VISOR++: A SOFTWARE VISUALISATION TOOL FOR TASK-PARALLEL OBJECT-ORIENTED PROGRAMS

Hendra Widjaja

A THESIS SUBMITTED FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE
IN THE DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ADELAIDE

March 1998

visor **n** 1. (*hist*) Movable part of a helmet, covering the face. 2. Peak of a cap. 3. (**'sun-**)~, oblong sheet of dark-tinted glass hinged at the top of a windscreen in a car to lessen the glare of bright sunshine.

(*Oxford Advanced Learner's Dictionary of Current English, by A.S. Hornby, Oxford University Press, 3rd edition, 1980, page 959.*)

VISOR Acronym for **Visual Instrument and Sensory Organ Replacement**. A remarkable piece of bioelectronic engineering that allowed Geordi La Forge to see, despite the fact that he was born blind. A slim device worn over the face like a pair of sunglasses, the Visor permitted vision in not only visible light, but across spectrum, including infrared and radio waves.

(*The Star Trek Encyclopedia, A Reference Guide to the Future, by M. Okuda, D. Okuda and D. Mirek, Pocket Books, New York, 1994, page 368.*)

Abstract

Applying software visualisation to task-parallel object-oriented programs poses interesting questions. The reason for this is that, typically, such programs exhibit complex behaviour as a result of the complex interaction among the program entities. Such interaction is caused, in part, by concurrency and distribution.

With the exception of a limited number of tools, many existing tools only focus on a narrow selection of language features for visualisation. However, to enable users to form a deep understanding, and subsequently fine-tune a program, a wide selection of such features is necessary for visualisation. Furthermore, multiple views depicting the program from multiple angles are also necessary.

This thesis describes Visor++, a tool for visualising programs written in CC++, a task-parallel, object-oriented language derived from C++. Visor++ provides a framework of visualising task-parallel object-oriented programs in the absence of language support for visualisation. In other words, Visor++ provides support for the visualisation of programs written in languages which are not "visualisation-conscious"; CC++ is one such language.

This thesis describes the techniques developed to enable the visualisation of task-parallel object-oriented programs by using a wide selection of language features. The effectiveness of this approach is testified by the experimentation with the tool. The design and experimentation with Visor++ are all described in this thesis.

Although the framework of Visor++ is implemented on specific platforms, it can also be applied to other similar systems.

Declaration

This is to certify that this thesis contains no material which has previously been accepted for the award of any degree or diploma in any university or other tertiary institution. To the best of my knowledge and belief, it contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

If this thesis is accepted for the award of the degree, permission is granted for it to be made available for loan and photocopying.

Hendra Widjaja

March 1998

Acknowledgements

First of all, I would like to thank my supervisor, Dr Michael J. Oudshoorn, for his advice, encouragement, and superb guidance during my candidature as a Master's student. He is the one who introduced me to the exciting field of software visualisation. I am also much indebted to Dr Jiannong Cao, who acted as co-supervisor during the first semester of 1995, particularly when Dr Oudshoorn was away from March to June 1995. His advice and guidance were of utmost importance during the infancy of the project.

In 1997, I presented a paper in San Jose, USA. Once again, I am much indebted to my supervisor, Dr Oudshoorn, for his tireless efforts in obtaining financial support for the trip. He is indeed a super "Visor". I also wish to thank Garuda Indonesia for providing special arrangements for the trip.

I owe special thanks to Prof. Peter Eades and Dr. Kang Zhang for reading the the thesis. Their comments are particularly insightful and invaluable, especially for further enhancements of the work.

Acknowledgment and thanks also go to the staff and members of the Department of Computer Science for all their help and support, particularly to Matthew, Stuart, Sam and Heath for their superb technical assistance. Special thanks to the DHPC (Distributed High-Performance Computing) group at the department for letting me print portions of the thesis by using their colour printer. Other members of the staff, the postgraduate students and visiting speakers have also contributed in many ways. Many thanks also to my office colleague, Lin Huang, for teaching me how to make dumplings.

I also wish to acknowledge the technical help I received that has made my work

possible. Special thanks to the CC++ language developers at California Institute of Technology, who made their system accessible, and provided much technical support through electronic mail by answering my questions, even the stupid ones. I would also like to express my special thanks to Professor John Stasko at Georgia Institute of Technology, who made his POLKA system publicly available, upon which my work is based.

Thank you, too, to all my friends, both at home in Indonesia, and here in Australia. They have made my stay in Adelaide more enjoyable. Many thanks, too, to the staff at CISSA (Council for International Students of South Australia), who have provided international students such as myself a chance to glimpse into the lives of Australians and the people of other cultures. I would also like to extend my thanks to AusAID for providing the Australian Development Cooperation Scholarship (ADCOS), without which my work and stay in Australia would not have been possible.

Finally, I would like to express my deepest gratitude to my parents and family, who have constantly supported and prayed for me from afar.

Contents

Abstract iii

Declaration iv

Acknowledgements v

1 Introduction 1

1.1 Motivation 5

1.2 Experimentation 8

1.3 Terminology 8

1.4 Thesis Structure 9

2 Software Visualisation 10

2.1 Aspects of Software Visualisation 10

2.1.1 Ideals 11

2.1.2 Purposes 12

2.1.2.1 Understanding 13

2.1.2.2 Debugging 13

2.1.2.3 Performance Analysis 15

2.1.3 Mechanisms 17

2.1.3.1 Data Collection 17

2.1.3.2 Visualisation 18

2.2 Visualisation Systems 20

2.2.1 Sequential Systems 21

2.2.2 Concurrent Systems 22
2.2.3 Concurrent Object-Oriented Systems 26
2.2.3.1 Visualisation of μC++ programs 26
2.2.3.2 Visualisation of pC++ programs 28
2.2.3.3 Visualisation of LAMINA programs 29
2.2.3.4 Visualisation of PARC++ programs 30
2.2.3.5 Observations 31
2.3 Summary 31

3 Overview of the CC++ Language **34**
3.1 The CC++ Language 34
3.1.1 Concurrency 35
3.1.1.1 Synchronous threads 35
3.1.1.2 Asynchronous threads 36
3.1.2 Synchronisation and Determinism 37
3.1.2.1 Synchronisation 37
3.1.2.2 Atomic Functions 37
3.1.3 Locality 39
3.1.3.1 Processor Objects 39
3.1.3.2 Remote Procedure Calls 40
3.2 Visualisation Support 43

4 Visor++ **45**
4.1 General Framework 46
4.2 Program Instrumentation Subsystem 48
4.2.1 Program Entities 49
4.2.2 Program Static Analysis 50
4.2.3 Program Instrumentation 52
4.2.3.1 Events 53
4.2.3.2 Location-ID 55
4.2.3.3 Instrumenting Functions 57

4.2.3.4 Instrumenting Destructors 62
4.2.3.5 Instrumenting Processor Object Classes 66
4.2.3.6 Instrumenting Synchronous Threads. 70
4.2.3.7 Instrumenting Asynchronous Threads. 73
4.2.4 Observations 74
4.3 Event Collection Subsystem 78
4.3.1 Establishing the Monitoring Environment 78
4.3.2 Collecting Traces 80
4.3.3 Consolidating traces 83
4.3.4 Observations 85
4.4 Event Visualisation Subsystem 87
4.4.1 POLKA 87
4.4.2 Architecture and Implementation 88
4.4.3 The Views 92
4.4.3.1 Design Considerations 93
4.4.3.2 Static Views 97
4.4.3.3 Auxiliary Views 101
4.4.3.4 Dynamic Views 102
4.4.4 Observations 119
4.5 Summary 119
4.6 Applicability to Other Systems 123

5 Using Visor++ **125**

5.1 Experiments 125
5.1.1 A Simple Example 126
5.1.2 Distributed Merge-Sort 127
5.1.3 Concurrent String Search 131
5.1.3.1 Implementation-1 131
5.1.3.2 Implementation-2 131
5.1.3.3 Implementation-3 135
5.1.4 An Electronic Transaction System 136

5.1.4.1 Implementation-1 138
5.1.4.2 Implementation-2 140
5.1.4.3 Implementation-3 146
5.1.4.4 Implementation-4 148
5.2 Discussion 151
5.2.1 Merits 151
5.2.2 Limitations on Visor++ Usage 152

6 Conclusions and Future Work 155
6.1 Summary 155
6.2 Conclusions 158
6.3 Future Work 158

A An Instrumentation Example 161
A.1 The Original Code 161
A.2 The Instrumented Code 163

B Transaction Subsystem Code 167
B.1 The Transaction Requests 167
B.2 The Stock and Customer Databases 168
B.3 The Business Logic Unit 170
B.4 The Transaction Servers 170
B.5 The Transaction Resolution Subsystem 171

List of Tables

5.1 Timing information from the merge-sort program. 129

5.2 Timing information from the parallel text-searching programs. 135

5.3 Timing information from Implementation-3 and Implementation-4. . . . 149

List of Figures

1.1 *Data visualisation of air flow near the airfoils of an aircraft.* 2

1.2 *Constructing a procedure in Hyperpascal.* 3

1.3 *The* **Processor Communication View** *in ParaGraph.* 4

1.4 *Algorithm animation of the Tower of Hanoi, using POLKA.* 5

1.5 *Mental models in software or program comprehension.* 7

2.1 *Aspects of software visualisation.* 10

2.2 *The* **Classy** *tool in the TAU visualisation toolset.* 14

2.3 *The* **Coarse Grained View** *in TPM.* 14

2.4 *The control panel of the parallel debugger* **Breezy** *in TAU.* 15

2.5 *Visualising data structures in* **Breezy**. 16

2.6 *The* **Utilisation View** *in ParaGraph.* 16

2.7 *Transformations to produce software visualisation.* 17

2.8 *The different views in visualisation tools.* 20

2.9 *The* **Spacetime Diagram** *in ParaGraph.* 23

2.10 *Statistics of communication traffic in ParaGraph.* 24

2.11 *The* **History View** *and the* **Mutex View** *in Gthreads.* 24

2.12 *Trace visualisation in MVD (POET).* 27

3.1 *Usage of the* **par** *construct.* 35

3.2 *Implicit synchronisation barrier at the end of a* **par** *or* **parfor** *block.* . . 36

3.3 *Usage of the* **parfor** *construct.* 36

3.4 *Usage of the* **spawn** *construct.* 37

3.5 *Usage of the* **sync** *construct.* 38

3.6 *Usage of the* **atomic** *construct.* 38
3.7 *Mapping of threads and processor objects in CC++.* 39
3.8 *Implementation of a processor object class.* 40
3.9 *Usage of global pointers.* 41
3.10 *Data communication and RPC between two processor objects.* 42
3.11 *A transfer function.* 43

4.1 *General framework of Visor++.* 46
4.2 *Information on functions in the static repository.* 50
4.3 *Information on classes in the static repository.* 51
4.4 *Ill-partitioned, but valid program.* 52
4.5 *Automatically obtaining a unique identifier for a processor object.* . . . 58
4.6 *A light-weight atomic function to obtain unique thread identifiers.* . . . 59
4.7 *Program trace structure.* 59
4.8 *Example of function instrumentation.* 61
4.9 *A function can have many exit paths.* 62
4.10 *Profiler class in for the instrumentation subsystem.* 63
4.11 *A function instrumented with an* **EntityProfiler** *object.* 64
4.12 *Example of an altered function call.* 65
4.13 *Constructor and destructor called from different threads.* 66
4.14 *Instrumentation of a class.* 67
4.15 *Changing the call to an object destructor.* 68
4.16 *Instrumenting a member function of a processor object class.* 69
4.17 *Marking the start and end of a RPC.* 70
4.18 *The generation of events for a RPC.* 71
4.19 *A drawback in RPC instrumentation.* 71
4.20 *Simple re-arrangment of source-code for RPC instrumentation.* 72
4.21 *Instrumentation of a* **parfor** *or* **par** *block.* 73
4.22 *Instrumenting a* **parfor** *block and its threads.* 74
4.23 *Instrumenting a* **par** *block and its threads.* 75
4.24 *Instrumenting a* **spawn** *block.* 76

4.25 *An unfortunate consequence of introducing a new scope.* 76
4.26 *Simple re-arrangement of source code as a way out.* 77
4.27 *Allocation of monitoring processes.* 79
4.28 *Collecting program traces.* 81
4.29 *Consolidating program traces.* 84
4.30 *Causality violations in event ordering.* 86
4.31 *The corrected DAG reflecting the "happened-before" relationships.* . . . 86
4.32 *Classes in POLKA, and their* **has-a** *relationships.* 88
4.33 *Example program of the animation of a circle.* 89
4.34 *The event visualisation subsystem.* 91
4.35 *Visor++ in execution.* 94
4.36 *Relationships among views.* 95
4.37 *The* **Source-Code View**. 98
4.38 *The* **Class Hierarchy View**. 99
4.39 *The* **Class Information View**. 100
4.40 *Information on a node.* 101
4.41 *Information on a processor object.* 101
4.42 *The* **Thread View**. 103
4.43 *The* **Status Information View**. 104
4.44 *Auxiliary view showing information of a function.* 105
4.45 *The* **Navigation View**. 107
4.46 *The* **RPC Statistics View**, *with the option menu popped up.* 109
4.47 *Auxiliary view on the number of RPCs from one PO to another.* 110
4.48 *The* **Processor/Processor-Object Activity View**, *with its option menu popped up.* 111
4.49 *Auxiliary view from the* **Processor/Processor-Object Activity View**. 112
4.50 *Program execution in a processor-object.* 113
4.51 *Two processor-objects executing on a processor.* 114
4.52 *The* **Function Usage View**. 116

4.53 *Selecting a function in the* **Function Usage View** *brings up further information.* . . . 117
4.54 *The* **Composite Function View**. . . . 118
4.55 *The auxiliary view from the* **Composite Function View**. . . . 118

5.1 *The* **Thread View** *of the simple master-slave program.* . . . 126
5.2 *Merge-sort using a 4-level binary tree. The left figure shows the initial placement of the nodes on processors, and the right figure the optimised placement.* . . . 127
5.3 *The* **Processor/Processor-Object Activity View** *reveals the inefficiency of processor usage.* . . . 128
5.4 *The merge-sort program is heavy with RPC activity.* . . . 129
5.5 *Higher efficiency in processor usage after program tuning.* . . . 130
5.6 *The* **Thread View** *shows that the master-PO is idle while the slave-POs are executing.* . . . 132
5.7 *The* **Source-Code View** *shows the source-code area where the master-PO is blocked.* . . . 133
5.8 *The* **Thread View** *of* **Implementation-2**, *with the compute-thread.* . . 134
5.9 *The interrupt-signaling mechanism in work.* . . . 136
5.10 *Three-tier architecture for the electronic transaction system.* . . . 138
5.11 *The* **Thread View** *shows the unfairness of transactions.* . . . 139
5.12 *Auxiliary view showing a transaction request being handled by a TS-PO on the machine "achilles".* . . . 141
5.13 *The* **Source-Code View** *reveals the inadequacy of the implementation of the transaction server.* . . . 141
5.14 *The* **Function Usage View** *reveals those functions which are heavily used or take much time to execute.* . . . 143
5.15 *Together with the* **Source-Code View**, *the* **Class Hierarchy View** *shows that the time-stamp subsystem does not depend on the implementation of other classes.* . . . 144

5.16 *The* **Class Information View** *of the class "TimeStampClass", displayed upon clicking the associated node on the* **Class Hierarchy View**. 144

5.17 *The* **Composite Function View** *shows which functions can be optimised for each processor-object.* 145

5.18 *This auxiliary view is the result of selecting the longest function bar in the second row from the top of the* **Composite Function View**. . . . 146

5.19 *The* **Function Usage View** *now reveals that the functions are within reasonable frequency and average time of execution.* 147

5.20 *The* **Function Usage View** *reveals approximately similar function profiles to those in Implementation-3.* 150

B.1 *Definition of the transaction request class.* 168

B.2 *The customer database class.* 168

B.3 *The stock database class.* 169

B.4 *Definition of the business logic unit.* 170

B.5 *Definition of the transaction server.* 171

B.6 *Definition of the transaction resolution subsystem.* 172

Chapter 1

Introduction

A picture speaks a thousand words. This proverb supports the belief that pictures have been used by mankind to convey information more effectively than words. Before the invention of written texts, pre-historic cavemen used pictures as a communication medium. Even after writing was invented, pictures were, and still are, widely used. Nowadays, engineers and architects, for example, use pictures to convey their ideas and to communicate with their colleagues.

The idea of conveying information through pictures also extends to the computing milieu to support new forms of human-computer interactions. Supported by the advances in computer graphics technology, such interactions are possible because they take advantage of the human visual capability, which can generally discern pictorial information better than textual information [45, 113]. These new forms of interaction include *visualisation*, which is concerned with the use of computer-generated graphical representation in the computing milieu [113].

Visualisation is a broad field which can be divided into three categories: scientific data visualisation, visual programming, and software visualisation [45, 109, 113].

Scientific data visualisation, sometimes called *scientific visualisation*, deals with the graphical representation of bulk scientific data. Such data can be acquired through experiments, or through simulations. To display the data within the limited real estate of a computer screen, the data is transformed, for instance, by data aggregation. Scientific visualisation is deployed in various fields such as meteorology, aeronautics, and

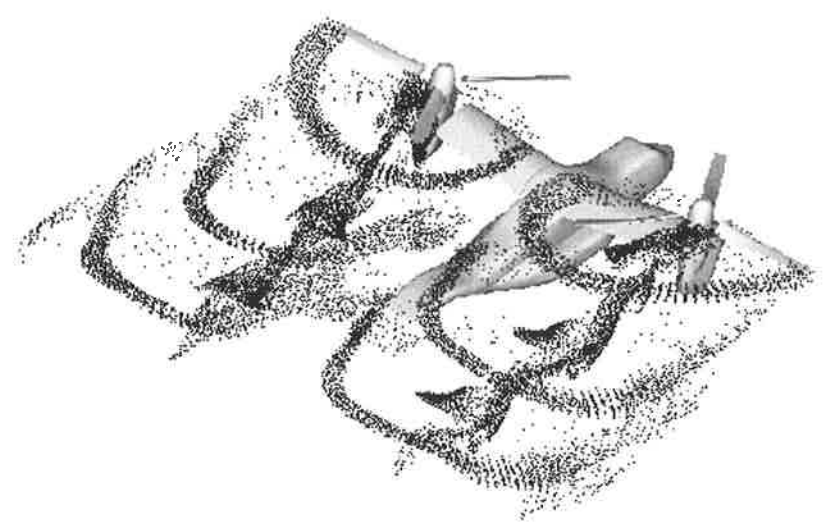

Figure 1.1. *Data visualisation of air flow near the airfoils of an aircraft.*

chemistry. Often, the graphical representations depict the real-world physical aspects of the data. For example, the huge numerical data relating to unsteady air flow near the airfoils of an aircraft can be represented on the screen as particle dots and lines. Figure 1.1 is the graphical rendering of such data, based on the position and other scalar quantities of the particles [85].

The second field, *visual programming*, is concerned with the use of graphical artifacts to specify or create computer programs. Through visual programming, iconic or graphical representations of language features can be manipulated interactively in some specific manner [93]. For example, a visual programming language may provide icons which represent program units (such as procedures), and other icons which represent control structures. A programmer then composes a program by using these icons, and by manipulating them graphically. Some examples of visual programming systems are HyperPascal [90] and Hence [9].

Figure 1.2 shows the construction of a procedure in Hyperpascal. This figure shows the graphical icons a programmer must use to create a procedure for the iterative multiplication of two integers, x and y. The topmost node, labeled *product* is the procedure to be created. The two boxes on the left of this node are the two input values. The other box on the right is the output type. The boxes above the *product*

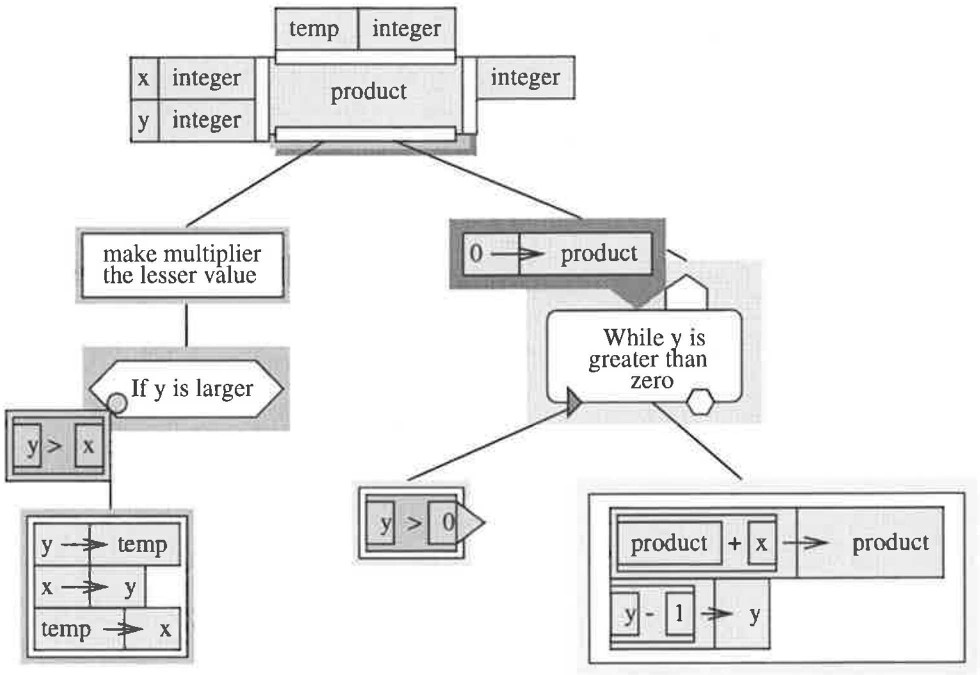

Figure 1.2. *Constructing a procedure in Hyperpascal.*

node specify entities local to the procedure. The two links extending from below the node specify two actions to be executed sequentially.

The third field, *software visualisation*, is concerned with the use of graphical constructs and methods to represent both the static and the dynamic aspects of software [109]. Software visualisation generically encompasses some other terms, whose definitions somewhat overlap with one another.

- *Program visualisation* is defined as the use of graphical artifacts to enhance the understanding of programs [45, 109, 113]. In program visualisation, the focus is usually on the graphical representations of program execution. The execution states, or elements of a program, such as function calls, or changes in data structures, can be represented as graphical icons which interact with one another on the computer screen. Several tools, such as Ovation [39], TraceViewer [73], and ParaGraph [69] are in this category. Figure 1.3 shows a communication diagram from ParaGraph which depicts the instantaneous communication among processors on which a message-passing program executes. Each processor is assigned a colour to indicate whether it is busy, idle, transmitting data, or receiving data.

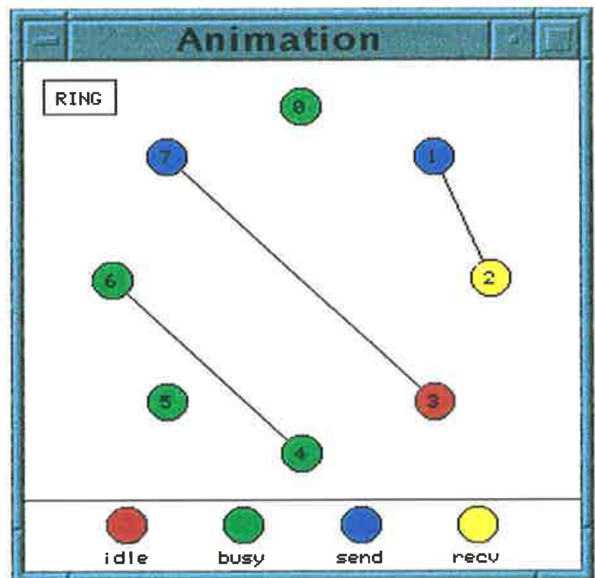

Figure 1.3. *The* **Processor Communication View** *in ParaGraph.*

- *Algorithm animation* is focused on the use of animation to depict a high-level description of algorithm operation. For example, the behaviour of a sorting algorithm inside a program can be represented as the animation of the movements of the data being sorted [5, 18]. Figure 1.4 shows the algorithm animation of the Tower of Hanoi through the use of the POLKA [122] toolkit. Other tools such as Zeus [18] and BALSA [22] are also examples of algorithm animation tools.

- *Code visualisation* is concerned with the graphical representation of the structures and elements of the source-code. This includes, for example, static program structures, class hierarchies in object-oriented programs, or the actual program text itself. The tools PIE [86], and TAU [17, 97] include code visualisation as part of their framework. Some other similar tools include SEE [6], and the work by Ball [7].

The above three fields have a common denominator in that they use graphics. However, the emphases are different. Scientific visualisation places emphasis on the representation of scientific data; visual programming focuses on program or software specification; and software visualisation highlights the static and dynamic aspects of a program or software system.

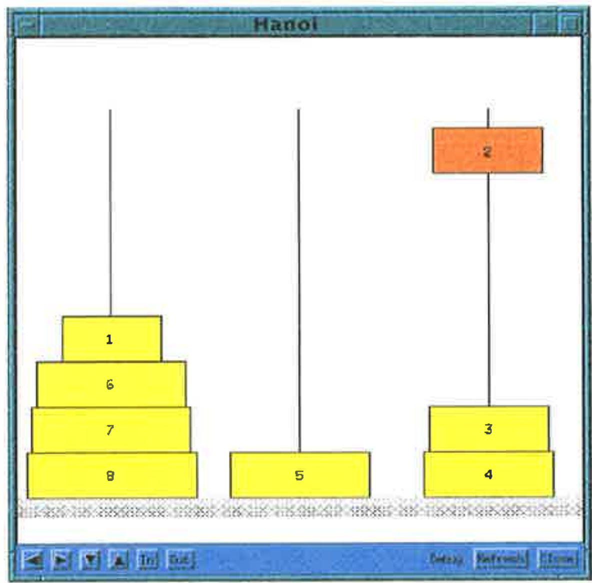

Figure 1.4. *Algorithm animation of the Tower of Hanoi, using POLKA.*

Scientific visualisation has proved successful and effective in presenting many types of scientific and engineering data to aid human comprehension. This is partly because scientific data usually has a natural correspondence to the physical world [71]. This is not so in visual programming and software visualisation. Software visualisation is of particular interest, because it is an attempt to provide users with a mental model of software execution, which can then be used for understanding, or for identifying defects in the software (see the next section). In addition, software visualisation is also interesting because it can be used for a variety of purposes, such as understanding, performance tuning, and debugging. Each purpose requires a different approach to ensure appropriate presentation. The focus of this thesis is software visualisation.

1.1 Motivation

Software visualisation can be used in different hardware and software domains [45, 103]. For example, it can be used for the visualisation of sequential, concurrent, or object-oriented software. In turn, the software can be executed on single-processor systems, tightly-coupled systems, or loosely-coupled multi-processor systems.

The usage of software visualisation is motivated by the fact that software execution

is usually a complex interaction of a collection of abstract objects which can be difficult to fathom [22]. It has been shown that in order to understand a program or software system, programmers and users form mental models which represent the software [51, 106]. This mental model formation generally relies on a textual representation, such as source code listings, and software design documents. However, software visualisation can be used to assist in the formation of this mental model, and to help confirm or strengthen any existing models. Software visualisation may also give users more insight into the static and dynamic aspects of the software.

The usage of software visualisation itself can be put into the framework of Figure 1.5 [103]. Firstly, a programmer creates a mental model of a program or software which is being developed [90, 106]. This original mental model is then transformed into the corresponding static model by using tools, such as text editors, CASE tools, and compilers. In this context, a static model comprises source code, the associated object code, and other intermediate code forms. After the static model is created, it is mapped into the corresponding dynamic model, which depicts the run-time behaviour of the program. The mapping is performed by using the program environment run-time control, such as operating systems and run-time libraries. This dynamic model can, furthermore, be mapped into the programmer's execution mental model, which is a mental representation of the program execution. This mapping is important [40], because it enables the programmer to superimpose the execution mental model onto the original mental model, which was conceived during the creation of the program. This makes it possible to understand, or identify defects in the program.

Software visualisation can be effective if it assists programmers or users of software visualisation tools to form or strengthen their mental models. However, unlike scientific visualisation, software visualisation usually does not have any apparent physical world correspondence[1] [71]. Therefore, in software visualisation, the choice of graphical

[1]Scientific visualisation usually has a clear physical world correspondence. For example, bulk scientific data on ocean currents (such as salinity, velocity, and pressure) can be naturally represented as ocean currents on the computer screen [74]. Another example is the graphical representations of the unsteady air flow near the airfoils of an aircraft (see Figure 1.1). Software execution, on the other hand, typically has no such natural correspondence. Therefore, different software visualisation tools represent, for example, function execution in widely different ways.

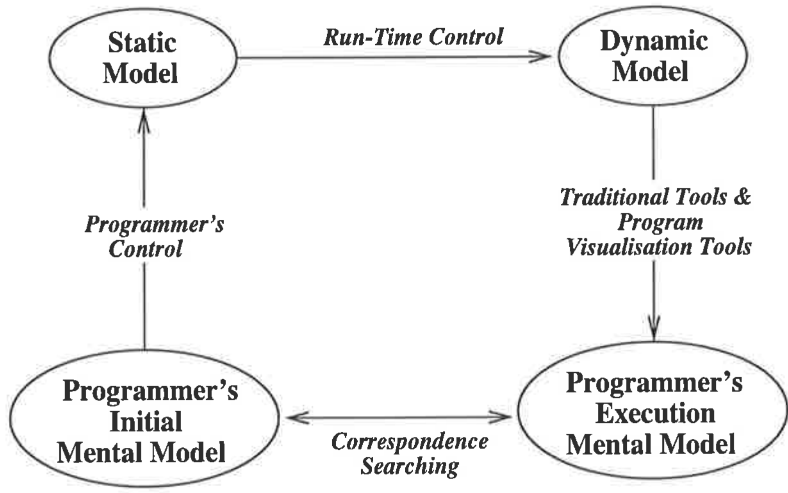

Figure 1.5. *Mental models in software or program comprehension.*

representation must be made carefully.

The above problem is exacerbated by the fact that software visualisation can be used in different combinations of hardware and software domains, such as the domain of distributed object-oriented programming.

This thesis examines program visualisation for concurrent object-oriented programs, particularly declarative task-parallel programs. This paradigm is chosen partly because it is relatively new, and little research on program visualisation in this area has been conducted. Consequently, there is still much to explore. In this regard, this thesis attempts to answer the following questions:

1. In order to help to form or strengthen users' mental models, how to best visualise task-parallel object-oriented programs? This is important, since such programs typically contain a rich collection of semantics and features [11, 23, 35, 129].

2. To ensure greater portability, what methods can be used to provide visualisation? Furthermore, how much information can be obtained from the programs, and what types of views can be generated?

1.2 Experimentation

To answer the above questions, a visualisation tool, called Visor++ (Visualisation of Software in the Task-Parallel Object-Oriented Realm)[2] is developed. It is used to visualise programs written in the CC++ language [32, 33]. This language is a declarative language which supports object-oriented task-parallel computation. Although the whole system, by implementation, is specific to CC++, nevertheless the findings uncovered are applicable to other similar systems or languages.

By nature, CC++ is a language of combined paradigms, i.e. it combines the concurrent, and the object-oriented paradigms. Furthermore, CC++ programs can be distributed. In this regards, the thesis examines the language and the manner in which CC++ programs can be sensibly visualised.

CC++ does not provide direct support for visualisation. In particular, it does not provide run-time data which can be used to produce sensible visualisation. Furthermore, CC++ is a declarative language, in the sense that, for example, thread or task invocation is implicit. Compared with imperative-style languages, it is more difficult to produce the necessary run-time data. Therefore, the thesis also presents the methods by which such data to drive the visualisation process can be produced. The methods are realised at the source-code level to ensure greater portability.

1.3 Terminology

The terms which are used throughout this thesis are explained below.

The terms **visualisation system**, **visualisation tool**, and **tool** are interchangeably used to refer to programs, tools, or systems which provide software visualisation or program visualisation.

A **programmer** is a person who writes or maintains a piece of software or program. This term also refers to the person who reads and tries to understand a program. On the other hand, the term **user** refers to the user of a visualisation tool, while a **visualiser**

[2]As the references on page *ii* suggest, in other contexts, the term "Visor" also refers to devices which are used to help the human vision or visual information processing.

is a person who produces a visualisation tool.

For the sake of simplicity, the term **concurrent systems** is used to denote systems employing concurrency and parallelism, including distributed systems[3].

The term **task-parallelism** implies the creation of independent threads of control, including synchronisation and communication among the threads. A closely related term is **data-parallelism**, which entails the synchronous/parallel application of a sequential or atomic operator over a set of operands [35, 38]. In connection thereof, **concurrent object-oriented** paradigms refer to both task-parallel and data-parallel object-oriented paradigms.

1.4 Thesis Structure

Chapter 2 examines some aspects of software visualisation tools. Some representative visualisation systems are also described. These include tools for sequential, concurrent, and concurrent object-oriented software. The focus of this chapter, however, is more on *program visualisation*. Chapter 3 provides an overview of CC++. Chapter 4 discusses the framework and architecture of Visor++, along with its implementation. Chapter 5 gives some examples of how Visor++ can be used to assist users in understanding the static and dynamic aspects of CC++ programs, and subsequently to carry out some program analysis and tuning. An evaluation of the Visor++ is also undertaken. Finally, Chapter 6 provides some concluding remarks.

[3]Following the definition by Fidge [49], the three terms can be defined as follows. *Concurrency* is "the logical concept of two or more actions appearing to occur at the same moment. *Parallelism* is the actual embodiment of this concept via a system which is physically capable of performing more than one action at a time ... A *distributed system* is a hardware architecture incorporating multiple processing units, and thus capable of implementing parallelism ..."

Chapter 2

Software Visualisation

2.1 Aspects of Software Visualisation

In general, software visualisation has three inter-related aspects (see Figure 2.1) [103]. Firstly, it has some general ideals which can be achieved. Secondly, it may be used for different purposes. Finally, based on the ideals and purposes, software visualisation is generated by using some specific mechanism. For each programming paradigm, and possibly for different application domains, these purposes, ideals, and mechanisms may differ. However, they generally share some common characteristics. The three aspects are described below.

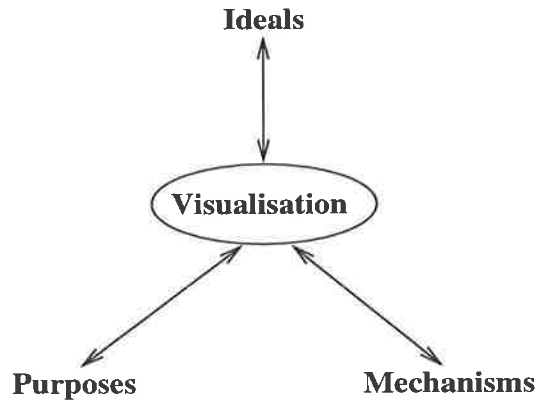

Figure 2.1. *Aspects of software visualisation.*

2.1.1 Ideals

The ideals of software visualisation may vary under different circumstances. Furthermore, since software visualisation involves the use of graphics, these ideals are subjective. Nevertheless, a small set of general ideals may be shared among the different tools [39, 79, 103, 109].

The ideals of visualisation can be loosely divided into two perspectives: the system's perspective, and the user's perspective. From the system's perspective, the ideals are as follows:

1. **Scalability.**

   A tool should be sufficiently scalable in terms of the problem size it can handle. For example, with a parallel program, the visualisation should be scalable in terms of the number of processors involved in the computation.

2. **Extendibility.**

   The mechanisms implemented in the tool should be flexible enough to accommodate new concepts or ideas, and to be changed accordingly. In a practical sense, this means that new views or displays can be added or changed relatively easily.

3. **Portability.**

   Portability means that neither the implementation nor the concepts of the tool significantly obstruct its applicability to other platforms. Minor changes are to be expected as different platforms may require slightly different implementations.

From the user's perspective, a visualisation tool has the following ideals:

1. **Minimal disturbances to the execution of user's program.**

   To drive visualisation, information must be obtained from a program. Obtaining such information may generate *probe effects* (Section 2.1.3.1) as the program executes. These probe effects are often inevitable [82]. However, it is desirable that they be kept to a minimum.

2. **Little or no programmer's intervention.**

   The generation of visualisation events should be automated as far as possible so that little or no programmer's intervention is necessary. This requirement stems from the fact that manually generating visualisation is both tedious and error-prone, especially for programs under development.

3. **Handle real-world problems.**

   One good way to test or measure a tool is to examine the size and nature of the problems it can handle. It is desirable that the tool can handle real-world problems. If this cannot be achieved, then at least an extended version of the tool should be able to achieve the same effects.

4. **Present the "right" things to the user.**

   From the perspective of the user interface, a good tool provides its user with the "right" information in the "right" way. What constitutes the "right" information and display mechanisms is subjective, and depends on the purpose served by a tool. In any case, a tool should help its users form or strengthen the mental models [31, 51] of the program being visualised. Providing a variety of views to present the same information goes some way to ensuring that the "right" views are available [41].

2.1.2 Purposes

Software visualisation and, in particular, program visualisation, can be used for several purposes. These purposes serve to specify the nature of the correspondence searching between the programmer's initial mental model and the execution mental model (see Figure 1.5).

In general, the majority of the purposes for software visualisation can be categorised into: understanding, debugging, and performance tuning.

2.1.2.1 Understanding

In terms of mental models, when used for understanding, software visualisation serves to help programmers *form* the mental models of the software. As described by DePauw [40], a visualisation tool supporting this purpose bridges the gap between the static specification and the software run-time behaviour. This is especially useful in the program development and maintenance process.

