), \( \xi \) and \( \lambda \). These are used to determine when processes should switch from collector back to mutator (mutator to collector switches occur when a service is required by the mutator from the collector). The parameter \( \tau \) determines how many nodes will be deleted for a call to the `process_pointer_deletion_stack` (see figure 6.28). This parameter also sets the number of nodes processed by `process_group_dispose_stack`. The parameter \( \theta \) sets the number of recursive calls to the Tarjan search (visit) that may be made at one time; \( \lambda \) sets the number of iterations of `emitgroup`. Finally, \( \xi \), controls the number of searches in the main heap that are to be made for overflow data; setting this larger than the main heap segment effectively means that the search will always be completed at a single call.

This mode of operation simulates the world in which the collector always runs faster than the mutator and no servicing of interrupts from the mutator have to be performed. This is not realistic for a practical implementation of the collector architecture, but allows data to be collected on the performance of the collector when it is free-running. This situation also allows the effective CPU resources set for part or all of the collector to be tailored for the purpose of experimentation.

The code of the simulator is calibrated (for all operational modes) so as to be capable of reporting the performance of all the primary collector functions. The results are logged in terms of the primitives \( L_m \), \( L_r/w \) and \( H_r/w \), providing data on the total number of calls to each function, that executed in a particular number of steps. A graph can then be drawn of the total number of calls to a procedure, against the number of collector operations \( (L_m, L_r/w \text{ or } H_r/w) \). This records the frequency of the return time for a function (collector service) versus the return time. Examples of such graphs are given later in the chapter.

### 9.2.2 Event-driven Mode.

This mode simulates the normal mode of operation of the collector, as described in earlier chapters. The user declares which procedure is of interest, for any particular run. The collector is fired normally, in a time-step manner, until the function of particular interest is called. This collector procedure is initiated in the normal fashion, but the collector receives a dummy interrupt from the mutator during the course of operation of the function. A dummy interrupt is used, that serves no function in the graph reduction process, to simulate a real mutator interrupt. (A real interrupt cannot be used since the mutator and collector activity is mutually exclusive, on a single tasking uniprocessor simulation). The time taken for this interrupt to be sent, is a random time from the point at which the function of interest commenced. The user sets limits in which the random firing of the dummy interrupt takes place. The collector records how long the mutator has to be halted before the mutator request can be serviced. No additional operations are performed by the collector, on receipt of a dummy interrupt, and the normal time-step mode of simulation is carried on until the next call to the function of interest.

The results of the mutator delay period (given in terms of \( L_m, L_r/w, H_r/w \)) are logged to a file each time that collector function is called. This mode of operation simulates the action of the `MAIN` module. The `MAIN` module also handles the queueing of mutator interrupts. These are stacked only one deep, in the current implementation, in accordance with the description in chapters 6 and 7.

-159-
Chapter 9. The Simulation

In this way, the time taken for an interrupt to be serviced, in a multi-tasking collector/mutator architecture, can be investigated for any particular function. The user can specify how many times the EASE program should be run in this event-driven mode. Data is logged for each run (for one nominated collector function only) and average performance figures, for the collector, are given at the end of the series of runs. This gives a picture of the observed collector latency.

Event-driven mode has two sub-modes of operation. These are predictive sub-mode and non-predictive sub-mode. In predictive mode, the MAIN module acts normally and allocates more resources to the pointer deletion task pool etc, if the free list and heap is nearing exhaustion (this is the on-line scheduling policy, described in chapter 7). In non-predictive mode, the swapping between task pools remains unchanged at all levels of heap occupancy. This allows the investigation of such effects as changing collector control parameters on the memory overhead ratio and final residual garbage count (whilst using small heaps). If predictive mode was used, then this would distort the results. Non-predictive mode also allows the investigation of the pathological cases where the collector fails to deliver any free cells to the mutator, as the heap is exhausted.

The user can also turn incremental operation off, to simulate a non-real time implementation of the limited-width collector techniques.

9.2.3 Control of the Collector's Cache.

The size of cache can be defined before reduction proceeds. The cache is a small area of collector memory, where the most recently used pointers to both group, or local, reference count data (in the main heap) is held. Before a search of the heap is made, to retrieve such overflow count data, the cache is searched. If the required data is present, then a pointer is simply followed, from the cache area, to the main heap, so avoiding a costly search. The cache area is controlled very simply; if the required data is not present, then when it is found it is placed in the cache, and the bottom of the cache is deleted.

9.2.4 Results Obtainable from the Simulator.

As well as the simulated time taken by the collector (in terms of the primitives \( Lm, Lr/w, Hr/w \)), various results are given by the simulator, these include: the number of Tarjan searches called (and which combinator caused the Tarjan search), the nature of each cycle (its size and how it splits during the reduction process), the maximum number of cycles formed, the maximum value of both local and group reference counts, the maximum length of the pointer deletion list, the group dispose stack, the maximum size of the handle for each group, the number of cells occupied in the heap to cope with both local and group reference count data overflow and the number of cells that were not recovered (if any) at the end of reduction, due to incremental collection techniques. This final figure represents a measure of how much the collector is lagging the mutator in resupplying redundant cells back for reuse. Such unrecovered cells are identified separately from other cells, which were possibly not collected due to a malfunction in the collector (no collector malfunctions were observed). In addition to these statistics, the number of simulated collector and heap operations is given, for each run.
9.3 Results from the Simulator.

This section presents a selection of the results obtained from the simulation, for a number of different EASE programs (see appendix A for a sample of these programs). The results were chosen to compare against the theoretical predictions made in the earlier chapters, both for memory occupancy patterns and collector response characteristics. Only a small selection of results are presented (because of space limitations), that were found to be typical of the behaviour of the simulator, over the entire range of programs run.

In figure 9.1, a comparison is drawn between the theoretical overhead ratio, developed in chapter 4, and actual experimental results. These were obtained for the Sieve of Eratosthenes program (see appendix A). The collector was in time-step mode, and taken out of incremental operation. This mode employs normal incremental procedures, but the collector control parameters are ignored, so that all procedures terminate before returning control back to the mutator. This minimises the effects of the stack size, necessary to support the real time incremental scheme.

![Figure 9.1 Comparison of empirical and theoretical memory overhead values.](image)

The results shown in figure 9.1, reflect typical memory occupancy trends obtained from all the EASE programs run on the simulator. It can be seen how the experimental results, for low $\alpha$ and $\beta$, more closely mirror the theoretical predictions, than for higher $\alpha$ and $\beta$. This is because in this region, overflow has taken place to the main heap (as assumed in the theoretical model). For higher values of $\alpha$ and $\beta$, no overflow occurred, and so there is a greater divergence from the expected behaviour. In general, a lower overhead ratio was
obtained in the experiments than predicted. This can be accounted for by the fact that the average handle length ($\eta$) was found to be only about half the size of the average cyclic structure size ($P$, the average group size typically varies from 10 to 35 nodes). The variations in the experimental points from a smooth curve, can be accounted for by variations in the cyclic occupancy factor ($c$), whilst reduction takes place. This was taken to be constant in the models of chapter 4. The factor $c$ was found to be only about 0.05 for a heap of 2000 cells, that again leads to lower than expected values for the overhead ratio obtained.

The standard reference counting overhead figure is calculated, but reduced to reflect the actual overhead of storing group handles, in this particular experiment. It can be seen how much lower the memory overhead is, for the limited-width technique.

Figure 9.2 shows the effect of varying $\alpha$ and $\beta$, on the number of searches made in the main heap, in order to recover overflow data.

![Figure 9.2 No of searches in the main heap against max. numerical-width of group descriptor number](image)

Figure 9.2 shows how the number of searches performed in the main heap, reduces with increasing values of $\alpha$ and $\beta$. It can be seen that increasing the local count width, $\alpha$, has a greater effect in reducing the search count, than increasing the group descriptor width, $\beta$. This is because the number of cycles, for any particular run-time session, was found to be lower than the number of cells exceeding a given local reference count value, (i.e., there tends to be only a few cycles, where as there usually is a small distribution of cells with large local counts). The graph shows the exponential type fall-off, in the number of searches called, as would be expected, referring to equation 4.7.

In a practical application, it would be necessary for the collector to operate in regions were very few heap searches are called, in order to maintain efficient collector operation. From simulation runs on numerous examples, values of $\alpha = 5$ bits and $\beta = 3$ bits, provide a collector regime which sustains garbage collection, with extremely few calls to the main heap in order to retrieve overflow data.
Figure 9.3 shows how the allocation of a small cache (in this case with the capacity to hold up to ten pointers) decreases the number of searches that have to be made in the main heap, in order to recover overflow data.

![Bar chart showing search rate in heap memory](image)

**Figure 9.3** Effect of cache on the number of heap searches, for Sieve of Eratosthenes.

Figure 9.3 is derived from the same Sieve test as before, run with a constant limited-width, maximum local reference count value of 2, and the maximum value of group descriptor number varying from one to seven. It can be seen that even with such a small size of cache, the search rate in heap memory is cut by between about 55% and 95%. This figure does not take into account the extra searches that occur within the cache itself; if this is done then the improvement is still better than 45% (this takes no account of the fact that cache searching is less expensive in CPU time and bus bandwidth than heap searches).

It should be noted that the maximum size of this cache should be strictly limited (experiment seems to indicate an upper bound of about fifty entries). If the cache is made too large, then one search space in the heap is simply swapped for another, in the cache area.

The following graphs show actual response times obtained from the collector. The collector primitive `addref_ext` is used to illustrate the collector response, in terms of `Hr/w, Lr/w,` and `Lm` operations, for the simulation operating in time-step mode. These graphs effectively reveal the processing requirement of the collector when responding to a mutator interrupt request (assuming all processing is done before the interrupt is serviced). Two different settings of $\alpha$ and $\beta$ were used, to illustrate the overhead of reference count data spilling into the main heap. The results were obtained from the `cycle2` EASE script (see appendix A). It was found that other primitive collector functions displayed similar response distributions to `addref_ext`, in every test EASE program run. These are not presented here because of space limitations.

All the bar charts show the complete frequency behaviour of the function, (this implies that there are no returns from values above the maximum indicated on the graph). The graphs are given in a suitable scale, so that even very small frequency of returns can be distinguished.
Figures 9.4 and 9.5 demonstrate important characteristics of the collector, operating in time-step mode. For $\alpha = \beta = 3$ (figure 9.5), the predicted worst case performance is indicated with no heap overflow (as given by table 7.1). The observed results show values slightly greater than would be expected. This is because there is a small overflow to the main heap. Figure 9.4 displays a larger overflow. It can be seen that there is a cluster of operations at
Section 9.3 Results from the Simulator

about 500. This is due to recovery of cell details from the main heap. The value of $\xi$ was set so that all searches were completed at one time (i.e. the main heap search is not interrupted); when $\xi$ is set to unity, then this cluster is found to disappear and so the collector displays a much improved observed worst case performance.

With the small values chosen for $\alpha$ and $\beta$ ($N^o$ of nodes in the heap, $n = 2000$), the worst case performance, actually obtained from the collector, is at least thirty times better than that predicted in table 7.2. This can be accounted for by the fact that in the worst case analysis it was assumed that an overflow data item would be found in the last location in any given segment, (on average the required cell would be encountered half way through the linear search of the segment). It was also assumed in the worst case analysis, that overflow would occur for every cell in the graph, this, in practice, is clearly not the case.

It should be noted that the collector displays most of its activity, at much shorter return times than the maximum values obtained. This shows that the majority of the operation occurs when no overflow in main memory has occurred. It can be seen that the main activity of the collector, in terms of response time frequency, is centred about the same part of the frequency chart, for varying $\alpha$ and $\beta$. The main difference between $\alpha = \beta = 2$ and $\alpha = \beta = 3$, is the maximum value of response time displayed. This implies that, in general, the greater the value of $\alpha$ and $\beta$, the faster the average response time of a collector procedure. Perhaps more importantly, as $\alpha$ and $\beta$ increase, the observed worst case performance of the collector also improves.

The comparison between the predicted worst case behaviour of the simple scheme (with full-width fields, so no main heap overflow occurs), and the results obtained in figures 9.4 to 9.5 (and figures 9.6 to 9.9) show that, in general, the majority of the collector activity is centred close to the theoretical worst case behaviour of the simple scheme, as taken from table 7.1.

Figures 9.6 to 9.9 show similar results for addref_ext, with respect to $L_r/w$ and $L_m$ operations.
Chapter 9. The Simulation

Figure 9.6 Number of collector local read/writes versus cumulative total, ($\alpha = \beta = 2$).

Figure 9.7 Number of collector local read/writes versus cumulative total, ($\alpha = \beta = 3$).
Figure 9.8 Number of collector operations versus cumulative total, \((\alpha = \beta = 2)\).

Figure 9.9 Number of collector operations versus cumulative total, \((\alpha = \beta = 3)\).
In contrast to the finding for figure 9.5, the average empirical value for $Lr/w$ performance is actually lower in figure 9.7 than the worst case value shown. This is because this calculated worst case value is heavily dependent upon the scan_state of the target node. This in practice is found to vary, on entry to the procedure, lowering the overall $Lr/w$ value.

The final figure in this set (figure 9.9), shows that the collector displays a constantly higher value of $Lm$ operations, than the simple case prediction. This can be accounted for by the computational overhead necessary to support the limited-width scheme.

Another important area to investigate, is the average performance of the mutator. Measurements were taken to see how many mutator operations (analogous to collector operations) and how many mutator heap read/writes (or read/writes to mutator scratch space) were performed between calls to the collector. The results obtained are presented in figures 9.10 and 9.11.

![Figure 9.10](image)

**Figure 9.10** Number of mutator operations versus cumulative total.
Figures 9.10 and 9.11 show that in general, the simple mutator employed with the experimental collector, returns requests to the collector much faster than the collector will process the previous mutator request. Thus the idealised model presented in figure 6.30 has to be modified, to show a greater number of suspensions in mutator activity (such as the interval $t_1$ shown in figure 6.30). In general, if we assume that the basic grain of operation of the mutator is the same as the collector, then for this simple implementation of the mutator, the mutator will be frequently halted because the collector is usually running slower than the mutator. In a larger, practical system, the operation of the mutator may well be more complicated, and hence slower. This allows more time for the collector to operate, and hence the mutator will be halted for less of the time. If the collector architecture could be made faster than the mutator’s, then mutator suspension could be further minimised. As mentioned above, if a supercombinator implementation were used then the load on the collector would be much lighter [HAR-88], so decreasing the time the mutator would have to halt for the collector.

Figures 9.10 and 9.11 also reveal that there are times when the mutator takes quite a considerable time between calls to the collector. It is in these periods that the collector is able to run unhindered on its background tasks.

A measure of how much the collector is lagging the mutator operation, is given by the number of uncollected nodes left behind after reduction ceases. The greater the number, then the more the collector was lagging the operation of the mutator. The graph below gives a chart of the number of uncollected cells for various $\tau = \theta = \lambda = \xi$. 

![Graph showing number of mutator read/writes versus cumulative total.](image)
Figure 9.12 Number of uncollected cells versus collector time allocation.