Different software aspects, i.e. static and dynamic aspects, can be visualised for the purpose of understanding. Visualisation of static aspects includes the visualisation of static elements, such as source-code, or in the case of object-oriented programs, the static class hierarchies. Tools such as Zeus [18] and BALSA [22] provide source-code views. Another example is the tool LOOK! [133], which displays static class hierarchies in C++ programs. Figure 2.2 shows the tool Classy, for browsing the class hierarchies in pC++ [15] programs. Classy itself is part of the TAU visualisation toolset [97] for visualising pC++ programs.

The visualisation of the dynamic aspects of software/program execution may take a variety of forms, such as the high-level animation of algorithms, and symbolic views of code execution. For the animation of algorithms (see Chapter 1), only a high-level description of the algorithm operation is visualised. Symbolic views of code execution, on the other hand, depict entities in a program by using symbolic icons. For example, a function may be represented as a coloured circle. Interactions among program entities are then visualised (and often, animated) on the screen as interactions among these symbolic entities. As an example, Figure 2.3 depicts the Coarse Grained View from the Transparent Prolog Machine tool (TPM) [44]. The view depicts the execution of a Prolog program as an AND/OR tree, in which nodes represent the goals of the program.

2.1.2.2 Debugging

Used for debugging, software/program visualisation serves to help programmers form program execution mental model, and *compare* it with the initial mental model. Discrepancies found can then be used to correct the program. In other cases, software

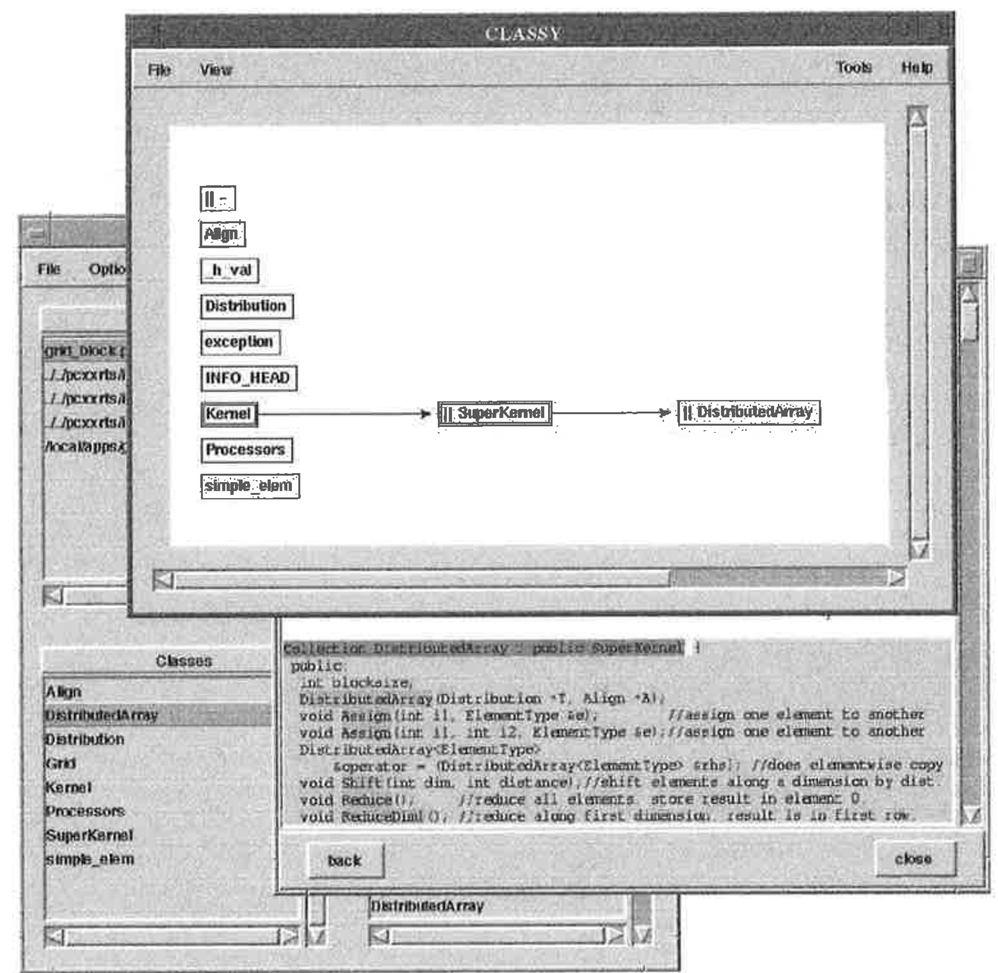

Figure 2.2. *The* **Classy** *tool in the TAU visualisation toolset.*

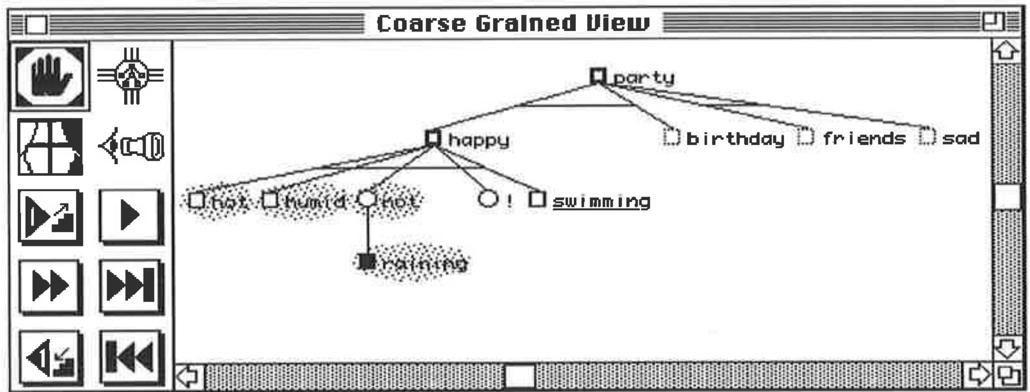

Figure 2.3. *The* **Coarse Grained View** *in TPM.*

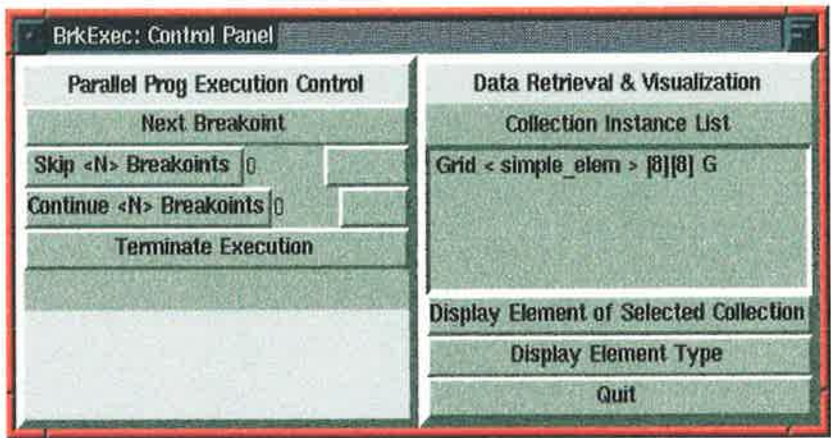

Figure 2.4. *The control panel of the parallel debugger* **Breezy** *in TAU.*

visualisation can be used to help users to reconstruct the original mental model (i.e. the "reverse-engineering" of the mental model). Such debugging by using software visualisation is sometimes called *visual debugging.*

One example of a visual debugging system is the tool Breezy [17], which is part of the TAU [97] tool system. Figure 2.4 shows the control panel of Breezy, while Figure 2.5 shows the display of the Breezy interactive window through which users can select portions of the data structures to view. Other examples include the tool KAESTLE [12] for visualising the dynamic list data structures in LISP programs, and CenterLine ObjectCenter [109] for visually debugging C++ programs.

2.1.2.3 Performance Analysis

Software visualisation can also be used for performance analysis. Such analysis is often carried out on parallel programs, for performance is generally the driving force for using parallel computation [104]. To be effective, performance visualisation tools should provide displays which provide insight into the relationship between a program's performance characteristics and its structure and operation [70]. Examples include tools such as ParaGraph [69] and AIMS [142] for the performance analysis of parallel (message-passing) programs. Figure 2.6 shows the Utilisation View in ParaGraph. The view displays the utilisation of processors in the execution of a message-passing program. It shows the amount of time processors spend performing useful computation

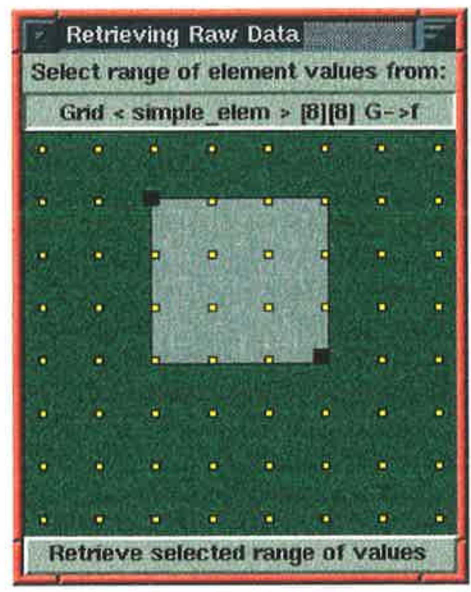

Figure 2.5. *Visualising data structures in* **Breezy**.

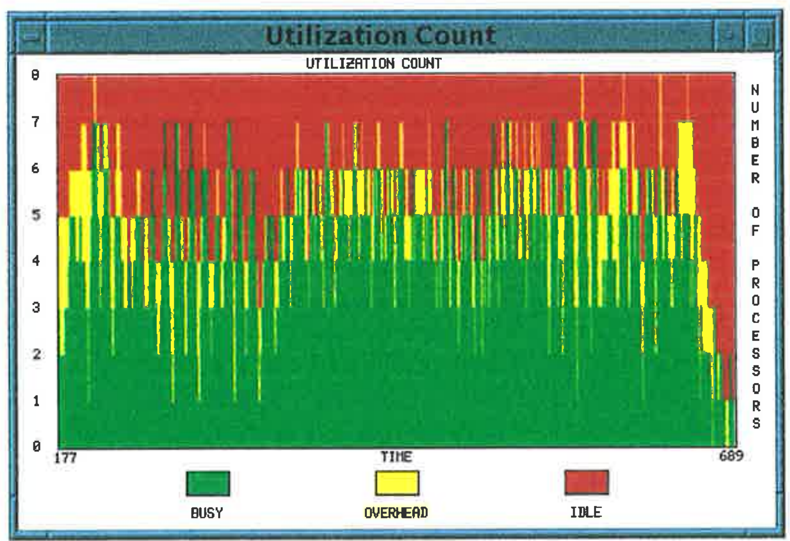

Figure 2.6. *The* **Utilisation View** *in ParaGraph.*

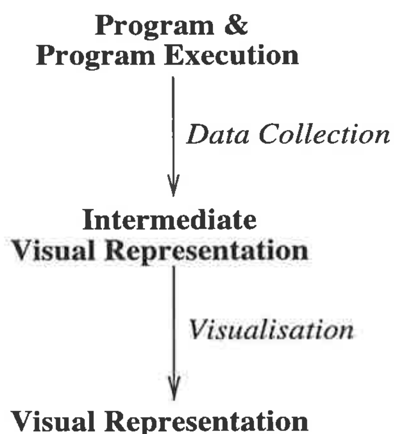

Figure 2.7. *Transformations to produce software visualisation.*

(being busy), performing computation overheads, and being idle.

2.1.3 Mechanisms

Following [103] and [113], software/program visualisation can be regarded as the mapping of programs or their execution into their visual representations (see Figure 2.7). To do this, several phases are followed. The first phase involves the collection of data to drive the visualisation. This phase is termed *data collection*. The data is then used in the next phase, *visualisation*, to produce and drive the final graphical representation. These two phases are discussed in the following sections. More detailed information can be found in [82] and [134].

2.1.3.1 Data Collection

Data collection involves the collection of data used for producing visualisation. The visualisation-driving data may be obtained, for example, from the program's source code, and from the program's execution. Normally, the data is then processed into an internal format, suitable for visualisation. This internal format is referred to as the *intermediate visual representation* (IVR).

Obtained from source code, the visualisation-driving data represents the software or program static structures, such as program source code and its block structures. In the case of object-oriented programs, the data could be the static class hierarchies.

On the other hand, the visualisation-driving data can also be derived from the program execution. To achieve this, a tool must have access to information on the program run-time states. Installing "probes" into the program is the most common approach. During program execution, these probes emit the program run-time states in the form of trace records. The trace records, possibly with the program static structures, are processed into the intermediate visual representation.

Installing probes into a program has its own difficulties. In particular, it produces a side effect known as *probe effects* [24, 82, 134][4]. The probe effects typically refer to the alteration of the duration of program execution due to the probe execution. In this context, the alteration means an additional timing factor. In multi-threaded programs, such timing variation may well cause the relative ordering of thread execution to be altered. Probe effects sometimes also refer to the alteration of the frequency of inter-task synchronisation errors [60] due to the probe execution. In this context, the synchronisation errors already exist in the program before the probes are installed. In this thesis, the latter type of probe effect is termed a *synchronisation-error effect*, while the previous is referred to as a *timing effect*.

Both synchronisation-error effects and timing effects cause the captured traces of a program to reflect only the states of the altered program. When the traces are used to drive the visualisation of the program, an incorrect interpretation of the program execution may occur. Therefore, the probes must be designed such that the program perturbation is minimal.

2.1.3.2 Visualisation

After the intermediate visual representation is obtained, it is mapped into views in which different types of graphical representations may be used. Graph-based views and statistics-based views are, perhaps, the most commonly used. Using graph-based views, software/program entities are represented as nodes, with their interactions as edges. Examples include Zeus [18], Virtual Images [132], GraphTrace [80], Ovation [39], and TPM (see Figure 2.3). For statistics-based views, graphical representations such

[4]Probe effects are sometimes also jokingly referred to as "Heisenbugs" [104].

as bar graphs and matrix-like views are used. Tools such as PIE [86], Pablo [4], and Traceview [91] incorporate such views (Figure 2.6 is the Utilisation Statistics View in ParaGraph [69]). Some other types of these views are novel or very specific to their environment. For example, the tool SIEVE [114] displays parallel program performance data using spreadsheet abstractions. Another example is the tool Voyeur [119], which uses fish and shark figures to symbolise performance data values.

The visualisation phase can be carried out concurrently (*on-the-fly*) with the data-collection phase. On the other hand, the two phases can also be executed separately in that the visualisation phase is carried out *after* data collection. Such a processing scheme is known as a *post-mortem* approach.

The on-the-fly approach has the advantage that the visualised software/program and the visualisation system interact directly [34, 81, 82]. This mechanism is particularly desirable for the debugging of concurrent programs [24, 47, 105, 139]. However, it also means that the monitored program may experience more timing effects or synchronisation-error effects due to the interaction. Furthermore, the rapid torrents of trace data may force the user to slow down the pace of the visualisation tool, thereby slowing down the monitored program as well [41]. Therefore, on-the-fly visualisation has to be carefully designed so as to minimise the probe effects. In contrast, with the post-mortem approach, the visualisation is devised by using traces collected during program execution. It has the advantage of providing the user with program traces which can be repeatedly analysed without having to re-execute the program. This results in relatively less program perturbation. However, the disadvantage is that it does not allow a direct interaction with, and manipulation of, the monitored program. Either the on-the-fly or the post-mortem approach may be more suitable for a particular visualisation scheme.

The above discussion provides a general overview of the various ideals and purposes of software/program visualisation and the general mechanisms by which visualisation can be devised. The discussion applies to many language paradigms, including the concurrent object-oriented paradigm, the focus in this thesis.

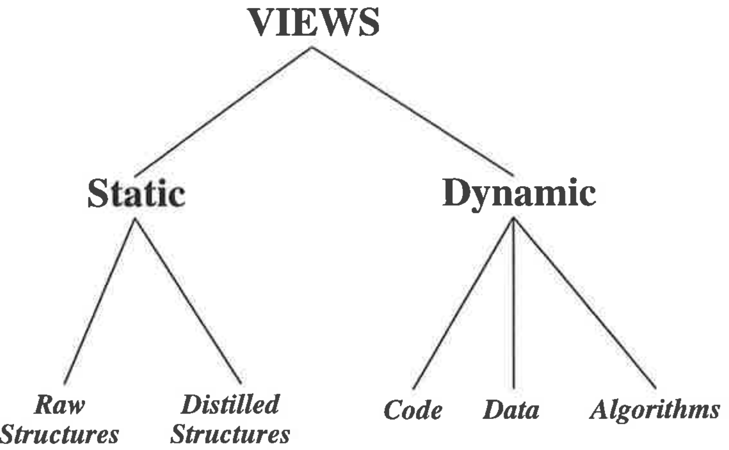

Figure 2.8. *The different views in visualisation tools.*

2.2 Visualisation Systems

In the light of the discussion in Chapter 1, software visualisation can be realised in a variety of forms. In practical terms, this means a variety of views can be constructed. The different views can be loosely categorised as in Figure 2.8. The discussion in this section places more emphasis on *program visualisation.*

A tool can provide static and dynamic views. Static views refer to those which display a program in terms of its source code. The views can display *raw structures*, in which the "raw" form of the source code is used. Source-code browsers, and pretty-printing of source code are in this category. Alternatively, *distilled structures* can also be used. The source code can be "distilled" to produce, for example, class hierarchy diagrams (for object-oriented programs) [97, 133], Nassi-Schneiderman diagrams [20], or code block structures [20, 36].

Dynamic views, on the other hand, are the views which depict program dynamic execution. Following the subdivision of software visualisation (Chapter 1), the views may pertain to *code*, *data*, or *algorithms*. Code views are those depicting code execution. Data views are those concerned with the dynamic changes in data structures and their contents during program execution, while algorithm views depict high-level algorithmic views of the program.

The above categorisation extends to the visualisation of concurrent object-oriented systems. Such systems are, to some extent, related to sequential, object-oriented and

concurrent systems as well. In fact, the concurrent object-oriented paradigm can be thought of as a blend of those paradigms. A discussion on the visualisation of concurrent object-oriented systems, therefore, is not complete without discussing the visualisation of sequential, object-oriented, and concurrent systems. The following sections describe the visualisation for these systems.

2.2.1 Sequential Systems

In this section, sequential systems refer to the systems or programs with sequential execution, while object-oriented systems are sequential systems which employ the object-oriented paradigm. They are grouped together because they share many features. For brevity, both are referred to as *sequential systems*.

Tools dealing with sequential systems typically use dynamic views. Some, however, provide only static views. For example, the tool SEE [6] displays the pretty-printed versions of C programs. Another tool, developed by Ball [7], displays the source code of large programs in a miniature format, tied to program execution. The colour in each of the miniature source code lines shows the frequency with which the line is executed.

These tools provide important views. However, the effectiveness of the static views can be enhanced if they can be linked more vividly to dynamic views displaying the associated program execution.

The majority of the tools for sequential programs, as a matter of fact, display only program's dynamic aspects. For example, tools displaying algorithm animation include BALSA [22] and Zeus [18]. Meanwhile, tools incorporating only code views include the Transparent Prolog Machine (TPM) (Section 2.1.2.1), VizBug++ [79], Ovation [39, 40], and LOOK! [133]. Finally, views displaying data structures and their contents are typically employed by visual debuggers. These include KAESTLE [12] for visualising dynamic list data structures in LISP programs and VIPS [78, 117], which display the list structures of Ada programs. More general-purpose tools include ObjectCenter [109] and DDD [143], which can both be used to visually debug C and C++ programs.

From the discussion, a general trend can be observed. The majority of the tools

use the strategy of exclusively employing the program's dynamic aspects. While this is a viable approach, the tools would be more effective by presenting both the static and dynamic aspects concurrently. However, this predicament is offset by the fact that the tools generally use a wide selection of program features for visualisation. For example, the tool LOOK! focuses not only on objects and classes; rather, it also uses function invocations, messages and pointer structures for visualisation. Other such tools include ObjectCenter, and DDD.

Another observation, which is minor, is that the majority of the tools for visualising sequential systems are used for understanding or debugging. Very few, if any, are used for performance tuning. The reason is, perhaps, that performance is not the main focus in such systems, as compared with concurrent systems.

2.2.2 Concurrent Systems

Software visualisation has also been extensively applied to concurrent systems. For these systems, code views appear to be the ones most frequently used. As previously described (Section 2.1.3.2), these code views are typically embodied as graph-based and statistics-based views. Some of the relevant tools are described below.

Graph-based views are most frequently used by visualisation tools for concurrent programs. The graphs used may take a variety of forms. For example, program activities may be presented as space-time diagrams depicting interactions among tasks (typically in message-passing programs), such as in ParaGraph [66, 69], AIMS [142], XPVM [65, 81], and ATEMPT [83]. Other graph-based views are used to depict history of program events in thread-based environments [48, 73, 144]. Yet, other graph-based views are used for depicting machine and process activity [27, 92].

These tools are invaluable in that they depict interaction among program entities in terms of the key features of the programs. For example, the tools for depicting message-passing programs provide visualisation in terms of message exchange among tasks, typically exemplified by Figure 2.9. Using these tools, patterns of program execution are generally easy to identify. However, the tools usually do not provide explicit pointers as to *why* a program behaves the way it does. Such assistance can be

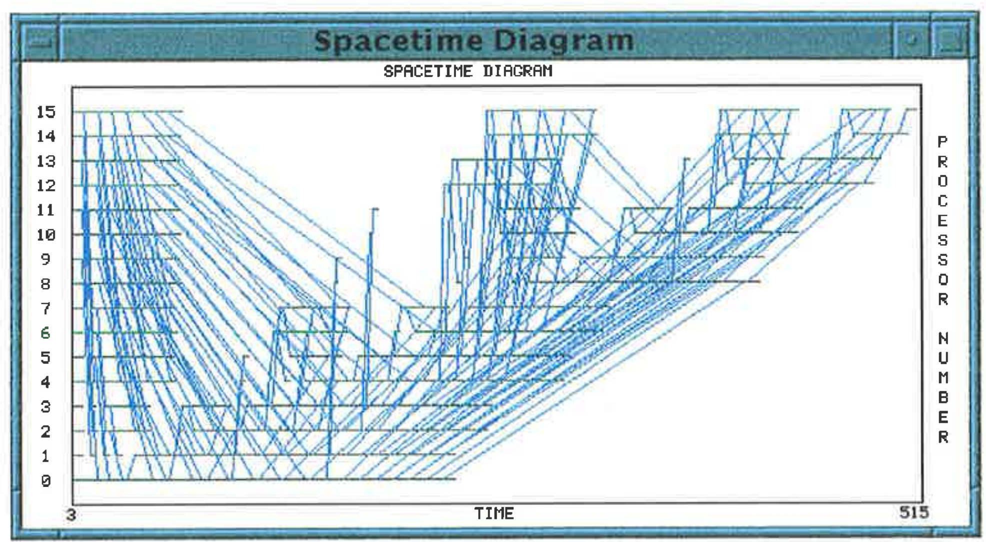

Figure 2.9. *The* **Spacetime Diagram** *in ParaGraph.*

facilitated by linking the views with the program's static structures, and by providing a variety of closely related views which depict other language/program features as well.

Tools for visualising concurrent systems can also use statistics-based views to display program performance. Bar-charts, pie-charts, matrices, and scatter-plots are some of the most commonly used views. Figure 2.10 is one of the views in ParaGraph, showing the statistics of communication traffic in a message-passing program. Other tools in this category include Medea [29] and Traceview [91][5]. Such tools provide a general idea to the user of how well a program performs, and of whether the desired performance has been achieved. However, without other views (including the program's static structures), it is difficult for users to pin-point potential sources of difficulty should they wish to improve the performance of the program.

Some tools present novel or unusual views for visualising concurrent systems. One example is the Gthreads visualisation tool [145], which visualises the execution of thread-based programs running on Kendall Square Research (KSR) machines. Figure 2.11 illustrates two views in Gthreads. The left view is the the History View, a space-time diagram depicting thread execution, while the right figure displays a mutex variable (the larger/outer circle) being accessed by a thread (the smaller inner circle). Other examples include SIEVE [114], which visualises performance data by using

[5]Traceview is not to be confused with TraceViewer [73], another tool.

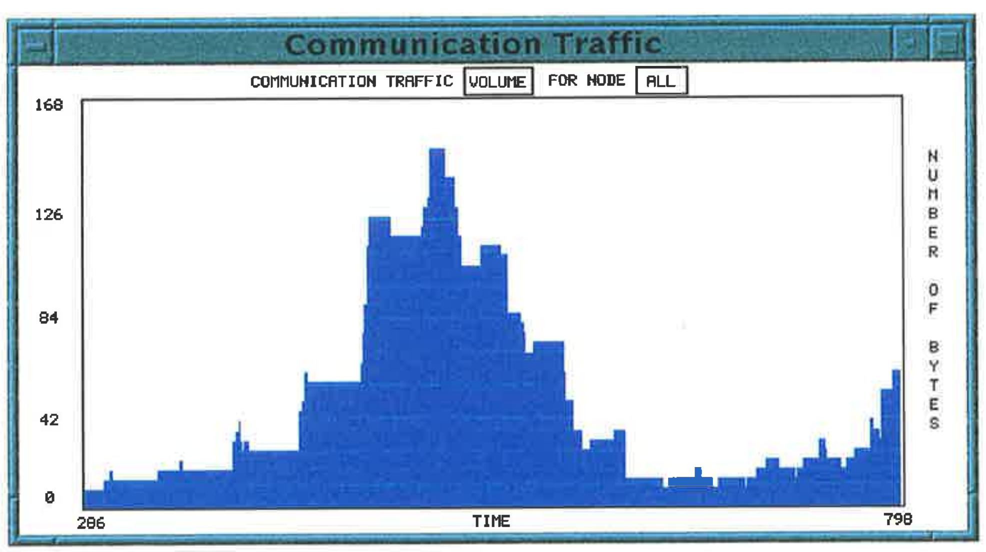

Figure 2.10. *Statistics of communication traffic in ParaGraph.*

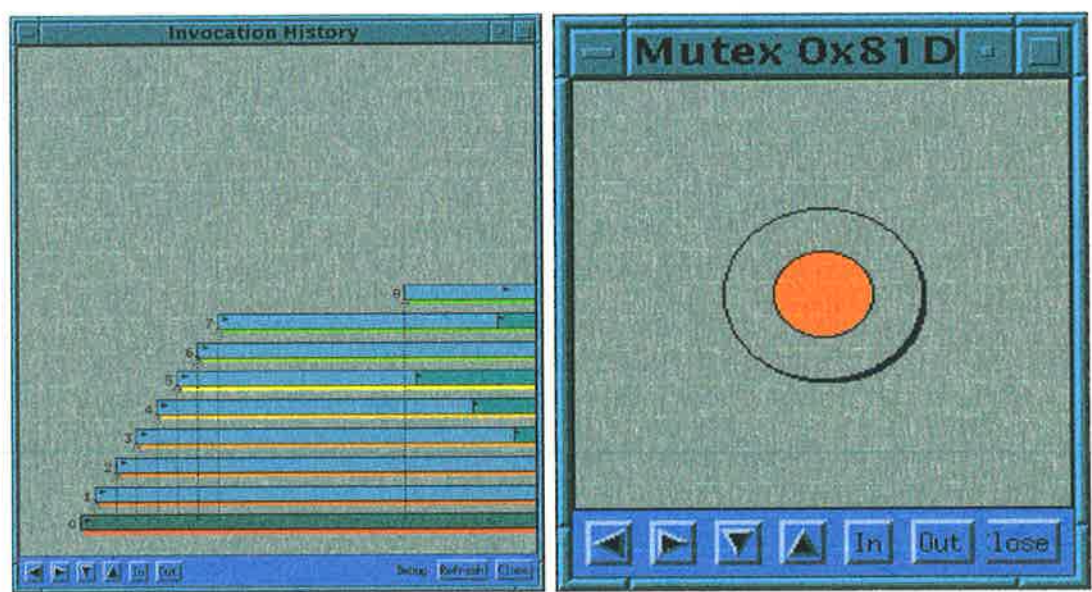

Figure 2.11. *The* **History View** *and the* **Mutex View** *in Gthreads.*

spreadsheet-like views; and Voyeur [119], which uses fish and shark figures to symbolise performance data value. These novel views provide breakthroughs in program visualisation. Their use, however, would be more effective if they are used in conjunction with other types of views to provide users with a multiple perspective of the program.

Based on the above discussion, some observations can be made as follows. Firstly, although many of the above systems employ both static and dynamic views, there is also a relatively large number which employ *only* dynamic views. This is the case, for example, in Voyeur. What is rather surprising is that a significant number of them only emphasise a *limited* number of aspects of the visualised programs. XPVM, for example, mainly depicts communication/message exchanges among tasks in its views. This, perhaps, is based on the notion that only the most important and distinguishing program elements or features should be displayed. However, the notion of what is "important" may differ from one user to another in a subjective manner.

Secondly, the tools generally include views which provide the information of *what* happens in a program. Sometimes, the notion of time is also considered; in other words, *when* a particular event happens is also indicated. However, most do not provide assistance to indicate *why* and *where* it happens in the program. This is problematic, since program understanding and program tuning require that such knowledge be available. Providing a set of inter-related views which expose a wide variety of program elements (or language features) can assist the user in obtaining such knowledge. Subsequently, the user would be able to form the program execution mental model, using it to modify the program, as necessary. Still, this scheme would be more effective if program's static structures are also visualised and linked to the rest of the system in a coherent manner.

Thirdly, another interesting side observation is that in the realm of concurrent systems, the majority of the tools seem to deal with the visualisation of message-passing programs. Perhaps this can be attributed to the fact that such programs have an inherently "clean" semantics in that program execution can be seen in a simplified way as a set of tasks communicating with each other. In contrast, in concurrent object-oriented systems, many program entities (such as threads, functions and RPCs, to name a few) interact in a complex manner. Once again, this may explain why there

are relatively few visualisation tools for concurrent object-oriented systems. The next section examines some of these tools.

2.2.3 Concurrent Object-Oriented Systems

There are relatively few tools for visualising concurrent object-oriented programs. Some of these are described in this section. For each, the associated concurrent object-oriented language is also described.

2.2.3.1 Visualisation of μC++ programs

μC++ [23, 26] is a language extension of C++ with some concurrency features. It is based on the notion of single-memory model execution in that it can operate on single-processor or shared-memory systems. μC++ extends C++ by providing four basic abstractions: *coroutine*, *monitor*, *coroutine-monitor*, and *task*. μC++ supports visualisation as part of its language design.

The visualisation of μC++ programs is embodied in the tool MVD (Monitoring, Visualisation and Debugging) [24, 25, 127]. As the name suggests, it comprises a set of tools for monitoring, visualising, and debugging the execution of μC++ programs. MVD is designed to help users understand the dynamic execution of the programs, and thereby debug them as necessary.

MVD supports two types of visualisation: statistical visualisation, and trace visualisation. Statistical visualisation is performed by taking program execution samples at regular time intervals and subsequently visualising them. The resulting view is displayed as icons representing tasks along with their activity status and stack high-water marks.

Trace visualisation, on the other hand, means that complete program events are visualised. These include important language elements, such as coroutine and thread operations. Such visualisation is carried out by using the tool POET [127], which is part of the MVD tool-set. The monitoring and visualisation can be done on-the-fly or through post-mortem analysis. In trace visualisation, entities such as coroutines and monitors are visualised in a view similar to a space-time diagram.

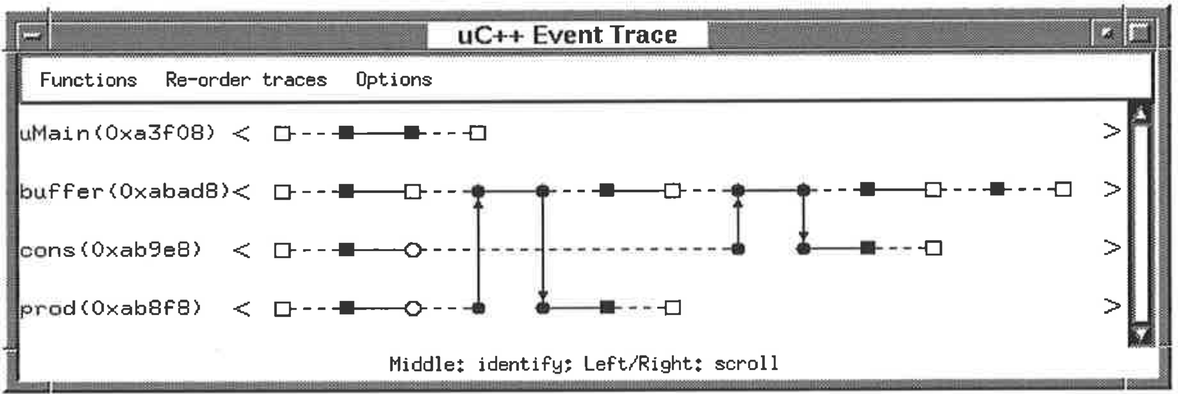

Figure 2.12. *Trace visualisation in MVD (POET).*

Figure 2.12 shows an example view from the tool POET. The figure shows the trace visualisation of a *producer-consumer* program, in which four tasks (*uMain*, *buffer*, *cons*, and *prod*) execute. A small empty square along the time-line of a task denotes an event causing the task to block, while a filled square represents an event causing the task to be ready. Vertical arrows represent inter-task communication. Finally, a solid horizontal line indicates that the task is ready, while a dashed line indicates that it is blocked.

MVD is relatively easy to use for visualisation. One factor which contributes to this fact is that both monitoring and visualisation are supported and integrated at the language level. Such support and integration has not been widely adopted in other languages. Secondly, MVD provides a number of different tools (or, rather, sub-tools) which can be used for different purposes. In particular, it provides statistical views [24], program trace views [127], and a concurrent debugger [25]. These sub-tools provide a rich metaphor for exploring the different aspects of program execution.

Although MVD has rich visualisation metaphors, there are also some difficulties. Firstly, the rich set of tools available are not well integrated, but are, rather, designed as separate sub-tools. This makes it difficult for users who need the assistance from the views concurrently to validate or compare observations from the views in one sub-tool with those in another. Secondly, MVD only uses a narrow selection of language features in the visualisation. In particular, it only visualises coroutines, monitors, coroutine-monitors, and tasks. Other language features, such as objects, and functions are not considered. Furthermore, except for the debugger, linkage to the program static

structures is rather poor. Resolution of these problems may make MVD more effective in helping users to understand and to pin-point sources of problem.

2.2.3.2 Visualisation of pC++ programs

TAU [17, 97, 98] is a program visualisation tool, which is part of the pC++ programming language system. pC++ [13] itself is an extension of the C++ language with additional constructs to support data parallelism.

TAU tools are implemented as a graphical hypertool in that it is a composition of several sub-tools, each supporting unique capabilities. The tools are divided into two groups. The first group is the static analysis tools. These include Fancy for browsing global functions and class methods, Cagey for displaying static program call-graphs, and Classy for displaying static class hierarchy. The second group of sub-tools are the dynamic analysis tools, which include Racy for displaying usage profiles of functions and concurrent object member functions, Easy for displaying program events on an X-Y graph, and Breezy (Figure 2.4 and Figure 2.5), which is a breakpoint-based debugger. The views in TAU are hierarchical. They also provide multi-faceted views of the same entities. There is also a consistent link-back into the program static structures.

The visualisation of programs via TAU are supported by the pC++ language environment. During program compilation, a program is transformed into an abstract syntax tree (AST) which can be manipulated by the tool Sage++ [14] for instrumentation. After manipulation, the AST is transformed into native C++ code. Through command line switches during compilation, the AST can be instrumented for profiling, tracing, or break-point debugging.

TAU provides a rich set of sub-tools which can be used to analyse both the static and the dynamic aspects of a program. Furthermore, unlike MVD, the sub-tools are integrated and can be used concurrently. This enables a user to compare the same program aspects or events from different angles, thereby helping users to more easily understand the program and pin-point sources of problem. It would, therefore, be interesting to see whether such a rich visualisation support is possible in other languages, such as CC++.

2.2.3.3 Visualisation of LAMINA programs

The LAMINA visualisation tool is used to visualise the execution of LAMINA [37] programs. The LAMINA language itself is a LISP-style concurrent object-oriented language, based on the notion of actors [101]. Both LAMINA and the LAMINA visualisation tool are built upon a simulation system for simulating parallel hardware models.

In the LAMINA language model, concurrent objects communicate with one another asynchronously, via the *stream* mechanism. In LAMINA, a message names the method to be executed and the parameters to be used. If object A wishes to invoke a method of object B, it sends a message to the task stream of object B. As soon as the message is sent, object A continues execution. Therefore, objects in LAMINA are asynchronous and message-driven.

The LAMINA visualisation tool provides a number of views which can be used for performance debugging. The Network Operator Map displays the load on each processor, and the communication among them. Latency and utilisation of each processor are also displayed. Another view, the Activity Table, textually displays the activities of each concurrent object, such as the number of messages which have been processed, the average execution time, and so forth. These views are produced by solely using the traces obtained by monitoring the messages exchanged during program execution.

There are several difficulties with the LAMINA visualisation tool. Firstly, the tool only provides program execution statistics, with a minimal reference to program's static structures. Secondly, the monitoring system of the visualisation tool is very platform-specific. It assumes that message exchanges among objects are the only factor that drives program execution. Other similar platforms may not behave similarly. Therefore, the portability of the concepts of the LAMINA visualisation tool is difficult. Finally, it uses a very limited number of program features, particularly only those pertaining to messages.

2.2.3.4 Visualisation of PARC++ programs

PARC++ [129] extends the C++ language with a set of class libraries for the construction of parallel programs. These libraries provide abstractions, such as thread management, communication, and synchronisation. Some of these abstractions relate to shared-memory (such as monitors), while others relate to message-passing (such as mailboxes). PARC++ is essentially the C++ language equipped with a library for supporting parallelism. This is in contrast with other languages such as pC++, ICC++ [35], and μC++, which provide new constructs.

There are three varieties of visualisation schemes supported in PARC++. The first method is by using the tool Visit [129]. Using Visit, threads, locks, mailboxes, and processor utilisation are visualised. The second method is by using ParaGraph [69] in which entities in PARC++, such as threads and mailboxes are each translated to nodes in the ParaGraph context. The third method is by utilising the tool POPAI [129], which is similar to Visit, except that threads are not visualised. All of the above visualisation schemes are supported by PARC++ through its libraries and run-time environments.

The visualisation of PARC++ programs has some merits. Firstly, it is supported by the language environment, in that program views are produced automatically by the system without user intervention. Secondly, it provides three different tools to the user for the visualisation. The tools provide relatively different views, enabling users to understand their programs better. However, there are also some difficulties, which are discussed below.

Firstly, the visualisation tools do not provide a comprehensive presentation of program activity. For example, the tool Visit does not provide views of monitors, and the tool POPAI does not visualise thread execution, while both monitors and threads are important concepts in PARC++. Secondly, the views provided by the tools do not address the visualisation of program's static structures. Understanding a program and subsequently pin-pointing sources of problem are, therefore, difficult. Providing static views, and merging the tools POPAI and Visit into a one coherent system may make the system more useful.

2.2.3.5 Observations

The concurrent object-oriented paradigm has a rich semantics in that it combines the notion of objects with concurrency. This, unfortunately, results in programs which may be difficult to understand. The visualisation of such programs, therefore, needs to incorporate a wide selection of *language features* in order to present program activity from *multiple angles* by using multiple views. The incorporation of these schemes, in turn, enables users to deduce the behavioural aspects of the programs, and change them as necessary.

The difficulty in visualising concurrent object-oriented programs can be testified by the relatively low number of tools, each having widely differing ways of presenting similar concepts. The tools, however, present several trends, as follows.

Firstly, with the exception of a few tools such as TAU, many of these tools provide only program's dynamic aspects. Secondly, from a wide variety of the available dynamic aspects, generally the tools only present a few of them. This trend can be seen in MVD and POET for μC++ programs, the LAMINA tool, and the tools Visit and POPAI for PARC++ programs. Thirdly, some tools present a number of sub-tools which present different aspects of a visualised program. However, these sub-tools are most frequently not related and not integrated. Usage of such sub-tools, therefore, present problems, especially for users who concurrently need the capabilities from all of the tools.

So far, it seems that the TAU visualisation framework has the most extensive support. The tool has been successfully used for visualising and tuning a number of pC++ programs. Therefore, it is interesting to see whether similar concepts can be applied to other similar languages, such as CC++.

2.3 Summary

The above discussion can be summarised as follows.

1. Firstly, the visualisation of programs written in "combined-paradigm" languages is usually more complex than the visualisation of those in "single-paradigm" languages. For example, in general, the visualisation of a sequential object-oriented

program is more complex than a purely sequential program. This may be the reason that the visualisation of combined-paradigm programs typically only focuses on a narrow selection of language features, compared with the visualisation of single-paradigm programs employing a wide selection. In Section 2.2.2, for example, it has been observed that many visualisation tools for message-passing programs *only* selects messages and communication as the basis of visualisation. These approaches are viable. However, a *wide* selection of features for visualisation could generate more in-depth knowledge and analysis of the visualised programs. Such a wide selection need *not* necessarily mean a cluttered display [123]. Rather, the structuring of such information in a sensible manner is necessary.

2. Secondly, a good visualisation tool provides users with the necessary assistance to visualise programs, in terms of *what*, *when*, *where*, and *why*. Without such qualities, users of a visualisation tool are frequently left to "guess" from the views what the real activity of a program really was. Many existing tools, however, do not seem to be equipped with these qualities. In the case of the visualisation of concurrent object-oriented programs, these qualities are of utmost importance. The reason is that such programs typically involve complex interactions of program elements, that it is difficult to understand their visualisation without good assistance from the tool.

3. Thirdly, based on the first observation, it can be seen that the visualisation of concurrent object-oriented systems poses an interesting venue for experimentation. As such visualisation is typically used for the purpose of understanding and fine-tuning programs, it is necessary that it use a wide selection of program features. As the concurrent object-oriented paradigm is relatively complex, such features need to be visualised from multiple angles by using multiple coherent views. These two schemes can ensure that a more thorough understanding of programs is possible. TAU, for example, is highly successful in this respect. It has been used by researchers to understand and fine-tune programs. Other tools are, of course, also successful. However, extending their functionalities and

capabilities by using the TAU approach can make them more effective.

4. Finally, the majority of the tools for concurrent object-oriented programs, as described in Section 2.2.3, use language support for providing visualisation. Such support is a highly desirable, as it facilitates automatic generation of visualisation. Unfortunately, many languages are not designed in this manner. In other words, visualisation is an *after-thought*. For such languages, visualisation support must be created as added functionality. It is one of the goals of the thesis to show how this can be achieved, particularly for task-parallel object-oriented languages derived from the C++ language.

Based on the above discussion, and on Section 1.1, the goals of the thesis are refined and re-formulated as follows:

1. To study how the visualisation of task-parallel object-oriented systems can be carried out. The visualisation should help form and strengthen users' mental models of the programs and subsequently enable them to understand and fine-tune the programs. In particular, the views should help users to determine what, when, where, and why program events occur. This should be achieved by employing a wide selection of language features as the basis of visualisation.

2. To study how such visualisation can be generated, particularly in absence of explicit support from the language environment. In other words, how sensible program visualisation can be generated in languages without visualisation support.

As a vehicle of experimentation, the language CC++ is used. It is a task-parallel object-oriented language without explicit/extensive visualisation support. Although the findings are specific to this platform, nevertheless it is expected that the concepts uncovered can be applied to other similar systems. To appreciate the relevant issues involved in providing such visualisation, the next chapter provides a brief overview of the CC++ language.

Chapter 3

Overview of the CC++ Language

3.1 The CC++ Language

The Compositional C++ language (CC++) [30, 32, 33, 53] is a strict superset of the C++ language [46, 124], with a small number of extensions. These extensions allow the construction of parallel programs from simpler components by using sequential and parallel composition. For example, to compose a distributed merge-sort program, the best algorithms can be used to create merging and sorting components. Later, these program components are composed to form a single concurrent or parallel program.

The CC++ extensions include six basic abstractions which can be loosely categorised into three groups, as follows:

- concurrency,
- synchronisation and determinism, and
- locality.

In the following discussion, it is assumed that the reader is familiar with the basic concepts of C++.

```
par {
   master();
   slave(1);
   slave(2);
}
```

Figure 3.1. *Usage of the* **par** *construct.*

3.1.1 Concurrency

Concurrency in CC++ is achieved via threads which can be synchronous or asynchronous.

3.1.1.1 Synchronous threads

Synchronous threads can be created through the use of one of two constructs. The first construct, `par`, is used as follows.

$$\texttt{par \{ } S_1; S_2; ...; S_N; \texttt{ \}}$$

The above construct results in each statement in the block being executed concurrently with all other statements in the same block. In other words, the statement S_1 executes concurrently with the statements S_2, S_3, and so on. Each such statement can be any valid CC++ statement, provided that it does not result in non-local changes to the flow of control, such as a `return` statement.

Figure 3.1 shows three threads being spawned by the `par` construct. One thread executes the `master()` function, while the other two execute the same `slave()` function with different parameters.

The execution of the `par` block terminates when all of its component statements terminate. This, in effect, means that barrier synchronisation is implicit at the end of the construct (Figure 3.2).

The second construct is `parfor`. Its syntax is identical to the C++ `for` keyword, except that the keyword is replaced by `parfor`. Similar to the `par` construct, a `parfor` block terminates when all of its threads have finished execution. Each such thread holds a copy of the loop control variable along with all other variables declared inside the

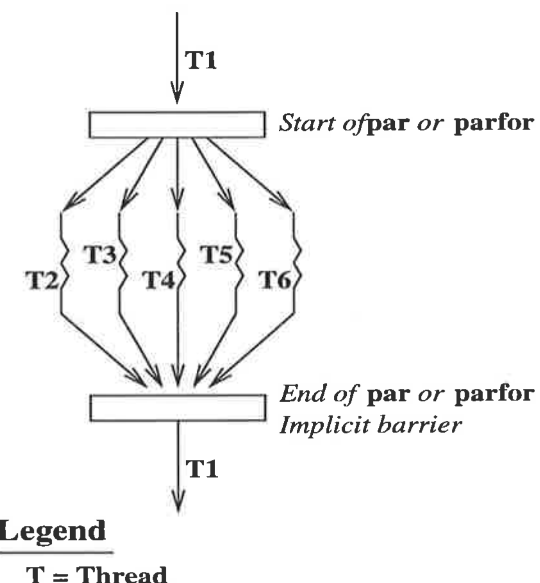

Figure 3.2. *Implicit synchronisation barrier at the end of a* **par** *or* **parfor** *block.*

```
parfor (int i=0 ; i<N ; i++) {
   think(i);
   eat(i);
   sleep(i);
}
```

Figure 3.3. *Usage of the* **parfor** *construct.*

block. Global variables are accessible by all threads, and the access can be controlled via `sync` variables or `atomic` functions. Figure 3.3 gives an example of a `parfor` block in which N threads are created, each of which executes 3 functions.

3.1.1.2 Asynchronous threads

Asynchronous threads can be created by using the `spawn` construct. This construct can only be used to spawn a thread executing a function which returns a `void` result. Unlike `par` or `parfor` threads, a `spawn`ed thread executes concurrently with its creator thread. The spawned thread is terminated when the function it executes finishes.

The syntax of the call is:

```
spawn some_function ();
```

Figure 3.4 gives an example of the usage of `spawn`.

```
void independent();
...
spawn independent();
```

Figure 3.4. *Usage of the* **spawn** *construct.*

3.1.2 Synchronisation and Determinism

Synchronisation and interleaving among threads in CC++ can be controlled by using `sync` variables and `atomic` functions. They provide a powerful mechanism to ensure data integrity and correctness of execution.

3.1.2.1 Synchronisation

A `sync` variable is a single-assignment variable which is used for synchronising thread execution. Any thread trying to read a `sync` variable will be blocked until a value is assigned to it. After assignment, the variable behaves as a constant; attempts to change its value result in run-time errors.

In Figure 3.5, the two slave threads will block until the master thread assigns a value to `data`. In principle, the `sync` construct can be used for synchronisation by any thread, be it synchronous or asynchronous.

3.1.2.2 Atomic Functions

CC++ guarantees the fairness of thread execution. However, it makes no guarantee on their order or interleaving which may vary from one execution to the next. The order of thread execution is sometimes important, particularly to ensure data integrity. The keyword `atomic` can be used in this respect. In CC++, only class member functions can be declared atomic. The execution of such a function will not be interleaved with the execution of other atomic functions of the same object.

Figure 3.6 shows the construction and usage of a class implementing a simple memory reader and writer. The class `ReaderWriterClass` contains two atomic functions: `read()` and `write()`. After the object `RW_Obj` is instantiated, each of its atomic functions is executed by a different thread. Therefore, the execution of the threads may

```
sync int data;
void master(int val) {
   ...
   data = val;
   ...
}
void slave(int id) {
   int masterdata = data;
   ...
}
...
par {
   slave(1);
   slave(2);
   master(0);
}
```

Figure 3.5. *Usage of the* **sync** *construct.*

```
class ReaderWriterClass {
   ...
public
   atomic int read();
   atomic void write(int internal_data);
   ...
};

...

ReaderWriterClass RW_Obj;
par {
   RW_Obj.read();               // thread 1
   RW_Obj.write(something);   // thread 2
}
```

Figure 3.6. *Usage of the* **atomic** *construct.*

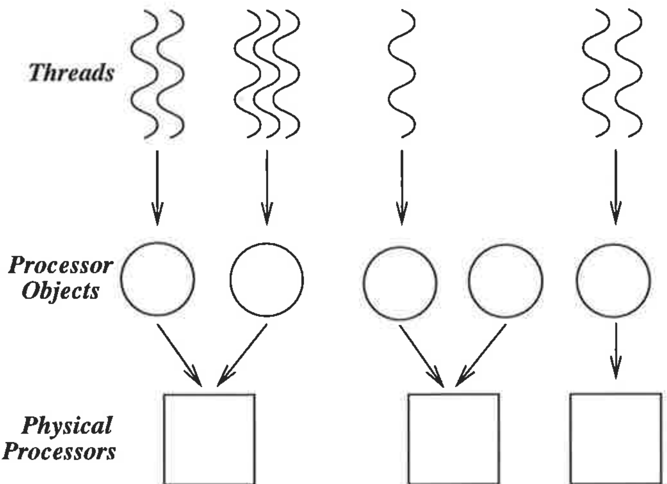

Figure 3.7. *Mapping of threads and processor objects in CC++.*

start with `read()`, then followed by `write()`., or vice versa.

3.1.3 Locality

The concepts of *locality* and computation are separated in CC++, in the sense that the specification of address space and threads of control are separated. In CC++, locality is specified by the notion of processor object (PO). A computation can be comprised of one or more POs, which can be executed on one physical processor, or distributed across a set of physical processors. In turn, a processor object may contain zero[6] or more threads executing concurrently (Figure 3.7).

3.1.3.1 Processor Objects

The concept of processor object (PO) is an extension of the C++ object concept [33], in that a PO embodies a separate address space. Analogous to the C++ object and class concepts, a PO is an instantiation of a specific PO class. A PO class is specified in the same manner as a C++ class is, but prefixed with the keyword `global`. The C++ class inheritance mechanisms also apply to PO classes. However, for the POs themselves, only those members which are declared under the keyword `public` are accessible outside their address space through the RPC and data transfer mechanisms.

[6] When a PO does not contain any thread, it is said to be idle.

```
//------------------
// File: activity.h
//------------------
global class ActivityClass {
private:
   int internal_data1;           // Inaccessible from without
protected:
   char internal_data2;          // Idem
public:
   int status;                   // Only these public members
   int walk(int activity_no);    // are accessible from
   int sit(int activity_no);     // without
};

//---------------------
// File: activity.cc++
//---------------------
#include "activity.h"
...
int data1;                       // Also inaccessible from
void function1 (void);           // without
...
int ActivityClass::walk(int activity_no)
{
   [function body]
}
...
```

Figure 3.8. *Implementation of a processor object class.*

All other data and functions are considered as private to the PO. This is illustrated in Figure 3.8.

3.1.3.2 Remote Procedure Calls

Cooperation among POs is achieved by accessing the public members of other POs. The public member accessed could be a piece of data or a function. Both types of access are facilitated through *global pointers*. Such pointers represent those data or functions which are potentially expensive to access. In the CC++ implementation, a global pointer is declared in the same manner as an ordinary C++ pointer, prefixed

```
1    char node_names[10][64];   // names of physical processors
2    ActivityClass *global gp[10];
3    parfor (int i=0 ; i<10 ; i++)  {
4       proc_t location = proc_t("activity.out", node_names[i]);
5
6       // instantiation and placement of PO of type "ActivityClass"
7       gp[i] = new(location) ActivityClass();
8       [user's code]
9       if (gp[i]->status == 0)
10         gp[i]->walk(i);
11      else
12         gp[i]->sit(i);
13      [user's code]
14   }
```

Figure 3.9. *Usage of global pointers.*

with the keyword `global` (line 2 of the code section in Figure 3.9).

When a processor object P_1 invokes a public member function of another processor object P_2, a RPC is issued. A new thread is then created in P_2 to handle the remote call, while the calling thread suspends. When the thread in P_2 has finished execution, the calling thread continues its execution flow. The results, if any, are transferred from the callee to the caller. Line 10 and 12 in Figure 3.9 are examples of such a RPC. The whole scheme is illustrated in the lower part of Figure 3.10

When the PO P_1 only accesses a piece of data in P_2, no thread is created in P_2; only data transfer occurs. Line 9 of the program in Figure 3.9 is one such example, and the mechanism is illustrated in the upper part of Figure 3.10

Both types of access, be it remote data access or remote function invocation, involve data transfer from the callee to the caller thread. The data transfer mechanisms can be defined by the user as *transfer functions*. This is especially necessary for non-trivial data structures, such as an array or a C++ structure. These transfer functions are essentially used for the marshaling and unmarshaling of data. These functions are automatically invoked whenever the associated type is to be transferred from one PO to another [32]. Figure 3.11 illustrates two transfer functions for the type `ComplexClass`. One such function (`operator<<`) is used for marshaling, and the other (`operator>>`)

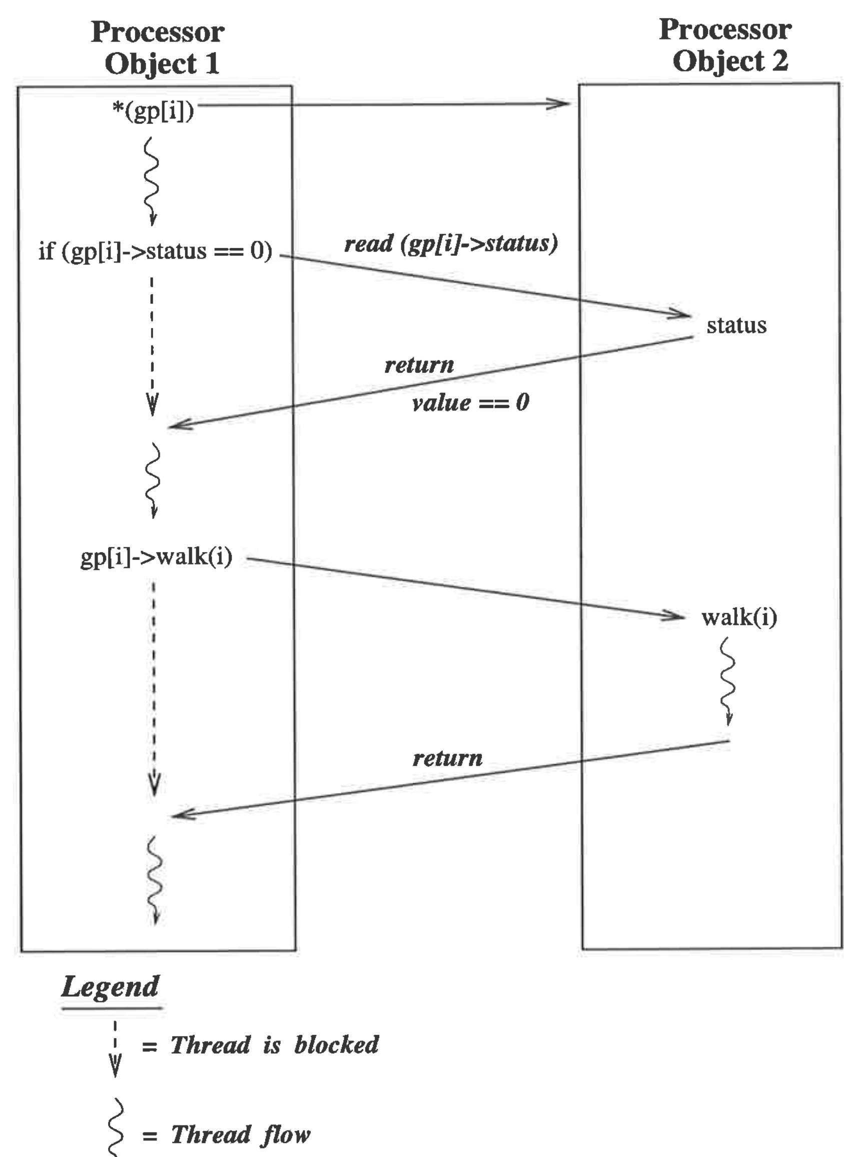

Figure 3.10. *Data communication and RPC between two processor objects.*

```
class ComplexClass {
private:
   double real_part;
   double imaginary_part;
   friend CCVoid& operator<<(CCVoid&,const ComplexClass &);
   friend CCVoid& operator>>(CCVoid&,ComplexClass &);
};

//--- marshaling ---//
// CCVoid is a special type in CC++, used for data
// transfer across different address spaces.
CCVoid& operator<<(CCVoid &v, const ComplexClass &c)
{
   v << c.real_part << c.imaginary_part;
   return v;
}

//--- unmarshaling ---//
CCVoid& operator>>(CCVoid &v, ComplexClass &c)
{
   v >> c.real_part >> c.imaginary_part;
   return v;
}
```

Figure 3.11. *A transfer function.*

for unmarshaling.

3.2 Visualisation Support

Unlike the pC++ language [13], CC++ does not provide full support for program visualisation. pC++, on the other hand, provides a source-code transformation tool, Sage++ [14], which is part of the language design. This tool can be used for program instrumentation. This facility is used by the program visualisation toolset TAU [97] to instrument pC++ programs for producing visualisation.

Although such support for visualisation is not available in CC++, some limited support is made available through its underlying run-time system. However, the support is rather rudimentary [63, 75, 130], and is not a part of the CC++ language

design [30, 33].

The visualisation support in CC++ is provided by the Nexus run-time system [55, 56], which serves as the target of the CC++ compiler. Nexus provides low-level facilities to support interactions between concurrently executing components of a program. These facilities include support for multiple threads of control, dynamic processor acquisition, and dynamic creation of address space.

Nexus allows the generation of profiling data of the execution of its applications. Since a CC++ program is essentially a Nexus application, it means that profiles of CC++ program execution can be obtained. This is achieved by providing appropriate switches during CC++ program compilation. The profiling data obtained can then be visualised by using Pablo [4].

Visualising the Nexus profile data, however, has several serious disadvantages. Firstly, the profiling data obtained is only in terms of low-level Nexus entities and contains the minimum amount of data necessary to drive meaningful visualisation. For example, the Nexus thread-creation profile does not contain information concerning which thread created a particular thread, or which line in the source code contributed to this event [54, 128]. Consequently, relating the views of these entities to the user's mental model is difficult.

Secondly, since the profile structures are fixed, to get further profiling information means changing Nexus itself. This can be difficult, and may lead to undesirable effects such as altered system behaviour. Furthermore, Nexus is designed as a compilation target, not a final system in itself. Any changes to Nexus, if necessary, must be done carefully.

Finally, Pablo requires that for different program profiles (i.e. different CC++ programs), different configuration files have to be created. This is cumbersome for users, especially non-experts. The approach of providing a specialised tool will provide significantly more benefits to more users [17].

Chapter 4

Visor++

This chapter describes the framework and architecture of Visor++ (Visualisation of Software in the Task-Parallel Object-Oriented Realm), the tool for visualising CC++ programs.

As expounded in Section 1.1, and Section 2.3, for the visualisation of concurrent object-oriented programs, there are two questions to be addressed. Firstly, in order to help form or strengthen the mental models of users, what features of the program or software system can be used for visualisation, and how should they be visualised? The answer hypothesised by Visor++ is that both the *static views* and *dynamic views* of a program should be visualised. The static views are the views of the static properties of a program. They include, for example, the views of source code and class hierarchies. Dynamic views refer to the views which depict program execution. These two types of views are important precisely because they represent a program static and dynamic models. This makes the comparison between a programmer's initial mental model and execution mental model possible (Figure 1.5).

Secondly, in the absence of adequate support for visualisation from the language system, what can be done to generate useful visualisation, and to what extent can this be done? The answer put forward by Visor++ is that meaningful visualisation can still be obtained, provided that two conditions are satisfied.

1. First, the visualiser must be able to obtain program execution snapshots (event traces). Such snapshots can be obtained by instrumenting the program either at

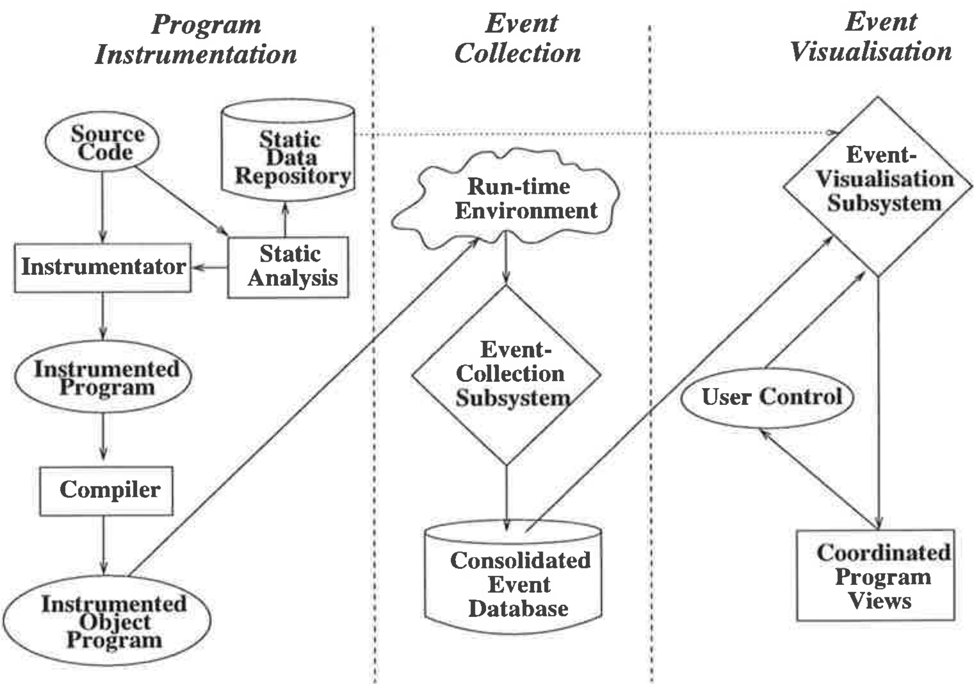

Figure 4.1. *General framework of Visor++.*

the source-code level or at other system levels, such as compilers or operating systems.

2. Second, the program snapshots must also include and reflect relationships among abstract program entities that they capture. For example, program snapshots on function execution must include information of which function and which thread invoked a particular function.

This chapter explores the above questions and answers. Specifically, the architecture and implementation of Visor++ are described.

4.1 General Framework

Visor++ [135, 136, 137] is implemented as a set of subsystems which transform the abstractions or constructs of a CC++ program from their initial forms to the final views. As Figure 4.1 shows, to provide the visualisation of a CC++ program, three steps are needed.

First, a program to be visualised is instrumented. This is achieved by automatically inserting probes into it. Simultaneously, *static analysis* is carried out on the program to extract the program static structures. These structures include such information as the original source code, class hierarchy, line numbers, and file names. This information is saved into a *static data repository*, which is later used by the visualisation subsystem. The instrumented program is then compiled by using the CC++ compiler, producing the instrumented object program. Note that both the instrumentation and the static analysis are carried out automatically by Visor++, without user intervention.

During program execution, the probes in the program emit traces depicting the program states. These records are captured by the Visor++ run-time environment, which is essentially a monitoring system. When the instrumented program has finished execution, all of the captured trace records are post-processed by the event collection subsystem. The records are then saved into a *consolidated event database*. The database, therefore, contains information on program states from all parts of the program, which may have executed on a single-processor or a multi-processor system. The event database, together with the static data repository, are then fed into the visualisation subsystem for producing the views. In other words, Visor++ employs the post-mortem visualisation approach. The same views can be reproduced at any time as long as the same static data repository and event database are used. Henceforth, the term **traces**, **trace records**, and **events** are used interchangeably. The term **event file** and **trace file** are also used to refer to the event database.

The event visualisation subsystem produces both static and dynamic views. The static views depict the program's static aspects, while the dynamic views the program's execution aspects. Both types of views are coordinated and animated coherently to reflect the original program structures and execution timing. The user can control and interact with the views and the visualisation subsystem through the graphical interface in the form of control panels.

By using the views, the user can gain an in-depth understanding of the program. The user can then fine-tune and modify the program as necessary. Subsequently, the modified program can be instrumented, visualised, and modified again. The cycle can

be repeated until the program behaves as expected.

From the diagram, it can be seen that the subsystems in Visor++ are loosely-coupled. The link from one subsystem to another is substantiated only through the event database and the static data repository. The design and implementation of these subsystems are described in the following sections.

4.2 Program Instrumentation Subsystem

Program instrumentation is executed only on programmer-defined entities (such as classes and functions). Those entities which are not defined by the programmer (henceforth termed **system code**) are not instrumented. Such system code includes, for example, the `iostream` library and the system-defined string functions such as `strcpy()`.

There are three main reasons for not carrying out instrumentation on system code. Firstly, programmers are generally interested only in their own code. Secondly, system code typically consists of many entities, such as classes and functions which are not efficient to extract, instrument and visualise. Even if the number of such entities were small compared to a programmer's code, it is still best to exclude them. This is necessary so that a user's cognitive perception is not inundated when using the system. Finally, the source code of the system code is typically not available for alteration or instrumentation.

The program instrumentation subsystem is constructed as a two-pass system. The first pass is the static analysis phase, which automatically extracts the information on programmer-defined entities from a program. The analysis is always carried out before the second pass, the program instrumentation. The reason is that program instrumentation needs to instrument only programmer-defined entities. The information on which parts of the source code constitute programmer-defined entities is precisely the one gathered during the static analysis. The entities instrumented are described in the next section.

4.2.1 Program Entities

As explained in Chapter 3, the CC++ language comprises many types of entities. These include C++ entities such as classes and functions, and CC++ extensions, such as threads and processor objects. However, not all of them are used, specifically variables and data structures. The reason is that a CC++ program can be distributed across a network of processors. This means that a CC++ computation may be large, in terms of CPU usage. For example, there may be a significant number of references to functions, objects, and threads. In such computation, in order not to inundate the users with excessive information, Visor++ does not display or focus on fine-grained program entities, such as data structures and pointers. Interactions and visualisation of such entities are perhaps better dealt with by a visual debugger. Furthermore, variables and data structures are typically used for temporary storage areas, and that Visor++ is not intended to be a debugging tool.

The CC++ entities visualised by Visor++ are described below.

- **Threads.** These include synchronous and asynchronous threads, including those initiated to cater for RPCs (Section 3.1.3.2).
- **Functions.** These include atomic functions, object member functions, ordinary C++ functions, and constructors and destructors[7]. Invocations of these functions reflect the activities of the threads in which they occur.
- **Logical Processor Objects.** The status and activity of processor objects also form an important part of visualisation. Such status information and activity are reflected by the operations of threads inside processor objects.
- **Remote Procedure Calls.** Remote procedure calls (RPCs) are an important mechanism by which cooperation among different parts of a CC++ computation is achieved. RPC activity, therefore, must be visualised.

[7]Object constructors and destructors, by definition, are not regarded as functions [124]. However, they can be treated as such for convenience.

```
struct FctInfoStruct {
   IdType FunctionId;     // Assigned by system
   IdType InWhichClass;   // Class index, -1 == ordinary function
   IdType InWhichFile;    // Source-file index
   IdType LineNumber;     // 1st line of function declaration

   NameType   FunctionName;
   RetValType ReturnValue;
   ParamType  Parameters;
};
```

Figure 4.2. *Information on functions in the static repository.*

4.2.2 Program Static Analysis

As previously described, the static analysis is used to automatically extract static information from a program. During the analysis, two types of information are extracted:

- **Source file information.**

  This includes the names of the source files, and the number of lines and characters for each file. Each file is given a unique identifier which is used for identifying the file in which a program event occurs.

- **Static program entities.**

  These include information on functions and classes. Information on functions comprises both ordinary functions and class member functions. The fields of function information in the static repository are shown in Figure 4.2. Class information, on the other hand, contains such information as class names, member functions, and friend classes. This information is illustrated in Figure 4.3.

  Note that all such information contains the following properties:

  - **Source-file identifier**, for identifying in which file a function or class is declared.
  - **Source line number**, for identifying the first line in which the entity declaration occurs in the source file.

```
enum AccessType { PRIVATE, PROTECTED, PUBLIC };

struct ClassInfoStruct {
   IdType ClassId;      // Assigned by system
   IdType InWhichFile;  // Source-file index
   IdType LineNumber;   // 1st line of class definition

   NameType   ClassName;

   IdType     *MemberFuncts;       // Array, pointing to funct-record
   AccessType *MemberFunctAccess;  // Access, for each funct-record
   int        NumMemberFuncts;     // Number of member functions

   IdType     *ParentClasses;      // idx to other class-info record
   AccessType *ParentClassAccess;  // Inheritance FROM parents
   int        NumParentClasses;

   IdType     *ChildClasses;
   AccessType *ChildClassAccess;   // Inheritance TO children
   int        NumchildClasses;

   IdType     *FriendFuncts;
   int        NumFriendFuncts;

   IdType     *FriendClasses;
   int        NumFriendClasses;
};
```

Figure 4.3. *Information on classes in the static repository.*

```
//--- File: included.h
   int a;
   struct struct1 *s;
};

//--- File: program.cc++
...
class class1 {
#include "included.h"
...
```

Figure 4.4. *Ill-partitioned, but valid program.*

Note that like C++, a CC++ program may span several files, and each file may include other files. In any case, the source line number of an entity is considered to be the first line in which the entity is *fully declared*. An entity is said to be fully declared if the body of the entity is fully specified. For example, a function header declaration with its body specified is said to be fully declared. On the other hand, a function prototype is *not* considered as a full declaration. In fact, it is sometimes known as a *forward declaration* [124]. It follows that for every function, there is only one full declaration. Therefore, the source-file and line number of a function are considered to be the first file and line number in which the header of a full function declaration occurs. This notion applies to other entities as well.

The example in Figure 4.4 illustrates an ill-partitioned, but otherwise valid CC++ program. In the example, the class `class1` is a fully declared class, which partly resides in the file `program.cc++`, and partly in `included.h`. In such a case, the source file and line number for the class are considered to be the first file and line number in which it occurs, which, in this case, is in `program.cc++`.

4.2.3 Program Instrumentation

Program static information, which is automatically extracted during the static analysis, is saved into a static repository. This repository is then used in the second pass,

program instrumentation, to automatically instrument only programmer-defined entities. The entities and the method of instrumentation are described in the following sections.

4.2.3.1 Events

The activities of CC++ program entities, as described in Section 4.2.1, are traced by "probes" which are automatically inserted into the program. A probe produces/transmits traces each time an entity in which the probe is installed *starts* and *finishes* execution. Such a design is necessary for several reasons. Firstly, it enables some performance data to be gathered, i.e. how much time is spent in each function, each thread, and so on. Secondly, it enables causality analysis to be carried out more accurately. For example, if a RPC is carried out by a thread, then since the CC++ RPC is synchronous, it follows that the RPC start and finish times must be "enclosed" by the thread start and finish times.

The events[8] implemented by Visor++ are as follows:

- **Threads.**

  In Visor++, events associated with threads are divided into two groups. The first group is associated with thread blocks, i.e. it records the entry and exit from `par` and `parfor` blocks. In the implementation, the events are given the following names.

  - `EV_PARFOR_START`
  - `EV_PARFOR_FINISH`
  - `EV_PAR_START`
  - `EV_PAR_FINISH`

  The second group is associated with threads themselves. This records the entry and exit from each thread, either synchronous (the threads created by a `par` or

[8] All CC++ events are given names with the prefix "EV", meaning "event". These names are self-explanatory.

parfor construct), or asynchronous (the threads created by the spawn construct). The event names are as follows:

- EV_PARFOR_THREAD_START
- EV_PARFOR_THREAD_FINISH
- EV_PAR_THREAD_START
- EV_PAR_THREAD_FINISH
- EV_SPAWN_THREAD_START
- EV_SPAWN_THREAD_FINISH

- **Functions.**

  For functions, Visor++ events record function entry and exit. They include ordinary functions, and object member functions, including atomic functions. The event names, which are self-explanatory, are as follows:

  - EV_MAIN_START
  - EV_MAIN_FINISH
  - EV_FUNCTION_START
  - EV_FUNCTION_FINISH
  - EV_CONSTRUCTOR_START
  - EV_CONSTRUCTOR_FINISH
  - EV_DESTRUCTOR_START
  - EV_DESTRUCTOR_FINISH
  - EV_MEMBER_START
  - EV_MEMBER_FINISH

  Note that the function main() has its own event names (EV_MAIN_START and EV_MAIN_FINISH). The reason is that in CC++, like C++, main() is a special function from which program execution starts.

- **Logical Processor Objects and RPCs.**

  The status and activities of processor objects comprise RPCs[9], and the creation and destruction of processor objects.

  - `EV_GLOBAL_CONSTRUCTOR_START`[10].
  - `EV_GLOBAL_CONSTRUCTOR_FINISH`
  - `EV_GLOBAL_DESTRUCTOR_START`
  - `EV_GLOBAL_DESTRUCTOR_FINISH`
  - `EV_GLOBAL_MEMBER_START`
  - `EV_GLOBAL_MEMBER_FINISH`
  - `EV_RPC_MARK_START`
  - `EV_RPC_MARK_FINISH`

The instrumentation methods and the generation of these events are explained in the next sections.

4.2.3.2 Location-ID

In order to obtain meaningful visualisation, relationships among program entities must be obtained. For example, for a function call, the traces for the events `EV_FUNCTION_START` and `EV_FUNCTION_FINISH` should contain information on the callee. The traces should also contain, among others, the information on the thread in which the caller resides. In other words, the location of both the caller and the callee must be known. This gives rise to the notion of *location-IDs.*

A location-ID is defined as the tuple **<Node-Id, Processor-Object-Id, Thread-Id>**. It describes the "location" in which the invocation of an entity (e.g. a function)

[9]From the definition of CC++, PO creation and destruction do not involve RPCs [30]. However, they are similar to remote invocation of PO member functions. For this reason, and for convenience, they are also regarded as RPCs.