Figure 9.12 uses a very coarse adjustment of setting $\tau = \theta = \lambda = \xi$. In practice, it is found that $\tau$ (the amount of work devoted to the processing of the pointer deletion stack) has the greatest bearing upon the residual cell count. This is why the MAIN module, whilst running in event-driven predictive mode, will concentrate on processing the pointer deletion stack (and group dispose stack) as the free list nears exhaustion, in order to free the maximum number of garbage cells as quickly as possible.

When the simulator is run in time-step mode, then the effects of varying $\tau$, $\theta$, $\lambda$, and $\xi$ can be observed. These parameters control the effective time that the collector is run for, as compared to the mutator. Lowering the value of any of these parameters, results in a smaller response time from the collector; however, as these parameter values decrease, the collector is given less time to perform garbage collection. Figure 9.12 gives a simple example of the relationship between these parameters and the amount of work performed by the collector, during the course of the entire reduction process. When reduction is complete, the number of uncollected garbage cells remaining in the heap is counted. This reflects the unperformed work by the collector, due to it lagging the mutator. Figure 9.12 gives the figures for the Fibonnaci series program (appendix A). It shows the general trend that if too small a "time-slice" is given to the collector, (i.e. the collector has to respond in a very short time), then the number of residual garbage cells rises sharply. (The maximum active heap size, for this example, was 267 cells). The time-slice can be varied to give either very low latency, if memory limitations are not critical, or lengthened if memory is at a premium.

The following figures, N\#s 13 to 15, show the behaviour of the collector in event-driven mode. They are taken for the procedure visit, running in the cycle2 EASE program. The results were gathered over 25 separate runs, and the average collector latency is given here. The collector was operated in non-predictive mode. They reflect the maximum length of time that a mutator would have to be suspended, after issuing an interrupt to the collector.
Section 9.3 Results from the Simulator

Theoretical maximum value

Figure 9.13 Number of collector local operations versus cumulative total.

Figure 9.14 Number of collector local read/writes versus cumulative total.
When the simulator is run in event-driven (non-predictive) mode, the effects of mutator interrupts can be observed. Figure 9.13 to 9.15 show how quickly the collector can respond to a mutator interrupt. The theoretical maxima are derived from table 7.1 (i.e. no main heap overflow occurs). No effects of memory overhead reduction techniques were taken into account, for this theoretical upper bound. This is because the collector can suspend a heap search, with very little extra processing. It only stores a marker to the point at which the main heap search had got to, so that this search can be resumed from that point, at a later time.

Figure 9.16 reflects the effect of changing $\tau$, against the overall collector memory overhead. This was performed for Sieve of Eratosthenes program, with $\delta = 11$ and $\alpha = \beta = 2$. 
Section 9.4 Comparison with Baker's Algorithm

Baker's real time copying algorithm [BAK-78] was implemented with the EASE functional system. This was done in order to compare the behaviour of this commonly encountered real time collector with our system.

The collector implemented is as described in chapter 5, section 5.5. The relocation constant, \( k \) was set to four, as suggested by Baker. In a similar fashion to our incremental reference counting collector, the code was calibrated in order to ascertain response times. Event-driven simulation was implemented to glean these figures (with the same mechanism as described above). The collector offered the same services as described in chapter 6.

Figure 9.17 gives the response time for Baker's collector, implementing the `newapp` service, in terms of local collector operations. Again the `cycle2` EASE program is used; the results were gathered over 25 separate runs with a semi space size set at 200 cells.

Figure 9.16 Effect of changing \( \tau \) on the memory overhead ratio.

Figure 9.16 shows the effect of varying \( \tau \) on the memory overhead. It can be seen that the smaller the value of \( \tau \), then the higher the overhead. This is because of the stack used to store pointer deletion operations and groups awaiting disposal. The smaller the value of \( \tau \), then the larger this stack becomes.

In event-drive predictive mode, the effects of the `MAIN` module switching priorities (to limited-panic mode) can be investigated. This occurs when small heap sizes are being used, and the free list and heap are nearing exhaustion. Testing shows that the heap can be made 8% smaller on average, than the equivalent non-predictive mode of operation, before full-panic mode is entered (where the mutator is suspended until a number of cells (usually set to three) is released).

9.4 Comparison with Baker's Algorithm.

Baker's real time copying algorithm [BAK-78] was implemented with the EASE functional system. This was done in order to compare the behaviour of this commonly encountered real time collector with our system.

The collector implemented is as described in chapter 5, section 5.5. The relocation constant, \( k \) was set to four, as suggested by Baker. In a similar fashion to our incremental reference counting collector, the code was calibrated in order to ascertain response times. Event-driven simulation was implemented to glean these figures (with the same mechanism as described above). The collector offered the same services as described in chapter 6.

Figure 9.17 gives the response time for Baker's collector, implementing the `newapp` service, in terms of local collector operations. Again the `cycle2` EASE program is used; the results were gathered over 25 separate runs with a semi space size set at 200 cells.
There are two distinct areas in the graph. The first reveals the very fast response time of the collector when garbage collection is not active. The second cluster shows the much increased collector latency that reflects the time taken for a mutator request for a new application cell, when garbage collection is active.

The overall "shape" of this performance is as expected. When garbage collection is active then $k$ (set to four in this case) cells are relocated, so lowering the response time. If no garbage collection is active then the response time is much faster. It is found that as the heap size is decreased, the average collector latency increases, since on average the collector spends more time in the phase where garbage collection is active.

For comparison figure 9.18 presents the results for the incremental reference counting collector (considering the newapp service once again).
Section 9.4 Comparison with Baker’s Algorithm

The reference counting collector displays a slightly better observed worst case performance, as compared to Baker’s collector. However, the latency is spread rather more than with Baker’s, there is not the concentration of activity at very small latencies.

Figure 9.19 shows the latency in Baker’s collector for the deletion of a pointer (using `deleteref`).

![Figure 9.19 Number of collector operations for Baker’s algorithm (for `deleteref`).](image)

As with the results for the `newapp` procedure, there are two distinct regions observed in figure 9.19. The majority of collector activity is centred around very low latencies. This is when a pointer is deleted directly. The small amount of activity observed higher up the time scale is due to pointers being updated when garbage collection is active (i.e. this represents the maintenance of the read barrier, as described in chapter 5).

The reference counting collector, for the operation of `deleteref`, gives a spread of results, similar to figure 9.18, but in a range 1 to 15 operations. This shows worse average case performance, but similar worst case performance. (The graph is not given here to conserve space).

In chapter 5 we showed that for a relocation constant, $k$, set to four, the memory overhead is $2\frac{1}{2}$ times the active heap. This is a massive memory overhead as compared to the limited width scheme proposed for reference counting in this work (c.f. figures 9.1 and 9.16).
9.5 Conclusions.

The first part of the chapter briefly described the simulator. Two primary modes of operation were identified, time-step and event-driven. Time-step mode allows the fundamental operation of the collector to be investigated, without undue interruption from the mutator. In event-driven mode, a fuller simulation of the MAIN module (see chapters 6 and 7) is made. This simulates a parallel mode of operation, between the collector and mutator.

In general, the results bear out the theoretical predictions made in earlier chapters. Where deviation has occurred, further analysis of the dynamic behaviour of the collector has yielded explanations.

The operation of the graph mutator was also explored. This showed that in general, the collector activity would impinge upon the mutator, causing it to halt occasionally. The latency problem observed in this simple simulation would probably be much less marked if a practical implementation of a mutator were employed, since more advanced implementation techniques require less frequent services from the collector [HAR-88].

For event-driven mode, figures for the collector latency are given. These again compare favourably with theoretical predictions.

The comparison of collector latency, for the addition of a new application node to the graph, shows that Baker's algorithm, has a worse observed worse case performance than the reference counting collector. However, the copying collector displays a lot of activity at very short latencies. This affect will be enhanced if large heaps are used and so garbage collection is called infrequently. As a result of this observation the use of copying collector may be more appropriate in a hybrid real time system, where the early satisfaction of a deadline gives enhanced value to the system. In a fast hard real time system, this consideration will not be as important as the much worse observed behaviour when garbage collection is active.

When Baker's algorithm was observed deleting a pointer, then the system is faster on average than the reference counting collector. However, Baker's algorithm displays a worse case performance, similar to the worst case latency observed for the reference counting collector.

Baker's algorithm shows a massive memory overhead, of the order of 1.5, as compared to our system (typically 0.6). This may not be a problem in virtual memory systems. For faster response times \( k \) must be decreased, so increasing the overhead ratio towards its worst case value of 2.

We have not explored generational collection [LIE-83] or varying \( k \) on the fly [BAK-78]. These techniques may improve the results for real time copying collection.

The results obtained show that the design of the collector is viable, in terms of its performance and interaction with the mutator.
Chapter 10. Conclusions.

10.1 Summary of the Thesis.

Chapters 1 to 3 presented a brief general introduction to parallel computing, functional programming, graph reduction and garbage collection. Various schemes for the collection of garbage were discussed. Reference counting was identified as one of the most promising techniques for parallel distributed systems, especially for ones that have to operate with real time constraints. In particular, the method developed by Hughes [HUG-82b] was introduced as a viable scheme for the reclamation of redundant cyclic structures using a modified reference counting technique. Some of the problems associated with standard reference counting techniques were introduced.

Chapter 4 addressed one of the most pressing problems of reference counting, namely the large memory overhead required to support the scheme. The problem is compounded by use of separate group reference counts to keep track of strongly connected components, within the reduction graph. A novel method of using limited width counts, within a block of memory reserved for the use of the collector, was discussed. In this chapter, the potential memory savings for the new technique were examined. Under an estimate of "average" operating conditions, a saving of at least 50 per cent in memory overhead was achieved. The worst case analysis showed that the memory overhead for the new techniques could actually be greater than with standard reference counting collection. Such operating conditions, however, have never been encountered in practice in this research, or in a number of other studies of the behaviour of functional languages [SAL-85, HAR-88].

In chapter 5, a general introduction to real time systems was given, and then a discussion of the questions raised by garbage collection in such systems. The inadequacies of modified mark-scan or copying collectors were highlighted and the advantages of reference counting for real time collection were presented.

Chapter 6 concentrated on the issues raised by using reference counting for real time systems. A method, first proposed by Hughes [HUG-82b], is used as a basis to identify and collect cycles within the graph. A novel incremental technique was developed to both scan cycles and collect redundant structures. This is designed to work as a parallel collector, with the graph mutator running with the minimum of interference from the collector. A high level architecture for the new collector is presented and also an idealised parallel interaction model, for the collector and mutator, is given.

The next chapter presents estimates for the worse case temporal behaviour of the algorithms. This analysis is based upon estimations of the critical path abstract execution times for each of the major services provided by the collector.

In chapter 8, a formal proof of the central incremental collector is given. The MALPAS theorem proving tool is used in this work.

Chapter 9 presents a discussion of the software simulation of the collector, including a brief introduction to the functional language, EASE, that is implemented by the simulator. There follows selected results from the simulator and then a discussion of these. Particular
attention is paid to comparing the results obtained with the theoretical predictions made in earlier chapters. The chapter also gives a selection of results obtained for an implementation of Baker's algorithm. These are compared to the results for the incremental reference counting collector.

10.2 Appraisal of the Research.

The aim of this project was to produce a garbage collection technique that would be suitable for the implementation of functional languages, using graph reduction, in real time systems. Because the action of reference counting is intrinsically distributed in time and space, it was chosen as the basis for the research.

The memory overhead problem, introduced by reference counting, was tackled by a method using limited-width counts. This does not rely on occasional mark-scan collection, to clear up cells that have entered an overflow condition, as with hybrid collection. This new scheme has the distinct advantage that mark-scan (or perhaps copying) collection, with all their inherent problems within real time architectures, does not have to be intermittently called. The new method stores any counts that cannot be accommodated in the memory of the collector, in the main heap space. This is expensive, both in terms of collector-heap communication bandwidth, and the CPU resources required to recover information by searching in specified areas of the heap. Any practical system, based upon this method, would have to appraise very carefully the widths of the fields used to store count data within the collector. Spillage of data into the heap should preferably be viewed as a last resort and hence should only occur very infrequently. This ensures overall collector performance is not unduly impaired, by having to perform an excessive number of searches in the heap.

Real time performance was gained by using various incremental techniques. In particular, the collection of cycles in such a fashion, complicates the original elegantly simple idea of tracking the behaviour of strongly connected cycles within the reduction graph. The amount of work that has to be performed by the collector is increased, in order to handle the possibility of the mutator "interfering" with parts of the graph that have been partly scanned by a Tarjan search.

The memory necessary to keep track of various stacks, which provide the background tasks for the collector, increases the memory overhead of the scheme. It is difficult to assess how great this memory usage will be, due to the complicated nature of collector and mutator interaction. Notwithstanding these objections to the extended reference counting technique, it does allow collection to be carried out in real time (the collector can be guaranteed to respond to a mutator request in a known, maximum time) and in parallel with mutator activity.

Since the collection system was designed with the ultimate realisation of a hardware implementation in view, the objections of increased collector memory and the amount of computational power required, can be mitigated by considering the relatively low cost of both memory and powerful processors. Both these commodities are currently decreasing in price, (at the time of writing about £25 per Mbyte and £3 per MIP are available).

The proving work carried out with the aid of MALPAS, has shown that the design of the collector is sound and so may form the viable basis for an industrial scale implementation of the collector.

The software simulation of the collector was implemented for two main reasons. The first was to test the viability of the ideas presented in the thesis. This was done by running
standard benchmark tests on the system, and appraising whether or not the collector was functioning correctly. A number of safeguards were placed in the software, in order to trap malfunctions of the collector system. To date, for all the tests run, the collector has always performed correctly. In this context, correctness of a garbage collector is defined as a system that identifies and appends to the free list cells that have become redundant (i.e., are playing no further rôle in useful computation), and does not perform this operation on any active cell. The collector must also respond to the requests of the mutator in the required manner, such as overwriting cells correctly, or deleting a pointer etc. All these operations have to be performed in a maximum bound time (to meet a real time systems *liveliness* constraints). This was simulated in the system by giving each small component of the collector an equivalent number of collector and heap operations and verifying that no operation took longer than the predicted upper bound to run.

The second reason for the simulation was to extract data from simulation runs, in order to assess the viability of the techniques that have been proposed. The worst case analysis presented in the thesis, for both memory overhead, and response time, seem never to be met in practice.

It was found by simulation, that some cessation of mutator activity will inevitably result from a scheme which employs a parallel mutator and collector. This represents a potential "bottleneck" for the mutator. However, this bottleneck appears to be limited in extent, and may prove to be even less constricting when these techniques are applied to a more sophisticated mutator (perhaps based upon supercombinator reduction). The process of halting the mutator has the added advantage, in this method, that there is a known upper fixed bound set on the time for which graph reduction is forced to halt.

The simulation of Baker's algorithm, which is currently a popular choice for real time functional systems, when a separate hardware collector is used, reveals that the temporal performance of this technique is in many ways comparable to our technique. It does, however, display a massive memory overhead as compared to our collector and the objections of copying collection in a distributed real time system still apply.

The research has culminated in a novel approach to real time garbage collection in graph reduction systems. It has made two major contributions in the area of alleviating the massive memory overhead normally associated with reference counting and also providing a novel scheme to incrementally collect all redundant structures (including cycles) in real time.