[10]The event names relating to processor objects are prefixed with "EV_GLOBAL", meaning that this is an event, which is generated through the use of a global pointer for invoking a RPC. For RPCs, some secondary events are generated, with their names prefixed with "EV_RPC" (Section 4.2.3.5).

occurs. It also describes the location from which the invocation is made. This location includes the physical node, the processor object, and the thread. The identifiers in the tuple can be generated by the system (such as compilers and run-time systems), or artificially generated by the program visualisation system.

The location-IDs in the generated traces make it possible to infer the run-time computation structure of a program. With this knowledge, a more compact layout and organisation of the views can be constructed.

However, in contrast to other imperative programming systems and languages, such as Pthreads [100] and PRESTO [11], such location-IDs do not exist in CC++ (Section 3.2). Pthreads and PRESTO involve, for example, the use of unique identifiers for thread operations. These identifiers, which are supplied by the programmer, are not available in CC++.

Nexus, the underlying run-time system of CC++ does, in fact, have internal structures which keep information on threads and contexts. However, this information is opaque to CC++ and is not meant to be accessible through CC++ programs. The reason is that exposing such information poses many scalability and portability problems in a dynamic distributed system [130]. As a solution, such identifiers must be artificially generated by the instrumentation subsystem for each CC++ program to be visualised. This involves the artificial generation of physical-node identifiers, processor-object identifiers, and thread identifiers.

A physical-node identifier is assigned to each node participating in a computation. Program traces from each processor-object are collected by a monitoring process which resides on the same physical node as the processor-object. The local monitoring processes are allocated by a central monitoring process, which gives each of them a unique identifier. This identifier acts as a unique node identifier for the physical processor in which the associated local monitoring process resides. Implementation details are addressed in Section 4.3.

Next, each processor object has a unique pair of physical node identifier and processor object identifier (PO-ID). The node identifier of the processor object is the same as the node identifier of its local monitoring process. The PO-ID, however, is allocated

by the local monitoring process and given to a processor object whenever the processor object is allocated (constructed). To achieve this effect, the code of each processor-object is instrumented in such a way that upon processor-object construction, the local monitoring process is automatically contacted for such a unique identifier pair. This is achieved by instrumenting the processor object with the static object `Instr`, which is instantiated from the class `RECORD_class`. When the PO is constructed, the constructor of the object `Instr` is also invoked. The constructor of `Instr`, in turn, invokes a special function `Ask_Monitor_For_Id()`, which contacts the local monitoring process for a unique pair of **<Node-Id, PO-Id>**. This is illustrated in Figure 4.5.

For threads, unique thread identifiers can be obtained by contacting a *thread-identifier server* within the processor object in which the threads reside. This mechanism is invoked every time a thread (either synchronous or asynchronous) is allocated. The identifier server is implemented as a light-weight atomic member function `get_thread_id()` in the class `RECORD_class` (Figure 4.5 and Figure 4.6).

For generating event traces, the member function `generate_trace()` of the class `RECORD_class` is used. With appropriate parameters, the function generates a trace record, and sends it to the local monitoring process (Section 4.3). The trace record itself is defined as in Figure 4.7.

4.2.3.3 Instrumenting Functions

In Visor++, instrumenting a function involves two steps. First, the body of the function *declaration* must be instrumented. Second, the *calls* made to the function must also be instrumented. In this discussion, the term "function" refers to ordinary functions, including member functions, constructors and destructors of ordinary classes and processor-object classes.

The general method of instrumenting the body of a function is to modify the parameter list of the function, and to place two trace calls — one at the beginning (*entry code*) and one at the end (*exit code*) of the function body. The parameter list of the function is augmented with parameters denoting the location-ID of the caller, along

```
//---- Initial code for processor object class ----//
// global class PO1 {
// ...
// };

//---- Instrumented code ----//
class RECORD_class {
private:
   IdType thread_id;     // keep track number of threads so far
   void Ask_Monitor_For_Id () {
      // code for contacting local monitoring
      // process to get a unique <Node-Id, PO-Id>
   }
   ...
public:
   IdType processor_id;  // communicated by monitoring process
   IdType PO_id;         // obtained by Ask_Monitor_For_Id()
   RECORD_class () {
      Ask_Monitor_For_Id ();
      //--- other setup code here
   }

   //--- for obtaining unique thread-id
   atomic IdType get_thread_id ();

   //--- A trace includes Location-ID of caller and callee
   generate_trace (IdType EventType, IdType Node_Id,
                   IdType PO_Id,     IdType Thread_Id,
                   IdType FromWhichNode_Id, IdType FromWhichPO_Id,
                   IdType FromWhichThread_Id, IdType Function_Id,
                   IdType File_Id, IdType LineNumber);

   determine_PO_thread_id (IdType FromWhichNode_Id,
                           IdType FromWhichPO_Id,
                           IdType FromWhichThread_Id);
   ...
};
static RECORD_class Instr;  // static, automatic object

global class PO1 {  // the instrumented class
...
};
```

Figure 4.5. *Automatically obtaining a unique identifier for a processor object.*

```
atomic IdType RECORD_class::get_thread_id ()
{
   thread_id ++;
   return (thread_id);
}
```

Figure 4.6. *A light-weight atomic function to obtain unique thread identifiers.*

```
struct TraceStructure {
   IdType event;              // Name of an event
   double t;                  // Time-stamp taken by a probe

   IdType node_id;            // Location-ID of this event
   IdType process_id;
   IdType thread_id;

   IdType from_node_id;       // Location-ID of caller ...
   IdType from_process_id;    // ... used if applicable
   IdType from_thread_id;

   IdType function_id;        // Used for function-related events
   IdType source_file;        // File information
   IdType source_line;
};
```

Figure 4.7. *Program trace structure.*

with the file-ID and line number to indicate the invocation point[11]. In particular, the location-ID is necessary because the same function can be invoked by another function or thread which may have a different location-ID. As a result, the same function can have different location-IDs in different calls. Subsequently, the location-IDs are used by `generate_trace()` to generate the associated events.

For the trace calls inside the function body, the requirement is that the entry code must be the first statement to be executed in the function block, and the exit code the last, before function return. Figure 4.8 illustrates this approach. The approach, however, has difficulties with respect to exit code, particularly for functions which have many exit points. For functions, an exit point can be reached when the execution path encounters an `exit()` statement, a `return` statement, or when the end of the function block is reached (Figure 4.9).

To overcome this problem, a simple technique of using automatic objects[12], can be used [94, 96]. If an automatic object is declared as the first line inside a function block, it ensures two things. First, the constructor of the object is called before other parts of the function are invoked. Second, its destructor is the last statement executed before the block is exited. The constructor can then be made to call the intended entry code and the destructor the exit code. To do this, a light-weight class `EntityProfiler`, which only contains a constructor and a destructor, must be constructed (Figure 4.10). Using this method, the function in Figure 4.9 is instrumented by inserting the object `ep` of class `EntityProfiler` as the first line in the function body. This is illustrated in Figure 4.11.

Since the parameter list of each user-defined function is altered, any call to such a function must also be altered. This is illustrated in Figure 4.12. Note that in the example, the call to `function1()` is supplied with the parameters `Instr.processor_id` and `Instr.PO_id`, both being member data of `Instr`, the instantiation of `RECORD_class` (Figure 4.5). The parameter `Ann_Thread_Id` is the identifier of the current thread in

[11]The augmented parameters are placed *before* the programmer's initial parameter list. This is to guard against the possibility that the initial parameter list may contain default parameters [124].

[12]An automatic object is defined as an object which is local to the invocation of a block. The object is created upon entry to the block, and automatically destroyed upon block exit [46, 124].

```
//--- Initial function declaration ---//

//  int function1 ([user's param-list])
//  {
//     [user's code here]
//  }

//--- Instrumented code ---//
int function1 (IdType Ann_Node_Id  , IdType Ann_PO_Id,
               IdType Ann_Thread_Id, IdType Ann_File_Id
               IdType Ann_Line_No,
               [user's param-list])
{
   // Generate function-entry trace
   // "Instr" is an instantiation of RECORD_class
   // For ordinary function, Location_IDs of caller and
   // callee are the same
   Instr.generate_trace (EV_FUNCTION_START,  // instr'ed statically
                         Ann_Node_Id, Ann_PO_Id, Ann_Thread_Id,
                         Ann_Node_Id, Ann_PO_Id, Ann_Thread_Id,
                         [function-id],      // instr'ed statically
                         Ann_File_Id, Ann_Line_No);

   [user's code here]

   // generate function-exit trace
   Instr.generate_trace (EV_FUNCTION_FINISH, // instr'ed statically
                         Ann_Node_Id, Ann_PO_Id, Ann_Thread_Id,
                         Ann_Node_Id, Ann_PO_Id, Ann_Thread_Id,
                         [function-id],      // instr'ed statically
                         Ann_File_Id, Ann_Line_No);
}
```

Figure 4.8. *Example of function instrumentation.*

```
int function1([user's param-list])
{
    ...

    if (a) then
       return (1);  // may exit from here

    if (error == 1)
       exit (-1);   // or from here

    [code]
    // or here at the end of function block
}
```

Figure 4.9. *A function can have many exit paths.*

which the call occurs. Note that for ordinary functions, both the caller and the callee have the same Location-ID. The other two parameters, `Ann_File_Id` and `Ann_Line_No`, indicate the source file and line number from which a call is invoked.

4.2.3.4 Instrumenting Destructors

The above approach applies to ordinary functions as well as the member functions and constructors of ordinary classes. Instrumentation for destructors, however, is a little different. This is because C++ and CC++ object destructors do not take any parameters, whereas the method described above works by modifying function parameters. This problem is particularly acute, because an object may be constructed in one thread and destroyed in another (Figure 4.13). Without parameters, it is virtually impossible to "inform" the trace calls in a destructor of which thread it is currently in.

To circumvent this problem, every class is augmented with a set of private members to record the location-ID of the call to a class constructor. When an object is constructed, the location-ID of the caller is recorded. If the object is an automatic object, then the destructor is automatically called when the object is out of scope [46]. When this happens, it means that the calls to the constructor and destructor both have the *same* location-ID. The location-ID saved into the augmented private members are then

```
class EntityProfiler {
private:
   IdType ep_EventType;
   IdType ep_Node_Id;
   IdType ep_PO_Id;
   IdType ep_Thread_Id;
   ...
public:
   EntityProfiler (IdType EventType,
                   IdType Node_Id,
                   IdType PO_Id,
                   IdType Thread_Id,
                   IdType FromWhichNode_Id,
                   IdType FromWhichPO_Id,
                   IdType FromWhichThread_Id,
                   IdType Function_Id,
                   IdType File_Id, IdType LineNumber)
   {
      ep_EventType = EventType;
      ep_Node_Id   = Node_Id;
      ep_PO_Id     = PO_Id;
      ep_Thread_Id = Thread_Id;
      ...
      // Instr==global variable, RECORD_class instantiation
      Instr.generate_trace (ep_EventType, ep_Node_Id,
                            ep_PO_Id, ep_Thread_Id, ... );
   }

   ~EntityProfiler ()
   {
      // find_exit_event() is a simple function that returns
      // the "matching exit event" of an event.  E.g.
      // if ev_EventType==EV_PAR_START, return EV_PAR_FINISH, etc
      IdType exit_event = find_exit_event (ev_EventType);
      Instr.generate_trace (exit_event, ep_Node_Id,
                            ep_PO_Id, ep_Thread_Id, ... );
   }
};
```

Figure 4.10. *Profiler class in for the instrumentation subsystem.*

```
// "Ann" == Annotated/instrumented
int function1 (IdType Ann_Node_Id,
               IdType Ann_PO_Id,
               IdType Ann_Thread_Id,
               IdType Ann_File_Id,
               IdType Ann_Line_No,
               [user's param-list])
{
   EntityProfiler ep (EV_FUNCTION_START,  // instr'ed statically
                      Ann_Node_Id,
                      Ann_PO_Id,
                      Ann_Thread_Id,
                      ... );
   ...

   if (a) then
      return (1);  // may exit from here

   if (error == 1)
      exit (-1);   // or from here

   [code]
   // or here at the end of function block
}
```

Figure 4.11. *A function instrumented with an* **EntityProfiler** *object.*

```
//--- The original fct-declaration & fct-call
//  int function1 ([function param-list])
//  {
//     function2 ([user's params]);
//     ...
//  }
//  ...
//  a = function1 ([user's params]);

//--- The instrumented ones
int function1 (IdType Ann_Node_Id,
               IdType Ann_PO_Id,
               IdType Ann_Thread_Id,
               IdType Ann_File_Id,
               IdType Ann_Line_No,
               [function param-list])
{
   EntityProfiler ep (EV_FUNCTION_START, ... );

   // The function call occurs in file 3, line 24
   function2 (Ann_Node_Id, Ann_PO_Id, Ann_Thread_Id,
              3, 24, [user's params]);
   ...
}

...

// The function call occurs in file 3, line 56
a = function1 (Instr.processor_id, Instr.PO_id,
               Ann_Thread_Id, 3, 56, [user's params]);
```

Figure 4.12. *Example of an altered function call.*

```
main () {
    ...
    SomeClass *sc = new SomeClass (...);  // created in one thread
    ...
    parfor (int i=0 ; i<3 ; i++) {
        ...
        if (i==0) delete sc;  // destroyed in another thread!
        ...
    }
}
```

Figure 4.13. *Constructor and destructor called from different threads.*

used for generating the proper trace.

However, the calls to the constructor and destructor may have *different* location-IDs. This is normally the case if they are invoked within different threads, as shown in Figure 4.13. To overcome this problem, each class is also augmented with a public member function, `Ann_delete()`, whose parameters are used for "recording" the location-ID. The destructor of the class is then explicitly called from within this function (Figure 4.14). Calls to the class destructor within a program are correspondingly replaced with calls to `Ann_delete()`, as shown in Figure 4.15.

4.2.3.5 Instrumenting Processor Object Classes

Instrumenting a processor object class and its member functions is similar to instrumenting an ordinary class. The only difference is that each time a member function of a processor object is called from another processor object, a RPC occurs, and a thread is created in the address space of the callee to cater for the call.

For this reason, member functions of processor object classes are augmented with a call to a member function of `RECORD_class`, `determine_PO_thread_id()`, to get a new thread identifier, if applicable. This means that each time a PO member function is called, the function `determine_PO_thread_id()` will be invoked to return a thread identifier T. If the PO member function is called from within the same PO (i.e. the same address space), the identifier T will be the same as the thread identifier of the caller.

```
//--- Initial class declaration ---//
//   class SomeClass {
//   public:
//      SomeClass  () { [user's code] }
//      ~SomeClass () { [user's code] }
//      ...
//   };

//--- Instrumented class declaration ---//
class SomeClass {
private:
   IdType Ann_Node_Id, Ann_PO_Id;
   IdType Ann_Thread_Id;
   IdType Ann_File_Id, Ann_Line_No;
public:
   virtual Ann_delete (IdType Node_Id  , IdType PO_Id,
                       IdType Thread_Id, IdType File_Id,
                       IdType Line_No) {
      this->Ann_Node_Id   = Node_Id;
      this->Ann_PO_Id     = PO_Id;
      this->Ann_Thread_Id = Thread_Id;
      this->Ann_File_Id   = File_Id;
      this->Ann_Line_No   = Line_No;
      this->~SomeClass();  // call destructor here
   }
   SomeClass (IdType Node_Id, IdType PO_Id, IdType Thread_Id,
              IdType File_Id, IdType Line_No) {
      this->Ann_Node_Id   = Node_Id;
      this->Ann_PO_Id     = PO_Id;
      this->Ann_Thread_Id = Thread_Id;
      this->Ann_File_Id   = File_Id;
      this->Ann_Line_No   = Line_No;
      EntityProfiler ep (EV_CONSTRUCTOR_START,
                         Ann_Node_Id, Ann_PO_Id, ... );
      [user's code]
   }
   ~SomeClass () {
      EntityProfiler ep (EV_DESTRUCTOR_START,
                         Ann_Node_Id, Ann_PO_Id, ... );
      [user's code]
   }
   ...
};
```

Figure 4.14. *Instrumentation of a class.*

```
SomeClass *sc = new SomeClass (...);
parfor (int i=0 ; i<3 ; i++) {
    ...
    // Changing the call "delete sc;"
    if (i==0) sc->Ann_delete (...);
    ...
}
```

Figure 4.15. *Changing the call to an object destructor.*

Otherwise, a new and unique thread identifier is returned. The resulting location-ID is then used in generating the associated trace record. Figure 4.16 shows the code of `determine_PO_thread_id()` and how the function is used in instrumenting a PO member function.

The events relating to PO member functions, as previously described, are associated with the processor object in which the functions are invoked. These events alone enable Visor++ to determine the amount of time spent in *remote function invocation.* However, the amount of time needed for RPC *overheads* is not known. To compute such overhead time, two secondary events, `EV_RPC_MARK_START` and `EV_RPC_MARK_FINISH` are generated in the address space of the caller PO. The event `EV_RPC_MARK_START` marks the start of a RPC, and `EV_RPC_MARK_FINISH` marks its end.

Using these secondary events, Visor++ recognises RPC execution as a three-phase execution: the *call-setup* phase, the *remote-execution* phase, and the *call-return* phase. For a RPC, the call-setup phase begins from the time when the `EV_RPC_MARK_START` event is generated up to the generation of the `EV_GLOBAL_MEMBER_START` event. The remote-execution phase is the execution phase between the generation of the events `EV_GLOBAL_MEMBER_START` and `EV_GLOBAL_MEMBER_FINISH`. Finally, the call-return phase refers to the time between `EV_GLOBAL_MEMBER_FINISH` and `EV_RPC_MARK_FINISH`. This notion extends to PO construction and destruction as well.

To illustrate the point, suppose the function `function3()` in processor object P_1, as illustrated in Figure 4.16, is invoked by a thread in another processor object P_2. Figure 4.17 illustrates the instrumentation of the source code. By having the two

```
IdType RECORD_class::determine_PO_thread_id (
                         IdType from_node_id,
                         IdType from_PO_id,
                         IdType from_thread_id)
{
   if (this->processor_id != from_node_id ||
       this->PO_id        != from_PO_id)
      return (this->get_thread_id());
   else
      return (from_thread_id);
}

...

int SomePO_Class::function3 (IdType Ann_FromWhichNode_Id,
                             IdType Ann_FromWhichPO_Id,
                             IdType Ann_FromWhichThread_Id,
                             IdType Ann_File_Id,
                             IdType Ann_Line_No,
                             [function param-list])
{
   Ann_Thread_Id = Instr.determine_PO_thread_id
                         (Ann_FromWhichNode_Id, Ann_FromWhichPO_Id,
                          Ann_FromWhichThread_Id);
   EntityProfiler ep (EV_GLOBAL_MEMBER_START, Instr.processor_id,
                      Instr.PO_id, Ann_Thread_Id,
                      Ann_FromWhichNode_Id,
                      Ann_FromWhichPO_Id,
                      Ann_FromWhichThread_Id, ... );
   ...

   [user's code]
}
```

Figure 4.16. *Instrumenting a member function of a processor object class.*

```
/*---------- Original code -----------
    ...
    a = gptr->function3 ([some params]);
    ...
 *--------------------------------------

// The instrumented code
...
Instr.generate_trace (EV_RPC_MARK_START, ... );
a = gptr->function3 (Instr.processor_id, Instr.PO_id,
                     Ann_Thread_Id, 4, 57,
                     [some params]);
Instr.generate_trace (EV_RPC_MARK_FINISH, ... );
...
```

Figure 4.17. *Marking the start and end of a RPC.*

additional events, both RPC start overhead and RPC finish overhead time can be calculated. The diagram, illustrating the generation of the events, is given in Figure 4.18.

The above approach, however, has a minor drawback, as illustrated in Figure 4.19. The figure shows that for the two functions `function1()` and `function2()`, only *one* pair of `EV_RPC_MARK_START` and `EV_RPC_MARK_FINISH` events is generated. Fortunately, a simple re-arrangement of the code can solve this problem. This is shown in Figure 4.20.

4.2.3.6 Instrumenting Synchronous Threads.

For synchronous threads, i.e. threads spawned by the `par` and `parfor` constructs, instrumentation is applied both to the `par` and `parfor` blocks, and to the individual thread.

For a `parfor` block, the start of the block is recorded as the `EV_PARFOR_START` event, and the end of the block the `EV_PARFOR_FINISH` event. Unlike the instrumentation for functions, the automatic object of type `EntityProfiler` need not be used. The reason is that the entry and exit points are fixed. The entry point for a `parfor` block is always

Caller PO

EV_RPC_MARK_START generated here

Caller is blocked here

EV_RPC_MARK_FINISH generated here

Time flow

Called PO

EV_GLOBAL_MEMBER_START generated here

function3() starts

function3() ends

EV_GLOBAL_MEMBER_FINISH generated here

Figure 4.18. *The generation of events for a RPC.*

```
/*----------- Original code -----------
    ...
    a = gptr->function1 ([some params]) +
        gptr->function2 ([some params]);
    ...
 *-------------------------------------

// The instrumented code
...
Instr.generate_trace (EV_RPC_MARK_START, ... );
a = gptr->function1 (Instr.processor_id, Instr.PO_id,
                     Ann_Thread_Id, 4, 89,
                     [some params]) +
    gptr->function2 (Instr.processor_id, Instr.PO_id,
                     Ann_Thread_Id, 4, 90,
                     [some params]);
Instr.generate_trace (EV_RPC_MARK_FINISH, ... );
...
```

Figure 4.19. *A drawback in RPC instrumentation.*

```
/*---------- Re-arranged code -----------
    ...
    SomeDataType a;
    ...
    a  = gptr->function1 ([some params]);
    a += gptr->function2 ([some params]);
    ...
 *-----------------------------------------

// The instrumented code
...
SomeDataType a;
...
Instr.generate_trace (EV_RPC_MARK_START, ... );
a = gptr->function1 (Instr.processor_id, Instr.PO_id,
                     Ann_Thread_Id, 4, 89,
                     [some params]);
Instr.generate_trace (EV_RPC_MARK_FINISH, ... );

Instr.generate_trace (EV_RPC_MARK_START, ... );
a += gptr->function2 (Instr.processor_id, Instr.PO_id,
                      Ann_Thread_Id, 4, 90,
                      [some params]);
Instr.generate_trace (EV_RPC_MARK_FINISH, ... );
...
```

Figure 4.20. *Simple re-arrangment of source-code for RPC instrumentation.*

```
Instr.generate_trace (EV_PARFOR_START, Instr.processor_id,
                       Instr.PO_id, Ann_Thread_Id, ... );
parfor (int i=0 ; i<5 ;i++)  {
   ...
   [user's code here]
   ...
} // implicit barrier synchronisation here
Instr.generate_trace (EV_PARFOR_FINISH, Instr.processor_id,
                       Instr.PO_id, Ann_Thread_Id, ... );
```

Figure 4.21. *Instrumentation of a* **parfor** *or* **par** *block.*

at the block beginning, and the exit point always at the end of the block[13]. This is illustrated in Figure 4.21.

Each thread spawned by a `parfor` block must, in turn, be instrumented. For each such thread, a new thread identifier must be obtained by using the function `get_thread_id()` of the object `Instr` (the global instantiation of the class `RECORD-_class`, Figure 4.5). Such an instrumentation method is shown in Figure 4.22. Note that the field `Function_Id` of the automatic object `ep` is set to "-1", since a thread is not a function.

Instrumenting a `par` block is similar to instrumenting a `parfor` block, except that each program statement inside the `par` block is instrumented by introducing a new scope region for each program statement inside the block. This is illustrated in Figure 4.23.

4.2.3.7 Instrumenting Asynchronous Threads.

Instrumenting an asynchronous thread (i.e. a thread created by the `spawn` construct) is quite straightforward. The `spawn` construct can only be used to spawn a thread executing a function which returns a `void` result (Section 3.1.1.2). Therefore, instrumenting the `spawn` construct essentially means instrumenting the function used by the construct. Similar to the instrumentation of `par` and `parfor` threads, a new thread

[13] An implicit synchronisation barrier is placed at the end of the scope of such a **par** or a **parfor** block (Section 3.1.1).

```
Instr.generate_trace (EV_PARFOR_START, ... );
parfor (int i=0 ; i<5 ;i++)  {
   IdType Ann_Parent_Id = Ann_Thread_Id;
   //--- get a new thread-id
   IdType Ann_Thread_Id = Instr.get_thread_id ();
   EntityProfiler ep (EV_PARFOR_THREAD_START,
                   Instr.processor_id, Instr.PO_id, Ann_Thread_Id,
                   Instr.processor_id, Instr.PO_id, Ann_Parent_Id,
                   -1, 5, 56);
   ...
   [user's code here]
   ...
} // implicit barrier synchronisation here
Instr.generate_trace (EV_PARFOR_FINISH, ... );
```

Figure 4.22. *Instrumenting a* **parfor** *block and its threads.*

identifier is also needed for each `spawn` statement. An additional scope is introduced so that the destructor of the profiler object `ep` of the class `EntityProfiler` is called upon block termination. This approach is illustrated in Figure 4.24.

The introduction of a new scope, as used for instrumenting *spawn* threads and *par* threads (Section 4.2.3.6), has a minor unfortunate consequence, especially if the new scope encloses a variable or data structure declaration. Figure 4.25 illustrates the case. In the figure, the user's code is instrumented such that the newly introduced scope serves to make the variable "a" local to the block. This forced locality could be dangerous, particularly if the code immediately after the instrumented block assumes the usage of the variable. This unfortunate effect may be nullified through compiler support. Fortunately, such a situation can be avoided by a simple re-arrangement of the source code, as shown in Figure 4.26.

4.2.4 Observations

Based on the experiments conducted during implementation, some remarks can be made. The instrumentation subsystem as described above is quite powerful, in that it can automatically extract important program elements and automatically instrument

```
//----- The original code -----//
//  par {
//     [statement 1]
//     [statement 2]
//     ...
//  }

//----- The instrumented code -----//
Instr.generate_trace (EV_PAR_START, ... );
par {
   //---> block 1
   {
      IdType Ann_Parent_Id = Ann_Thread_Id;
      //--- get a new thread-id
      IdType Ann_Thread_Id = Instr.get_thread_id ();
      EntityProfiler ep (EV_PAR_THREAD_START,
                     Instr.processor_id, Instr.PO_id, Ann_Thread_Id,
                     Instr.processor_id, Instr.PO_id, Ann_Parent_Id,
                     -1, 5, 56);
      [statement 1]
   }

   //---> block 2
   {
      IdType Ann_Parent_Id = Ann_Thread_Id;
      //--- get a new thread-id
      IdType Ann_Thread_Id = Instr.get_thread_id ();
      EntityProfiler ep (EV_PAR_THREAD_START,
                     Instr.processor_id, Instr.PO_id, Ann_Thread_Id,
                     Instr.processor_id, Instr.PO_id, Ann_Parent_Id,
                     -1, 5, 57);
      [statement 2]
   }
   ...
} // implicit barrier synchronisation here
Instr.generate_trace (EV_PAR_FINISH, ... );
```

Figure 4.23. *Instrumenting a* **par** *block and its threads.*

```
//----- Initial code -----//
//   void f ([user's params])
//   {
//      [user's code]
//   }
//   ...
//   spawn (f ( [user's supplied params] );

//----- Instrumented code -----//
{  // <--- scope introduction
   IdType Ann_Parent_Id = Ann_Thread_Id;
   //--- get a new thread-id
   IdType Ann_Thread_Id = Instr.get_thread_id ();
   EntityProfiler ep (EV_SPAWN_THREAD_START,
                  Instr.processor_id, Instr.PO_id, Ann_Thread_Id,
                  Instr.processor_id, Instr.PO_id, Ann_Parent_Id,
                  -1, 5, 56);
   spawn (f (Instr.processor_id, ...
              [user's supplied params] );
}
```

Figure 4.24. *Instrumenting a* **spawn** *block.*

```
/*--------- The initial code ---------*
    [code block 1]
    ...
    int a = some_function();
    ...
    [code block 2]
 *------------------------------------*/

 //--- The instrumented version
    [code block 1]
    {  // <--- introducing a new scope
       ...
       int a = some_function([instrumented params]);
       ...
    }  // scope end
    [code block 2]
```

Figure 4.25. *An unfortunate consequence of introducing a new scope.*

```
/*--------- The re-arranged code ---------*
    int a;
    [code block 1]
    ...
    a = some_function();
    ...
    [code block 2]
 *---------------------------------------*/

 //--- The instrumented version
    int a;
    [code block 1]
    {  // <--- introducing a new scope
       ...
       a = some_function([instrumented params]);
       ...
    }  // scope end
    [code block 2]
```

Figure 4.26. *Simple re-arrangement of source code as a way out.*

a program with powerful probes. This is significant, because CC++ is a declarative language with virtually no support for program visualisation. This is in contrast, for example, with many imperative and message passing systems (such as PVM [65] and PThreads [100]) in which interactions among program elements are effected by issuing specific library primitives. Generally, these primitives are available to programmers for instrumentation. Such manual instrumentation can usually be done relatively easily. This is, perhaps, the primary reason that instrumentation and visualisation of message-passing programs generally focus exclusively on these primitives.

Although the instrumentation method previously described is powerful, there are some limitations. Firstly, these methods may be insufficient when low-level information, such as data structure access, is required. Furthermore, due to the approach used, the activities of some CC++ entities, particularly the transfer functions, cannot be traced. In this case, compiler support, such as that provided by the pC++ language environment [13, 97] is required.

Secondly, the instrumentation methods have some minor drawbacks. These pertain

to the possibility of improperly instrumenting RPC events (Section 4.2.3.5), and the introduction of new scope regions (Section 4.2.3.7). However, as described in the sections, these minor unfortunate consequences can be easily rectified by simple re-arrangements of the source code. A better solution to nullify such effects may need full compiler support.

Finally, the instrumentation methods described in this chapter do not cater to customised instrumentation. In other words, insertion and execution of instrumentation probes can not be controlled by users. Such control, however, is important because it enables users to control the amount of perturbation that can occur to instrumented programs [111]. Provision of such control is outside the scope of Visor++. Incorporation of the control can, perhaps, be experimented with in future enhancements of Visor++.

The instrumentation subsystem has been successfully implemented by using the approach described in [138]. A program to be instrumented is first parsed into its associated syntax graph. The graph is then annotated in the manner described in the previous sections of this thesis. The resulting annotated graph is then written into output files which constitute the instrumented version of the program.

Appendix A provides a simple example program, along with its instrumented version. The program is simple, yet provides a compact summary of the instrumentation approach used in Visor++.

4.3 Event Collection Subsystem

The event collection subsystem executes in three phases. First, the event monitoring environment is established. Second, program traces are collected, and third, the traces are consolidated.

4.3.1 Establishing the Monitoring Environment

When an instrumented program is executed, the probes in the program transmit event traces which are captured by the event collection subsystem. The monitoring environ-

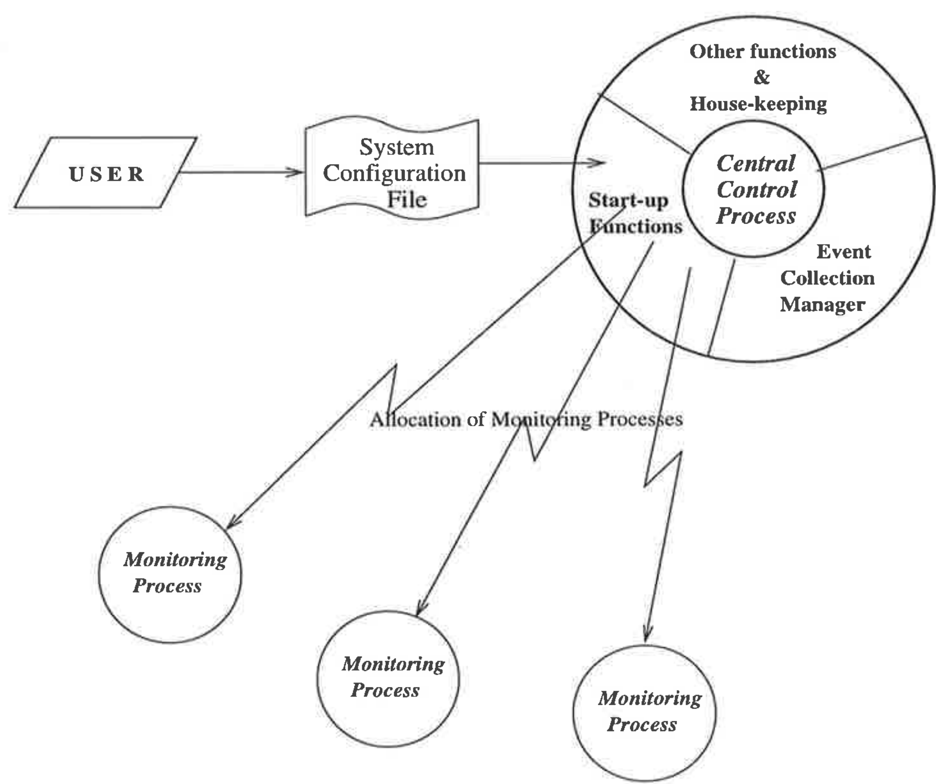

Figure 4.27. *Allocation of monitoring processes.*

ment essentially consists of a *central control process* which allocates one *local monitoring process* to each physical node on which the program to be monitored executes.

Prior to the execution of an instrumented program, the event-collection subsystem must already be active and ready to monitor and collect traces. The central control process is activated by the user by passing along a *system configuration file.* This file contains the names or network addresses of all the physical nodes participating in the computation. The central control process can be activated on any one node, provided that there is a communication path between the node of the central control process and all other nodes specified in the system configuration file. This mechanism is illustrated in Figure 4.27.

The central control process itself consists of three major parts. The first part is the *start-up functions*, which are used for establishing the environment for monitoring. The second part is the *event collection manager*, used for managing the collection of event-traces. Finally, the third part is *house-keeping functions*, used for the internal management of the process itself.

Upon activation, the central control process executes its start-up functions. The process allocates a local monitoring process to each of the nodes as specified in the system configuration file. Subsequently, the central process establishes a dedicated communication path with each local monitoring process. Through this path, the central process allocates a unique node identifier to each local monitoring process. This identifier then becomes the node-identifier for all the traces transmitted by the probes on that particular node (Section 4.2.3.2). After the allocation of the local monitoring processes, the central control process switches its functionality to that of an event-collection manager.

4.3.2 Collecting Traces

When a local monitoring process is allocated to a node, the process performs three activities. First, it establishes an *event queue* which accepts all the traces transmitted by the processor objects of a program being monitored on that node. Next, an *internal buffer* which holds the traces is created. Finally, a *query port* is set up for the processor objects to query/obtain a unique processor object identifier (Section 4.2.3.2). When each local monitoring process is ready on each node, the instrumented program can commence execution.

When an instrumented program executes, it may create one or more processor objects. Upon creation, each processor object contacts its local monitoring process to obtain a unique pair of **<Node-Id, PO-Id>**. This request is made to the monitoring process via its query port. Upon receipt of the request, the monitoring process allocates a unique pair of **<Node-Id, PO-Id>** and sends it back to the processor object. The handling of this request is synchronous. Therefore, it is designed as a lightweight computation.

After a unique **<Node-Id, PO-Id>** is received by a processor object, it continues its execution. During this execution, the probes residing in the processor-object generate program traces. These traces are transmitted *asynchronously* to the event queue of the monitoring process. This means that once a trace record is constructed, it is sent immediately by the associated trace call in the processor object, without blocking.

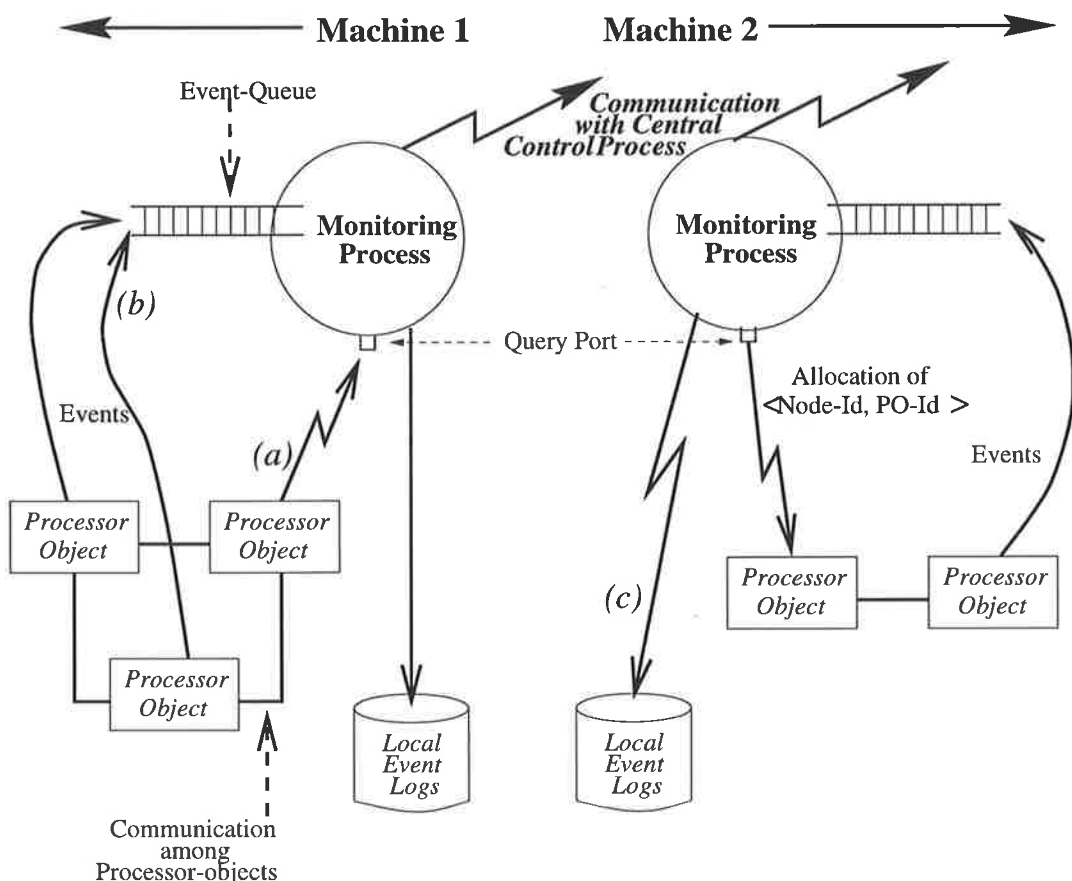

Figure 4.28. *Collecting program traces.*

Therefore, a thread (or PO) is *never* blocked in dispatching trace records.

For each trace arriving at the event queue, the monitoring process buffers it in its internal buffer. When the buffer is full, it is dumped to a local event log file. This happens in the address space of the monitoring process. In other words, the management of the traces are not delegated to each processor object, but rather, centrally managed. This reduces the amount of program perturbation which would otherwise be more invasive.

The whole mechanism, which is repeated until program termination, is shown in Figure 4.28. The figure shows the query port of the local monitoring process, which is being contacted for a unique PO-ID (label a). The figure also shows the POs of the monitored program sending events (trace records) asynchronously to the event queue of the monitor (label b). Finally, the figure shows the events being recorded by the monitoring process into a local event log file (label c).

By the definition of the CC++ (and C++) semantics, the execution of a program is terminated whenever one of the following cases occurs.

1. The `main()` function of the program terminates normally, i.e. the end of the function is reached. Note that CC++ retains the C++ semantics in that there is only one function called `main()` in a program.

2. An `exit()` statement is executed in any part of the program.

3. The program receives an error signal, which is not caught by an exception handler. Such a condition usually forces the program to terminate prematurely. In C++ and CC++, this can be caused, for example, by the `abort()` statement [46].

The above cases can occur in any processor object of a program. If the first or the second case occurs, then by the semantics of C++, the destructor of the static object `Instr`, i.e. instantiation of `RECORD_class` (Section 4.2.3.2), is called automatically. This mechanism is used to signal program termination, hence the termination of monitoring activities, to all local monitoring processes and the central monitoring process. When one of the above two cases occurs in a processor object, the destructor of `Instr` (`Instr.~RECORD_class()`) in that processor object sends an internal termination signal to the local monitoring process before it terminates[14]. This signal is then propagated to the central monitoring process, which then broadcasts it to the rest of the local monitoring processes.

If, however, the third case occurs, then the destructor of the object `Instr` (and, in fact, the destructors of all static objects [46, 124]) is not invoked. When this happens, there is no way for both the central and the local monitoring processes to detect that the monitored program has terminated (abnormally). Two methods can be used to handle this condition. First, the system can supply an option to the user to manually invoke a termination signal to the monitoring system. This is quite simple and straightforward. The second method involves the use of specially designed system interrupts which

[14]The allocation of the static object "Instr" is the first statement in every processor object (Section 4.2.3.2).

generate such a signal upon detecting an abnormal termination. Visor++ adopts the first method.

In any of the above three cases, when a local monitoring process has sent or received a termination signal, the process begins its shutdown sequence. The process carries out three activities. Firstly, it shuts down its event queue, and its query port. Subsequently, it also dumps to a local file any remaining trace records which still reside in its internal buffer (Figure 4.29). Secondly, the local trace file is sent to the central monitoring process for consolidation. This is achieved by using the dedicated link between the local monitoring process and the central monitoring process (Section 4.3.1). Thirdly, both processes engage in an activity to synchronise their clocks (both processes may reside within the same or different nodes). The description of this activity is given in the next section. The shutdown sequence is complete once the third step has been executed.

4.3.3 Consolidating traces

Trace consolidation at the central monitoring process begins when it carries out clock synchronisation with each local monitoring process. The result of clock synchronisation is used to adjust the timestamps of the traces sent by each local monitoring process.

Clock synchronisation is used to determine both the drift and offset between two clocks [2, 43, 110, 126]. The clock synchronisation algorithm used in Visor++ is the Christian algorithm [126]. The idea is to estimate clock drift and offset by using the timestamps from a series of message exchanges between two processes or physical nodes. Such clock synchronisation is carried out between the node in which the central monitor resides with all the other nodes. It is assumed that the clocks as seen by the program probes in all processor objects of each node are synchronised (because they reside on a physical node which uses only one clock).

The clock drifts and offsets are then used by Visor++ to adjust the timestamps of the traces against the clock of the central monitoring process as the reference point. Such a choice is, in fact, arbitrary. Any other clock could have been used.

After the timestamps of the traces are adjusted, it is possible that some of the traces

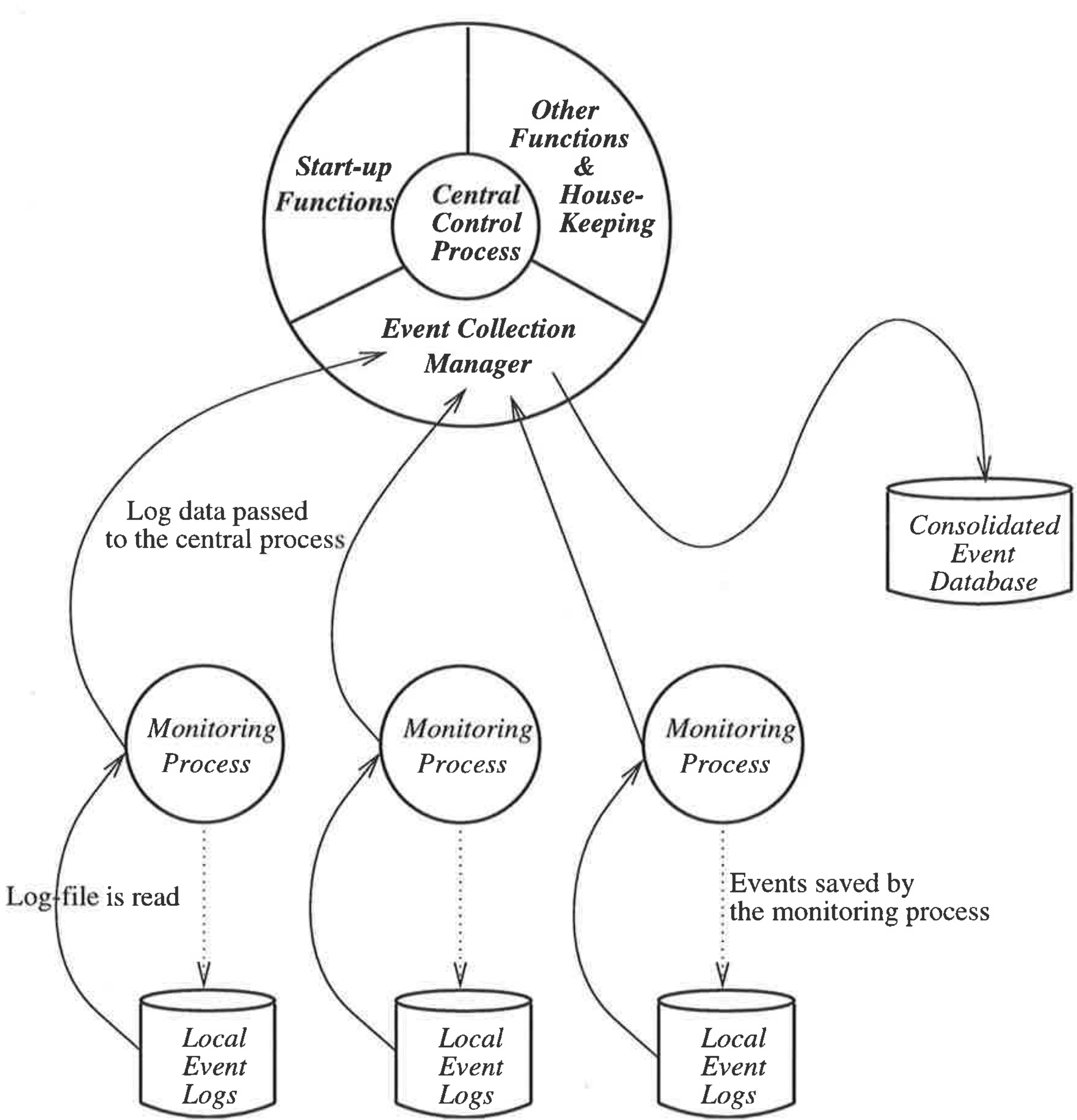

Figure 4.29. *Consolidating program traces.*

still violate the causality relationships [84]. This can cause, for example, an event of type EV_PAR_THREAD_START seemingly to happen before its associated EV_PAR_START event, or a EV_GLOBAL_MEMBER_START before its associated EV_RPC_MARK_START event. Such "impossible" pairs of events are also sometimes called "tachyons" [10, 82][15]. To eliminate the tachyons, a causality analysis is carried out so that the traces obey Lamport's "happened-before" relationships [10, 82, 84].

Causality analysis is performed in two steps. Firstly, a directed acyclic graph (DAG) is formed of all the events, based on their location-IDs. Secondly, the DAG is traversed to correct their timestamps according to the approach used in [10] and [84]. This traversal is repeated until no further causality violations are found.

Figure 4.30 partially shows the DAG formed of one possible instance of the execution of the program in Figure 4.17[16]. The DAG shows the events, with their already synchronised timestamps. However, the happened-before notion is still violated. Assuming that the timing correction is carried out at the microsecond order, the adjusted timestamps of the events are as shown in Figure 4.31. At this point, the traces are ready for input into the visualisation subsystem.

4.3.4 Observations

Several remarks can be made about the implementation of the event collection subsystem. For the processing of trace records, the clock synchronisation as previously described works well. To obtain still better accuracy, however, more refined algorithms may be needed, particularly for program execution on machines with very high clock resolution [43]. Such an approach assumes that the machines which produced the trace files are accessible to carry out clock synchronisation. If such on-line clock synchronisation is not possible, an offline clock synchronisation mechanism is needed, as is described in [2, 43, 110]. Such work, however, typically deals with message-passing

---

[15] In particle physics, a tachyon is an hypothetical particle which can travel faster than the speed of light [146].

[16] See also Figure 4.18.

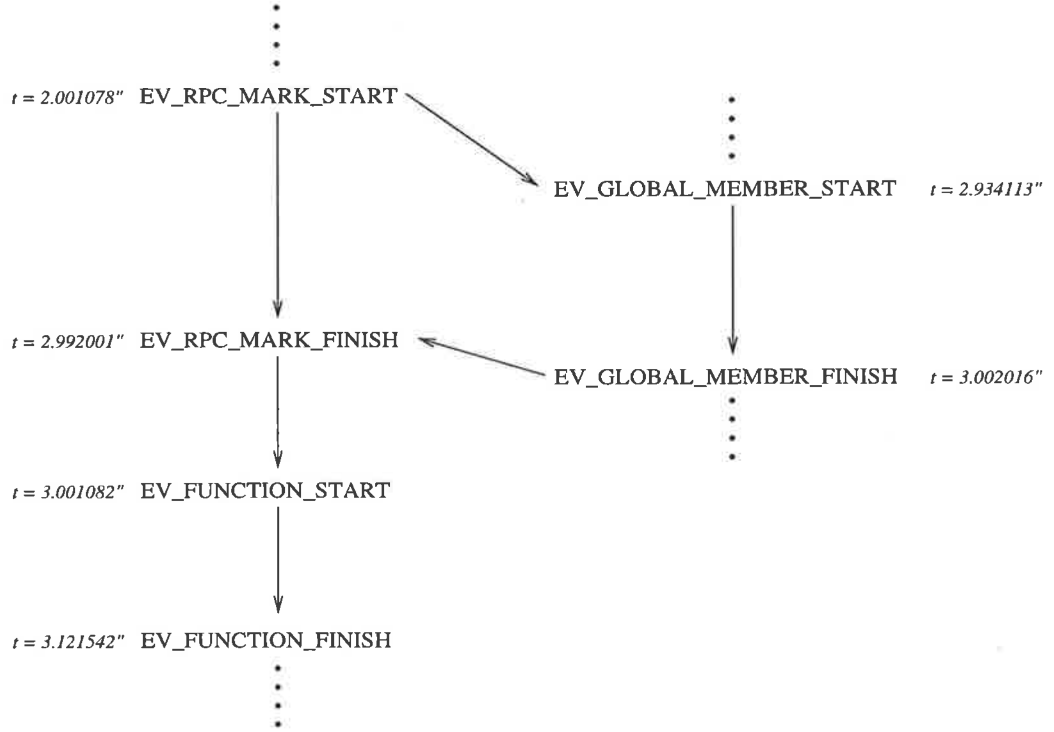

Figure 4.30. *Causality violations in event ordering.*

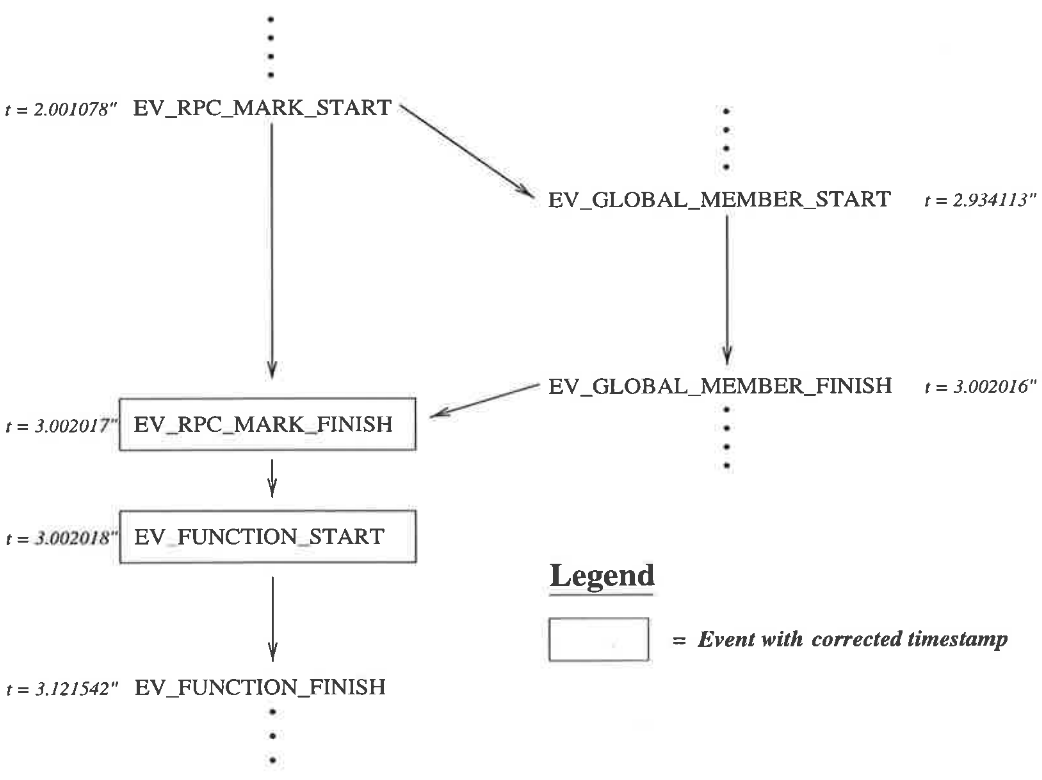

Figure 4.31. *The corrected DAG reflecting the "happened-before" relationships.*

programs which are communication-intensive[17]. For CC++ programs which are typically compute-intensive, new algorithms may be needed. This is because the underlying assumptions for both types of programming styles are quite different.

4.4 Event Visualisation Subsystem

The visualisation subsystem uses two inputs to produce displays. The first one is the static repository, which is obtained during program static analysis. The second is the consolidated event database, which is produced as the result of post-processing program traces. The visualisation subsystem then converts these two inputs into displays.

To make the visualisation subsystem more manageable, it is implemented by using the POLKA program visualisation toolkit from Georgia Institute of Technology [121, 122]. This system is described in the next section, followed by descriptions of the architecture of the visualisation subsystem and the views.

4.4.1 POLKA

POLKA is a general-purpose graphical animation toolkit, particularly suited to building visualisation of concurrent programs. It has two-dimensional and three-dimensional versions, implemented in C++, and executes on top of the X-Window system [1, 72, 102]. POLKA provides many types of objects and primitives for generating smooth, concurrent, overlapping animation in multiple windows. In effect, it provides the capabilities to properly reflect concurrent operations of a program.

Figure 4.32 shows the POLKA classes which are used to create visualisation/animation. Each animation program derives an animation-specific subclass of the abstract class `Animator`. This subclass can contain one or more objects, instantiated from user-defined subclasses of the abstract class `View`, each being a separate animation window.

The animation in each POLKA window is realised by using various animation

[17]For example, the author has tried to adapt one such approach to the CC++ trace files with little success.

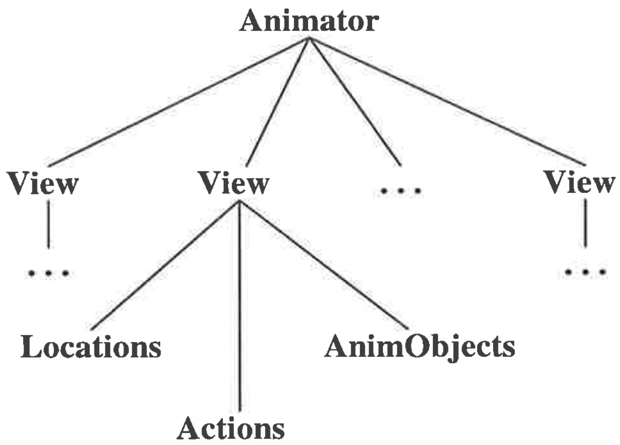

Figure 4.32. *Classes in POLKA, and their* **has-a** *relationships.*

objects (`AnimObject`), operated and animated by using `Location` and `Action` objects, and *animation frames.* `Location` objects are used to specify the locations (coordinates) of animation objects. Movements or animations from one window coordinate to another are then specified by using `Action` objects. These animations are decomposed into a sequence of animation frames, each of which specifies which actions are to be executed on which objects. These frames are indexed by a POLKA timer, which is advanced manually in a program. Figure 4.33 gives a short example of how to create a smooth animation of a circle, moving from coordinate $A(0.2, 0.2)$ to coordinate $B(0.8, 0.8)$.

4.4.2 Architecture and Implementation

The general architecture of the event visualisation subsystem is shown in Figure 4.34. The *visualisation manager* is the entity which drives the whole visualisation. Its operation is controlled by the user. It creates several views each of which is controlled by a *view controller.* Each view controller translates the input stream provided by the visualisation manager into actions to update its view respectively. The views produced are linked together in the sense that they are updated coherently. When the user interacts with any part of the views, changes are propagated to other views as well, whenever applicable.

The architecture of the visualisation subsystem is fully implemented in POLKA. To suit the needs of Visor++, some slight modifications and additions are made to

```
//--- global variables ---//
double x, y;
int time;

//--- Subclass of the "View" class ---//
class CircleView : public View {
private:
   Circle *c;
public:
   ...
   int move_circle (double new_x, double new_y);
};

int CircleView::move_circle (double new_x, double new_y)
{
   Location *loc = c->Where(PART_C);
   float old_x = loc->XCoord();
   float old_y = loc->YCoord();
   float diff_x = new_x - old_x;
   float diff_y = new_y - old_y;
   ...
   // "1" == 1 frame
   Action *mov = new Action ("MOVE", 1, &diff_x, &diff_y);
   c->Program (time, mov);
   ...
}

//--- Subclass of the abstract class "Animator" ---//
class ProgramAnimator : public Animator {
private:
   CircleView *cv;
public:
   ...
   int Controller ();
};
```

Figure 4.33. *Example program of the animation of a circle.*

```
int ProgramAnimator::Controller () {
   ...
   if (!strcmp (AlgoEvtName, "MoveCircle"))
      cv->move_circle(AnimDouble[0], AnimDouble[1]);
   ...
}
...

ProgramAnimator MyAnimator;

//--- Main program ---//
main()
{
   ...
   MyAnimator.RegisterAlgoEvt ("MoveCircle", "ff");
   ...
   x = y = 0.2;
   time = 0;
   while (x < 0.8) {
      x = x + 0.05;
      y = x;
      MyAnimator.SendAlgoEvt ("MoveCircle", x, y);
      time = MyAnimator.Animate (time, 1);
   }
   ...
}
```

Figure 4.33 *(Continued).*

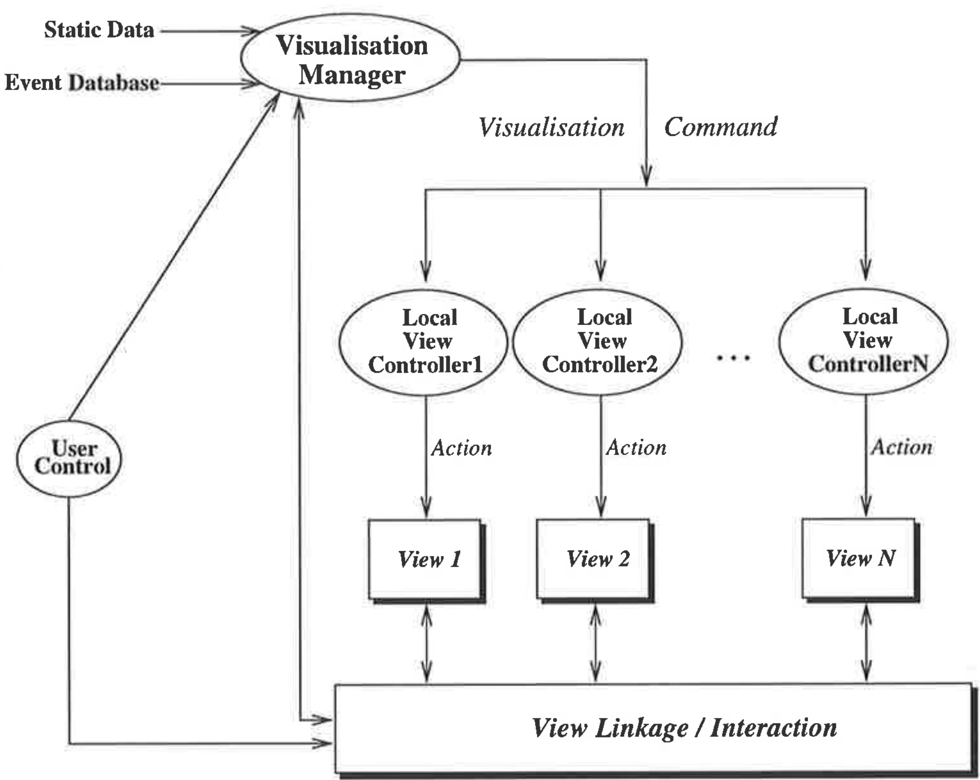

Figure 4.34. *The event visualisation subsystem.*

POLKA. The modifications are relatively minor, the biggest one being the addition of a new subclass of `View` to permit the creation of windows with multiple subwindows. This is necessary for the *Thread View* (Section 4.4.3).

Some views which are otherwise difficult or cumbersome to implement with POLKA are also added. There are three such views. The first one is the *Source-Code View*, which displays the source code of the program being visualised (Section 4.4.3.2). The second view added is the *Class Information View*, which shows the details of a class hierarchy (Section 4.4.3.2). The third are the *auxiliary views*, which are used to display additional information on entities from the dynamic views (Section 4.4.3.3). All three views are similar in that they textually display information which frequently changes. Implementing such views with POLKA would be cumbersome, for it would be memory-intensive and relatively difficult to handle. The Source-Code View, for example, may have multiple source files to handle, and each source file may have hundreds of lines of code. In POLKA, each such line would have to be represented as an animation object, which is memory-intensive and likely to be slow. Implementing such views

with Motif [72] is more convenient and straightforward.

Using the modified POLKA system and the additional modules previously described, the implementation of the visualisation subsystem is straightforward. The visualisation manager is implemented as a subclass of the `Animator` class. Each local view controller is implemented as a set of object methods that translate each visualisation command into local actions. In other words, they translate the command into a set of transformations on their animation objects to effect the required changes in their views.

The visualisation commands issued by the visualisation manager can be divided into two groups. The first is the *visual update command* group, which instructs the local view controllers to change their view displays. Such a command consists of an event with an instruction to the controllers to effect the required changes. The command could also consist of an instruction to the controllers to update and refresh the views to animate the passage of time[18] in the program being visualised. These visual update commands are automatically constructed by the visualisation manager based on its two inputs (static repository and event database).

The second group of commands is the *auxiliary command* group, which instructs the controllers to perform auxiliary actions, such as reducing the speed or pausing the animation. These actions are issued by a user and translated by the visualisation manager into their corresponding auxiliary commands, which are then sent to the local view controllers.

4.4.3 The Views

The CC++ event database along with the program static repository are processed to produce Visor++ views. In designing the view, however, there are several issues which must be considered. These issues are described in the next section.

[18]In Visor++, the timestamps of the trace records are proportionately mapped to the internal POLKA timing. Given any three events A, B, and C from the trace file, Visor++ will map their recorded timestamps t_A, t_B, and t_C into their internal POLKA time t'_A, t'_B, and t'_C. The mapping is such that the ratio $(t'_A - t'_B)/(t'_B - t'_C)$ is maintained to be as close as possible to $(t_A - t_B)/(t_B - t_C)$. In effect, the passage of time between any two events is also reflected and animated by Visor++.

4.4.3.1 Design Considerations

In designing the views, two considerations must be taken into account: the *view contents and presentation structures*, and *the human-computer interaction* issues.

As discussed in Chapter 1, there are two types of mental model which can be formed of a program: the original mental model, and the execution mental model. To help form (or reconstruct, in the case of the original mental model) both mental models, Visor++ provides three types of views: the program *static views*, the program *dynamic views*, and the *auxiliary views*. The overall Visor++ appearance is shown in Figure 4.35.

The static views depict the static aspects of a program. These include the class information, and the source code. These views are generated by using the program static repository. The dynamic views, however, depict the execution aspects of the program. The views are mainly generated by using the event database. Both the static and dynamic views are strongly inter-related and coherent.

The static and dynamic views are structured and presented in a hierarchical structure. Such organisation can make the incorporation of either horizontal or vertical expansion of the views more natural. Such a presentation layout is chosen because hierarchical structures, so far, seem to be the most natural way for presenting complex information [16].

To make the presentation of the views more effective and helpful to users, the third type of view, the auxiliary views, are used. The purpose of the auxiliary views is to help users to better grasp the meaning of the entities presented in other views. This, in turn, enables the user to form a consistent and more compact mental representation of the program. These views functionally serve to help "inform" users of the meanings of the entities in the views. Another way is to look at the auxiliary views as a "glue" of the static views with the dynamic views.

Both the static and the dynamic views are grouped into high-level and low-level views. High-level views are those which describe computation-wide program activity or components, while low-level views describe those of lower-level program entities. Such a division and the relationships among the static, dynamic and the auxiliary views are illustrated in Figure 4.36. More detailed descriptions of the static views, the dynamic

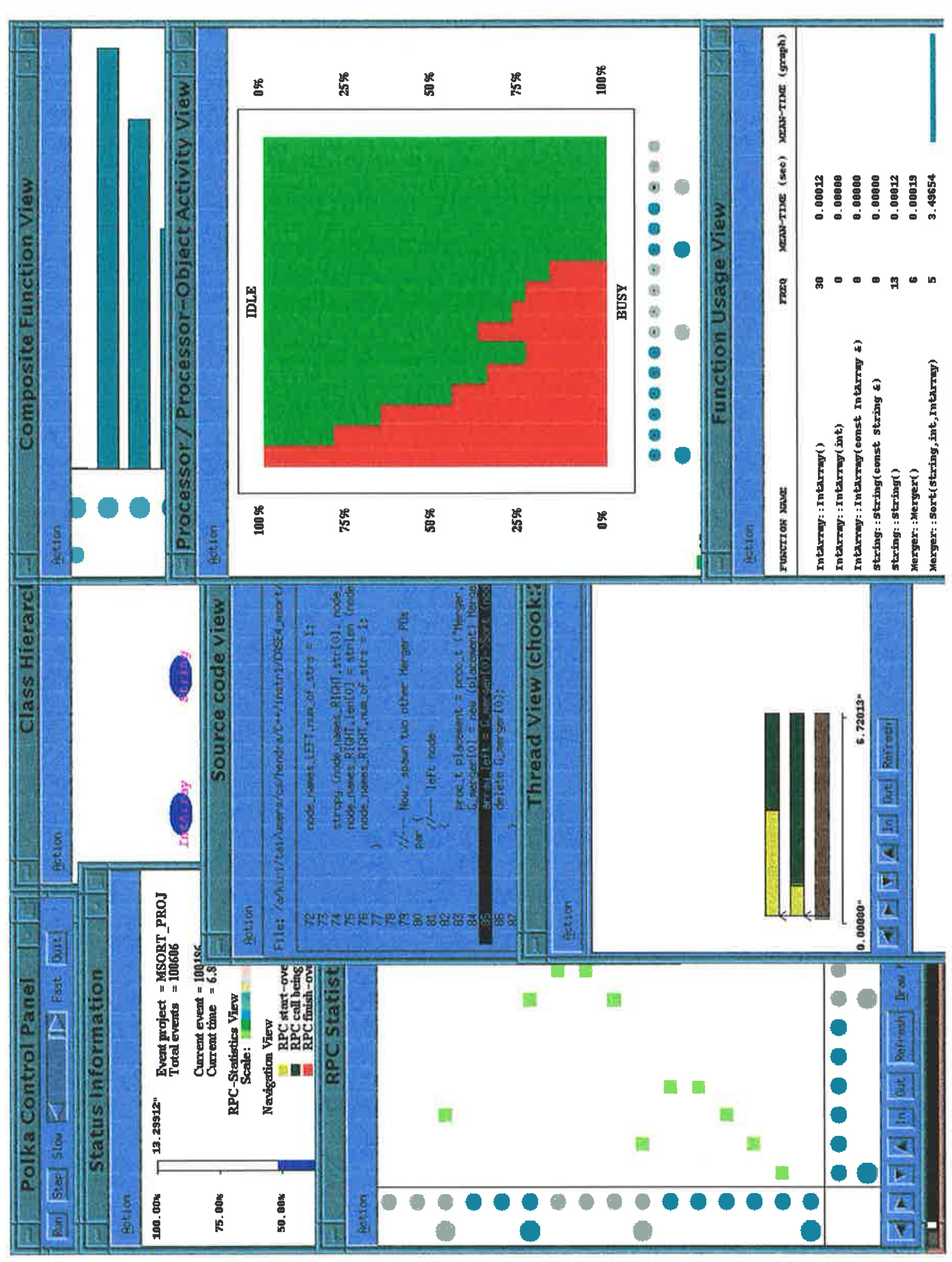

Figure 4.35. *Visor++ in execution.*

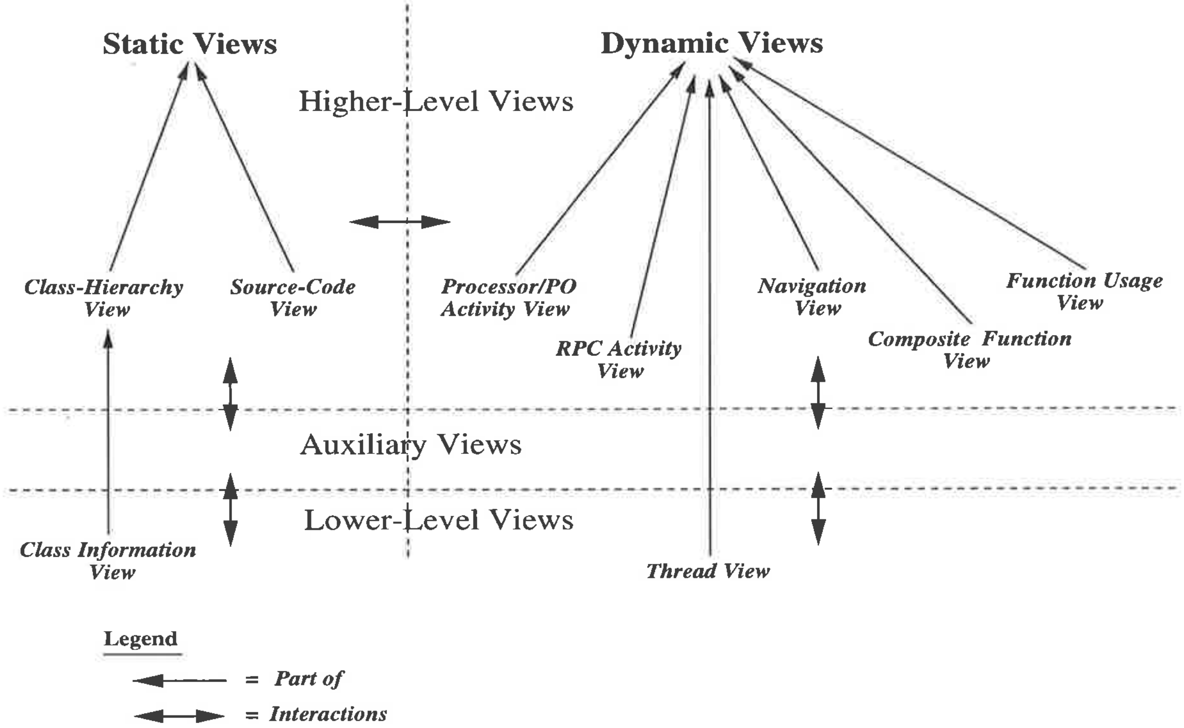

Figure 4.36. *Relationships among views.*

views and the auxiliary views is given in the next sections.

The second consideration when designing the views relate to the human-computer interaction issues. These issues cover two domains as follows.

1. **Colour consistency.** Colour is an important element in a graphical interface. Inappropriate use of colour can seriously reduce the effectiveness of the interface. There are many guidelines which can be used to achieve such effectiveness [99, 141], especially for software visualisation [5]. However, they can generally be summed up into the following notions: simplicity, consistency and clarity. These notions translate into Visor++ view design, as follows:

   - **Economy of colours.** Only a limited number of colours are used. The reason is that the human short-term memory can only handle a limited number of items at one time [59, 95]. This is the reason, for example, that the Class Hierarchy View only uses a single colour for the coloured circles which represent classes (Section 4.4.3.2).
   - **Minimal overlapping of colour usage.** Colours in Visor++ are also used

in a non-overlapping manner as far as possible. For example, the colours representing processor objects are not used for other purposes.

2. **Presentation consistency.** The views and the entities inside the views are also presented consistently. This includes the following:

   - **Shape consistency.** Shapes are maintained, as far as possible, to be consistent in all the views. For example, both physical nodes and processor objects are represented as coloured circles in all the views which contain them.

   - **Behavioural consistency.** Entities representing the same concept but located in different views maintain the same behaviour. For example, the coloured circles representing processor objects may contain a small empty circle (○), a black fill (●), or a bow-tie symbol (⋈) in it to indicate their status. This applies to all views which contain a representation of processor objects. Changes to any one of such entities are propagated in a coherent manner to other views as well. In other words, Visor++ employs *the principle of least astonishment*[19].

   - **Placement consistency.** The placement of entities representing the same concept is also made as consistent as possible. For example, the coloured circles representing both the physical processors and the processor objects are arranged identically in both the Navigation View and the Cumulative RPC Statistics View (Section 4.4.3.4). The arrangement is different, however, in the Processor/Processor-Object Activity View. The reason is that while the Navigation View and the Cumulative RPC Statistics View represent almost identical concepts, the Processor/Processor-Object Activity View represents something entirely different. However, all these views still maintain shape and behavioural consistency.

[19]The use of this term in the discipline of building large-scale object-oriented software can be found in [16].

Visor++ views are designed by using the above principles.

In this thesis, it is assumed that a user interacts with Visor++ by using a three-button mouse[20]. Selecting a graphical entity (a function representation, for example) in a dynamic view (Section 4.4.3.4) by clicking the left button of the mouse causes the associated source code line to be highlighted in the Source-Code View (Section 4.4.3.2). Selecting the entity by using middle button causes additional information on the entity to be displayed in an auxiliary view (Section 4.4.3.3). Finally, using the right button expands the entity into a detailed view.

4.4.3.2 Static Views

Static views are the views of the static properties of a program. These views are constructed by using the program static repository. There are two such views: the *Source-Code View* and the *Class Hierarchy View*.

- **Source-Code View.**

  The Source-Code View displays the original source files of the program being visualised. This view does not operate alone, rather it works in conjunction with other views. When an entity in a dynamic view is selected, the associated source code line in the original program, rather than the instrumented program, is highlighted. This is shown in Figure 4.37. Such source code link-back, however, does not apply to every graphical entity in the dynamic views. For example, physical nodes, as displayed in the Cumulative RPC Statistics View (Section 4.4.3.4), do not have an associated source-code line. Finally, depending upon context, reference to the Source-Code View can be made to *declaration points* or *invocation points*. For example, in the Thread View, when an entity representing a function invocation is selected, the associated invocation point in the source code is highlighted. On the other hand, selecting an function entity in the Function Usage

[20]The operation of a three-button mouse can generally be simulated in a two-button or a single-button mouse. One way to do this is to "map" the mouse button events into other event combinations, as desired. In the X Window environment [1, 102], pressing the middle mouse button can be changed, for example, to a combination of pressing a mouse button and one key on the keyboard. A more obvious example can be found in the Macintosh computer.