10.3 Future Work.

One of the drawbacks with the present simulator is use of the simple EASE interpreter. The language is very restrictive and programming large examples in EASE proves to be a difficult task. Preliminary work has been carried out in investigating the possibility of implementing the collector on a practical functional programming system. A promising approach appears to be offered by the use of FLIC [PEY-87b, PAR-91], which is intermediate code generated by use of HASKELL [HUD-90]. A separate FLIC reducer has been developed [PAR-91], the storage management interface of which matches the one given above very closely. In this way, large applications, programmed in HASKELL, would be able to run with the collector.

The analysis and results given above, show how costly is the retrieval of overflow data, from the main heap. A possible avenue of research would be to explore an asymmetric division of the heap, for the purposes of storing overflow data. This system would employ
two free lists. One would deal with cells in (say) the top tenth of the heap, whilst the other incorporated cells from elsewhere. The allocation of cells, from the free lists, would be biased in accordance with the use of the cell. Overflow data would normally be incorporated in the top of the heap. This would enable much smaller search spaces to be employed. Normal cell allocation would be biased towards the lower heap partition. When memory is nearly exhausted, then cells could be drawn for any purpose from either free list. Overflow count locations would have to have a mark bit distinguishing which free list was used for the overflow allocation.

Asymmetric heap division will obviously improve average performance, whilst slightly improving repose times in extreme operating conditions (when overflow data is held in the larger partition). This would involve the extra expense of a mark bit per cell and the small collector overhead of maintaining two free lists.

The results from the simulation show how effective a small cache is in reducing the number of searches in the main heap. The present cache controlling algorithm is very simple. Work needs to be carried out in order to appraise more sophisticated caching algorithms. Account must be taken of the amount of extra collector processing overhead, necessary to implement such cache controllers.

Some discussion has been given of the important connexion between functional languages and parallel architectures. To date, the research has concentrated on simulation of a uniprocessor machine. Little mention has been made of how the collection scheme would be extended to a multiprocessor environment. In principal little change is envisaged in the proposed architecture. Each collector would be associated with a mutator and heap (in a loosely coupled system). The interaction between the collector and any mutator would not need to be significantly different to that outlined in chapters 6 and 7, since the system of signalling to the mutator that it is safe to proceed with graph reduction, could be used to control the behaviour of remote processors. It would, however, be necessary for collectors to communicate between themselves, to indicate which parts of the graph are being scanned by Tarjan searches etc. This would ensure that nodes are traced properly in their S.C.C. structures, which may spread across more than one heap. Simulation of such a parallel regime would be useful in optimising a final design. This would also facilitate work to minimising this class of interaction.

Once the basic techniques have been developed to extended the system for a parallel architecture, other improvements could be investigated. The advantages of weighted reference counting [THO-81, BEV-85], or external node pointers [HUG-85], would enhance the system in a distributed machine. The possibility of employing recursively structured control mechanisms [WIS-88], between each collector, could also prove to be useful in a multi-processor environment. After initial analysis and appraisal of these techniques, with reference to the work presented in this thesis, the simulator could be extended to incorporate some of these ideas.

A possibly fruitful area of research would be to investigate compile-time garbage collection techniques, which could be used to reduce the loading on the run-time reference counting collection. Hughes [HUG-91] suggests that both compile-time garbage collection and a technique of destructive allocation, could reduce the load on the run-time garbage collector. This would be achieved by removing the need to maintain reference counts in certain parts of the machine's memory. Hughes [HUG-91] points out that no compile-time garbage collector is likely to replace a run-time collector, but employing compile time optimisation may be worthwhile.
Section 10.3 Future Work

At the moment, the main scheduler module only stacks one mutator interrupt. It is conceivable that stacking requests deeper than this may improve average collector latency. The effects on the safety of the collector, however, are unclear. This technique would have to be investigated further, to quantify any benefits and ensure that the safety of the collector is not compromised.

The current simulation work has mainly concentrated on the implementation of the collector, in order to demonstrate its functionality and viability. Only preliminary work has been carried out to optimise its performance. From the analysis of the behaviour of the various functions within the collector, further work in minimising the length of the critical (and average) path behaviour of these procedures, may prove to be a valuable exercise in improving the collector's overall performance.

The final stage in the implementation of such an extended collector architecture, would be to design and build a hardware implementation and then run this with a practical functional machine. This would represent the final culmination to the research, and provide a possible commercial product for use with functionally programmed machines.
Appendix A. Some Functional Programs.

```lisp
(program (fact 8))
((fact n) (if (= n 0) 1 (* n (fact (- n 1)))))
```

Figure A.1 Factorial (8).

```lisp
(program (fibseries 8))
((fibseries n) (if (= n 0) 0 (cons (fib n) (fibseries (- n 1)))))
((fib n) (if (<= n 0) 0 (if (= n 1) 1 (+ (fib (- n 2)) (fib (- n 1)))))
```

Figure A.2 Fibonacci series (8).

```lisp
(program primes)
(primes (sieve (from 2)))
((sieve s) (if (= (head s) 31) 31 (cons (head s) (sieve (filter (relprime (head s)) (tail s)))))
((relprime p n) (if (= p n) false (if (<= p n) (relprime p (- n p)) true)))
((filter p l) (if (p (head l)) (cons (head l) (filter p (tail l))) (filter p (tail l)))))
((from n) (cons n (from (+ 1 n))))
(false 0)
(true 1)
```

Figure A.3 Sieve of Eratosthenes (to 31).
Appendix A. Some Functional Programs

\{(program (ackermann 3 2))

\{(ackermann n m) \{if (= n 0) (+ m l)
  (if (= m 0) (ackermann (- n 1) l)
  (ackermann (- n 1) (ackermann n (- m 1))))\}\}

\[\text{Figure A.4} \text{ Ackermann function (3,2).}\]

\{(program (sort list))

\{(sort l) \{if (= l nil) nil \{insert \{head l\} \{sort \{tail l\}\}\}\}
  \{(insert x l) \{if (= l nil) \{cons x nil\}
    (if (<= x \{\{- (head l)\}\}) \{cons x l\} \{cons \{head l\} \{insert x \{\{tail l\}\}\}\}\}\}\}

\{list \{cons 99 \{cons 23 \{cons 1 \{cons 102 \{cons 4
  \{cons 3 \{cons 9 \{cons 1 \{cons 7
  \{cons 2 \{cons 5 nil\}\}\}\}\}\}\}\}\}\}\}\}\}

\[\text{Figure A.5} \text{ Sort.}\]

\{(program (cycle 1 2 4))

\{(cycle p q r) \{if (= p l) \{cons q \{cycle 2 99 4\}\}
  (if (= p 2) \{cons \{cycle 3 99 4\} \{cycle 3 99 4\}\} q))\}

\[\text{Figure A.6} \text{ Cycle2.}\]
Appendix A. Some Functional Programs

{ 
(program (queens nil))

((queens board) (if (ok board) (if (complete board) board 
(queens (addqueen board))) (queens (new board))))

((complete board) (if (= (len board) 8) TRUE FALSE))

((addqueen board) (cons 1 board))

((ok l) (if (= l nil) TRUE (if (not (mem (tail l) (head l)))
(not (hit (- (head l) 1) (+ (head l) 1) (tail l))) FALSE )))

((addqueen board) (cons 1 board))

((hit ql qr l) (if (= l nil) FALSE
(if (= ql (head l)) TRUE
(if (= qr (head l)) TRUE
(hit (- ql 1) (+ qr 1) (tail l))))))

((new l) (if (= l nil) 1
(if (< (head l) 7) (cons (+ (head l) 1) 
(tail l)) (new (tail l)))))

((len l) (if (= l nil) 0 (+ 1 (len (tail l))))))

((mem l a) (if (= l nil) FALSE (if (= a (head l)) TRUE mem (tail l) a)))

((not x) (if (= x TRUE) FALSE (if (= x FALSE) TRUE ERROR)))

(TRUE 1)

(FALSE 0)

(ERROR error)
}

Figure A.7 Eight queens.
tag == num  || just a type synonym to make things clearer

node * ::= Node tag * [node *]

|| an example graph (this is cyclic)

graph1 = hd list
where
  list = [(Node 1001 1 [elem 3 list, elem 2 list]),
          (Node 1002 2 [elem 4 list, elem 1 list]),
          (Node 1003 3 [elem 4 list]),
          (Node 1004 4 [elem 3 list])]

|| another example (again, cyclic)

graph2 = hd list
where
  list = [(Node 101 11 [elem 3 list, elem 2 list]),
          (Node 102 12 [elem 4 list, elem 1 list]),
          (Node 103 13 [elem 4 list]),
          (Node 104 14 [])]

|| addgraph takes a graph of numbers and adds an increment to each
|| node in the graph (it constructs another graph with the new values
|| and hence generates new cycles dynamically).

addgraph : : num -> node num -> node num
addgraph n nd = update (+ n) nd

|| Visit every reachable node once only and return the list of
|| values associated with the nodes, given the root node of a graph.

visit :: node * -> [*]  || pass in the initial state and ..
visit = snd.(visit’ [])) || .. strip off the final state

visit’ :: [tag] -> node * -> ([tag],[*])

visit’ tags (Node t v children)
  = (tags, []), if member tags t  || node already seen
  = (new_tags, v:new_nodes), otherwise  || unseen node

where
  (new_tags, visit_list) = iterative_map visit’ (t:tags)
  new_nodes = concat visit_list


graph_state * == ([tag],[node *])

|| Copy a possibly cyclic graph, updating the value parts of the
|| respective nodes according to the supplied value update function.

update :: (*->*) -> node * -> node *
update vfn root = snd (update’ vfn ([],[]) root)

update’ :: (*->*) -> graph_state * -> node * -> (graph_state *, node *)

update’ vfn (tags, nodes) (Node tag val children)
Appendix A. Some Functional Programs

= ((tags, nodes), noderef), if member tags tag
= (newstate, newnode), otherwise

where

    noderef = nodes!(listpos tags tag) | | ref newly built node
    newnode = Node tag (vfn val) newchildren | | build new node
    (newstate, newchildren)
    = iterative_map (update' vfn) state' children
    where state' = (tag:tags, newnode: nodes)

|||==-------------------------------||
|| auxilliary functions .... ||
|||]==-------------------------------||

|| elem is the list index function, taking arguments from 1 upwards
|| (i.e. `elem 1 list' returns the 1st item in `list').

elem x = (!{(x-1)})

|| listpos just finds the position of a value within a list

listpos :: [*] -> * -> num
listpos list item = #(takeWhile ((=) item) list)

|| iterative_map takes:
|| (i) a function of the form:
||    (state,value)->(newstate,newvalue)
|| (ii) an initial state
|| (iii) a list of values
|| and maps the function along the list, passing each new state to the
|| state input of the next successive application of the function. It
|| returns the new list of values paired with the final state.

iterative_map :: (*->(*->(*,[*]))) -> * -> [*] -> (*, [*])

iterative_map f state [] || end of list reached, return the ..
    = (state, []) || .. current state and the empty list

iterative_map f state (val:vals)
    = (final_state, val':vals')
    where (state', val') = f state val
    (final_state, vals') = iterative_map f state' vals

Figure A.8 Dynamic cycles.
Appendix B. The MALPAS Model of the Collector.