Figure 4.37. *The* **Source-Code View.**

View (Section 4.4.3.4) brings up its declaration point instead.

- **Class Hierarchy View.**

  The Class Hierarchy View displays the class hierarchy graph, in which each class is represented as a coloured oval (circle), with a label indicating its name. In the view, CC++ processor-object classes are represented as double-edged ovals. A directed edge with an arrow pointing from an oval representing class A to another oval representing class B indicates that A is the superclass or parent of B. This view is shown in Figure 4.38. By the principle of the economy of colours, only two colours are used in this view. The colour blue is used to represent the class entities, while the colour magenta is used for class names.

  When an oval in the Class Hierarchy View is selected, a *Class Information View* is constructed and displayed. This view contains detailed information on the selected class, such as its member functions, its friend functions, friend classes, and other general information. This is shown in Figure 4.39.

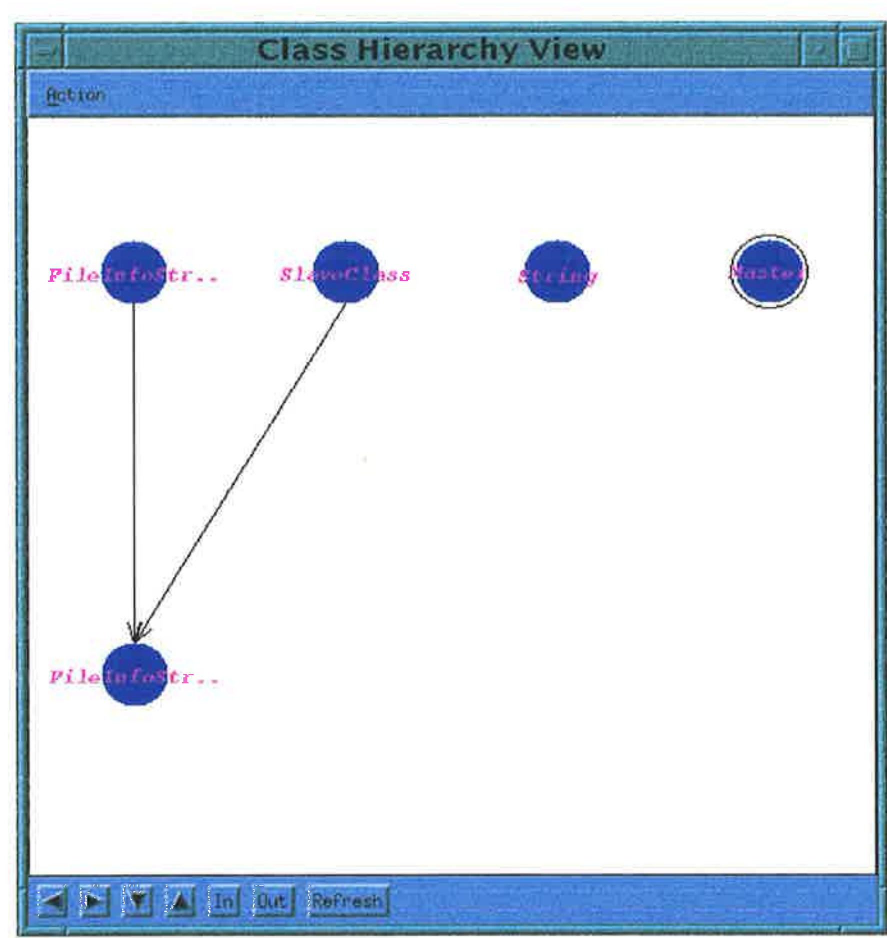

Figure 4.38. *The* **Class Hierarchy View.**

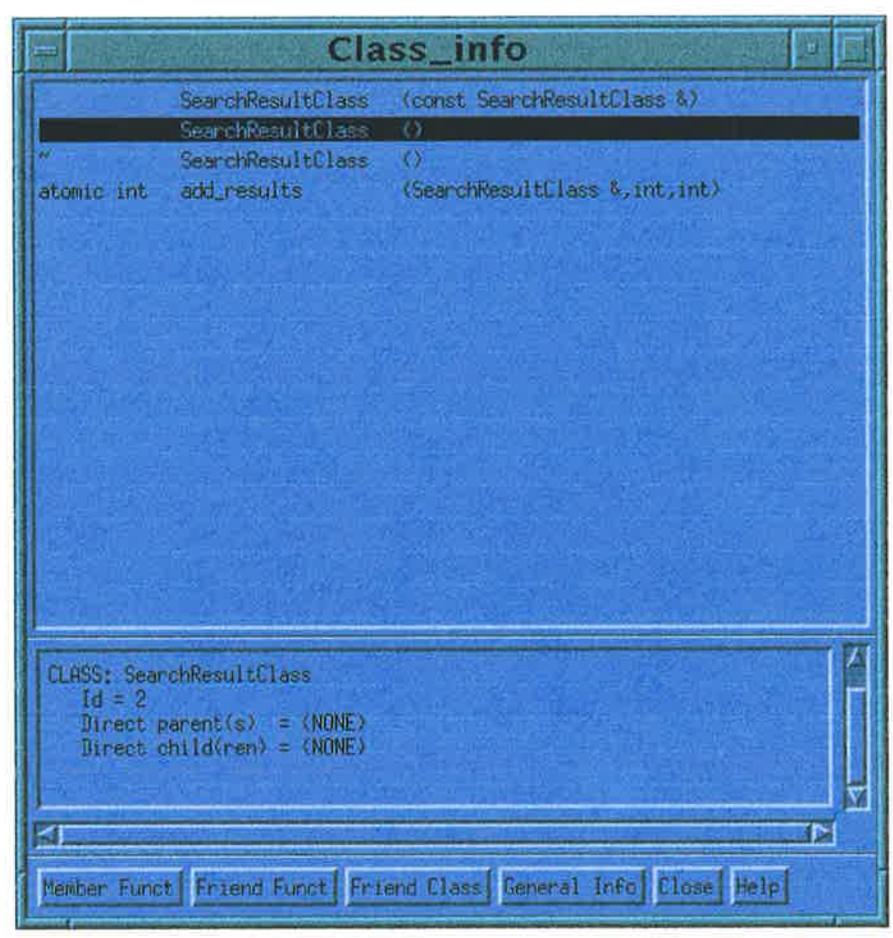

Figure 4.39. ***The*** **Class Information View.**

Figure 4.40. *Information on a node.*

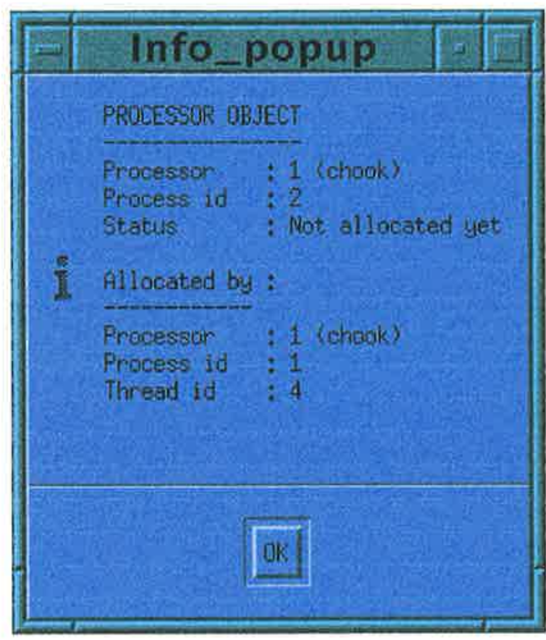

Figure 4.41. *Information on a processor object.*

4.4.3.3 Auxiliary Views

Auxiliary views provide additional information as needed by the user. The additional information is chosen by the user by selecting (clicking the middle mouse button) any graphical entity on the dynamic views. For example, selecting the representation of a physical node brings up an auxiliary view as shown in Figure 4.40. As another example, selecting the representation of a processor object brings up the view as shown in Figure 4.41. Auxiliary views are, therefore, *contextual.*

The next section describes each of the dynamic views. Whenever applicable, the associated auxiliary views will be described concurrently.

4.4.3.4 Dynamic Views

Dynamic views are those views which depict the execution of a program. These views are constructed by using the program event database. As shown in Figure 4.36, there are two types of such views: *low-level views* and *high-level views*.

High-level views describe program activities computation-wide, while low-level views display program activities inside a particular processor object. In particular, the low-level views display thread creation and function invocation. There are two types of low-level views: the *Thread Activity View* and the *Function Stack View*. Since the two views are closely related, they are displayed collectively as a stack of two subwindows of a single view. For brevity, they are collectively referred to as the *Thread View* (Figure 4.42).

1. **Thread Activity View.**

   This view is displayed as the topmost subwindow in the Thread View. The Thread Activity View displays the threads which are active at each point in time. This view is essentially a space-time diagram, in which the vertical axis represents the threads, and the horizontal axis represents time. At the bottom of the view, a line extending to the right shows the program execution time in seconds. Each thread is then displayed as a light-green bar, extending to the right, as execution time advances. Thread creation and destruction is indicated by an up-arrow and a down-arrow respectively. A function invocation by a thread is represented as a segment inside the bar, with a unique colour. If this function calls another function, then the invoked function is, again, represented as another segment, juxtaposed to the right of the caller function's segment. When the invoked function returns, the caller segment continues to be extended to the right, until it returns. In this view, several colours are used, as follows:

   - **Light green**. This indicates that a thread is active, but no function is being invoked (i.e. it is executing its own local code).
   - **Brown**. This colour indicates that a `par` or a `parfor` block is active.

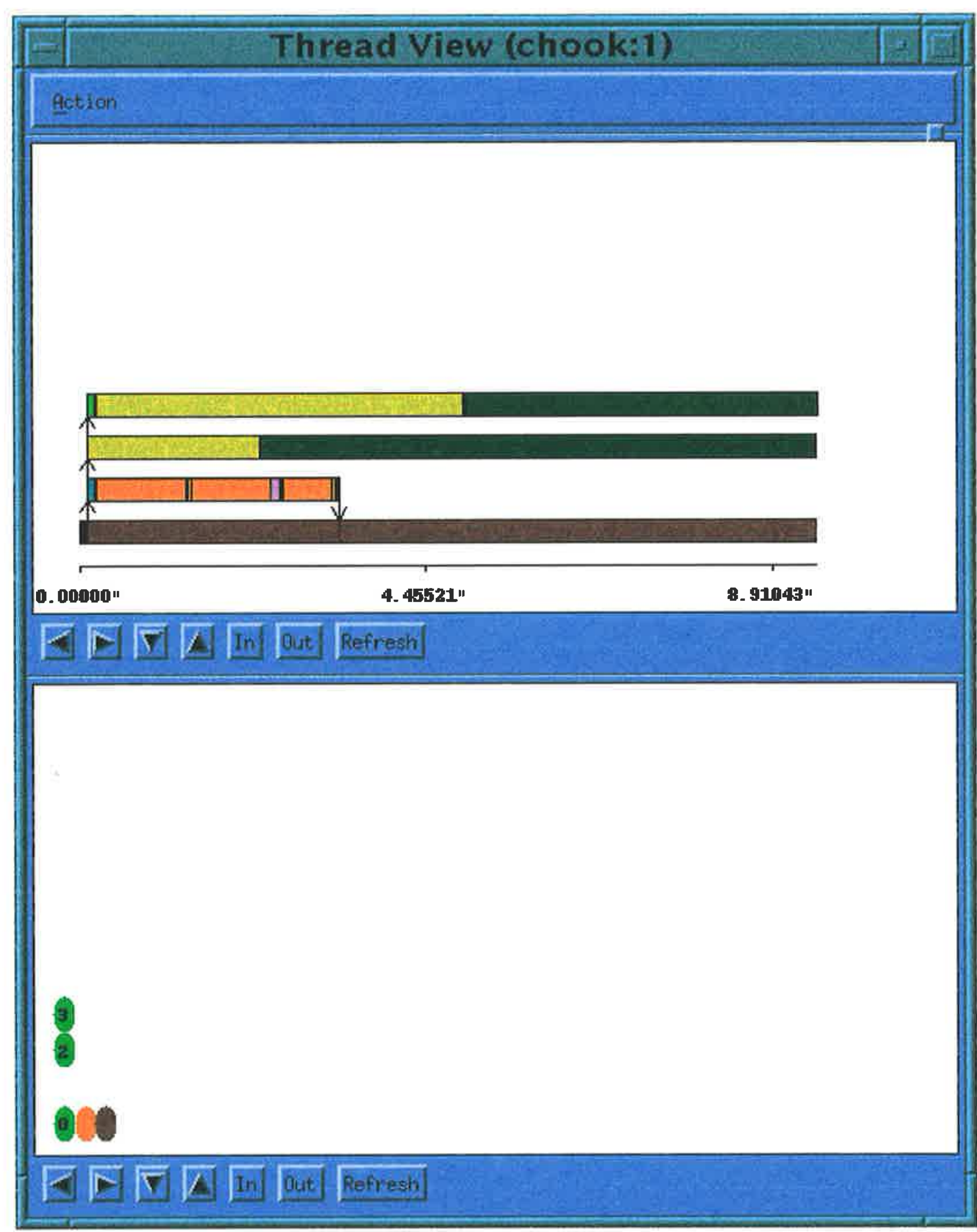

Figure 4.42. *The* Thread View.

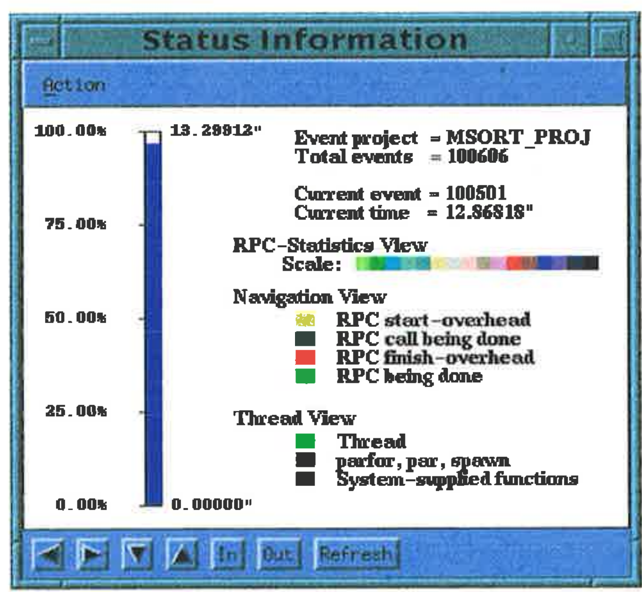

Figure 4.43. *The* **Status Information View.**

- **Yellow, dark green, and red.** The colour yellow indicates that the thread is invoking a RPC in another PO; the thread being in the call-setup phase. The colour dark green indicates that the RPC is in its remote-execution phase, while the colour red indicates the call-return phase (Section 4.2.3.5).
- Other colours indicate function invocations.

This colouring scheme is summarised in a special auxiliary view, the *Status Information View* (Figure 4.43).

2. **Function Stack View.**

The Function Stack View is displayed as the lower subwindow in the Thread View. This view is essentially similar to the Thread Activity View, except that for each thread on the vertical axis, a thread stack grows to the right when a function is called, and shrinks to the left when it returns. Each function being invoked is represented as an oval, with its unique colour. When such an oval is selected (i.e. mouse-clicked), an auxiliary view is popped up as shown in Figure 4.44. Source-code link-back to the original source code can also be exercised to show the invocation point of the function. The same auxiliary views also pop up if a bar segment in the Thread Activity View is selected in the same manner. Similar

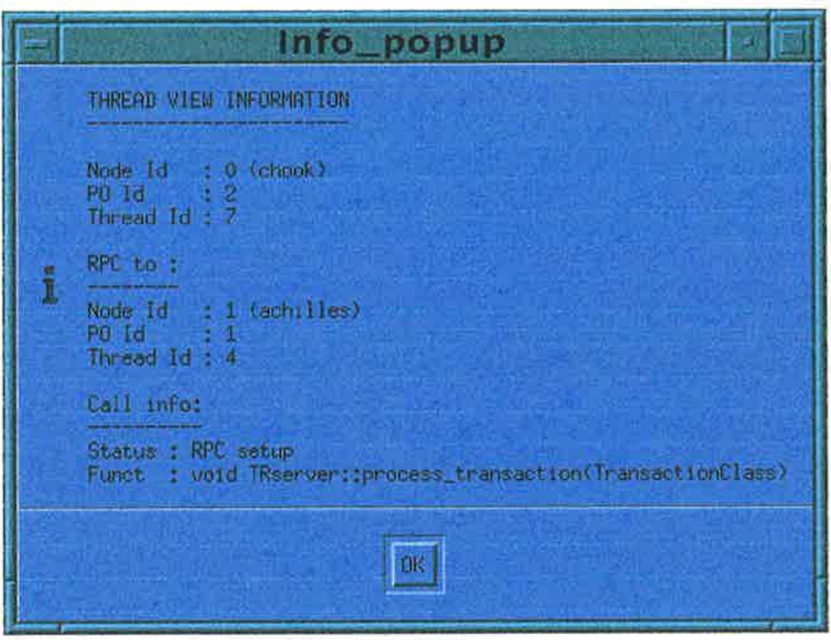

Figure 4.44. *Auxiliary view showing information of a function.*

to the Thread Activity View, if a thread is active but no function is being invoked in it, then only one green oval is displayed for the thread.

Figure 4.42 shows an example of a Thread View. The title bar indicates that this is the Thread View of a processor object executing on the machine "chook", and its PO-ID is 1. There are four threads in the PO. With the auxiliary view, it is revealed that the lower-most bar represents the **main()** thread. This thread executes a **parfor** block which creates three threads. The first of these threads (the second bar from the bottom) executes several functions and is destroyed after approximately 3 seconds of execution. The other two threads execute RPCs to other POs.

The above two subviews can be used to analyse the details of thread execution, including function invocations and thread call structures. Such analysis can reveal some program inefficiencies, for example which functions are most often called, or which functions use excessive amounts of time to execute within a processor object.

It is worth noting that the Thread View of a processor object is not displayed until the user specifically requests so by selecting (i.e. mouse-clicking) a processor object representation in one of the global views, which are described below.

Global views depict program activities at the node/processor level. These include

the following:

1. **Navigation View.**

   The Navigation View is a matrix-like view displaying the RPC activities of the visualised program. As Figure 4.45 shows, this view consists of the X and Y axes, each of which has nodes and processor objects as axis points. The first row of circles below the X-axis are the *PO-circles*. A PO-circle is a coloured circle representing a processor object that participated in the computation of the visualised program. The second row of circles, below the PO-circles are the *node-circles*, each representing a physical node participating in the computation. Each PO-circle may have a smaller circle inside it to indicate its operating status. When a PO-circle does not have such a circle, it means that it has not been allocated yet. A small empty circle (○) denotes that the processor object is being allocated/constructed. Later, it may change to a black fill (•), indicating that the PO is currently active (has been constructed). In the end, each PO is deallocated, which is indicated by a small bow-tie symbol (⋈).

   The node-circles and the PO-circles are arranged in such a way that the PO-circles are clustered together based on the node to which the associated POs belong. For example, Figure 4.45 shows that the first physical node has six POs, all of which are already constructed and active. On the other hand, the third node only has three POs, one of which is being constructed and the other two are not active yet. To accommodate the principle of the economy of colours, only two colours (dark turquoise and grey) are used for both the PO-circles and the node-circles. The use of these colours are for visual differentiation only. The two colours are assigned to the node-circles in an alternating fashion. Each PO-circle, in turn, is assigned the same colour as the node-circle to which it "belongs".

   The Y-axis is similar to the X-axis, i.e. they both represent the same entities with the same graphical properties. The Y-axis effectively is the X-axis rotated ninety degrees counter-clockwise. It is worth noting that the location-IDs (Section 4.2.3.2) are used as the "key" to arrange the placement of the circles. In the

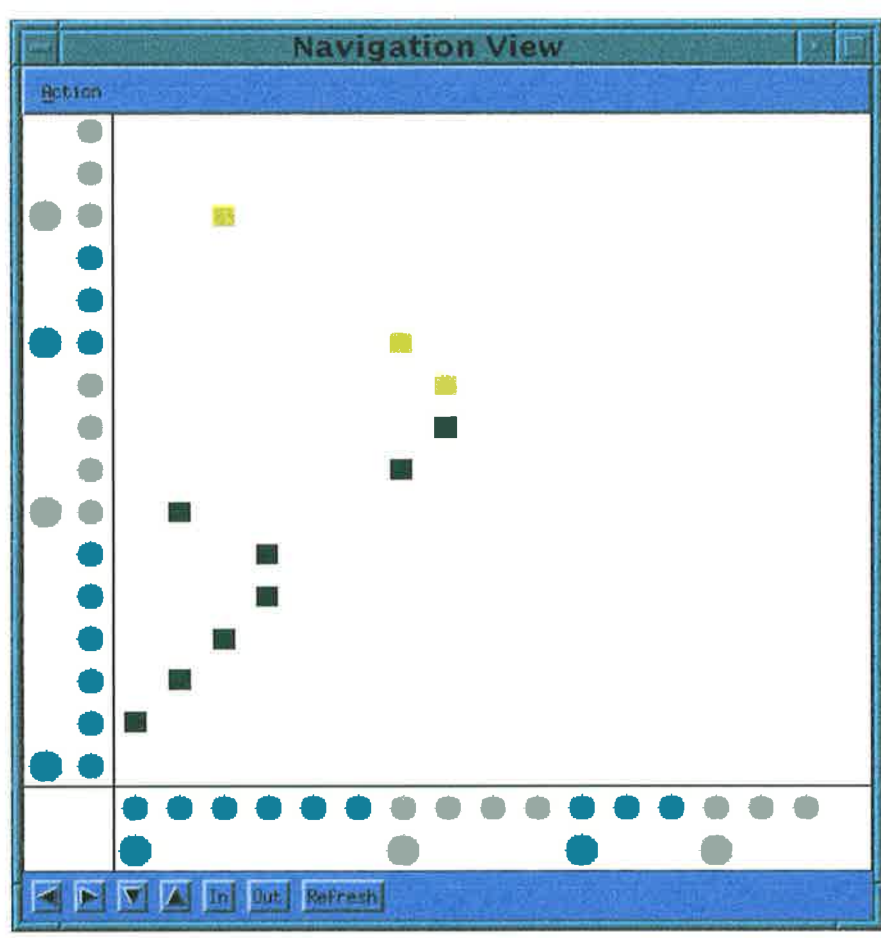

Figure 4.45. *The* **Navigation View**.

view, the node-circles and the PO-circles are arranged in an "ascending" order. These location-IDs are also used in auxiliary views.

The X and Y-axes, as previously described, form a grid system. When both the node-circles and PO-circles are present, a coloured rectangular point in the grid at coordinate (x, y) represents a RPC being made by one PO (represented by a PO-circle at the X-axis position x) to another PO (represented by a PO-circle at the Y-axis position y). Such a rectangular point is given a colour which represents the status of the call. The colouring scheme for the rectangles follows that for RPC bars in the Thread View. However, a PO may place more than one RPC to another PO. If this is the case, then if any one of the RPCs is still active, the rectangle is given a light green colour.

If the number of processor objects in a program is relatively large, then the above display is difficult to discern. Therefore, the user has the option to "collapse" the view. In such a collapsed view, only the physical nodes are represented. A grid coordinate (x, y) then represents the aggregate RPC activities from one node to another. This "collapse" operation is performed by clicking the right button of the mouse on any one of the node-circles. Clicking again on one of the node-circles expands the view. When this view is collapsed or expanded, other views which have similar node-circles and PO-circles (such as the Processor/Processor-Object Activity View) also collapse and expand accordingly.

Each of the PO-circles in this view can also be selected (by clicking the right button of the mouse) to "expand" its view. When this happens, a Thread View is displayed for the associated processor object. Another important operation is to select the entities in the view by clicking the middle button of the mouse. This is used to bring up additional information of an entity in an auxiliary view. Some examples of auxiliary views of nodes and processor objects are shown in Figure 4.40 and 4.41.

Used in conjunction with other views, the Navigation View can be used to visualise the current distribution of RPCs among nodes or processor objects. This

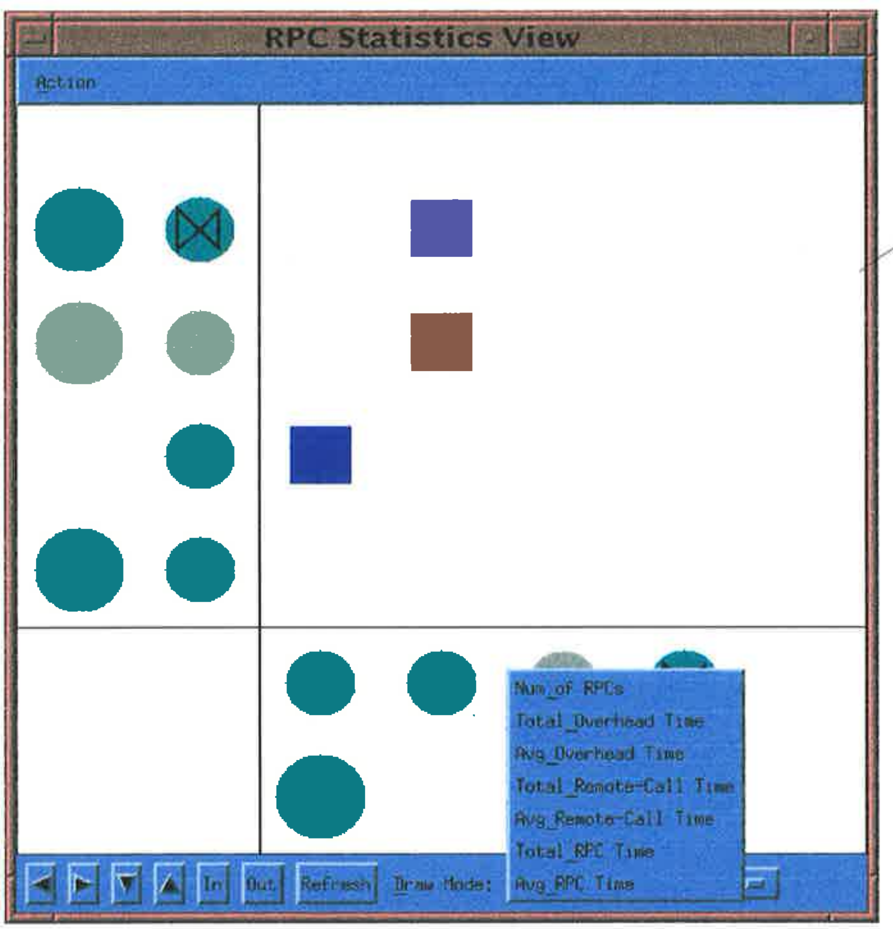

Figure 4.46. *The* **RPC Statistics View**, *with the option menu popped up.*

can be used to re-arrange a program to have a better distribution of communication patterns, for example.

2. **Cumulative RPC Statistics View.**

   This view is similar to the Navigation View, except that it shows the cumulative RPC statistics among nodes and processor objects (Figure 4.46). For each grid point, the smaller the value of the statistics it represents, the brighter its colour. The colour shades, reflecting the relative value of the statistics, is summarised in Figure 4.43.

   There are several types of statistics in this view. To change or toggle the display from one view to another, the user can use the option menu at the bottom right-hand corner of the view. The statistics provided are as follows:

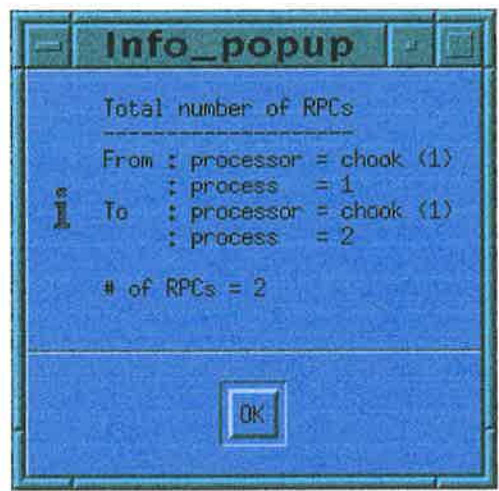

Figure 4.47. *Auxiliary view on the number of RPCs from one PO to another.*

(a) **Number of RPCs.** This shows the number of RPCs made so far between physical nodes or processor objects.

(b) **Total RPC overhead time** and the average value. In this view, RPC overhead time comprises both the call-setup and the call-return overhead times, but excludes the RPC remote-execution times (Section 4.2.3.5). The user can also elect to display the average value.

(c) **Total remote-call time** and its average. In contrast to the total RPC overhead time, this shows the total time spent in the remote-execution phase alone. The user can also opt to display its average.

(d) **Total RPC time** and its average. This displays the total time spent in RPCs, i.e. the summation of RPC overhead times and remote-call times. Once again, its average can also be displayed.

When selected, a grid point in this view can bring up an auxiliary view, such as shown in Figure 4.47.

Used with the Navigation View, this view can show the (cumulative) pattern of the distribution of communication among nodes or processor objects. This is important, since in CC++ RPCs are expensive activities. Minimising or balancing the execution of RPCs is highly desirable. Another use of this view is to

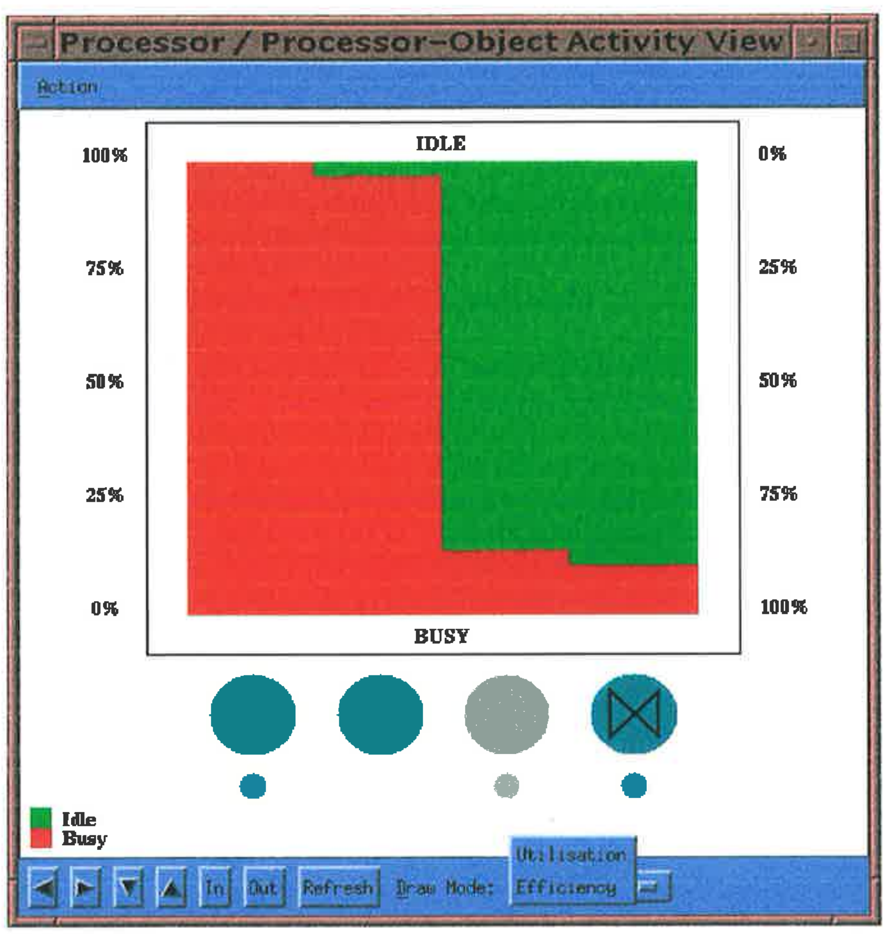

Figure 4.48. *The* **Processor/Processor-Object Activity View**, *with its option menu popped up.*

re-arrange the placements of processor-objects in such a way that those which communicate frequently with each other can be placed on the same physical node.

3. **Processor/Processor-Object Activity View.**

This view shows the percentage of time that processor-objects or physical processors spend in computation, and being idle (Figure 4.48). Selecting one of the display regions brings up an auxiliary view as in Figure 4.49.

In the view, information on computation time (busy time) actually consists of the time that a processor or processor-object spent doing real computation *and* communication. This is in contrast to the visualisation of message-passing programs, in which the two quantities are separated. The reason is that each processor object can consist of more than one thread, whose execution (i.e. either

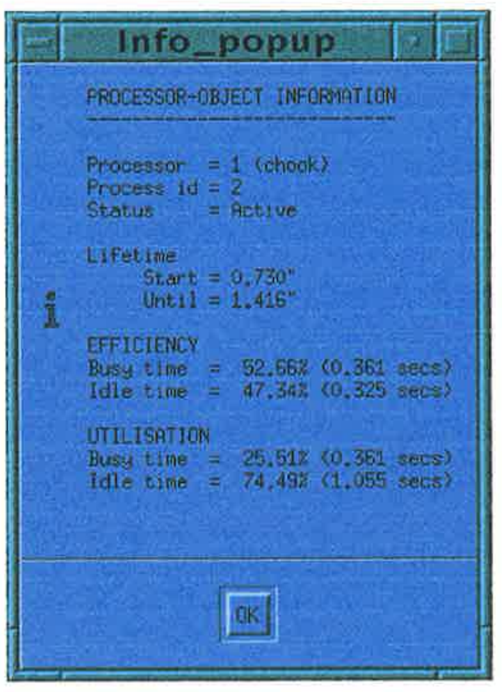

Figure 4.49. *Auxiliary view from the* **Processor/Processor-Object Activity View**.

computation or communication) is automatically interleaved (context-switched) by the system. To present highly precise information on such an activity as in message-passing programs, low-level information on such context switching is required. Acquiring such information is costly, if not impossible. To worsen the situation, there is also a difference between thread blocking caused by thread context switching and the blocking caused by communication wait-time. For these reasons, both computation and communication time are aggregated as computation time.

The above concept is illustrated in Figure 4.50. The figure schematically shows events relating to the execution of a processor-object. The program execution starts at the time t_0, and has been animated in the visualisation up to the time t_6. Meanwhile, the PO starts executing at the time t_1; at the time t_6, it has not been deallocated yet. In this scheme, the computation time (busy time) is defined as $t_{busy} = (t_3 - t_2) + (t_5 - t_4)$. For the PO, two metrics can be defined. In *efficiency statistics*, the fraction of the PO busy time is defined as $F_{busy} = t_{busy}/(t_6 - t_1)$, and the fraction of the PO idle time as $F_{idle} = 1 - F_{busy}$. For *utilisation statistics*,

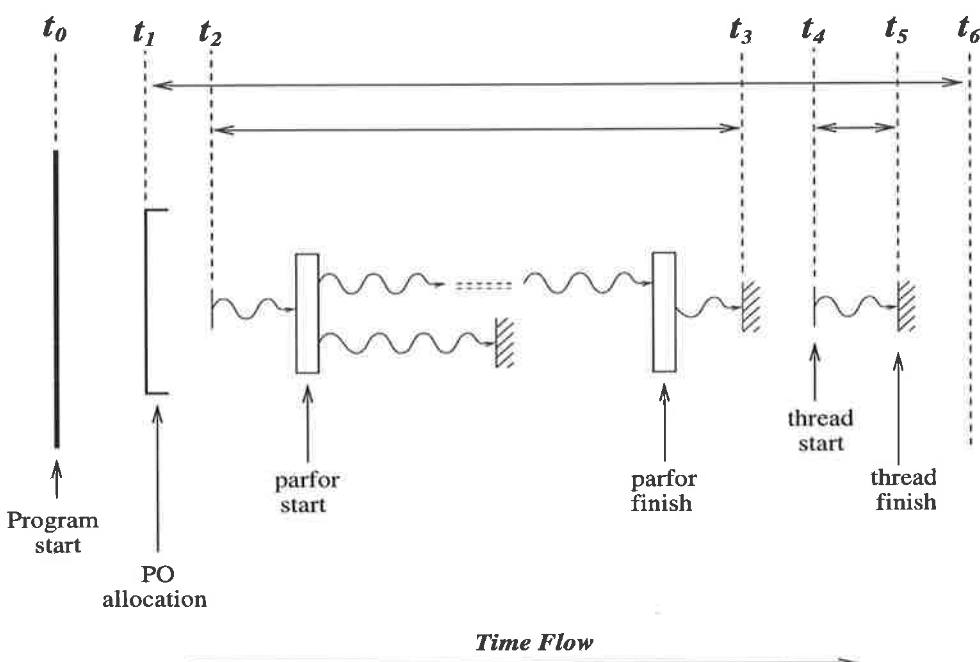

Figure 4.50. *Program execution in a processor-object.*

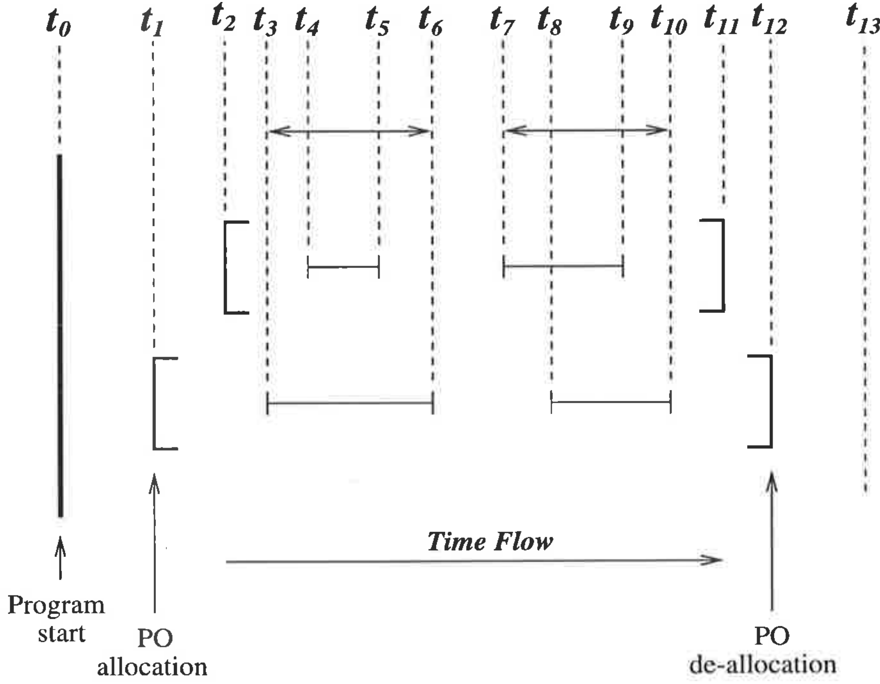

Figure 4.51. *Two processor-objects executing on a processor.*

however, the fraction of busy time is defined as $F_{busy} = t_{busy}/(t_6 - t_0)$, while the fraction of idle time is, again, defined as $F_{idle} = 1 - F_{busy}$. In other words, for the utilisation statistics, time reference is made against the *program* execution time, while for the efficiency statistics, reference is made against the *local* PO execution time. These two metrics are provided to enable the user to carry out both local and global optimisation of the program.

The computation time (busy time) of a physical processor, on the other hand, is defined as the aggregated computation time-spans of the POs it hosts. Figure 4.51 illustrates this concept. The figure depicts one of the processors in a program execution. The program starts execution at the time t_0, and it has been animated up to the time t_{13}. The processor hosts two POs, one of which starts from t_2 to t_{11}, and the other from t_1 to t_{12}. The busy time of the processor is then defined as $t_{busy} = (t_6 - t_3) + (t_{10} - t_7)$. For both the efficiency and utilisation statistics, the fraction of the processor busy time is defined as $F_{busy} = t_{busy}/(t_{13} - t_0)$,

and the fraction of the processor idle time as $F_{idle} = 1 - F_{busy}$. Note that both metrics are similarly defined because processor-level activity is considered to be program-wide activity, i.e. global activity. The metrics, therefore, should refer to the global program timing.

Either the efficiency statistics or the utilisation statistics can be displayed in the view. The statistics to be displayed can be selected by using the option menu button at the lower right-hand corner of the view. When the view is expanded, i.e. when the PO-circles are displayed, the view depicts the relevant statistics for POs. When the view is collapsed, it shows the statistics for processors.

The statistics shown by these views, can be useful for determining the computation efficiency, and for subsequently making the necessary alterations to the program.

4. **Function Usage View.**

   This view (Figure 4.52) shows the frequency of invocations and the average invocation time of all the functions computation-wide. Note that the bar graphs for the average times are scaled to fit into the window. Selecting a function in this view can bring up an auxiliary view as in Figure 4.53. Function declaration in the original source code can also be displayed through link-back to the Source-Code View.

   By the principles set out in Section 4.4.3.1, the bar graphs are assigned a limited number of colours[21], specific only for the representation of functions. The colour assignment is consistent with that in other views, such as the Thread View, and the Composite Function View.

   The Function Usage View can be useful for optimising the functions which are most often used, thereby reducing overall computation time.

[21]The colours can be assigned to the functions by using some "hashing" scheme. For example, if only 5 colours are to be used, then every function starting with the letter *a, f, k, . . .* are assigned the first colour, while those starting with *b, g, l, . . .* are assigned the second colour, etc. Some other grouping scheme, such as based on class membership, may also be used. Visor++ uses the first approach.

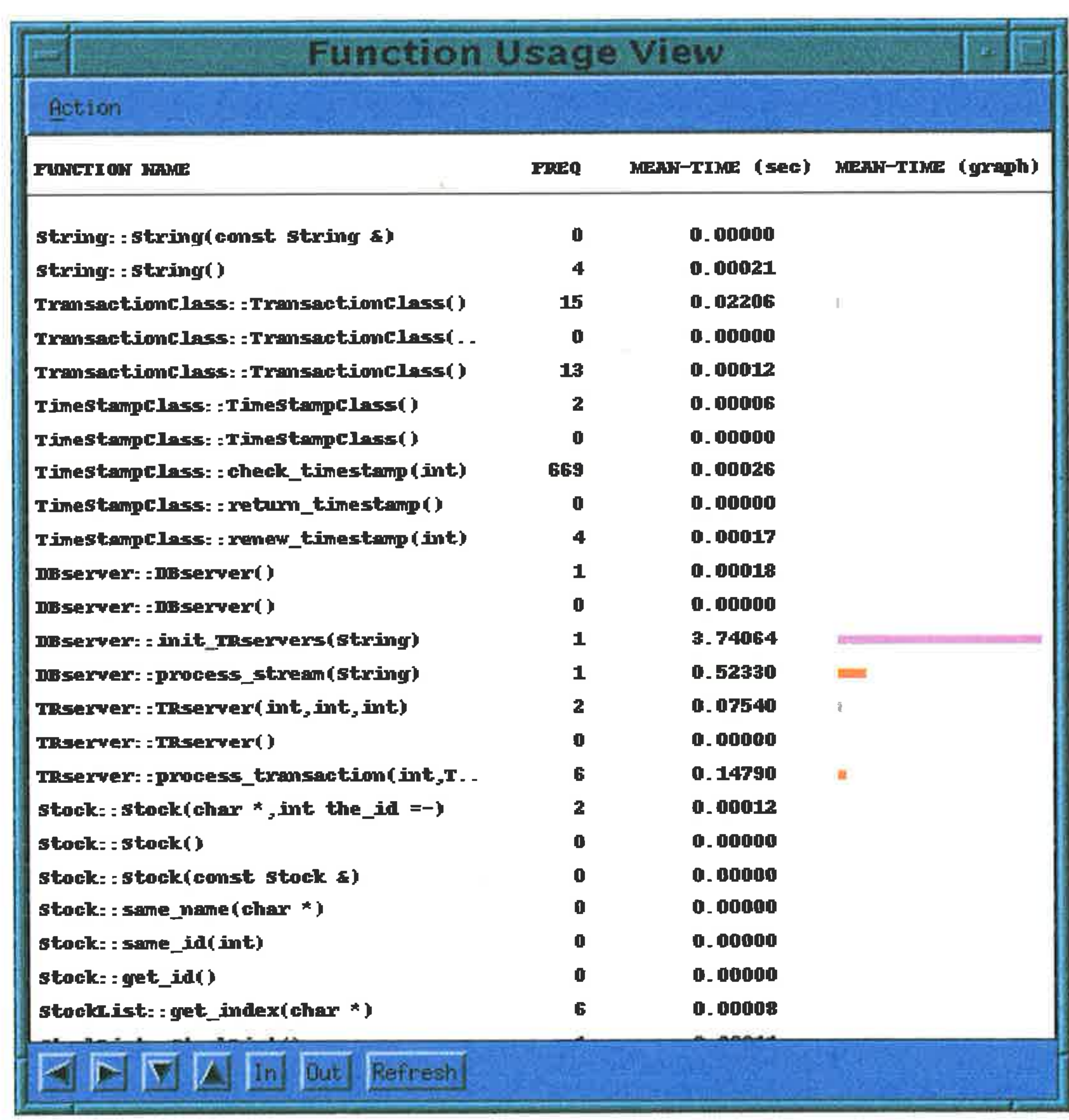

Figure 4.52. *The* Function Usage View.

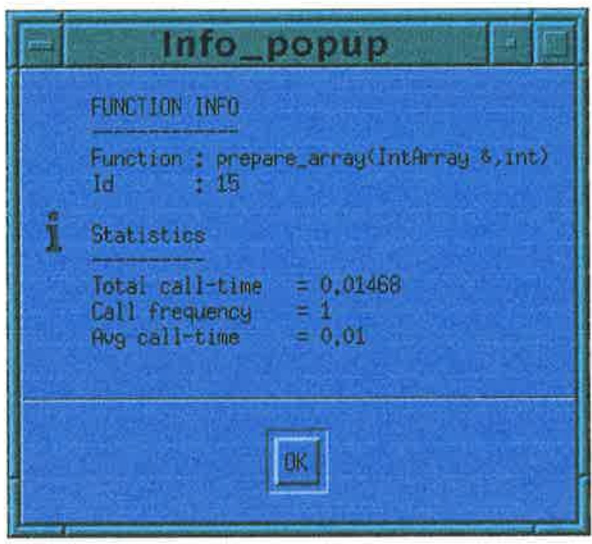

Figure 4.53. *Selecting a function in the* **Function Usage View** *brings up further information.*

5. **Composite Function View.**

The view (Figure 4.54) shows the average invocation time of functions for each node or processor object. The node-circles and PO-circles are placed on the left-hand side of the view, with a bar extending to the right. Each bar depicts a maximum of ten functions with the largest average invocation times compared with other functions in the same node or processor object. Each such function is denoted as a bar segment, having a distinctive colour, with a colouring scheme consistent with that used in other views. For each bar, the function segments are placed in a "descending" order to the right. The bars are drawn to scale to fit into the available window space. Selecting one of the bar segments brings up an auxiliary view as shown in Figure 4.55. Similar to the Function Usage View, source-code link-back can also be exercised.

The colour assignment for the functions in this view is the same as that of the Function Usage View. However, the Composite Function View is different from the Function Usage View in that the previous depicts function usage time per node or processor object. Used together, these two views can identify the particular functions in which a particular node or PO spends the most time. This, in turn, facilitates the improvement of function implementation.

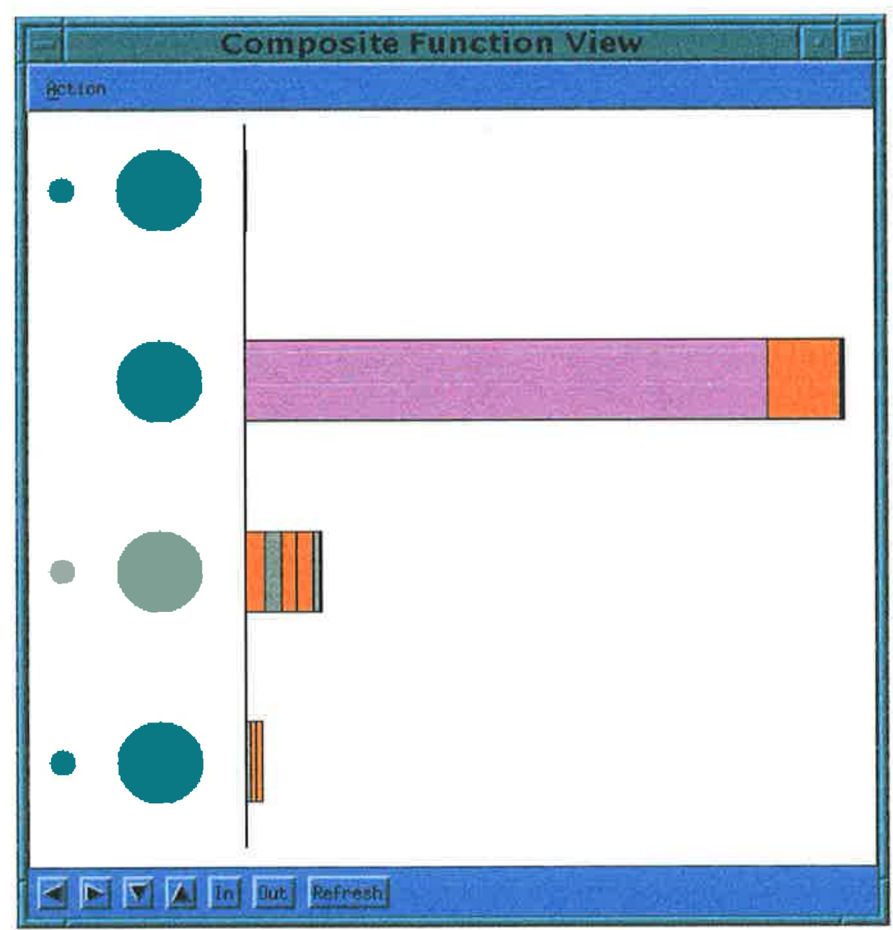

Figure 4.54. *The* **Composite Function View***.*

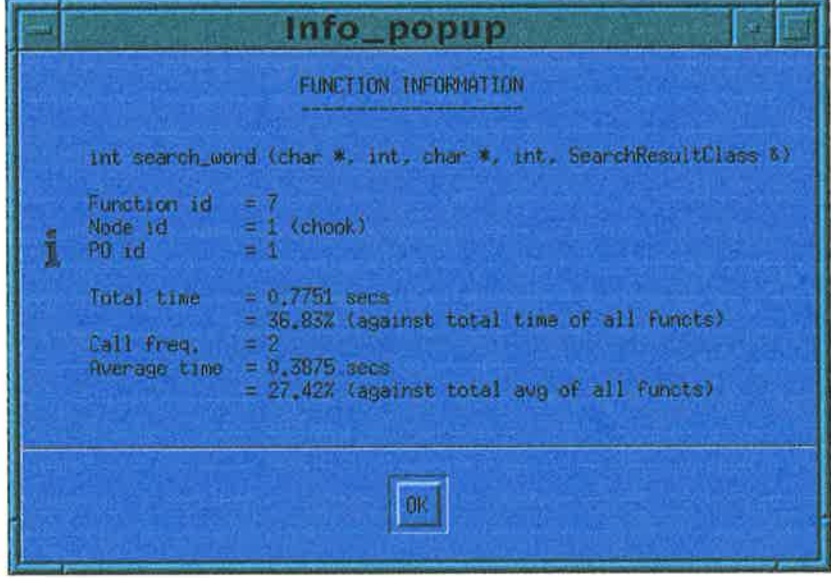

Figure 4.55. *The auxiliary view from the* **Composite Function View***.*

4.4.4 Observations

The design and architecture of the views in Visor++ are clean in that it is modular and allows relatively easy expansion and addition of new views. Such expansions could be horizontal (i.e. adding new views), or vertical (adding new level of view abstractions). This is possible because the views are designed by observing sound principles in user-interface design (Section 4.4.3.1).

The views, however, have some limitations. Most of these limitations are not inherent in the design of the views. Rather, they are caused by the lack of information which can be gathered by the instrumentation methods which are employed by Visor++. In the light of the goals of Visor++, these limitations are rather minor. They are described below.

4.5 Summary

Visor++ is a post-mortem visualisation tool for visualising CC++ program execution. The CC++ language itself does not provide visualisation support as in pC++ and μC++. However, by using the Visor++ framework, a sensible visualisation scheme is possible. In fact, by using the methods described in this chapter, a wealth of important information can be visualised. The framework is embodied in the tool Visor++.

Visor++ has several desirable properties. Firstly, to make the framework and implementation of Visor++ portable, it is designed to work at the source-code level. It provides automatic instrumentation of users' programs, which can then be used to produce visualisation. Visor++ views are implemented by using POLKA, which makes them more portable, and relatively easy to implement.

Secondly, Visor++ uses a wide selection of language features as the basis for visualisation. In particular, it uses threads, functions, objects, processor objects, and RPCs to drive the visualisation. Such a wide selection provides the possibility of constructing multiple views representing aspects of a program from multiple angles.

Thirdly, Visor++ provides both the static and dynamic aspects of CC++ programs. The types of views and their representation make it easy for users to determine

what, when, where, and why CC++ events occur. Furthermore, Visor++ also presents auxiliary views, which are also vaguely used in several tools such as MVD [24, 127] and Interaction Network [3]. However, the use and construction of auxiliary views are formalised in Visor++ for the first time.

Finally, the views in Visor++ are constructed by using some strong principles in the graphical user interface. In particular, it employs the notions of colour consistency and presentation consistency. These principles make the views relatively easy to understand. Furthermore, unlike the majority of other tools for visualising concurrent object-oriented programs, the views in Visor++ are integrated. They are presented in a hierarchical manner with their visuals animated coherently.

As with many other tools, the implementation of Visor++ has some limitations. These limitations are discussed in terms of the instrumentation subsystem, the event-collection subsystem, and the visualisation subsystem.

In the instrumentation subsystem, certain constructs cannot be handled well by the tool. In some cases, the instrumentation method *changes* program semantics to some extent. This pertains to the introduction of new scopes into the program (Section 4.2.3.7), and the instrumentation for RPCs (Section 4.2.3.5). However, as shown in the associated sections, these can generally be overcome with relative ease.

The subsystem also does not provide the ability to gather lower-level information, such as information on data structures, global pointers, and *sync* variables, in terms of their contents and changes during program execution. Such information can be important, especially for debugging and for the visualisation of data-parallel programs. The instrumentation approach as described in this thesis does not seem to be adequate for such a task. This, to some extent, confirms the notion that source-code instrumentation cannot accommodate the broad needs of visualisation [111]. On the other hand, such low-level information may not be needed, especially for large programs which involve a large number of processors, processor objects, and threads.

The current implementation of the system also does not cater to the provision of control for the insertion and execution of instrumentation probes. Such control is important for users to control the amount of perturbation in program execution [111].

Incorporation of this feature can, perhaps, be experimented with in future enhancements of the system.

The instrumentation methods described in this thesis also assume that the organisation and structure of the compiler-generated code are similar to that in the source code [108]. This can be achieved, to some extent, by using traditional compilers which do not violate this principle[22]. For the above reasons, compiler-supported instrumentation and visualisation, as in pC++ and TAU [13, 97] seem to be one possible solution.

Finally, it is to be noted that the instrumentation approach as described in this chapter applies primarily to C++ based languages. However, the fundamental ideas are relatively flexible to be applied to other language systems, as described in the next section (Section 4.6).

In terms of the event collection subsystem, the execution of the instrumented version of a program causes probe effects, which include timing effects and synchronisation-error effects (Section 2.1.3.1). Generally, the probe effects are inevitable [82]. For the timing effects, at best the visualiser (the person designing the tool) could strive to minimise such effects [24]. One method to minimise these effects is by providing control of insertion and execution of instrumentation probes, as described above. The synchronisation-error effects, however, depend highly on the user's programs themselves [60]. If the programs contain synchronisation error, then timing effects only serve to exacerbate synchronisation-error effects. In this context, Visor++ cannot be used effectively to visualise programs with synchronisation errors. However, as to the timing effects, significant effort has been put into Visor++ to minimise them. In particular, the probes have been designed in such a way as to minimise the amount of information contained in the generated traces. Furthermore, unlike some other visualisation systems [24, 34], the handling of the traces is the responsibility of Visor++, i.e. it is *separated* from the user's program. Finally, the generation and dispatching of these traces from the program to Visor++ are carried out in an asynchronous manner. The exact measurements of the timing effects are established and discussed in Chapter 5.

[22]Turning off the optimisation options in a compiler during the compilation of both original and instrumented programs may be of assistance.

In the event visualisation subsystem, program traces are visualised by using a set of views composed of high-level and low-level views, with a clean architecture which allows for smooth integration and expansion of new views. Furthermore, the views are constructed by adhering to some strong principles, as described in Section 4.4.3.1. However, the types of views, and the information provided by the views are bounded by the information collected by the tool. These boundaries are determined by the power of the instrumentation method used. For example, the views on data structures are not available. As another example, the nature of the instrumentation methods used and the nature of the CC++ language itself limit the ability of the system to gather information on thread context switching and thread blocking. Until such issues are resolved, no associated views can be produced.

Finally, the presentation of the views can still be improved. For example, the code structure diagram (CSD) [36], pretty-printing [5], the Nassi-Schneiderman diagram [20], and better graph drawing algorithms [62, 88] can be used to enhance the presentation of the static views. Further improvement can also be carried out on the dynamic views. In particular, the addition of a high-level algorithmic view would be useful. Another area of improvement is the provision of customised visualisation constructs [77] which enable users to construct and/or customise views. Given the complexity of concurrent object-oriented systems, such provision warrants further investigation.

Although Visor++ has some limitations as described above, it can indeed be used in a variety of circumstances for understanding and fine-tuning programs. In some cases, it can even be used as a debugging tool.

Visor++ itself has been successfully implemented by using C++, CC++, and POLKA. The system sizes, in terms of the instrumentation subsystem, the event-collection subsystem, and the visualisation subsystem, are approximately 6,500 lines, 4,000 lines, and 10,000 lines respectively, excluding program comments.

4.6 Applicability to Other Systems

Given the open architecture and implementation of Visor++, the tool framework, particularly the instrumentation subsystem, can also be applied to other languages which are similar to CC++.

Concurrent languages based on C++ can generally be divided into two main groups: those using *library extensions*, and those using *language extensions* [38]. In languages with library extensions, concurrency is implemented by a set of libraries which are opaque to users. With language extensions, concurrency is implemented by the introduction of new language constructs.

The implementation of the Visor++ framework on language libraries is rather straightforward. The reason is that the libraries are implemented with standard C++. As such, the libraries can be instrumented to implement the Visor++ concepts. PRESTO [11] and PARC++ [129] are examples of such language libraries.

For languages with extensions, the implementation of Visor++ can be carried out, provided that several preconditions are met. Firstly, the language constructs can be instrumented such that program traces reflecting full program states can be obtained. Secondly, the concurrency constructs in the language express *guaranteed* concurrency. This means that the application of the constructs to a program block should guarantee that the block is executed concurrently. Languages with such constructs include CC++ and Concurrent C++ [64]. If these two preconditions are satisfied, then the Visor++ framework can be adapted for the associated language. If either one of the preconditions is violated, then the application of the Visor++ framework may need either new approaches for source-level instrumentation, or full compiler support. In particular, compiler support would be needed if program traces cannot be conveniently produced by the source-code instrumentation. The support may also be needed if the language has constructs which provide *potential* concurrency, i.e. application of the construct to a program block may or may not result in concurrency. Examples of languages which do not fulfill the preconditions are ICC++ [35], and Mentat [68].

The framework of the Visor++ *visualisation subsystem*, however, can be applied to

any of the above types of languages, provided that program traces conforming to the Visor++ instrumentation requirements can be obtained. Once the traces are obtained, program analysis can be commenced. The next chapter describes several experiments and their analysis by using Visor++.

Chapter 5

Using Visor++

By using Visor++, program analysis and tuning can be done. The typical session in using Visor++ is as follows. First, the instrumentation subsystem of Visor++ is invoked by the user to automatically instrument a program. Next, the instrumented program is executed, and the resulting traces are visualised. Note that during visualisation, all references to the program static structures are made to the original program, *not* the instrumented one (Section 4.4.3.2 and Section 4.4.3.4). Using the views, the user may find anomalies or loopholes for optimisations. Equipped with this knowledge, the user may modify the program, which, again, is instrumented and visualised by using Visor++. The whole cycle is repeated until the program exhibits the desired behaviour. The final product is the modified, uninstrumented program with the desired properties.

This chapter describes some experiments in using Visor++. A discussion of the merits and limitations of Visor++ usage is also given.

5.1 Experiments

Four experiments are conducted for visualising CC++ programs. The description of these experiments is given in the order of their complexity, from the most simple experiment to the most complex.

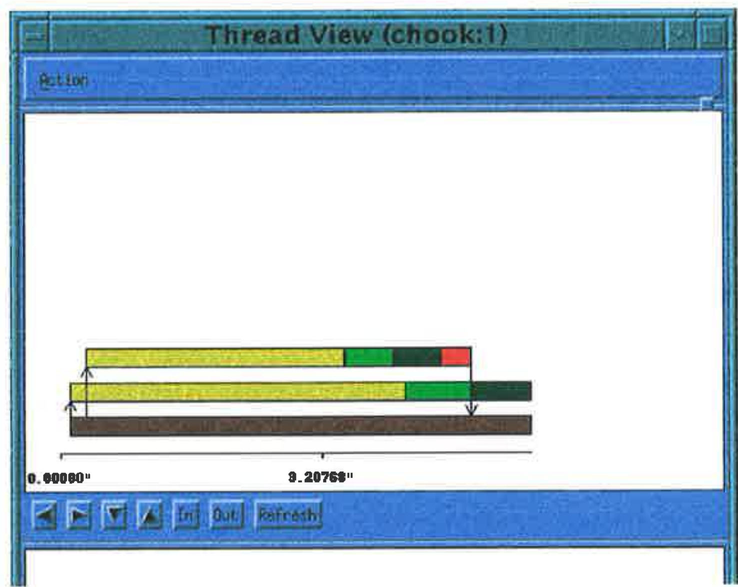

Figure 5.1. *The* **Thread View** *of the simple master-slave program.*

5.1.1 A Simple Example

The first experiment is the visualisation of a small and simple distributed master-slave program, adapted from the CC++ tutorial [32]. The program involves one master processor object (PO) spawning one or more slave POs, each of which carries out some data processing, sends a message to be printed back to the master PO, and terminates. In this example, the program is executed with two slave POs. The master PO and the slave POs are placed on different physical processors.

Figure 5.1 illustrates the Visor++ Thread View of the program's master PO. The view indicates that there are three threads running in the PO. The first thread, positioned at the lowest position, is the main thread. This thread subsequently creates two synchronous threads by using the `parfor` construct (execution of this construct is indicated in the figure by the colour brown and two up-arrows). The colour yellow in the child threads indicates that each of the threads is engaged in a RPC call-setup overhead. The Thread View and the Cumulative RPC Statistics View reveal that this overhead takes approximately 3 to 4 seconds, which occupies a considerable portion of the overall program execution of 6 seconds. Upon following the link-back to the source code, it is apparent that the overhead is mainly for the creation of the slave POs. This can be explained by the fact that the creation of a PO means using a RPC to create a

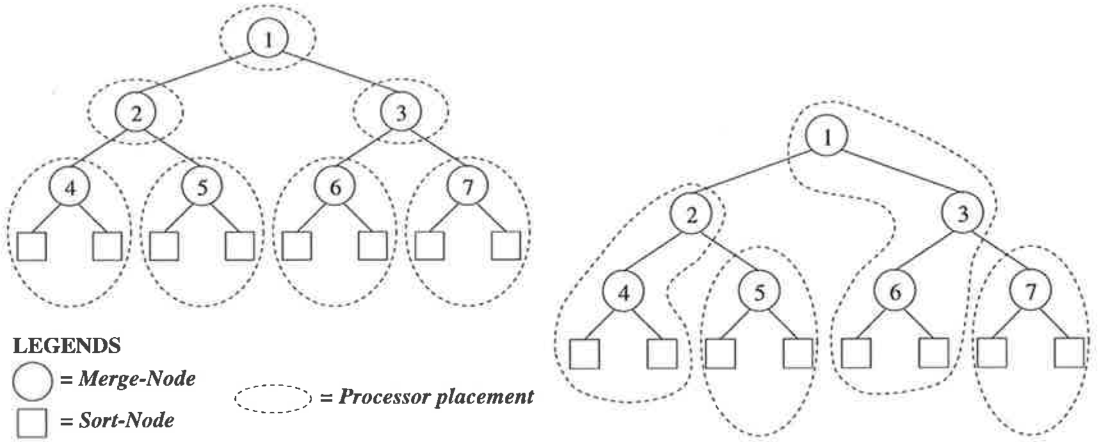

Figure 5.2. *Merge-sort using a 4-level binary tree. The left figure shows the initial placement of the nodes on processors, and the right figure the optimised placement.*

separate execution address space, possibly on a different physical processor. The same views also reveal that RPCs, in general, are very expensive in terms of execution time. The full source code of the program, as well as its instrumented version, can be found in Appendix A.

Although this example is simple, it illustrates how the usage of Visor++ can reveal important facts about a program, hence assisting programmers to better understand their code. In the following examples, it is demonstrated that using Visor++ views can help programmers *both* to understand and to fine-tune their programs.

5.1.2 Distributed Merge-Sort

The second experiment is the optimisation of a distributed merge-sort program, also adapted from [32]. The program uses a master-slave configuration in the form of a binary tree of processor-objects (POs). The internal nodes are the *merge-PO*s which implement the merge operation, while the leaves are the *sort-POs* implementing the sorting operation. Each merge-PO divides the data it receives into two equal halves, spawns two other merge-POs (or two sort-POs at the leaves of the tree), merges the results, and passes them back to its parent PO.

An analysis of a merge-sort program using a 4-level binary tree is conducted with Visor++. Initially, each merge-PO is placed on a different physical processor (the left portion of Figure 5.2), using 7 different machines. The rationale is that as each merge-

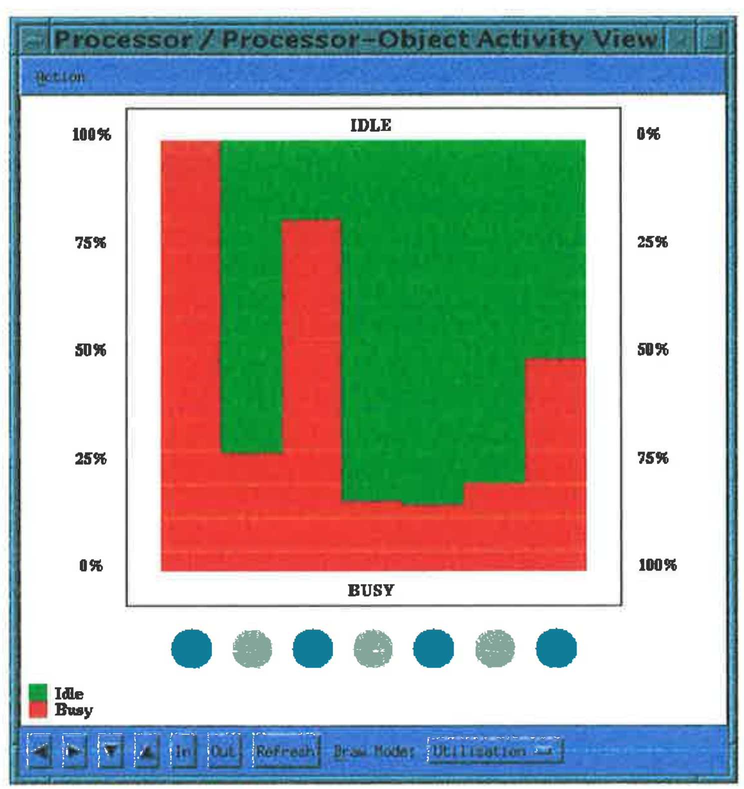

Figure 5.3. *The* **Processor/Processor-Object Activity View** *reveals the inefficiency of processor usage.*

PO is computationally expensive, it needs to be placed on a separate processor. Each sort-PO, however, is placed on the same processor as its parent. The rationale is that since a UNIX time-sharing system is being used, as many processor cycles as possible should be used for sorting, while at the same time reducing the amount of time spent in RPCs. Using Visor++, the program is automatically instrumented, and the execution traces of the instrumented program are visualised.

During visualisation, the Processor/Processor-Object Activity View reveals that the POs do not have much idle time. However, the overall utilisation of the physical processors is rather poor (Figure 5.3). As in the first example, by using the Cumulative RPC Statistics View, the Thread View, and the link-back to the corresponding source-code, it is found that synchronous RPCs dominate the computation. One of the views that reveal this is the Cumulative RPC Statistics View (Figure 5.4). By using the Thread View, it is also revealed that while a merge-PO is performing two RPCs to its slave merge-POs (or sort-POs), it is effectively blocked, hence being idle. This

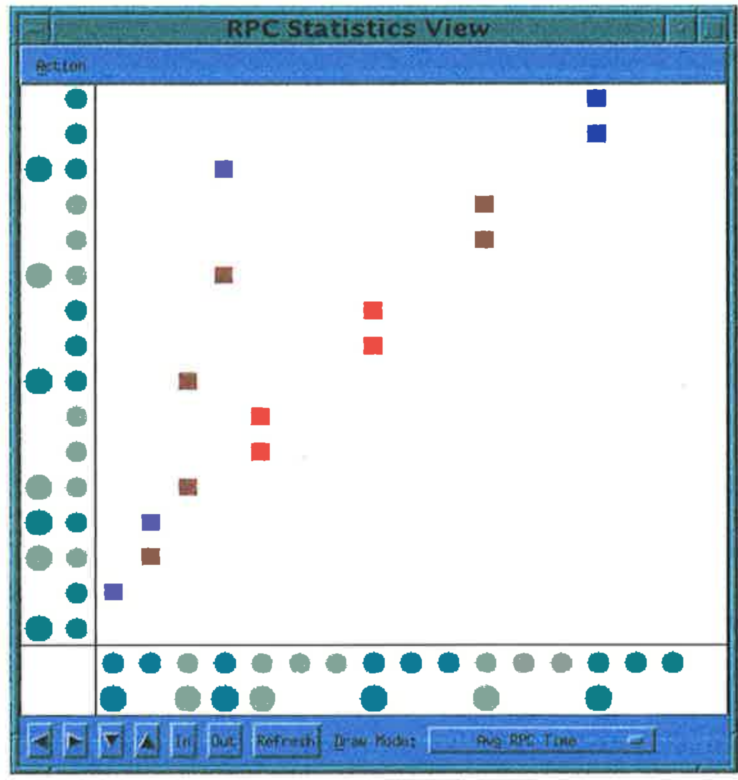

Figure 5.4. *The merge-sort program is heavy with RPC activity.*

	Un-instrumented	Instrumented	% change
Original placement	18.057 secs	26.515 secs	46.8 %
Final placement	9.723 secs	11.981 secs	23.2 %
Improvement	185.70 %	221.31 %	–

Table 5.1. Timing information from the merge-sort program.

observation leads to the idea of placing those POs into a reduced number of physical nodes to increase efficiency. The final placement is shown on the right side of Figure 5.2, and the resulting processor utilisation is shown in Figure 5.5.

Table 5.1[23] provides the timing information of the program for sorting 20,000 integers. The program with the initial PO placement takes approximately 18 seconds to execute. With the alternative placement, execution time is reduced to 9.7 seconds while using less processors. The column "instrumented" gives the measurements of

[23]To level out the spikes in machine and network loads, the original program, the un-instrumented and the instrumented versions, are executed 10 times in an interleaved fashion, and the averages are taken. This measurement is carried out on both the original program and the improved program in a similar fashion. Other tables in this chapter are produced in a similar fashion.

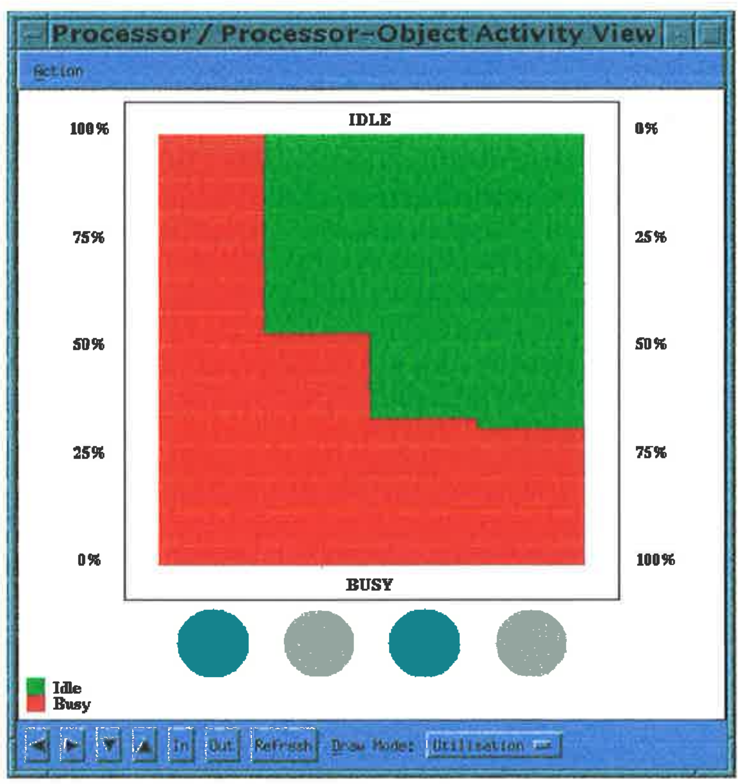

Figure 5.5. *Higher efficiency in processor usage after program tuning.*

the instrumented versions of the same programs. The column "% change" refers to the difference in execution time between the un-instrumented and the instrumented versions of the program. It is computed as $(i - u)/u$, where i is the execution time for the instrumented version of a program, and u for the un-instrumented one. Finally, the row "improvement" gives the speedup of the program. It is calculated as o/f, where o is the execution time of the original version of the program, and f for the final version. As a comparison, the sequential version of the merge-sort program, employing the same sorting algorithm, takes approximately 49 seconds.

Further optimisations are, of course, possible. For example, the Function Usage View can be used to highlight which functions are most frequently invoked or are the longest to execute, and optimise accordingly. However, it is the intention of the discussion here to show that optimisation is possible by cleverly arranging computation structures, even without changing a single line of code. This insight is achieved by using Visor++.

5.1.3 Concurrent String Search

The third example is the visualisation of a parallel text-searching program, which is also briefly described in [136]. A particular string or text S is to be searched for among a given set of N text files. The final output is a list of L files in which the string is found, where $0 \leq L \leq N$. In the implementation, P processor-objects (POs) are used, in which $N \ mod \ P$ of them are assigned $\lceil N/P \rceil$ files, and each of the remainder is assigned $\lfloor N/P \rfloor$ files. Such a system can be adapted to many situations, one of which is for an Internet search engine.