B.1 IL Source.

```il
TITLE reference_counting_garbage_collector;

_INCLUDE "diskthree:tesa.command.|prelude"
_INCLUDE "diskthree:tesa.command.|list.prelude"
_INCLUDE "diskfour:prelight.malpas|array|prelude"

TYPE EXT tree;
PREFIX LEPT(%l-tree)!  %1-tree;
PREFIX RIGHT(%1 tree): %1-tree;
PREFIX ROOT{%1-tree):  % l ;
PREFIX T_EMPTY(%1-tree)!  boolean;
FUNCTION min(Integer, integer): integer;
REPLACE ( X ,  y :  integer)
  min(x, y )
  BY if(x <  y ,  X ,  y);
FUNCTION tree(%1-tree, %1, %1-tree):  %1-tree;
REPLACE (left, right: %1-tree; root: % 1 )
  T EMPTY tree(left, root, right)
  BY false;
REPLACE ( X :  %1-tree)
  TEMPTY X
  BY true IF X EQ empty:%1 tree;
REPLACE (left, right: %1-tree; root: % 1 )
  LEFT tree(left, root, right)
  BY left;
REPLACE (left, right: %1-tree; root: % 1 )
  ROOT tree(left, root, right)
  BY root;
REPLACE (left, right: %1-tree; root: % 1 )
  RIGHT tree(left, root, right)
  BY right;

TYPE vertex = RECORD id ,  lowlink, number :  integer;
           visited, stacked: boolean
ENDRECORD;
TYPE SC CO =  (a ,  b ,  C);
TYPE scan_condition =  (xx, nn, pp, ff, x, f, e);
TYPE node details •  RECORD l_count, group id :  integer;
           garbage: boolean; scan: scco;
           cyclic free: boolean
ENDRECORD;
TYPE ngcl •  (one, two, three, four);
CONST upper_heap_limit: integer;
CONST lower_heap_limit: integer;
FUNCTION length ( % 1  list): integer;
REPLACE ( 1 :  %1-list)
  length ( 1 )
  BY 0  IF EMPTY 1 ,
  BY 1  .  length (REST 1 ) ,
FUNCTION ngcfl (vertex-list, node details-array):  boolean;
REPLACE ( 1 :  vertex -  list;  n_a: node_details-array)
  ngcfl ( 1 ,  n_a)
  BY true IF EMPTY 1 ,
  BY false IF GARBAGE (nallD FIRST 1 ) ,
  BY ngcfl (REST 1 ,  n_a)7
FUNCTION nondeterm (ngcl-list): ngcl;
REPLACE ( 1 :  ngcl-list)
  nondeterm ( 1 )
  BY FIRST 1 ;
FUNCTION ra_no (ngcl list, vertex-tree, integer, node_details -  array):  vertex;
REPLACE ( 1 :  ngcl-list; gr: vertex-tree; i:  integer; na:  node_details array)
  ra_no ( 1 ,  gr, i ,  n a )
  BY ROOT gr IP T_EMPTY LEFT gr AND T EMPTY Right gr
  AND NOT GARBAGE (nallD (ROOT gr)) ,
  BY ROOT gr IP i  =  0 AND NOT GARBAGE (nallD (ROOT gr)),
  BY ra_no (REST 1 ,  LEFT gr, i-1, n a) IF FIRST 1  -  one AND
  NOT T_EMPTY RIGHT gr AND NOT GARBAGE (nallD (ROOT LEFT gr)) ,
  BY ra_no (REST 1 ,  RIGHT gr, i-1, n a) IF FIRST 1  -  one AND
  NOT T_EMPTY LEFT gr AND NOT GARBAGE (nallD (ROOT RIGHT gr)) ,
  BY ra_no (REST 1 ,  LEFT gr, i-1, n a) IF FIRST 1  -  two AND
  NOT T_EMPTY RIGHT gr AND NOT GARBAGE (nallD (ROOT LEFT gr)) ,
  BY ra_no (REST 1 ,  RIGHT gr, i-1, n a) IF FIRST 1  -  three AND
  NOT T_EMPTY LEFT gr AND NOT GARBAGE (nallD (ROOT RIGHT gr)) ,
  BY ra_no (REST 1 ,  RIGHT gr, i-1, n a) IF FIRST 1  -  three AND
  NOT T_EMPTY RIGHT gr AND NOT GARBAGE (nallD (ROOT LEFT gr)) ,
  BY ra_no (REST 1 ,  LEFT gr, i-1, n a) IF FIRST 1  -  root;
FUNCTION exists ( % 1  list, %1):  boolean;
REPLACE (1: %1-list; X: %1)
  exists (1,  X )
  BY false IF EMPTY 1 ,
  BY true IF FIRST 1  EQ X ,
  BY exists (REST 1 ,  X );
FUNCTION element test (vertex-list, integer, node_details array):  boolean;
REPLACE ( 1 :  vertex-list; X :  integer; na: node_details array)
  element_test ( 1 ,  x ,  na)
  BY true IF EMPTY 1 ,
  BY false IF NUMBER FIRST 1  =  x ,
  BY element_test (REST 1 ,  x, na);;
FUNCTION ret graph (integer, vertex-tree):  vertex-tree;
REPLACE (node: integer; g: vertex-tree)
  ret_graph (node, g)
  BY g IF TEMPTY LEFT g AND T EMPTY RIGHT g
  AND NOT GARBAGE (nallD (root g)) ,
  BY root g IF i  =  0 AND NOT GARBAGE (nallD (root g)),
  BY ra_no (REST 1 ,  LEFT g, i-1, na) IF FIRST 1  -  one AND
  NOT T_EMPTY LEFT g AND NOT GARBAGE (nallD (ROOT LEFT g)) ,
  BY ra_no (REST 1 ,  RIGHT g, i-1, na) IF FIRST 1  -  one AND
  NOT T_EMPTY RIGHT g AND NOT GARBAGE (nallD (ROOT RIGHT g)) ,
  BY ra_no (REST 1 ,  LEFT g, i-1, na) IF FIRST 1  -  two AND
  NOT T_EMPTY RIGHT g AND NOT GARBAGE (nallD (ROOT LEFT g)) ,
  BY ra_no (REST 1 ,  RIGHT g, i-1, na) IF FIRST 1  -  three AND
  NOT T_EMPTY LEFT g AND NOT GARBAGE (nallD (ROOT RIGHT g)) ,
  BY ra_no (REST 1 ,  LEFT g, i-1, na) IF FIRST 1  -  three AND
  NOT T_EMPTY RIGHT g AND NOT GARBAGE (nallD (ROOT LEFT g)) ,
  BY ra_no (REST 1 ,  RIGHT g, i-1, na) IF FIRST 1  -  root;
FUNCTION exists ( % 1  list, %1):  boolean;
REPLACE (1: %1-list; x: %1)
  exists (1,  x )
  BY false IF EMPTY 1 ,
  BY true IF FIRST 1  EQ x ,
  BY exists (REST 1 ,  x );
FUNCTION element test (vertex-list, integer, node_details array):  boolean;
REPLACE ( 1 :  vertex-list; x: %1-integer; na: node_details-array)
  element_test ( 1 ,  x ,  na)
  BY false IF EMPTY 1 ,
  BY true IF NUMBER FIRST 1  =  x ,
  BY element_test (REST 1 ,  x, na);;
FUNCTION ret graph (integer, vertex-tree):  vertex-tree;
REPLACE (node: integer; g: vertex-tree)
  ret_graph (node, g)
  BY g IF TEMPTY LEFT g AND T EMPTY RIGHT g ,
  BY root g IF i  =  0 AND NOT GARBAGE (nallD (root g)),
  BY ra_no (REST 1 ,  LEFT g, i-1, na) IF FIRST 1  =  one AND
  NOT T_EMPTY LEFT g AND NOT GARBAGE (nallD (ROOT LEFT g)) ,
  BY ra_no (REST 1 ,  RIGHT g, i-1, na) IF FIRST 1  =  one AND
  NOT T_EMPTY RIGHT g AND NOT GARBAGE (nallD (ROOT RIGHT g)) ,
  BY ra_no (REST 1 ,  LEFT g, i-1, na) IF FIRST 1  =  two AND
  NOT T_EMPTY RIGHT g AND NOT GARBAGE (nallD (ROOT LEFT g)) ,
  BY ra_no (REST 1 ,  RIGHT g, i-1, na) IF FIRST 1  =  three AND
  NOT T_EMPTY LEFT g AND NOT GARBAGE (nallD (ROOT RIGHT g)) ,
  BY ra_no (REST 1 ,  LEFT g, i-1, na) IF FIRST 1  =  three AND
  NOT T_EMPTY RIGHT g AND NOT GARBAGE (nallD (ROOT LEFT g)) ,
  BY ra_no (REST 1 ,  RIGHT g, i-1, na) IF FIRST 1  =  root;
```
Appendix B. The MALPAS Model of the Collector

FUNCTION deleteref_ga (node_details-array, vertex, vertex, integer-array, boolean, integer, integer): integer-array;
REPLACE (n_a: node_details-array; to: vertex; from: vertex; g_a: integer-array; ef: boolean; et: integer; eg: integer; s_g: integer)

FUNCTION deleteref_ga_ass (node_details-array, vertex, vertex, integer-array, boolean, integer, integer): integer;
REPLACE (n_a: node_details-array; to: vertex; from: vertex; g_a: integer-array; ef: boolean; et: integer; eg: integer; s_g: integer)

FUNCTION addref_ext_ga (node_details-array, vertex, vertex, integer-array, boolean, integer, integer): integer-array;
REPLACE (n_a: node_details-array; to: vertex; from: vertex; g_a: integer-array; ef: boolean; et: integer; eg: integer; s_g: integer)

FUNCTION addref_ext_ga_ass (node_details-array, vertex, vertex, integer-array, boolean, integer, integer): integer-array;
REPLACE (n_a: node_details-array; to: vertex; from: vertex; g_a: integer-array; ef: boolean; et: integer; eg: integer; s_g: integer)

FUNCTION addref_ext_ps (node_details-array, vertex, vertex, boolean, integer, integer, integer): integer-list;
REPLACE (n_a: node_details-array; to: vertex; from: vertex; ef: boolean; et: integer; eg: integer; s_g: integer; g_a: integer-array; s_q: integer-list)
\[ s_c (t_0, n_a, ef, et, eg, s_g) \times \]
Appendix B. The MALPAS Model of the Collector

FUNCTION newapp_ga(node_details-array, vertex, vertex, vertex-tree, integer-array, integer-array, integer, integer, integer): integer;
REPLACE (na: node_details-array; new: vertex; par: vertex; gr: vertex-tree;
newapp_ga (na, new, par, gr, ef, et, eg, sg)
REPLACE (na: node_details-array; new: vertex; par: vertex; gr: vertex-tree;
newapp_ga_ass (na, new, par, gr, ef, et, eg, sg)

FUNCTION newapp_ga_same(node_details-array, vertex, vertex, vertex-tree, integer-array, integer-array, integer, integer, integer): integer;
REPLACE (na: node_details-array; new: vertex; par: vertex; gr: vertex-tree;
newapp_ga_same (na, new, par, gr, ef, et, eg, sg)

FUNCTION newapp_left(node_details-array, vertex, vertex, vertex-tree, integer-array, integer-array, integer, integer, integer): integer;
REPLACE (na: node_details-array; new: vertex; par: vertex; gr: vertex-tree;
newapp_left (na, new, par, gr, ef, et, eg, sg)
Section B.1 IL Source

NOT VISITED ROOT RIGHT gr,
BY tree(visit(LEFT gr, dfsn). x, gp, spl) gr, n_a)
ROOT up, merge (gr, dfsn), LEFT gr, RIGHT gr,
RIGHT gr) IF NOT T EMPTY RIGHT gr AND NOT T EMPTY LEFT gr AND
NOT VISITED ROOT LEFT gr AND VISITED ROOT RIGHT gr AND
STACKED ROOT RIGHT gr AND dfsn > NUMBER ROOT RIGHT gr,
BY tree(visit(LEFT gr, dfsn), x, gp, spl) gr, n_a)
ROOT up, merge (gr, dfsn), LEFT gr, RIGHT gr,
RIGHT gr) IF NOT T EMPTY RIGHT gr AND NOT T EMPTY LEFT gr AND
NOT VISITED ROOT LEFT gr AND VISITED ROOT RIGHT gr AND
STACKED ROOT RIGHT gr AND dfsn > NUMBER ROOT RIGHT gr,
BY tree(visit(LEFT gr, dfsn), x, gp, spl) gr, n_a)
ROOT up, merge (gr, dfsn), LEFT gr, RIGHT gr,
RIGHT gr) IF NOT T EMPTY RIGHT gr AND NOT T EMPTY LEFT gr AND
STACKED ROOT RIGHT gr AND STACKED ROOT LEFT gr AND
dfs = NUMBER ROOT RIGHT gr AND dfsn > NUMBER ROOT LEFT gr,
BY tree(LEFT gr, ROOT up, merge (gr, dfsn), LEFT gr, RIGHT gr),
RIGHT gr) IF NOT T EMPTY LEFT gr AND T EMPTY RIGHT gr AND
NOT VISITED ROOT LEFT gr AND NOT VISITED ROOT RIGHT gr,
BY tree(visit(LEFT gr, dfsn), x, gp, spl) gr, n_a)
ROOT up, merge (gr, dfsn), LEFT gr, RIGHT gr,
RIGHT gr) IF NOT T EMPTY RIGHT gr AND T EMPTY LEFT gr AND
NOT VISITED ROOT LEFT gr AND VISITED ROOT RIGHT gr AND
STACKED ROOT RIGHT gr AND STACKED ROOT LEFT gr AND
dfsn > NUMBER ROOT RIGHT gr AND dfsn > NUMBER ROOT LEFT gr,
BY tree(LEFT gr, ROOT up, merge (gr, dfsn), LEFT gr, RIGHT gr),
RIGHT gr) IF NOT T EMPTY LEFT gr AND T EMPTY RIGHT gr AND
NOT VISITED ROOT LEFT gr AND NOT VISITED ROOT RIGHT gr,
BY tree(LEFT gr, ROOT up, merge (gr, dfsn), LEFT gr, RIGHT gr),
RIGHT gr) IF NOT T EMPTY LEFT gr AND NOT T EMPTY RIGHT gr AND
NOT T EMPTY LEFT gr AND NOT T EMPTY RIGHT gr AND
NOT VISITED ROOT LEFT gr AND NOT VISITED ROOT RIGHT gr AND
STACKED ROOT LEFT gr AND STACKED ROOT RIGHT gr AND
dfs = NUMBER ROOT RIGHT gr AND dfsn = NUMBER ROOT LEFT gr,
BY tree(LEFT gr, ROOT up, merge (gr, dfsn), LEFT gr, RIGHT gr),
RIGHT gr) IF T EMPTY LEFT gr AND T EMPTY RIGHT gr,
BY tree(visit(LEFT gr, dfsn), x, gp, spl) gr, n_a)
ROOT up, merge (gr, dfsn), LEFT gr, RIGHT gr,
RIGHT gr) IF NOT T EMPTY LEFT gr AND T EMPTY RIGHT gr AND
NOT VISITED ROOT LEFT gr AND VISITED ROOT RIGHT gr AND
STACKED ROOT RIGHT gr AND STACKED ROOT LEFT gr AND
dfs = NUMBER ROOT RIGHT gr AND dfsn > NUMBER ROOT LEFT gr,
BY tree(LEFT gr, ROOT up, merge (gr, dfsn), LEFT gr, RIGHT gr),
RIGHT gr) IF NOT T EMPTY LEFT gr AND T EMPTY RIGHT gr AND
NOT VISITED ROOT LEFT gr AND VISITED ROOT RIGHT gr AND
STACKED ROOT RIGHT gr AND STACKED ROOT LEFT gr AND
dfs = NUMBER ROOT RIGHT gr AND dfsn = NUMBER ROOT LEFT gr,
BY tree(LEFT gr, ROOT up, merge (gr, dfsn), LEFT gr, RIGHT gr),
RIGHT gr) IF T EMPTY LEFT gr AND T EMPTY RIGHT gr,
Appendix B. The MALPAS Model of the Collector

IN group_handle: vertex-list-array
IN free_list: vertex-list
IN emit_flag: boolean
IN emit_group_id: integer
IN emit_interior_count: integer
IN spl_grp: integer
IN g_array: integer-array
INOUT node_array: node_details-array
IN grp_id: integer)

DERIVES graph AS
visit (graph, i, 0, g_array, spl_grp, node_array),
node_array FROM node_array & graph,
emit_thresh AS i, 0 < 1
PRE NOT EMPTY hidden AND NOT T EMPTY graph AND
grp_id > 0 AND NOT emit_flag AND
SCAN (node_array;ID ROOT graph) • 4 AND
not_garbage (ID ROOT graph, graph, node_array) AND
NOT EMPTY free_list
POST graph EQ visit
('graph, ' i, 0, g_array, 'spl_grp, node_array)
AND (node_array;ID ROOT graph) SCAN b
node_array, ID ROOT graph, (node_array;ID ROOT graph) SCAN c) AND
NOT EMPTY hidden AND
not_garbage (ID ROOT graph, graph, node_array);

PROCSPEC collect_group (INOUT cyc_stk): vertex-list
IMPLICIT (IN nodearray: node_details array
IN g array: integer-array
INOUT free_list: vertex-list)
DERIVES cyc_stk PROM cyc_stk,
free_list PROM free_list
PRE NOT GARBAGE(node_arrayI (ID FIRST cyc_stlr))
and
NOT EMPTY cyc_stlc AND
g_array I  GROUP_ID( nodearray I  ID FIRST cyc_ s t l t )  < -  0
POST EMPTY cyc_stk;

PROCSPEC stack_group_body (IN grp_id: integer
IN start_node: vertex
IN gr_str: vertex tree
INOUT cyc_stk: vertex-list)
IMPLICIT (IN node_array: node_details-array
INOUT group handle :  vertex -list array
IN emitstop: boolean
IN emit_interior count :  integer
IN split_queue: integer-list
INOUT cyclicfreelist:  vertex-list
IN g_array: integer-array
IN stacic vertex-list
IN spl_grp: integer
IN free_list: vertex-list)
DERIVES cyc_stk AS c_stk (node_array, startnode, grpid, gratr, garray,
cyc_stk)
PRE NOT GARBAGE(node_arrayl (ID start_node))  AND not_garbage (ID start_node, graph, node_array)
POST NOT GARBAGE(  node_arraylID startnode) AND cyc_stk EQ
c_stk (  node_array, startnode, grpid, gr_str, garray,
cyc_stk);

PROCSPEC stackgroup (IN grp_id: integer
IN startnode:  vertex
IN gr_str: vertex-tree
INOUT cyc_stk: vertex-list)
IMPLICIT (IN graph :  vertex tree
INOUT nodearray:  node details array
INOUT group_handle: vertex-list-array
IN cyclicfreelist:  vertex-list
INOUT node_array: node_details-array
INOUT group_handle: vertex-list-array
IN g_array: integer-array
IN stacic vertex-list
IN spl_grp: integer
IN free_list: vertex-list
DERIVES cyc_stk AS c_stk (node_array, start_node, grp_id, 'gr_str, 'g_array, 'cyc_stk))
PRE NOT GARBAGE(node_array;ID FIRST cyc_stkr) AND NOT GARBAGE(node_array;ID FIRST cyc_stkr)
POST NOT GARBAGE(node_array;ID FIRST cyc_stkr) AND NOT GARBAGE(node_array;ID FIRST cyc_stkr)

PROCSPEC emit (IN emit_thresh :  integer
INOUT stack :  vertex-list
INOUT g_array: integer-array
IMPLICIT (IN graph :  vertex tree
INOUT hidden: apq-list
IN emit_stop: boolean
INOUT T_i, ser-id: integer
INOUT T_i, h, id: integer
IN split_queue: integer-list
IN gr_str: vertex tree
INOUT node_array: node_details-array
IN group_handle: vertex-list-array
INOUT group_handle: vertex-list-array
INOUT grouphandle: vertex-list
INOUT node_array: node_details-array
IN spl_grp: integer
INOUT grp_id: integer
IN free_list: vertex-list
INOUT emit_interior_count: integer
INOUT emit_group_id: integer
INOUT emit_flag: boolean)
DERIVES stack FROM stack & emit_thres,
grp_id AS grp id + 1,
emit_group_id AS grp id,
emit_interior_count FROM node_array & stack & emit_flag &
Section B.1 IL Source

exit thread & emit group_id & spl grp & g_array FROM g_array & emit_group_id & node_array & stack & exit thread & emit group_id & emit interior_count & t_iec & t_g & g_array FROM g_array & emit group_id & node_array & stack & exit thread & emit group_id & emit interior_count & t_iec & t_g & g_array FROM g_array & emit group_id & node_array & stack & exit thread & emit group_id & emit interior_count & node_array AS emit na (node_array, stack, emit group_id, emit thread)

.emit_flag is derived as an abstract function

PRE NOT EMPTY hidden AND NOT EMPTY stack AND NUMBER FIRST stack = emit thread AND grp_id > 0 AND NOT emit flag AND NOT emit_stop AND emit interior count = int count (node_array, stack, graph, emit thread, emit group_id, spl_grp)

AND NOT EMPTY hidden AND node_array & emit na (node_array, stack, emit group_id, emit thread);
Appendix B. The MALPAS Model of the Collector

```plaintext
■emit_group_id, ' splgrp) AND
emit_interior_count EQ newapp_lec ('node_array, FIRST ' freelist,
par, g array, ' graph,
' emitinteriorcount,
' emit_flag, ' emit_thresh, ' emit_group_id,
' splgrp, ' garray, ' split_queue') AND
split_queue EQ newapp_jps ('node_array, FIRST ' freelist, ' par, graph,
' emit_flag, ' emit_thresh, ' emit_group_id,
' emit_interior_count, ' emit_stop, ' emitthresh,
' emit_group_id, ' splgrp) AND
group_handle EQ newapp_ph ('node_array, FIRST ' freelist,
par, g_array, ' graph,
' emit_flag, ' emit_thresh, ' emit_group_id,
' emit_interior_count, ' emit_stop, ' emitthresh,
' emit_group_id, ' splgrp, ' garray, ' group_handle') AND
(s_c (new_app, node_array, emit_flag, ' emit_thresh,
' emit_group_id, ' splgrp) •  xx OR s_c (new_app, node_array,
' emit_flag, ' emit_thresh, ' emit_group_id, ' splgrp) •  nn OR
s_c (new_app, node_array, emit_flag, ' emit_thresh,
' emit_group_id, ' splgrp) •  ee);

PROCSPEC addref (IN tO:vertex
IN from:vertex
INOUT g_array :  integer-array
INOUT nodearray:  node_details array)
IMPLICIT (IN spl grp: integer
IN emit_interior_count: integer
INOUT emit_group_id :  integer
INOUT split_queue:  boolean)
DERIVES node_array AS addref_na (node_array, to),
g_array AS addref_ga (node_array, to, from, g_array, emitflag,
emitinteriorcount, emit_group_id, spl_grp)
PRE NOT GARBAGE(node_array(I.D to))
POST node_array EQ addref_na (node_array, ' to) AND
split_queue EQ addref_ga ('node_array, ' to, from, ' g_array, emit_flag,
' emit_interior_count, ' emit_stop, ' emit_group_id, ' spl_grp);

PROCSPEC addref_ext (IN to:  vertex
IN from:vertex
INOUT g_array: integer array
INOUT node_array: node_details array)
IMPLICIT (IN splgrp: integer
INOUT emit_interior_count: integer
INOUT emit_stop: boolean
INOUT split_queue: integer-list
INOUT group_handle: vertex-list array
INOUT emit_group_id :  integer
INOUT split_queue:  boolean)
DERIVES node_array AS addref_na (node_array, to),
g_array AS addref_ext_ga (node_array, to, from, g_array, emit_flag,
emit_interior_count, emit_group_id, spl_grp)
emit_stop AS (s_c (from, node_array, emit_flag, emit_group_id,
' emit_stop, ' spl_grp) •  wz) AND
(s_c (from, ' g_array, ' emit_stop, ' emit_group_id,
' spl_grp) •  wz) AND
emit_interior_count AS addref_ext_iec (node_array, ' to, from,
g_array, emit_interior_count, emit_flag, emit_interior_count,
emit_group_id, spl_grp)
splitqueue FROM node_array & from & to & emit_flag & emit_group_id & split_queue & g_array & split_queue
AS addref_sp (node_array, ' to, from, emit_flag, emit_group_id, emit_interior_count, emit_stop, emit_group_id, ' spl_grp & split_queue)
PRE NOT GARBAGE(node_array(I.D to))
POST node_array EQ addref_na (node_array, ' to) AND
g_array EQ addref_ext_ga ('node_array, ' to, from, ' g_array, emit_flag,
' emit_interior_count, ' emit_stop, ' emit_group_id, ' spl_grp) AND
split_queue EQ addref_ext_ps (node_array, ' to, from, emit_flag, emit_group_id, emit_interior_count, emit_stop, emit_group_id, ' spl_grp & split_queue)
PRE NOT GARBAGE(node_array(I.D to))
POST node_array EQ addref_na (node_array, ' to) AND
g_array EQ addref_ext_ga ('node_array, ' to, from, ' g_array, emit_flag,
' emit_interior_count, ' emit_stop, ' emit_group_id, ' spl_grp)
split_queue EQ addref_ext_ps (node_array, ' to, from, ' g_array, split_queue, emit_interior_count, emit_stop, emit_group_id, ' spl_grp & split_queue)
PRE NOT GARBAGE(node_array(I.D to))
POST node_array EQ addref_na (node_array, ' to) AND
g_array EQ addref_ext_ga ('node_array, ' to, from, ' g_array, emit_flag,
' emit_interior_count, ' emit_interior_count, emit_stop, emit_group_id, ' spl_grp)
splitqueue FROM node_array & from & to & emit_flag & emit_group_id & split_queue & g_array & split_queue
AS addref_sp (node_array, ' to, from, ' g_array, emit_flag,
' emit_interior_count, emit_stop, emit_group_id, ' spl_grp & split_queue)
PRE NOT GARBAGE(node_array(I.D to))
POST ...
```

-198-
Appendix B. The MALFAS Model of the Collector

ID FIRST 'Stack, (node_array)ID FIRST 'Stack) SCAN a AND not_garbage (ID x, ret_graph (ID x, graph), node_array) AND lrc_gto (ID x, ret_graph (ID x, graph), node_array) AND NOT VISITED x AND NUMBER x \(\leq 0 \) AND LOWLINK x = 0;

MAINSPEC:
- IN node_array: node_details-array
- IN group_handle: vertex-list-array
- IN stack: boolean
- IN lrc: integer
- IN stack: integer
- IN stack: vertex-list
- IN splitqueue: integer-list
- IN stack: vertex-list
- PRE lower_heap_limit < upper_heap_limit AND NOT EMPTY hidden AND not_garbage (ID ROOT graph, graph, nodearray) AND lrc_gto (ID ROOT graph, graph, node_array) AND NOT EMPTY free_list AND EMPTY cyclic_free_list AND NOT emit_flag AND EMPTY split_queue AND EMPTY stack AND NOT emit_stop;

PROC push;
x \rightarrow STACKED true;
stack \rightarrow (L x & stack);
ENDPROC

PROC push_handle;
store (group_handle, group, L (x) & (group_handleIgroup)) ASSUME POST;
ENDPROC

PROC pop_handle;
VAR h: vertex-list;
h \rightarrow group handleIgroup;
FIRST h \rightarrow h;
REST h \rightarrow h;
store (grouphandle, group id, remove from list (group_handleIgroup_id, x)) ASSUME POST;
ENDPROC

PROC pop_split;
Z \rightarrow FIRST Stack;
stack \rightarrow REST Stack;
ENDPROC

PROC pop_free;
VAR t n; node_details;
FIRST stack \rightarrow (node_arraylID x);
stack \rightarrow REST stack;
ENDPROC

PROC freecell;
VAR t n; node_details;
VAR X_sc: 3can_condition;
scan_con (x, x_sc) ASSUME POST;
IF x_sc \neq X THEN
    t n \rightarrow (node_arraylID n);
    t n \rightarrow GARBAGE true;
    t n \rightarrow GROUP ID 0;
    store (node array, ID n, tn) ASSUME POST;
push free (x, free_list) ASSUME POST;
ELSE
    cyclic_free_list \rightarrow (L X & cyclic_free_list);
ENDIF;
ENDPROC

PROC process_cyclic_free_list;
VAR n: vertex;
VAR t n: node_details;
VAR l, m: integer;
l \rightarrow length (cyclic_free_list);
m \rightarrow l + 1; (establishes loop variant)
WHILE
   ASSERT l > m;
   NOT EMPTY cyclic_free_list LOOP
        l \rightarrow length (cyclic_free_list);
        m \rightarrow FIRST cyclic_free_list;
pop handle (m, GROUP ID(node_arraylID m)) ASSUME POST;
tn \rightarrow (node_arraylID m);
tn \rightarrow GARBAGE true;
tn \rightarrow GROUP ID 0;
store (node_array, ID m, tn) ASSUME POST;
push free (x, free_list) ASSUME POST;
cyclic_free_list \rightarrow REST cyclic_free_list;
l \rightarrow length (cyclic_free_list);
ENDLOOP;
ENDPROC

-200-
Section B.1 IL Source

PROC kblog_con;
IF SCAN (node_arrayID x) = a AND GROUP_ID(node_arrayID x) = aplgr THEN as := m;
ELSIF SCAN (node_arrayID x) = b AND NOT emit_flag THEN ac := p;
ELSIF SCAN (node_arrayID x) = b AND emit_flag THEN ac := q;
ELSIF SCAN (node_arrayID x) = c AND NOT emit_flag THEN ac := f;
ELSIF SCAN (node_arrayID x) = c AND emit_flag AND NUMBER x = emit_thread AND GROUP_ID(node_arrayID x) = aplgr THEN ac := f;
ELSIF SCAN (node_arrayID x) = c AND emit_flag AND NUMBER x < emit_thread AND GROUP_ID(node_arrayID x) = aplgr THEN ac := f;
ELSE ac := xx;
ENDIF;
ENDPROC

PROC pop;
  z := FIRST stack;
  stack := REST stack;
ENDPROC

PROC collect group;
VAR y : vertex;
VAR tn: node_details;
VAR 1, ■ : integer;
  t := length (cyc_stk);
  ■ - 1 := 1;  (Establishes loop variant)
WHILE NOT GARBAGE (node_arrayI( ID FIRST cyc_stk)) AND t > m;
NOT EMPTY cyc_stk
LOOP
  t := length (cyc_stk);
  y := y STACKED false;
  tn := (node_arraylID y);
  tn := tn GARBAGE true;
  store (node_array, ID y, tn) ASSUME POST;
  push_free (y, free_list) ASSUME POST;
ENDLOOP;
ENDPROC

PROC stackgroup;
VAR s_n: vertex;
remove_from_split_queue (grp_id) ASSUME POST;
Ensures the we do not subsequently try to split this group)
WHILE NOT EMPTY (group_handleIgrp_id) OR
GARBAGE (node_array (cyc_stk)) AND NOT EMPTY (group_handleIgrp_id) AND NOT GARBAGE (node_array)
LOOP
  remove_from_handle (s_n, grp_i d) ASSUME POST;
  stack_group_body (grpid, s_n, retgraph (ID s_n, graph),
ENDLOOP;
ENDPROC

PROC stack group body,
VAR ah, at: vertex;
VAR sh, st: boolean;
IF NOT STACKED start_node THEN
  head := ROOT LEFT grstr;
  tail := ROOT RIGHT grstr;
push (start_node, cyc_stk) ASSUME POST;
same_group (start_node, head, garray, node_array, sh) ASSUME POST;
same_group (start_node, tail, garray, node_array, st) ASSUME POST;
ELSE
ENDIF;
ENDIF;
ENDPROC

PROC same group;
IF g_array!(GROUP_ID(node_arrayI ID p)) > g_arrayl(GROUP_ID(node_arraylID q)) THEN result := true;
ELSE result := false;
ENDIF;
ENDPROC

PROC newapp;
VAR smhd, smtl : boolean;
VAR par_sc, hdsc, tl_sc: scan_condition;
VAR nd, tn: node_details;
VAR new_node, hd, tl: vertex;
VAR tgc: integer;
popfree (free_list, newnode) ASSUME POST;
(Give the new node an initial local rc of one)
tn := (node_arraylID newnode);
store (node_array, ID newnode, tn) ASSUME POST;
hd := ROOT LEFT ret_graph (ID newnode, graph);
ENDPROC

PROC kblogapp;
VAR ma_md, ma tl: boolean;
VAR par_sc, hd_sc, tl_sc: scan_condition;
VAR ac, tn: node_details;
VAR new_node, b, tl: vertex;
VAR tgc: integer;
popfree (free_list, new_node) ASSUME POST;
(Give the new node as initial local rc of one)
tn := (node_arraylID new_node);
ENDPROC

PROC newspaper;
VAR mm, ac, tl: scan_condition;
VAR ac, tn: node_details;
VAR tgc: integer;
ENDPROC

-201-
Appendix B. The MALPAS Model of the Collector

11 := ROOT RIGHT ret_graph (ID new_node, graph);
scan_con (par, par_sc) ASSUME POST;
scan_con (hd, hd_sc) ASSUME POST;
same_group (par, hd, g_array, node_array, em, hd) ASSUME POST;
same_group (par, tl, g_array, node_array, em, hd) ASSUME POST;
IF par_sc = = OR par_sc = = OR par_sc = = OR par_sc = = q @ THEN
(The new structure will be taken into account by the Tarjan search when it returns to the parent node)
IF sae hd OR sae tl THEN
(cannot have a node spanning two groups, so incorporate it into the parent's group)
nd := new GROUP_ID (GROUP_ID (node array ID par));
(Do not set the STACKED, lowlink, number values since child will be visited by Tarjan search)
store (node array, ID new node, nd) ASSUME POST;
ELSE (must be a singular node)
grp_id := grp_id + 1;
(gives a unique id to this singular node)
nd := new GROUP_ID (grp_id);
store (node array, ID new_node, nd) ASSUME POST;
ENDIF;
ELSEIF par_sc = = THEN
IF sae hd OR sae tl THEN
(some group is to be re-split, after emission. Note that the child and parent are both in the old group, cannot certify SCC structure)
push_split (spl grp, split queue); (only push if not already on stack)
Push the parent onto the handle to guarantee that there is a starting point for the Tarjan search)
push_handle (par, splgrp) ASSUME POST;
(Place parent in the old group handle so it will be visited by Tarjan search later)
nd := new GROUP_ID (spl grp);
(Do not set the STACKED, lowlink, number values since newnode will be visited by Tarjan search)
store (node array, ID new node, nd) ASSUME POST;
ELSE (must be an external node, including child_sc = =)
grp_id := grp_id + 1;
(gives a unique id to this singular node)
nd := new GROUP_ID (grp_id);
store (node array, ID new_node, nd) ASSUME POST;
ENDIF;
ELSEIF par_sc = = THEN
IF sae hd OR sae tl THEN
(old group is to be re-split, before emission)
push_split (spl grp, split queue); (only push if not already on stack)
(Do need to push the parent onto the handle to guarantee that there is a starting point for the Tarjan search)
push_handle (par, spl grp) ASSUME POST;
(Place parent in the old group handle to guarantee that there is a starting point for the Tarjan search)
nd := new GROUP_ID (spl grp);
(Do not set the STACKED, lowlink, number values since newnode will be visited by Tarjan search)
store (node array, ID new node, nd) ASSUME POST;
ELSE (must be an external node)
grp_id := grp_id + 1;
(gives a unique id to this singular node)
nd := new GROUP_ID (grp_id);
store (node array, ID new_node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSE (must be an external node)
grp_id := grp_id + 1;
(gives a unique id to this singular node)
nd := new GROUP_ID (grp_id);
store (node array, ID new_node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSIF par_sc = = THEN
IF sae hd OR sae tl THEN
(tgc := g array emit group id)
store (g array, emit group id, tgc) ASSUME POST,
nd := new GROUP_ID (em, g array)
store (node array, ID new node, nd) ASSUME POST;
ENDIF;
ELSE (must be an external node)
grp_id := grp_id + 1;
(gives a unique id to this singular node)
nd := new GROUP_ID (grp_id);
store (node array, ID new_node, nd) ASSUME POST;
*
Section B.1 IL Source

nd := nd GROUP_ID org_id;
store (node_array, ID new_node, nd) Assume POST;
ENDIF;
ENDIF;

addref (ld, new_app, g_array, node_array) Assume POST;
addref (tl, new_app, g_array, node_array) Assume POST;
ENDPROC

PROC addref:
VAR sm : boolean;
VAR to: node_details;
VAR tic, tgc: integer;
VAR parse, ch_sc: scan_condition;
tn := (node_arraylID to);
tic := L_COUNT tn;
tic := tic + 1;
store (node_array, ID to. tn) Assume POST;
scan_con (to, ch_sc) Assume POST;
scancon (from, parse) ASSUME POST;
IF par_sc = xx OR par_sc = nn THEN
THE new pointer will be looked at by the Tarjan search when it returns
THE parent)
same_group (to, from, g_array, node_array, sm) ASSUME POST;
IF NOT sm THEN
ip := g_arraylGROUP_ID(node_array|ID to);
store (g_array, GROUP_ID(node_array|ID to), tgc+l) ASSUME POST;
ELSEIP par_sc = fe THEN
IF ch_sc = fe OR ch_sc = ee THEN
[Skip]
ELSEIF ch_sc = fa OR ch_sc = qq OR ch_sc = pp THEN
[This is an external pointer when the group is emitted,
DO not increment the old group’s count since this will
be done by emit]
ELSEIP ch_sc = xx THEN
ip := g_array|GROUP_ID|node_array|ID to;
store (g_array, GROUP_ID|node_array|ID to, tgc+l) ASSUME POST;
ENDIF;
ELSEIP

ENDIF;
ENDIF;
ENDPROC

PROC addref_ext:
VAR sm : boolean;
VAR to: node_details;
VAR tic, tgc: integer;
VAR par_sc, ch_sc: scan_condition;
tn := (node_arraylID to);
tic := L_COUNT tn;
tic := tic + 1;
store (nodearray, ID to, tn) ASSUME POST;
scan_con (to, ch_sc) ASSUME POST;
scancon (from, parse) ASSUME POST;
IF par_sc = xx OR parse = nn OR parse = pp OR par_sc = qq THEN
(The new pointer will be looked at by the Tarjan search when it returns
THE parent)
same_group (to, from, g_array, node_array, sm) ASSUME POST;
IF NOT sm THEN
ip := g_arraylGROUP_ID|node_array|ID to;
store (g_array, GROUP_ID|node_array|ID to, tgc+l) ASSUME POST;
ELSEIP par_sc = fe THEN
IF ch_sc = fe OR ch_sc = ee THEN
[Skip]
ELSEIF ch_sc = xx THEN
ip := g_array|GROUP_ID|node_array|ID to;
store (g_array, GROUP_ID|node_array|ID to, tgc+l) ASSUME POST;
ELSEIF par_sc = fe THEN
IF ch_sc = fe THEN
same_group (to, from, g_array, node_array, sm) ASSUME POST;
IF sm THEN
(old group is to be re-split, after emission). The child and
parent are both in the old group, cannot rectify
GCD structure)
push_split (splitgroup) ASSUME POST;
ELSEIP

ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDIF;
ENDER
Appendix B. The MALPAS Model of the Collector

do not increment the old group's count since this will be done by emit)
ELSEIF ch_sc = xx THEN
  tpc = tpc-GROUP_ID(node_array(ID to to), to); store (g_array, GROUP_ID(node_array(ID to to), to); tpc+1) ASSUME POST; ENDIP;
ELSEIF par_sc = nn OR parse = fe OR par_sc = ee THEN
  the child is in the group being split - no need to add to handle, since we have already visited the child);
ELSEIF par_sc = xx OR par_sc = fe OR par_sc = ee THEN
  an external pointer to the new group, or other group, to the old, when the parent is (if par_sc = fe OR par_sc = ee)
  emitted, no need to decrement old group's grc since this will be recalculated by emit)
ELSEIF ch_sc = nn OR parse = fe OR par_sc = ee THEN
  an external pointer from the new group, or other group, to the old, when the parent is (if par_sc = fe OR par_sc = ee)
  emitted, no need to decrement old group's grc since this will be recalculated by emit)
ELSEIF ch_sc = xx OR ch_sc = nn OR parse = fe OR par_sc = ee THEN
  [i.e. we are already splitting the group the child belongs to]
ELSEIF ch_sc = nn OR parse = fe OR par_sc = ee THEN
  no need to add current group to the split_queue)
ELSEIF par_sc = xx OR par_sc = fe OR par_sc = ee THEN
  child will NOT form part of the SCC currently being emitted, but has been visited by the Tarjan search]
ELSEIF par_sc = xx OR par_sc = fe OR par_sc = ee THEN
  child is NOT on the Tarjan stack, but is in the group being split, the child is in the group being split - no need to add
  to handle, since we have already visited the child];
ELSEIF par_sc = xx OR par_sc = fe OR par_sc = ee THEN
  [have to re split the old group after emission has occurred]
ELSEIF ch_sc = xx OR ch_sc = nn OR par_sc = xx OR par_sc = ee THEN
  we know that this is an external pointer to the new group, if the parent was going to be on the new group then it would have
par_sc = fe or ee (i.e. have been visited by now by the Tarjan search, emit will calculate the new group's grc)

-204-
Section B.1 IL Source

PROC strong_connect;
VAR v, w: vertex;
VAR wp: vertex tree;
VAR t: node_details;
VAR g: ngcl;
VAR aa, bb, cc, dd: vertex;
VAR i, dfsn: integer;
com (g) ASSUME POST;
IF g = one THEN
  aa := ra_no (hidden, graph, i, node_array);
  newapp (aa, g_array, nodearray, bb) ASSUME POST;
ELSIF g = two THEN
  dd := ra_no (hidden, graph, i, node_array);
  cc := ra_no (hidden, graph, i, node_array);
  deleteref (dd, cc) ASSUME POST;
ELSIF g = three THEN
  dd := ra_no (hidden, graph, i, node_array);
  cc := ra_no (hidden, graph, i, node_array);
  addref ext (dd, cc, garray, nodearray) ASSUME POST;
  IF (g_array 1 (GROUP_ID (node_array 1 (ID ROOT graph))))
    v := ROOT graph;
  ELSE Viv = VISITED true;
  ENDIF;
  t := t SCAN b;
  dfsn = dfsn + 1;
  store (node_array, ID v, t) ASSUME POST;
  graph = tree(LEFT graph, v, RIGHT graph);
  push (v, stack);
  IF NOT T_EMPTY LEFT graph THEN
    wg := LEFT graph;
    w := ROOT wg;
    IF NOT VISITED w THEN
      strong_connect (w, dfsn) ASSUME POST;
      graph := tree (LEFT graph, w, RIGHT graph);
      v := LOWLINK min (LOWLINK v, LOWLINK w);
      push (v, stack);
    ENDIF;
  ENDIF;
ELSE
  IF (g_array 1 (GROUP_ID (node_array 1 (ID ROOT graph))))
    spl_grp THEN
    spl_grp := spl_grp - 1;
  ENDIF;
ENDIF;
ENDPROC

IP L_COUNT (node_array (ID to)) = 0 THEN
  deleteref (ROOT LEFT (ret_graph (ID to, graph)), to) ASSUME POST;
  deleteref (ROOT RIGHT (ret_graph (ID to, graph)), to) ASSUME POST;
  freecell (to, free_list, cyclic_free_list) ASSUME POST;
ENDIF;
ELSIF par_sc = xe OR par_sc = xx THEN
  IF L_COUNT (node_array (ID to)) = 0 THEN
    remove_from_handle (to, emit_group_id) ASSUME POST;
    deleteref (ROOT LEFT (ret_graph (ID to, graph)), to) ASSUME POST;
    deleteref (ROOT RIGHT (ret_graph (ID to, graph)), to) ASSUME POST;
    freecell (to, free_list, cyclic_free_list) ASSUME POST;
  ELSE
    push_handle (to, emit_group_id) ASSUME POST;
  ENDIF;
ENDIF;
ELSIF ch_sc = xe THEN
  IF par_sc = xx OR par_sc = fx OR par_sc = qq THEN
    (an internal pointer in the new group, split the new group after emission)
    push_split (emit_group_id, split_queue) ASSUME POST;
    IF L_COUNT (node_array (ID to)) = 0 THEN
      remove_from_handle (to, emit_group_id) ASSUME POST;
      deleteref (ROOT LEFT (ret_graph (ID to, graph)), to) ASSUME POST;
      deleteref (ROOT RIGHT (ret_graph (ID to, graph)), to) ASSUME POST;
      freecell (to, free_list, cyclic_free_list) ASSUME POST;
    ELSE
      push_handle (to, emit_group_id) ASSUME POST;
    ENDIF;
  ELSE
    push_handle (to, emit_group_id) ASSUME POST;
    emit_stop := true;
  ENDIF;
ELSIF ch_sc = xe THEN
  IF par_sc = xx OR par_sc = fx OR par_sc = qq THEN
    (an internal pointer in the new group, split the new group before emission)
    push_split (emit_group_id, split_queue) ASSUME POST;
    IF L_COUNT (node_array (ID to)) = 0 THEN
      deleteref (ROOT LEFT (ret_graph (ID to, graph)), to) ASSUME POST;
      deleteref (ROOT RIGHT (ret_graph (ID to, graph)), to) ASSUME POST;
      freecell (to, free_list, cyclic_free_list) ASSUME POST;
    ELSE
      push_handle (to, emit_group_id) ASSUME POST;
    ENDIF;
  ELSE
    push_handle (to, emit_group_id) ASSUME POST;
  ENDIF;
ENDIF;
Appendix B. The MALPAS Model of the Collector

IF STACKED w THEN
  v := LOWLINK min(LOWLINK v, NUMBER w);
  graph := tree(LAST graph, v, RIGHT graph);
END IF;
IF NOT T_EMPTY RIGHT graph THEN
  w := RIGHT graph;
END IF;
IF NOT VISITED w THEN
  strong_con(wg, dfsn) := ASSUME POST;
  graph := tree(LAST graph, ROOT graph, w);
  v := LOWLINK min(LOWLINK v, LOWLINK w);
  graph := tree(LAST graph, v, RIGHT graph);
END IF;
IF NUMBER w := NUMBER v THEN
  v := LOWLINK min(LOWLINK v, NUMBER w);
  graph := tree(LAST graph, v, RIGHT graph);
END IF;
END IF;
END IF;
END IF;
END PROC

PROC emit;
VAR y, head, tail; vertex;
VAR nfs, group_count, integer;
VAR nd: node_details;
VAR bd, tbl; node, scan_condition;
VAR cfs, temp_stack; vertex-list;
VAR n : NUMBER;
VAR strs; number-first stack;
VAR g0; number-first stack;
VAR int_co: integer;
VAR comm: led
VAR t_g: integer;
VAR t_i: integer;
VAR d, e, f, g, h: integer;
VAR g, gnl, g2, g3, g4: integer;
VAR g0: number-first stack;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
VAR t_d: integer;
VAR t_c: integer;
VAR t_h: integer;
VAR t_i: integer;
VAR t_g: integer;
VAR t_f: integer;
VAR t_e: integer;
Section B.2 The MALPAS Log File

The MALPAS Log File

MALPAS 6.0
B batch command file
command m a i n
command using MALPAS.
command procedure.
Command files for analysis
MALPAS/precedent.stap/precedent/precedent.data
Appendix B. The MALPA S Model of the Collector

| CPU time | 0:04:04.18 | false pre_store err was reduced to false |
| CPU time | 0:04:06.18 | false post_store err was reduced to false |
| CPU time | 0:04:07.49 | Threat was reduced to false |
| CPU time | 0:04:07.02 | All threats was reduced to false |
| CPU time | 0:04:07.03 | Path 1 |
| CPU time | 0:04:08.68 | false pre_store err was reduced to false |
| CPU time | 0:04:09.12 | false post_store err was reduced to false |
| CPU time | 0:04:09.13 | Threat was reduced to false |
| CPU time | 0:04:09.14 | All threats was reduced to false |
| CPU time | 0:04:10.47 | false pre_store err was reduced to false |
| CPU time | 0:04:10.48 | false post_store err was reduced to false |
| CPU time | 0:04:10.49 | Threat was reduced to false |
| CPU time | 0:04:10.50 | All threats was reduced to false |
| CPU time | 0:04:10.59 | Threat was reduced to false |
| CPU time | 0:04:10.67 | All threats was reduced to false |
| CPU time | 0:04:10.70 | Path 2 |
| CPU time | 0:04:10.71 | Threat was reduced to false |
| CPU time | 0:04:10.72 | All threats was reduced to false |
| CPU time | 0:04:10.73 | Path 3 |
| CPU time | 0:04:10.74 | Threat was reduced to false |
| CPU time | 0:04:10.75 | All threats was reduced to false |
| CPU time | 0:04:10.76 | Path 4 |
| CPU time | 0:04:10.77 | Threat was reduced to false |
| CPU time | 0:04:10.78 | All threats was reduced to false |
| CPU time | 0:04:10.79 | Path 5 |
| CPU time | 0:04:10.80 | Threat was reduced to false |
| CPU time | 0:04:10.81 | All threats was reduced to false |
| CPU time | 0:04:10.82 | Path 6 |
| CPU time | 0:04:10.83 | Threat was reduced to false |
| CPU time | 0:04:10.84 | All threats was reduced to false |
| CPU time | 0:04:10.85 | Path 7 |
| CPU time | 0:04:10.86 | Threat was reduced to false |
| CPU time | 0:04:10.87 | All threats was reduced to false |
Appendix B. The MALPAS Model of the Collector

CPU tlm • 009,56,01.31
CPU tlm • 009,56,01.26
CPU tlm • 009,55,22.56
CPU tlm • 009,55,17.64
CPU tlm • 009,55,17.63
CPU tlm • 009,55,11.72
CPU tlm • 009,54,46.77
CPU tlm • 009,53,59.00
CPU tlm • 009,53,56.96
CPU tlm • 009,53,57.99
CPU tlm • 009,53,57.94
CPU tlm • 009,53,55.09
CPU tlm • 009,53,55.07
CPU tlm • 009,53,55.03
CPU tlm • 009,53,55.02
CPU tlm • 009,53,55.01
CPU tlm • 009,53,55.00
CPU tlm • 009,53,54.46.77
CPU tlm • 009,53,53.59.02
CPU tlm • 009,53,53.56.96
CPU tlm • 009,53,53.57.99
CPU tlm • 009,53,53.57.94
CPU tlm • 009,53,53.55.09
CPU tlm • 009,53,53.55.07
CPU tlm • 009,53,53.55.03
CPU tlm • 009,53,53.55.02
CPU tlm • 009,53,53.54.46.77
CPU tlm • 009,53,53.53.59.02
CPU tlm • 009,53,53.53.56.96
CPU tlm • 009,53,53.53.57.99
CPU tlm • 009,53,53.53.57.94
CPU tlm • 009,53,53.53.55.09
CPU tlm • 009,53,53.53.55.07
CPU tlm • 009,53,53.53.55.03
CPU tlm • 009,53,53.53.55.02
CPU tlm • 009,53,53.54.46.77
Section B.2 The MALPAS Log File
Appendix B. The MALPAS Model of the Collector

MALPAS Release 8.0.2 (3.0) 20-Sep-1993 16:16  Banner Page  

The Compliance Analyser.

B.3 Output of the Compliance Analyser.

MALPAS Release 8.0.2 (3.0) 20-Sep-1993 16:16  IL Reader  

The following file, qualifiers and keywords were specified for this run. . .

FILE = DISKDIR/COMPLIANCE/MALPASITAR_69.IL;  
COMPLIANCE: <-----

Qualifiers  

--- Applied to MALPAS ---

MALPAS Release 8.0.2 (3.0) 20-Sep-1993 16:16  IL Reader  

--- Applied to IL Reader ---

--- Include files ---

--- Keywords ---

--- Greenhous ---

--- Difference ---

--- Output ---

--- Statistics ---

--- Summary ---

--- Compliance ---

--- PRIORITY ---

--- End ---
Appendix B. The MALPAS Model of the Collector

```plaintext
REPLACE [a] (x: hi-array; i: integer; m: hi));
REPLACE [b] (x: hi-array; i: integer; m: hi);
REPLACE [c] (x: hi-array; i: integer; m: hi);
REPLACE [d] (x: hi-array; i: integer; m: hi);
REPLACE [e] (x: hi-array; i: integer; m: hi);
REPLACE [f] (x: hi-array; i: integer; m: hi);
REPLACE [g] (x: hi-array; i: integer; m: hi);
REPLACE [h] (x: hi-array; i: integer; m: hi);
REPLACE [i] (x: hi-array; i: integer; m: hi);
REPLACE [j] (x: hi-array; i: integer; m: hi);
REPLACE [k] (x: hi-array; i: integer; m: hi);
REPLACE [l] (x: hi-array; i: integer; m: hi);
REPLACE [m] (x: hi-array; i: integer; m: hi);
REPLACE [n] (x: hi-array; i: integer; m: hi);
REPLACE [o] (x: hi-array; i: integer; m: hi);
REPLACE [p] (x: hi-array; i: integer; m: hi);
REPLACE [q] (x: hi-array; i: integer; m: hi);
REPLACE [r] (x: hi-array; i: integer; m: hi);
REPLACE [s] (x: hi-array; i: integer; m: hi);
REPLACE [t] (x: hi-array; i: integer; m: hi);
REPLACE [u] (x: hi-array; i: integer; m: hi);
REPLACE [v] (x: hi-array; i: integer; m: hi);
REPLACE [w] (x: hi-array; i: integer; m: hi);
REPLACE [x] (x: hi-array; i: integer; m: hi);
REPLACE [y] (x: hi-array; i: integer; m: hi);
REPLACE [z] (x: hi-array; i: integer; m: hi);

WARM UPG:

WARM UP:
No PEG condition supplied for procedure store.

WARM UP:
No PEG condition supplied for procedure store.

End of file.
```

---

This page contains the code for the MALPAS Model of the Collector, which is a part of the METIS garbage collection system. The code includes various procedures and functions for manipulating data structures, such as arrays and records, and conditions for determining the state of the system during garbage collection. The code is written in a procedural style, typical of low-level programming languages used in system software at that time.
Section B.3 Output of the Compliance Analyser

FUNCTION g_cout_aaa (noda_datail array, vecto-list, integer, integer)

REPLACE [1] (noda_datail array, vector-list, integer, integer)
Appendix B. The MALPAS Model of the Collector

FUNCTION delete_read (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)

FUNCTION delete (node_detailary, vector, vector)

FUNCTION update (node_detailary, vector, vector)

FUNCTION add (node_detailary, vector, vector)

FUNCTION replace (node_detailary, vector, vector)
Section B.3 Output of the Compliance Analyser
**Section B.3 Output of the Compliance Analyser**

```plaintext

```

-229-
Appendix B. The MALPAS Model of the Collector

11423)

11512)

11513)

11514)

11515)

11516)

11517)

11518)

11519)

11520)

11521)

11522)

11523)

11524)

11525)

11526)

11527)

11528)

11529)

11530)

11531)

11532)

11533)

11534)

11535)

11536)

11537)

11538)

11539)

11540)

11541)

11542)

11543)

11544)

11545)

11546)

11547)

11548)

11549)

11550)

11551)

11552)

11553)

11554)

11555)

11556)

11557)

11558)

11559)

11560)

11561)

11562)

11563)

11564)

11565)

11566)

11567)

11568)

11569)

11570)

11571)

11572)

11573)

11574)

11575)

11576)

11577)

11578)

11579)

11580)

11581)

11582)

11583)

11584)

11585)

11586)

11587)

11588)

11589)

11590)

11591)

11592)

11593)

11594)

11595)

11596)

11597)

11598)

11599)

11600)

11601)

11602)

11603)

11604)

11605)

11606)

11607)

11608)

11609)

11610)

11611)

11612)

11613)

11614)

11615)

11616)

11617)

11618)

11619)

11620)

11621)

11622)

11623)

11624)

11625)

11626)

11627)

11628)

11629)

11630)

11631)

11632)

11633)

11634)

11635)

11636)

11637)

11638)

11639)

11640)

11641)

11642)

11643)

11644)

11645)

11646)

11647)

11648)

11649)

11650)

11651)

11652)

11653)

11654)

11655)

11656)

11657)

11658)

11659)

11660)

11661)

11662)

11663)

11664)

11665)

11666)

11667)

11668)

11669)

11670)

11671)

11672)

11673)

11674)

11675)

11676)

11677)

11678)

11679)

11680)

11681)

11682)

11683)

11684)

11685)

11686)

11687)

11688)

11689)

11690)

11691)

11692)

11693)

11694)

11695)

11696)

11697)

11698)

11699)

11700)

11701)

11702)

11703)

11704)

11705)

11706)

11707)

11708)
Section B.3 Output of the Compliance Analyser

```
(1537) PROC pop_handle;
(1538) VAR h, vector(list);
(1539) $1 = procedure pop_handle pop_expression;
(1540) $2 = handle

(1541) PROC
(1542) h , FIRST h;
(1543) $2 = (L (h , & start));
(1544) $2 = FIRST;
(1545) $2 = STOP;

(1546) STOP, MAP
```

```
Appendix B. The MALPAS Model of the Collector

1649 Appendix B. The MALPAS Model of the Collector
1650
1651 # XXX (SKIP)
1652
1653 # XXX (SKIP)
1654
1655 # XXX (SKIP)
1656
1657 # XXX (SKIP)
1658
1659 # XXX (SKIP)
1660
1661 # XXX (SKIP)
Section B.3 Output of the Compliance Analyser
Appendix B. The MALPA Model of the Collector

```plaintext
VAR par_ac, ch_ac, ac_con;  
VAR ttm, integer;  
PROC addref;  
VAR par_ac, ch_ac, ac_con;  
PROC addref;  
VAR par_ac, ch_ac, ac_con;  
PROC addref;
```

The new pointer will be looked at by the Tarjan structure when it returns to the parent.

The new pointers will be looked at by the Tarjan structure when it returns to the parent.

Do not allow emit to proceed by setting the emit.top flag.

Puah the parent on to the handle to guarantee that there is a
"

The new pointers will be looked at by the Tarjan structure when it returns to the parent.

Do not allow emit to proceed by setting the emit.top flag.

Puah the parent on to the handle to guarantee that there is a
"

The new pointers will be looked at by the Tarjan structure when it returns to the parent.

Do not allow emit to proceed by setting the emit.top flag.

Puah the parent on to the handle to guarantee that there is a
"

The new pointers will be looked at by the Tarjan structure when it returns to the parent.

Do not allow emit to proceed by setting the emit.top flag.

Puah the parent on to the handle to guarantee that there is a
"

The new pointers will be looked at by the Tarjan structure when it returns to the parent.

Do not allow emit to proceed by setting the emit.top flag.

Puah the parent on to the handle to guarantee that there is a
"

The new pointers will be looked at by the Tarjan structure when it returns to the parent.

Do not allow emit to proceed by setting the emit.top flag.

Puah the parent on to the handle to guarantee that there is a
"
Section B.3 Output of the Compliance Analyser

[Page 237]
Appendix B. The MALPAS Model of the Collector

[0001] 177: #BSP
[0002] 178: IF LCOUNT(nodea) = 0 THEN
[0003] 179: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0004] 180: DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1
[0005] 181: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0006] 182: s"DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1"
[0007] 183: #BSP

[0008] 184: ELSE
[0009] 185: IF LCOUNT(nodea) > 0 THEN
[0010] 186: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0011] 187: DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1
[0012] 188: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0013] 189: s"DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1"
[0014] 190: #BSP

[0015] 191: ENDIP.

[0016] 192: #BSP

[0017] 193: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0018] 194: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0019] 195: #BSP

[0020] 196: ELSE
[0021] 197: IF LCOUNT(nodea) > 0 THEN
[0022] 198: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0023] 199: DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1
[0024] 200: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0025] 201: s"DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1"
[0026] 202: #BSP

[0027] 203: ENDIP.

[0028] 204: #BSP

[0029] 205: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0030] 206: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0031] 207: #BSP

[0032] 208: ELSE
[0033] 209: IF LCOUNT(nodea) > 0 THEN
[0034] 210: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0035] 211: DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1
[0036] 212: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0037] 213: s"DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1"
[0038] 214: #BSP

[0039] 215: ENDIP.

[0040] 216: #BSP

[0041] 217: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0042] 218: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0043] 219: #BSP

[0044] 220: ELSE
[0045] 221: IF LCOUNT(nodea) > 0 THEN
[0046] 222: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0047] 223: DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1
[0048] 224: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0049] 225: s"DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1"
[0050] 226: #BSP

[0051] 227: ENDIP.

[0052] 228: #BSP

[0053] 229: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0054] 230: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0055] 231: #BSP

[0056] 232: ELSE
[0057] 233: IF LCOUNT(nodea) > 0 THEN
[0058] 234: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0059] 235: DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1
[0060] 236: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0061] 237: s"DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1"
[0062] 238: #BSP

[0063] 239: ENDIP.

[0064] 240: #BSP

[0065] 241: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0066] 242: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0067] 243: #BSP

[0068] 244: ELSE
[0069] 245: IF LCOUNT(nodea) > 0 THEN
[0070] 246: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0071] 247: DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1
[0072] 248: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0073] 249: s"DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1"
[0074] 250: #BSP

[0075] 251: ENDIP.

[0076] 252: #BSP

[0077] 253: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0078] 254: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0079] 255: #BSP

[0080] 256: ELSE
[0081] 257: IF LCOUNT(nodea) > 0 THEN
[0082] 258: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0083] 259: DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1
[0084] 260: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0085] 261: s"DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1"
[0086] 262: #BSP

[0087] 263: ENDIP.

[0088] 264: #BSP

[0089] 265: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0090] 266: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0091] 267: #BSP

[0092] 268: ELSE
[0093] 269: IF LCOUNT(nodea) > 0 THEN
[0094] 270: PUSHDATA(nodea) = PUSHDATA(nodea) + 1
[0095] 271: DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1
[0096] 272: s"PUSHDATA(nodea) = PUSHDATA(nodea) + 1"
[0097] 273: s"DELETE(dataDAG(nodea)) = DELETE(dataDAG(nodea)) + 1"
[0098] 274: #BSP

[0099] 275: ENDIP.
Section B.3 Output of the Compliance Analyser

```plaintext
[137x733] BLSIF g . th r a a  THEN
[139x728] dd t«  E*_oo (h id d an . g ra p h , l i .  n o d a _ a rra y );
[139x723] c c  I .  r« _ o o  (h id d an , g ra p h , 11 . o c d a _ a rra y ),
[139x718] a d d ra (_ a x t (d d . c c . g a r r a y , n o d a _ a rra y ) AfiSUKB POST;
[137x709] l«K IP) KLSB
[139x704] IF  (g _ a rra y  »  (GROOP^ZD (n o d a _ a rra y  ! (10 ROOT g r a p h ) )) )
[141x685] V  V  LOWLINI i ,
[141x680] V  » • V  N U M B B R  i;
[141x675] V  V  VISITED t r u a ;
[141x656] d fa n  I - d f a n t i ;
[141x652] a t o r a  (n o d a _ a rra y . ID v. to ) ASSUM E POST;
[141x647] g ra p h  • •  traa(L E F T  g ra p h , v. RIGHT g ra p h );
[141x642] IP  NOT T_BMPTY LEFT g ra p h  THEN
[143x638] wg >• LEFT g ra p h ;
[143x628] IP  HOT VISITED W  T H E M
[146x623] a tro n  g r a p h  (w g . d fan ) ASSUM E POST;
[146x618] g ra p h  ■ > traa(W E L T g ra p h , v. ROOT g ra p h . RIGHT g ra p h );
[145x609] g ra p h  i« LOWLINK (LOWLINK v. LOW LINK w );
[145x604] g ra p h  ,» traa(L E F T  g ra p h , v. RIGHT g ra p h )
[143x600] BLSIF N U M B B R  w <  N U M B E R  V  THEN
[146x595] IF STACKED w T H E M
[148x590] V  V  LOW LINK BiO(LOWLIMK V. N U M B E R  w ),
[148x595] g ra p h  ,■ traa(L E PT  g ra p h , v. RIGHT g ra p h )
[141x496] ENDIF,
[136x492] (SKIP)
[140x468] a t o r a  (nod# a rra y . ID v . tn ) ASSUM E POST,
```

### Additional Code Snippets

```
VAR y. head , tall  . v e r t e x ;
VAR  nfa . g r o u p _ c o u n t. l i . in t e g e r ,
VAR  nd , node d e t a i l a ,
VAR  hd _ a c. t l _ a c i  a c a n c o n d l t i o n ,
VAR  Cfa. te « p _ a ta c k , v e r t e x . H a t ,
VAR  g . n g c i;
VAR  aa bb, c c . d d ; v e r t e x ,
VAR  la c _ l n f . g _ ln f , ln t_ c o «  I n te g e r ,
VAR  aaauaie , « <p ro c e d u re  em it PRE ex p re
```

### Variables and Expressions

- `VAR y. head , tall  . v e r t e x ;`
- `VAR  nfa . g r o u p _ c o u n t. l i . in t e g e r ,`
- `VAR  nd , node d e t a i l a ,`
- `VAR  hd _ a c. t l _ a c i  a c a n c o n d l t i o n ,`
- `VAR  Cfa. te « p _ a ta c k , v e r t e x . H a t ,`
- `VAR  g . n g c i;`
- `VAR  aa bb, c c . d d ; v e r t e x ,`
- `VAR  la c _ l n f . g _ ln f , ln t_ c o «  I n te g e r ,`
- `VAR  aaauaie , « <p ro c e d u re  em it PRE ex p re`
Appendix B. The MALPAS Model of the Collector

```plaintext

(2369) fEND. BM M A IH
```

WARNING. ASSUME used with unproved POST or USE in call of attr

WARNING. ASSUME used with unproved POST or USE in call of declare

WARNING: Procedure body for select has not been defined

WARNING: ASSUME used with unproved POST or USE in call of attr

WARNING: The following identifier is declared but not used:
```
Section B.3 Output of the Compliance Analyser

Operator • with argument result specification (real) : real
  was declared but not used

Operator • with argument result specification (integer) : integer
  was declared but not used

Operator • with argument result specification ([real, real] : real)
  was declared but not used

Operator • with argument result specification ([real, real] : real)
  was declared but not used

Operator • with argument result specification (atring, atring)
  was declared but not used

All definitions of operator • exist.
All definitions of operator / exist.
Operator / with argument result specification (integer, integer) : boolean
  was declared but not used

Operator / with argument result specification (real, real) : boolean
  was declared but not used

Operator / with argument result specification (real, real) : boolean
  was declared but not used

Operator / with argument result specification (integer, integer) : boolean
  was declared but not used

Operator / with argument result specification (real, real) : boolean
  was declared but not used

Operator / with argument result specification (atring, atring) : boolean
  was declared but not used

All definitions of operator / exist.
All definitions of operator • exist.
References.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Journal or Publication Details</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>References</th>
</tr>
</thead>
</table>


References


### References

<table>
<thead>
<tr>
<th>Reference Code</th>
<th>Author(s)</th>
<th>Title</th>
<th>Publisher/Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>KOV-83</td>
<td>A.B. Kovaleski</td>
<td>&quot;Time Multiplexed Communications for Parallel Processors&quot;</td>
<td>Ph.D. Thesis, Department of Electronic and Electrical Engineering, University College London. (January 1983)</td>
</tr>
<tr>
<td>LAN-82</td>
<td>P.J. Landin</td>
<td>&quot;A Correspondence between Algol 60 and Church's Lambda Notation: Part 1&quot;</td>
<td>Communications of the A.C.M. 8. (February 1982). pp. 29 - 59</td>
</tr>
</tbody>
</table>
References


PAR-91  D. Parrott. Personal communication, Department of Computer Science, University College London. (February 1991).


PEY-87b  S.L. Peyton Jones. "FLIC - a Functional Language Intermediate Code". Internal note 2048, Department of Computer Science, University College London. (February 1987).


References


SAL-85 J.D. Salkild. "Implementation and Analysis of Two Cyclic Reference Counting Algorithms". M.Sc., Department of Computer Science, University College London. (June 1985).


<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title and Details</th>
</tr>
</thead>
</table>
References


References


Index

α 56
β 56
δ 54
η 57, 162
φ 57
λ 159
θ 159
τ 159
ξ 159
ψ 57, 162
@ 26
_assume 142
_as_thread 142
_thread 142

A
abstract function 142, 144
Æ 127-128, 131
active memory 14
Ada 14
addition of pointers and nodes 102
addr 115
addr_ext 127, 129, 132-133
Aerts 37
algebra of programs 19
Ali 32
ALICE 16
Appel 66, 76-77
application node 26
architecture of the collector 118
assignment statement 18
ASSUME_POST 142
atomic 27
Axford 37

B
background tasks 118, 121
Backus 15
Baker 28, 72
Ben-Ari 72
Bevan 34
bit-slice architecture 15
Bobrow 37
bottleneck 18
boxed representation 27
Brownbridge 37
bugs 17
built in variables 142
Burnett 24, 30, 82

C
c 55, 57-58, 82
C.mmp 16
cache 54
California Institute of Technology 16
Carnegie Mellon University 16
cell 26
cell colours 71
child 93, 114
Church 19
Church Rosser property 21
Cm* 16
combinators 23
CommonSpeak 82
compaction 28, 31
comparison of memory overhead 13, 161
compile time garbage collection 30
compliance analyser 139
compound collector operation 128
concurrent languages 17
concurrent mark-scan algorithms 70
Connection Machine 16
coroutine statements 18
copying garbage collection 28, 32
copying garbage collection in real time 72
correlation factor 57
CPU 15
critical section 85, 90
CTDNNet2 82
cyclic_free_list 95, 123
cyclic_free_flag 123, 135

d
Dannenberg 82
data flow 16, 23
data flow languages 19
data-driven 23
definition of a real time algorithm 66
deleteref 127, 129, 132-133
deletion types 124
Dennis 35
depth first search number (dfsn) 39, 55
derived graph 38
DERIVES relation 139, 141
deutsch-Schorr-Waite 28, 31
Index

dfsn 39-40, 54, 89, 91
digital signal processing 24, 82
Dijkstra 71, 137
distributed system 34, 72
dummy interrupt 159
Duprat 77
duration calculus 82

eager evaluation 21
EASE 157
embarrassing pause 70
emit_flag 92, 123
emit_group_id 92, 123
emit_interior_count 92, 109-110, 123
emit_stop 91, 97, 106, 123
emit_thresh 39, 91, 123
En 127-128, 131
environment-based architecture 23
Erlang 82
event-driven simulation mode 158
explicit pointer addition 115
external collector services 124

F
fast real time systems 68
FC 127-128, 131
fifth generation computers 24
flavours of pointer operation 124
FLIP operation 74
forwarding address 28, 73
free list 26, 123
freespace 73
Friedman 37
from-space 28, 72
Fugue 24, 82
full pointer deletion 124
full-panic mode 173
function 20
function type 128

G
garbage 26
Garbage collection 27
GCD 127-128, 131
GDC 52
GDH 52
GDP 52
generation number 29
Gi 127-128, 130
Glaser and Thompson 78
global machine state 18
global variable 18
graph traversal rule 85
GRIP 16
group descriptor 52
group descriptor count 52
group descriptor number 47, 52
group descriptor number block 50
group descriptor pointer 52
group_dispose_stack 96, 123
group handle 50, 95-96, 123, 125
group handle stack 50
group reference count 39, 92
 Górski 82

H
handle (see group handle)
handshake 84
handshake protocol 121
hard real time system 67
Hartel 157
hash link 48
head operation 102
heap 26
heap overflow 58
HECHT 137
HECHT-KASAI 137
Hewitt 29, 76
higher order functions 20
Hr/lw 127
Hudak 32, 47
Hughes 32, 37, 40, 72
hybrid collector 36
hybrid real time systems 68
hypercube architecture 16

I
Ichisugi 34
ICL DAP 15
Icon 82
ID 19
IL 137
IL reader 137
Illiac IV 15
imperative language 15, 17
incremental collection 84
INMOS 17
instruction-prefetch 15
intra-cyclic pointer deletion 45
invisible pointer 28
Index

**K**
KASAI 137  
Kung and Song 72

**L**
lambda calculus 19  
Lamport 72  
lazy evaluation 21  
lazy garbage collection 78  
Li 127-128, 130-131  
Lieberman 29, 76  
limited-panic mode 173  
limited-width fields 47  
linearising compaction 28, 32  
LISP 19, 23, 77  
liveliness 66  
Lm 127  
LO tag 48  
local count 39  
local count overflow 47  
logic languages 18  
loop assertion 153  
lowlink 39-40, 89  
Lr/w 127

**M**
Mago 24  
MAIN module 159  
MALPAS 137  
MALPAS intermediate language 137  
Malvern 137  
Manchester University 16  
mark-scan collection 28, 31, 70  
massively parallel processor 15  
maximal strongly connected cycle 38  
McCarthy 19, 70  
MIMD 16  
Minsky 28, 70, 72  
modified Tarjan search 90  
Modula-2 17  
Morris 27, 29, 69  
multi-processor architecture 82  
mutator interrupt 84-85

**N**
newapp 102, 105, 127, 129, 132-133  
newnode 102, 121  
Xg 127-128, 131  
Nilsen 24, 77, 82  
oisy heap operation 31  
on predictive sub-mode 160  
normal form 157

**O**
OCCAM 17  
ONE-ONE 137  
onstack 40, 54, 89, 91  
overflow 58  
overflow ratio 59  
overhead ratio 55, 58

**P**
panic mode 133  
parallel architecture 15  
parallel C 17  
parallel FORTRAN 17  
parent node 93, 102  
Peyton Jones 31  
pointer 27  
pointer deletion stack 85-86, 123  
pointer-bit 29  
pointer-reversal 31  
postcondition 139-140  
precondition 139-140  
predicate logic 18  
predictive sub-mode 160  
procedure 12, 101  
process_cyclic_free_list 118  
process_group_dispose_stack 118  
process_pointer_deletion_stack 118  
program maintenance 17  
Prolog 18

**R**
race conditions 34  
read barrier 74, 175  
real time systems 69  
recovery of cyclic structures 88  
redex 157  
reduced memory overhead 56  
redundant acyclic structures 84  
reference count 29, 33, 77  
reference counting garbage collection 29  
referential opacity 20  
referential transparency 20  
risk 140  
Rosser 19  
Rovner 77  
Runciman 30

**S**
S.C.C. (see strongly connected component)  
S.E.C.D. machine 23  
safe state 84, 86  
safety 66
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>same group</td>
<td>40, 89, 91</td>
</tr>
<tr>
<td>save space</td>
<td>30</td>
</tr>
<tr>
<td>scanned</td>
<td>91, 123, 135</td>
</tr>
<tr>
<td>scan state</td>
<td>106</td>
</tr>
<tr>
<td>scratch space</td>
<td>30</td>
</tr>
<tr>
<td>segment</td>
<td>49</td>
</tr>
<tr>
<td>semaphore primitives</td>
<td>70</td>
</tr>
<tr>
<td>semispaces</td>
<td>72</td>
</tr>
<tr>
<td>Sg</td>
<td>127-128, 131</td>
</tr>
<tr>
<td>side-effects</td>
<td>18, 21</td>
</tr>
<tr>
<td>SIMD</td>
<td>15</td>
</tr>
<tr>
<td>simple pointer operation</td>
<td>124</td>
</tr>
<tr>
<td>simulation</td>
<td>157</td>
</tr>
<tr>
<td>single instruction single data (SISD)</td>
<td>15</td>
</tr>
<tr>
<td>single-bit reference count</td>
<td>36</td>
</tr>
<tr>
<td>SISAL</td>
<td>19</td>
</tr>
<tr>
<td>SLIP</td>
<td>79, 84</td>
</tr>
<tr>
<td>slow real time system</td>
<td>68</td>
</tr>
<tr>
<td>soft real time system</td>
<td>67</td>
</tr>
<tr>
<td>SOLOMON</td>
<td>15</td>
</tr>
<tr>
<td>split queue</td>
<td>95, 97, 105, 123</td>
</tr>
<tr>
<td>spl_grp</td>
<td>105-107, 123</td>
</tr>
<tr>
<td>stack</td>
<td>123</td>
</tr>
<tr>
<td>static graph</td>
<td>39</td>
</tr>
<tr>
<td>status word</td>
<td>57</td>
</tr>
<tr>
<td>Steele</td>
<td>70</td>
</tr>
<tr>
<td>stop-the-world collection</td>
<td>32</td>
</tr>
<tr>
<td>storage</td>
<td>157</td>
</tr>
<tr>
<td>string reduction</td>
<td>24</td>
</tr>
<tr>
<td>strongly connected component (S.C.C.)</td>
<td>37</td>
</tr>
<tr>
<td>supercombinators</td>
<td>23</td>
</tr>
<tr>
<td>syntax tree</td>
<td>26</td>
</tr>
<tr>
<td>systolic array</td>
<td>16</td>
</tr>
<tr>
<td>transputer</td>
<td>17</td>
</tr>
<tr>
<td>trivial marker</td>
<td>46, 50, 54</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>unboxed representation</td>
<td>29</td>
</tr>
<tr>
<td>Unger</td>
<td>15</td>
</tr>
<tr>
<td>useful heap size</td>
<td>55</td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
<tr>
<td>VAL</td>
<td>19</td>
</tr>
<tr>
<td>value of an event</td>
<td>67</td>
</tr>
<tr>
<td>variable</td>
<td>18</td>
</tr>
<tr>
<td>vector nodes</td>
<td>33</td>
</tr>
<tr>
<td>vector pipeline</td>
<td>15</td>
</tr>
<tr>
<td>visit</td>
<td>39, 88, 127, 129, 132-133</td>
</tr>
<tr>
<td>visited</td>
<td>91, 124, 135</td>
</tr>
<tr>
<td>von Neumann architecture</td>
<td>15</td>
</tr>
<tr>
<td>von Neumann bottleneck</td>
<td>15, 18</td>
</tr>
<tr>
<td>W</td>
<td></td>
</tr>
<tr>
<td>Wadler</td>
<td>30, 70</td>
</tr>
<tr>
<td>Wakeling</td>
<td>30</td>
</tr>
<tr>
<td>weak pointer</td>
<td>37</td>
</tr>
<tr>
<td>weighted reference counting</td>
<td>34</td>
</tr>
<tr>
<td>Weizenbaum</td>
<td>79, 84</td>
</tr>
<tr>
<td>Winter</td>
<td>82</td>
</tr>
<tr>
<td>Wise</td>
<td>37, 77</td>
</tr>
<tr>
<td>Wong</td>
<td>76</td>
</tr>
<tr>
<td>Woodger's scenario</td>
<td>144</td>
</tr>
<tr>
<td>word-at-a-time programming</td>
<td>15</td>
</tr>
<tr>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Yonezama</td>
<td>34</td>
</tr>
</tbody>
</table>

T
- B.D. stack 78
- tag of a node 26
- tail pointer 102
- Tarjan search 39, 79, 88
- Tarjan stack 54-55
- Thomas 34
- Thompson 90
- threat 139-140
- time-step mode 158
- timeliness 66
- to-be-decremented stack (see T.B.D. stack)
- tospace 28,72-73
- total path accessor predicate 142
- TOTAL reduction 137
- traditional languages 17

-257-