5.1.3.1 Implementation-1

The solution is implemented as a branch-and-bound algorithm, in which a master-PO allocates and places slave-POs on separate physical processors. The master-PO then initiates RPCs to obtain results from the slave POs. To do this, the master-PO creates P threads, each of which allocates, places, and performs RPCs to a slave-PO. Henceforth referred to as **Implementation-1**, the program is instrumented and visualised by Visor++.

Using Visor++ views, an optimisation loophole is pin-pointed. The Thread View in Figure 5.6 shows that the master-PO is idle while the slave-POs are executing. The view shows that the master-PO creates two synchronous threads using the `parfor` construct (colour brown). The two threads are indicated as the two bars above the `parfor` bar. The colour yellow indicates that the thread is experiencing a RPC call-setup overhead, while dark green indicates that the RPC is being executed. The Thread View, with the source-code link-back (Figure 5.7), indicates that the master-PO has been idle (blocked) for more than 10 seconds. Consequently, the implementation of the master-PO can be modified so that instead of being blocked, it also participates in the search. This modification is embodied in **Implementation-2**.

5.1.3.2 Implementation-2

In **Implementation-2**, another thread is created in the master PO to carry out similar computation as the slave POs. This new thread is referred to as the *master PO's*

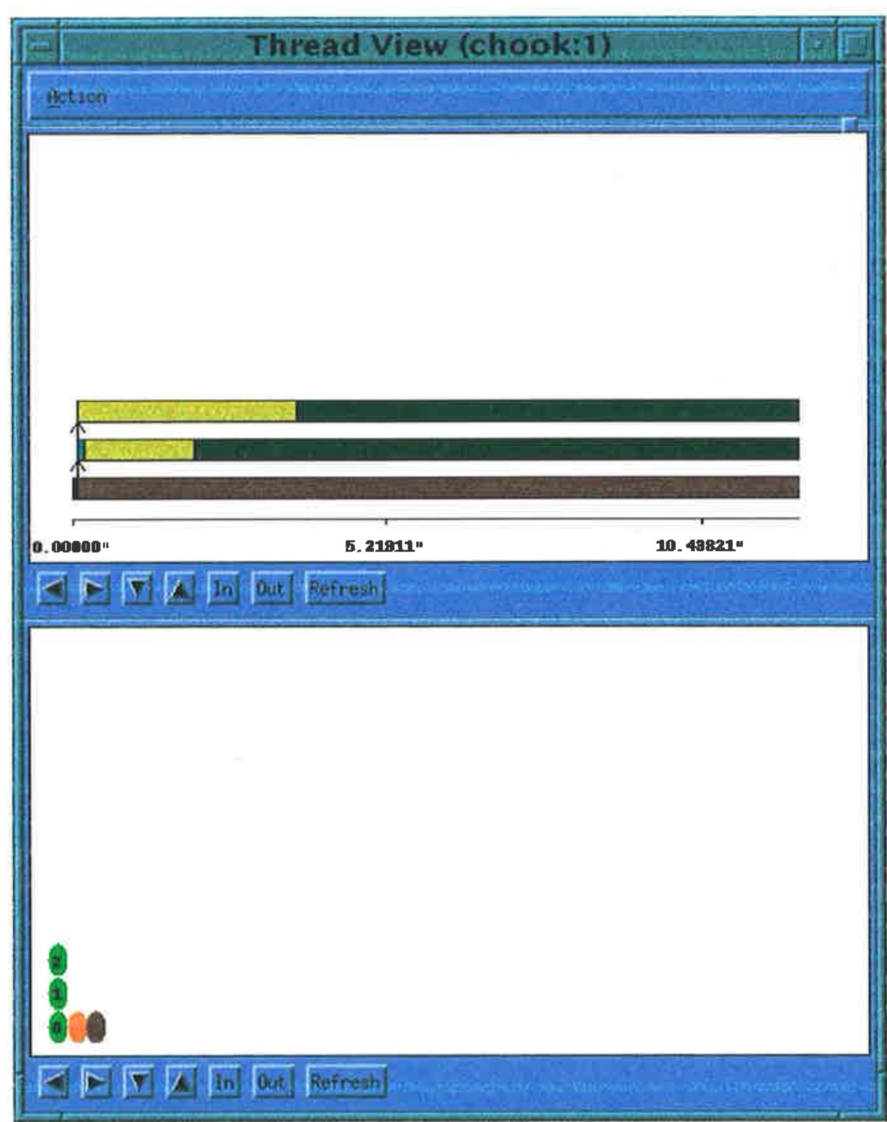

Figure 5.6. *The* **Thread View** *shows that the master-PO is idle while the slave-POs are executing.*

Source code view

Action

File: /a/kiri/tai/users/cs/hendra/C++/instr1/CASE1/ORIG/master2.c++, line 73

```
60      read_file_info (masterfname.str, 0, MAX_FILES, fis);
61      base = fis.num_of_items / num_of_hosts;
62      if (base == 0)  {
63         cout << "File nums < processor nums" << endl;
64         exit (1);
65      }
66
67      remainder = fis.num_of_items - base * num_of_hosts;
68      globalres.num_of_items = 0;
69
70      for (i=0 ; i<num_of_hosts ; i++)
71         G_sl[i] = NULL;
72
73      parfor (int i=0 ; i<num_of_hosts ; i++)  {
74         SearchResultClass res;
75         int idx_start, idx_end;
76
77         if (i < remainder)  {
78            idx_start = (base+1) * i;
79            idx_end   = (base+1) * (i+1) - 1;
80         }
81
82         else {
83            idx_start = remainder * (base+1) + (i-remainder) * base;
84            idx_end   = idx_start + base - 1;
```

Figure 5.7. *The* **Source-Code View** *shows the source-code area where the master-PO is blocked.*

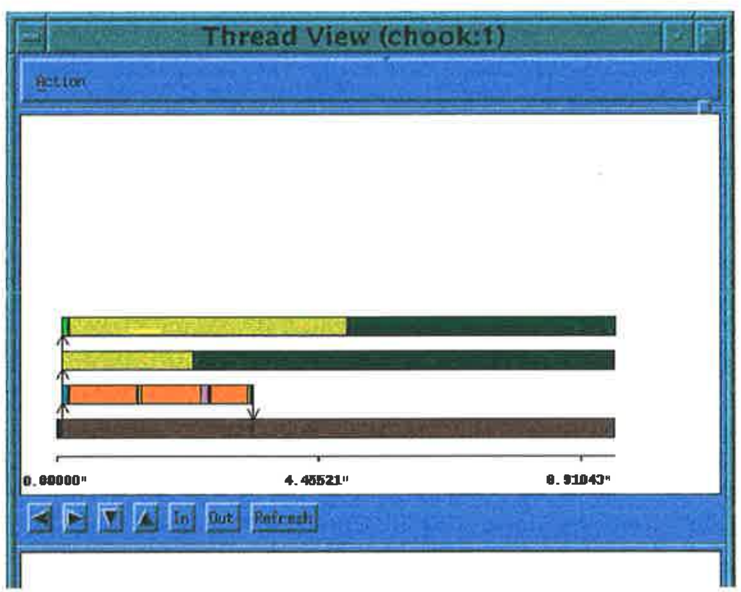

Figure 5.8. *The* **Thread View** *of* **Implementation-2**, *with the compute-thread.*

compute-thread. Figure 5.8 shows the Thread View generated by Visor++. In the view, the compute-thread is represented as the second bar from the bottom. It is apparent that the compute-thread finishes execution much sooner than the two threads executing the RPCs. There are two possibilities. First, the compute thread is given too few files to work with, hence it finishes its execution quickly. Second, it is also possible that the compute-thread *does* have enough files to work with. However, it stops quickly because it has found exactly L number of files in which the string S occurs. In other words, the string S occurs in *at least* L number of files among the $\lceil N/(P+1) \rceil$ or $\lfloor N/(P+1) \rfloor$ files[24] with which the compute thread is assigned to work. Hence, it is useless to continue the search. Accordingly, it is possible to improve program performance by incorporating two changes. Firstly, the compute-thread is modified so that it is given more files to work with. Secondly, the program can be modified such that as soon as L files are found, the remaining POs (possibly including the compute thread) are instructed to stop executing. Hence an *interrupt-signaling system* is developed. These changes are incorporated in **Implementation-3**.

[24]In the original implementation (**Implementation-1**) P processor objects are used, in which each PO is assigned $\lceil N/P \rceil$ or $\lfloor N/P \rfloor$ files to work with. With the addition of the master PO's compute thread, this number becomes $\lceil N/(P+1) \rceil$ and $\lfloor N/(P+1) \rfloor$ respectively.

	Un-instrumented	Instrumented	% change
Implementation-1	21.403 secs	29.025 secs	35.6 %
Implementation-3	13.043 secs	16.640 secs	27.6 %
Improvement	164.10 %	174.43 %	–

Table 5.2. Timing information from the parallel text-searching programs.

5.1.3.3 Implementation-3

In **Implementation-3**, it is found that using the same set of inputs and the same operating environments, the program does exhibit better performance. The Thread View in Figure 5.9 shows that the interrupt-signaling mechanism is working. The right-hand green portion in the second bar from the bottom shows that the compute-thread is *not* calling any more functions. In fact, by using the Source-Code View, it can be deduced that the thread is ready to exit from the `parfor` block. The third bar from the bottom, with its down arrow, indicates that the RPC to one of the slave POs has finished execution. Although it is not very indicative[25], it is very likely that the slave PO has received a signal from the compute-thread. Note that the dark green portion of the thread represents a RPC being active. As indicated by the view, the RPC finishes execution *after* the compute-thread does not call any further functions (the colour green). Therefore, it is likely that the compute-thread signals the other two threads to finish, after which the compute-thread itself terminates. This is "supported" by the fourth bar from the bottom, which represents the second RPC. It is apparent that the RPC (the remote-execution phase, excluding the RPC call-setup and call-return overheads) is short-lived, considerably shorter than the first RPC. Hence, it may also have received the signal from the compute-thread.

The results of this experiment is presented in Table 5.2, where $L = 5$, $N = 150$, $P = 2$, with the files having various sizes, ranging from 6 Kbytes to 80 Kbytes, and the word to search for is "nanocomputer". Note that the table only shows the timing information for **Implementation-1** and **Implementation-3**.

[25]It is not indicative because the actual interleaving of thread execution is not known. With the current implementation, such information cannot be easily captured by Visor++ (Section 5.2.2).

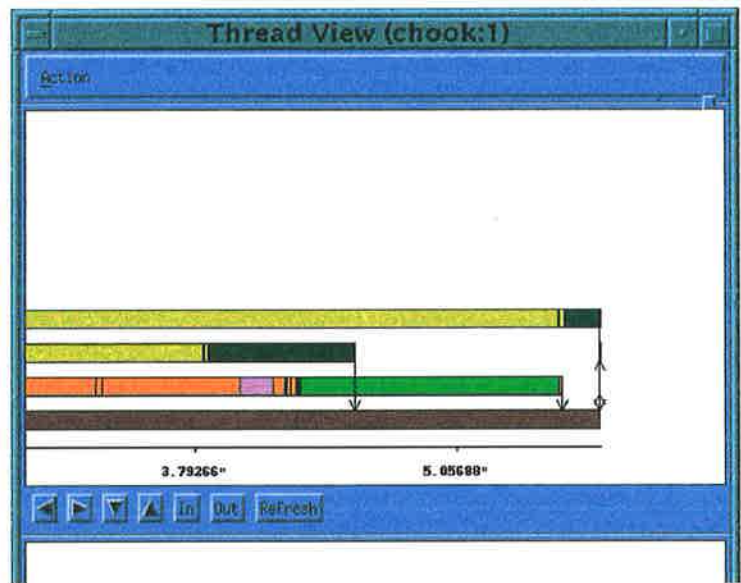

Figure 5.9. *The interrupt-signaling mechanism in work.*

5.1.4 An Electronic Transaction System

The final example is the development and refinement of an electronic transaction system. This section shows how Visor++ can be used not only in understanding and tuning of ready-made programs. It can also be used during the *program development stage*. In this perspective, Visor++ is used as a refining tool, to achieve the desired results.

The problem is to build an electronic transaction subsystem which can record the transactions that occur among a set of parties. Building all the components of such a large system *at once* is not a recommended approach. On the contrary, it is best to build such systems as a set of subsystems. It means that the system should be decomposed into meaningful subsystems, possibly orthogonal and independent, which are later composed to build the final product [16]. Therefore, the components of the electronic transaction system must be identified, and should be designed in a highly decoupled manner.

An electronic transaction system could consist of many subsystems, for example: the user-interface subsystem, the database subsystem, the network subsystem, and the transaction subsystem itself. The discussion in this section pertains to the development of one of the subsystems, namely the transaction subsystem (referred to as the *trans-*

action system). Therefore, the scope of the problem is the development of a prototype of a transaction system.

The prototype to be developed is the transaction system for a commodity market, whereby many parties may commit transactions. In particular, a transaction occurs whenever a party P_1 who wants to sell m units of a commodity C, can be matched with another party P_2 who wants to buy n units of the same commodity. Note that it is not necessary that $n = m$. If, for example, the party P_1 cannot find any other party to buy the commodity, then P_1's *intention to sell* is kept by the system until there is another party P_2 who declares an *intention to buy* the same commodity. The two parties are then matched by the system, and the transaction is recorded in a transaction database. If $m > n$, then, after the transaction, P_1's intention to sell $(m - n)$ units of commodity C is retained by the system. On the other hand, if $m < n$, then P_2's intention to buy $(n - m)$ units of C is retained by the system. Otherwise, both intentions can be safely discarded. The term *intention* is used to denote either an intention to sell or an intention to buy.

In this system, no party can have any bias or preference. In other words, no party can choose its preference as to whom it wants to sell to or to buy a commodity from. The system must also be designed in such a way that it guarantees fairness. It means that it must employ a first-come-first-served (FCFS) approach. For example, a party P_1 declares first an intention to sell a commodity C, and then followed by another party P_2 who declares the same intention. If there is another party P_3 who declares an intention to buy that commodity, then the system will first match P_1 with P_3. If, after the transaction is completed, P_3's intention to buy is still retained by the system, only then will P_2 be matched with P_3.

The system is developed in CC++, as a three-tier client-server system, consisting of the clients, the servers, and the business logic unit (BLU) in between [67]. The clients are the entities which are used to place buyers' intentions to buy and sellers' intentions to sell. The servers are those entities which try to match the buyers and the sellers. The BLU is the entity which translates the intentions from clients into their corresponding *transaction requests* (Figure 5.10). These transaction requests are then

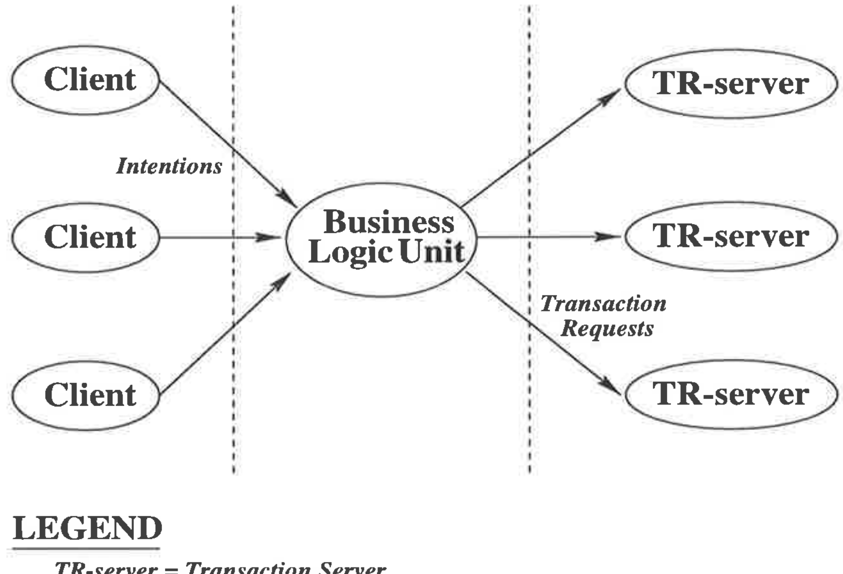

Figure 5.10. *Three-tier architecture for the electronic transaction system.*

redirected to suitable transaction servers for processing. The results of this processing are then passed from the servers back to the BLU, to be redirected to the clients. In other words, the BLU regulates the flow and transformation of information between the clients and the servers. The general architecture of the whole system is illustrated in Figure 5.10. Without loss of generality, the prototype developed here consists only of *one* client, one BLU, and multiple transaction servers.

5.1.4.1 Implementation-1

In the initial implementation (henceforth referred to as **Implementation-1**), the client accepts intentions to sell and to buy from any party or customer. Each intention is then passed to the business logic unit which translates it into a suitable internal data format as a transaction request. This translation is carried out by using the customer database and the commodity (stock) database. In other words, both databases are placed in the BLU. Later, the transaction request is passed on to one of the transaction servers for processing. It is these servers that match potential buyers with potential sellers. The decision as to which transaction server handles a request is based on the stock specified in the request. In other words, the transaction servers are stock-partitioned. When a transaction server succeeds in matching a buyer and a seller, a transaction is then

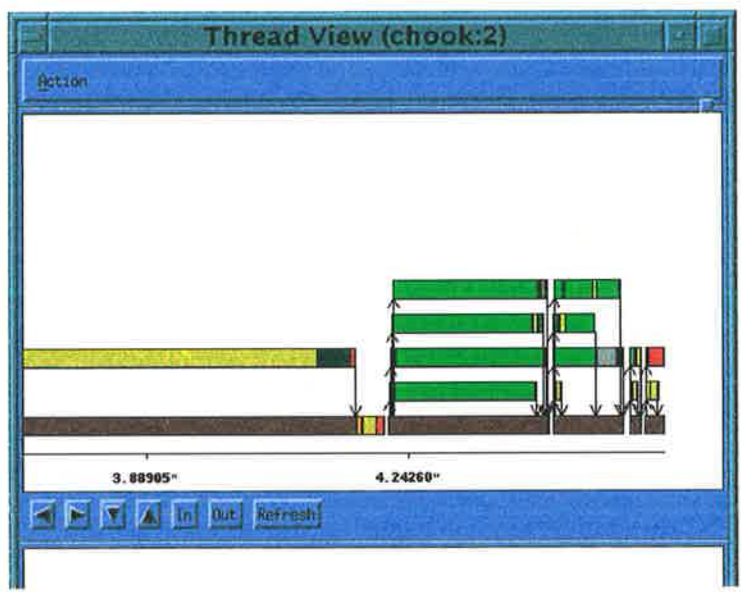

Figure 5.11. *The* **Thread View** *shows the unfairness of transactions.*

committed and recorded in a transaction database. The resulting transaction is also coded in a *status value* which is passed back to the BLU, and subsequently to the client. To increase program efficiency, the BLU is implemented such that it does not send off a transaction request to servers as soon as the associated intention arrives. Instead, the requests are batched until a certain number are accumulated. At this point, the processing of these requests on the transaction servers is invoked by the BLU via the RPC mechanism.

The client, the BLU, and the transaction servers are all implemented as processor objects. They are referred to as the client-PO, the business logic PO (BL-PO), and the transaction server PO (TS-PO). The code is implemented such that the placement of the client-PO, the BL-PO and the TS-POs is parameterised. The number of transaction servers is also parameterised. In this example, the program is executed with one client, and two transaction servers. For analysis, the code of this implementation is instrumented and visualised using Visor++.

During visualisation, it transpires that the system does *not* exhibit the required transaction fairness criteria. The Thread View of the BL-PO indicates this (Figure 5.11). The figure shows that the main thread in BL-PO buffers four transaction requests (to sell or to buy), processes them, then dispatches them concurrently to the

transaction servers. This action is repeated until the requests are exhausted. In the figure, the main thread is the lower-most bar, while the concurrent dispatching of requests is handled by the four threads created by the main thread. This is indicated by subsequent groups of four bars above the bar representing the main thread. By using the auxiliary views, it is revealed that the threads represented by the left-most group of four bars execute RPCs to the same TS-PO located on the machine "achilles" (Figure 5.12). Similar information can also be obtained from the Navigation View. If the fairness criteria is satisfied, then for any two threads A and B in the thread group, if thread A is created earlier than B, then thread A should finish its execution earlier than thread B. However, the view shows that thread 3 finishes before thread 2 (thread 2 is represented by the second lower-most thread bar in the group). Furthermore, by using the source-code link-back facility, it is revealed that the the concurrent dispatching of transaction requests by the BLU does not guarantee that the order of request dispatching is preserved. This is not a problem if the transaction ordering can be resolved by the transaction servers. However, the Source-Code View of the implementation of the transaction servers shows that this is not the case (Figure 5.13). This analysis is confirmed by inspecting the transaction databases in the transaction servers. The ordering of the transactions does not preserve the ordering of the intentions to sell and to buy from the client.

The above analysis shows that **Implementation-1** is not a sufficient and correct implementation. The implementation needs modification in such a way that the ordering of the resolution of transaction requests is preserved to satisfy the fairness criteria. The modification is embodied in **Implementation-2**.

5.1.4.2 Implementation-2

In **Implementation-2**, each transaction request is time-stamped by the BLU. As in **Implementation-1**, the requests are then dispatched by the BLU to the transaction servers via the RPC mechanism. The following scenario, however, is possible.

The BLU could invoke the first RPC to one transaction server to dispatch transaction request A, and then the second RPC to the same server for request B. Each of

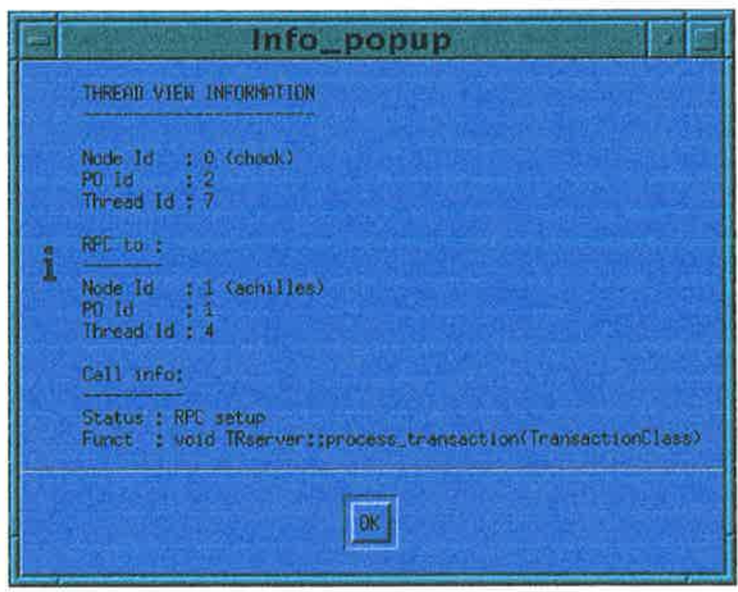

Figure 5.12. *Auxiliary view showing a transaction request being handled by a TS-PO on the machine "achilles".*

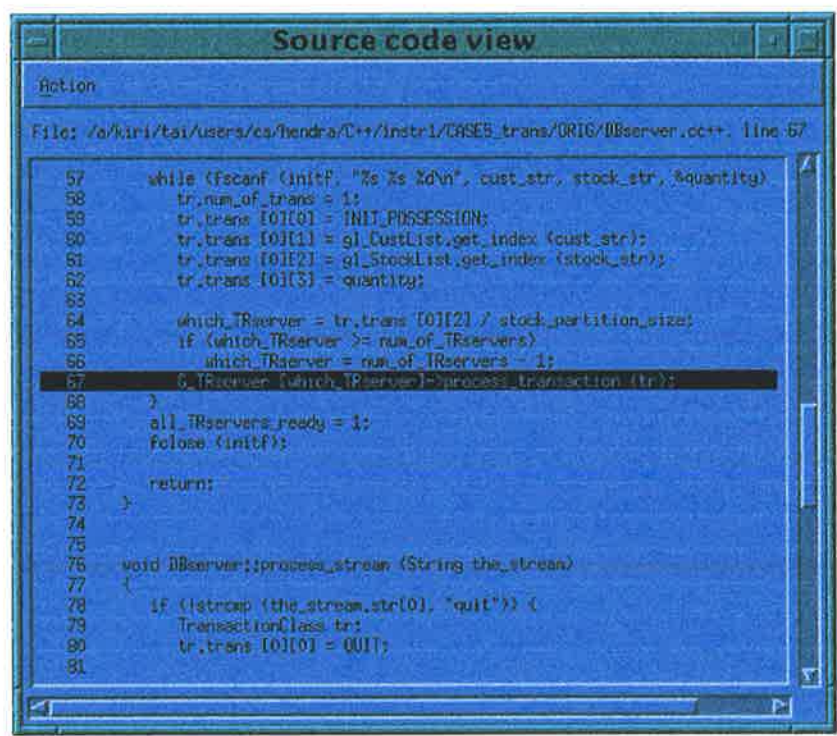

Figure 5.13. *The* **Source-Code View** *reveals the inadequacy of the implementation of the transaction server.*

the RPCs would then create a thread in the server. It is possible, however, that the thread for request B would be started before the thread for request A. To guarantee serialisation, each transaction server maintains a time-stamp subsystem. Each thread in the server then checks and compares the timestamp of the request it is to handle with the time-stamp subsystem. Only when permission is granted by the subsystem will the thread complete the request. Therefore, it uses an *arbitration mechanism*. The buffering/batching of requests and their concurrent dispatching, as in **Implementation-1**, are still maintained.

By using Visor++ to instrument and visualise the execution of **Implementation-2**, it is revealed that the fairness criteria is now satisfied. However, the views also reveal that due to the method used by the transaction servers to handle transaction requests, many processor cycles are wasted in dealing with the time-stamp subsystem. This is illustrated in Figure 5.14.

The Function Usage View in Figure 5.14 shows that the function `TimeStampClass::check_timestamp()` is most frequently invoked. Other views, i.e. the Thread View and the Composite Function View, reveal that the number of threads invoked in the transaction servers is less than the frequency of invocations of `check_timestamp`. Consequently, the operation of the time-stamp subsystem can be improved. The Class Hierarchy View (Figure 5.15) and the Class Information View in (Figure 5.16) show that the class `TimeStampClass`, of which the function is a member, does not inherit from any other class. The Source-Code View also shows that in the implementation, the subsystem does not depend on other objects (classes). Therefore, modifying the time-stamp subsystem does not entail modifying other parts of the code.

The Function Usage View also shows that `DBserver::init_TRservers()` takes a relatively long time to execute. Aided by the Composite Function View (Figure 5.17), this observation is confirmed. By using the auxiliary view, it is found that the function `init_TRservers()` is called in the BL-PO placed on the machine "chook" (Figure 5.18). The auxiliary view is displayed by selecting the left-most (longest) function bar in the second row from the top of the Composite Function View. However, upon source-code inspection, it is apparent that no significant improvement can be made to

Function Usage View

Action

FUNCTION NAME	FREQ	MEAN-TIME (sec)	MEAN-TIME (graph)
String::String(const String &)	0	0.00000	
String::String()	4	0.00021	
TransactionClass::TransactionClass()	15	0.02206	
TransactionClass::TransactionClass(..	0	0.00000	
TransactionClass::TransactionClass()	13	0.00012	
TimeStampClass::TimeStampClass()	2	0.00006	
TimeStampClass::TimeStampClass()	0	0.00000	
TimeStampClass::check_timestamp(int)	669	0.00026	
TimeStampClass::return_timestamp()	0	0.00000	
TimeStampClass::renew_timestamp(int)	4	0.00017	
DBserver::DBserver()	1	0.00018	
DBserver::DBserver()	0	0.00000	
DBserver::init_TRservers(String)	1	3.74064	
DBserver::process_stream(String)	1	0.52330	
TRserver::TRserver(int,int,int)	2	0.07540	
TRserver::TRserver()	0	0.00000	
TRserver::process_transaction(int,T..	6	0.14790	
Stock::Stock(char *,int the_id =-)	2	0.00012	
Stock::Stock()	0	0.00000	
Stock::Stock(const Stock &)	0	0.00000	
Stock::same_name(char *)	0	0.00000	
Stock::same_id(int)	0	0.00000	
Stock::get_id()	0	0.00000	
StockList::get_index(char *)	6	0.00008	

In Out Refresh

Figure 5.14. *The* Function Usage View *reveals those functions which are heavily used or take much time to execute.*

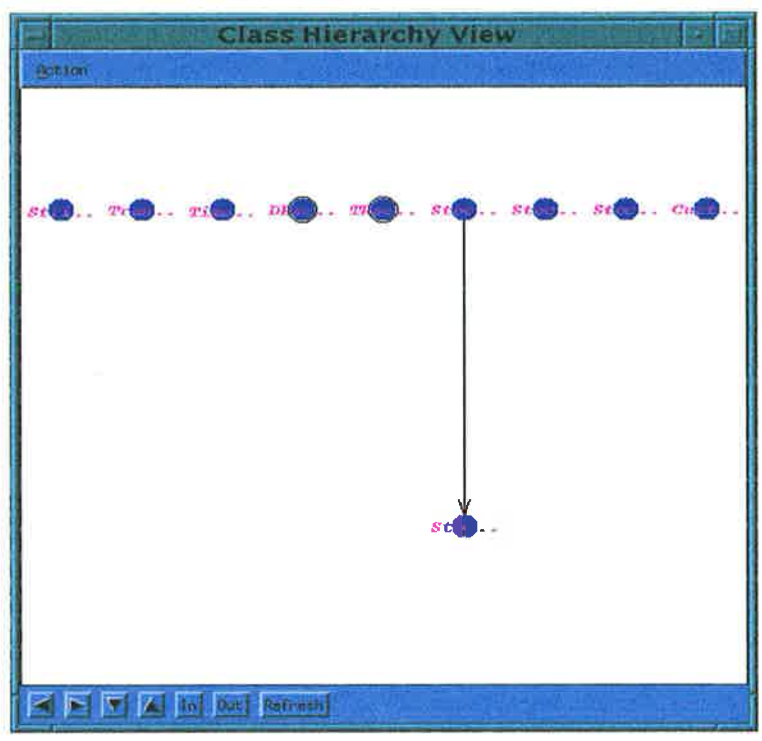

Figure 5.15. *Together with the* **Source-Code View**, *the* **Class Hierarchy View** *shows that the time-stamp subsystem does not depend on the implementation of other classes.*

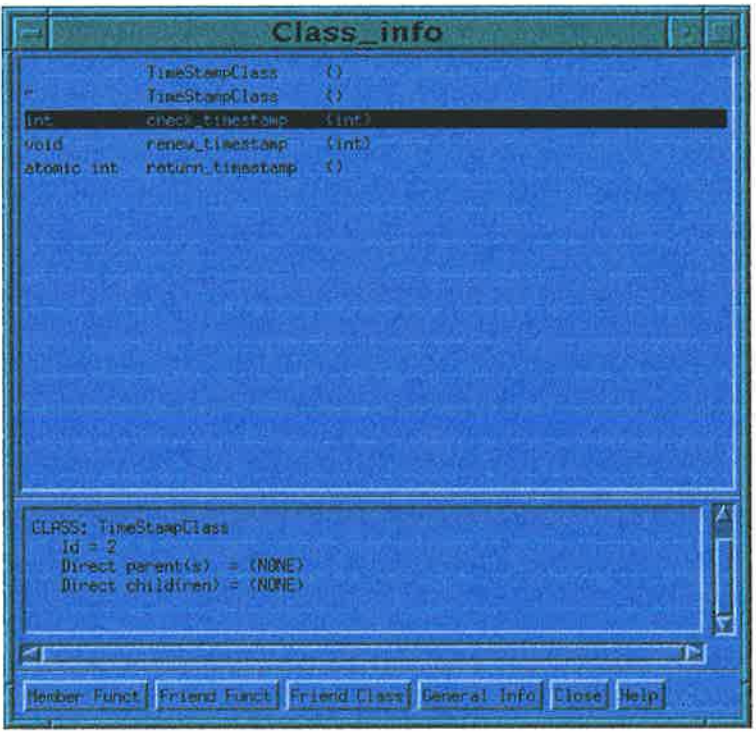

Figure 5.16. *The* **Class Information View** *of the class "TimeStampClass", displayed upon clicking the associated node on the* **Class Hierarchy View**.

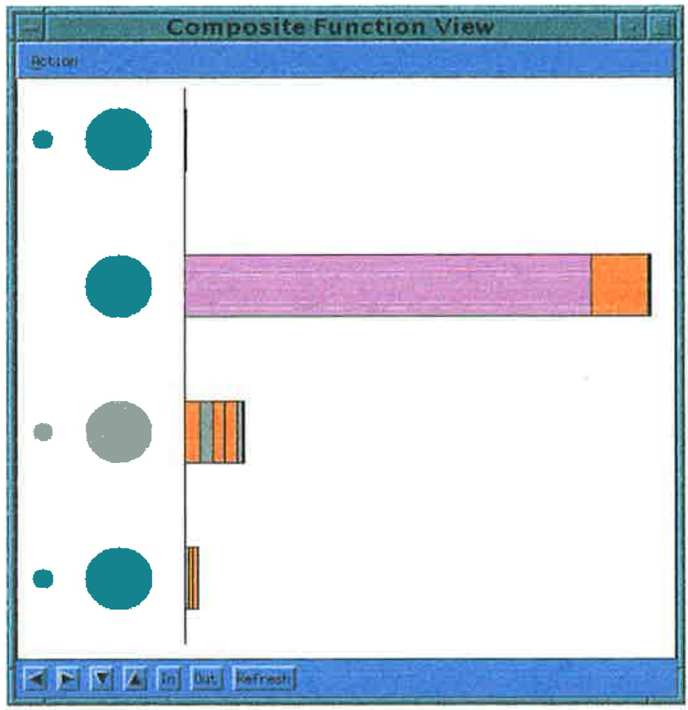

Figure 5.17. *The* **Composite Function View** *shows which functions can be optimised for each processor-object.*

the function. All the above views are taken when Visor++ is visualising the program after 5.30 seconds of the overall execution of 5.96 seconds.

As a final note on **Implementation-2**, the program analysed above operates with approximately less than 50 transactions. Upon closer observation, for a bigger transaction count, e.g. 300 or 400 transactions, the arbitration mechanism as previously highlighted does not guarantee an upper bound on program execution time. During experimentation, using a transaction count of approximately 300, it is observed that sometimes the program completes in less than 10 seconds, while on other occasions, it might complete in more than 50 seconds. This fact makes it is safe to deduce that **Implementation-2** is not a correct solution.

Based on the above analysis, the implementation of the transaction servers is altered, which is embodied in **Implementation-3**.

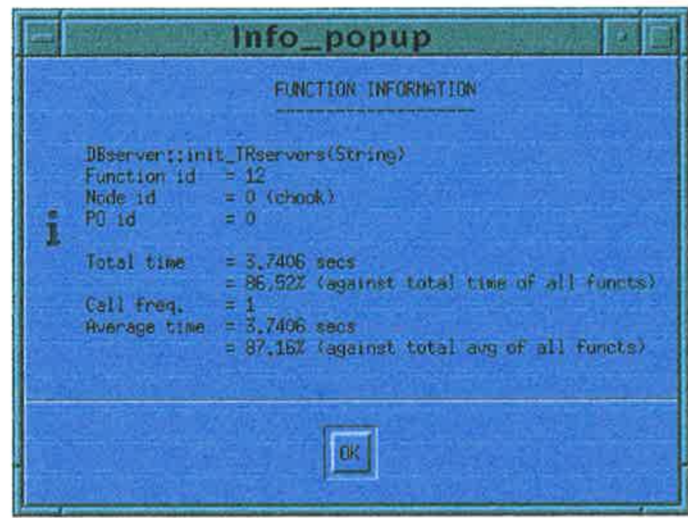

Figure 5.18. *This auxiliary view is the result of selecting the longest function bar in the second row from the top of the* **Composite Function View**.

5.1.4.3 Implementation-3

In **Implementation-3**, the threads in the transaction servers do not actively check and compare the timestamp of the transaction request it has to handle, with the time-stamp subsystem. Instead, whenever such a thread is created, it will register its transaction request along with the time-stamp of the request to a central *transaction resolver*[26]. The transaction resolver contains a *seller queue* and a *buyer queue*. Transaction requests received by the threads in the transaction server will be channeled to the resolver into the appropriate queue. In other words, transaction requests representing the intentions to sell will be channeled into the seller queue, and the intentions to buy into the buyer queue. These requests are queued based on their timestamps. In other words, the seller queue and the buyer queue are both priority queues. It is also the transaction resolver which will match buyers and sellers to complete transactions by using the priority queues.

During visualisation, it is apparent that some improvements have been made. The Function Usage View in Figure 5.19 shows that all the functions are relatively low in their call frequencies and average execution time. The view shows that although the function `init_TRservers()` have not been changed, yet its average execution time

[26]Each transaction server has one transaction resolver.

Function Usage View

Action

FUNCTION NAME	FREQ	MEAN-TIME (sec)	MEAN-TIME (graph)
String::String(const String &)	0	0.00000	
String::String()	5	0.00052	
TransactionClass::TransactionClass()	22	0.00076	
TransactionClass::TransactionClass(..	0	0.00000	
TransactionClass::TransactionClass()	21	0.00052	
DBserver::DBserver()	1	0.00017	
DBserver::DBserver()	0	0.00000	
DBserver::init_TRservers(String)	1	2.53182	
DBserver::process_stream(String)	2	0.02737	
TRserver::TRserver(int,int,int)	2	0.02192	
TRserver::TRserver()	0	0.00000	
TRserver::process_transaction(int,T..	7	0.00093	
TimeStampedTrans_PriorityQ::TimeSta..	2	0.00010	
TimeStampedTrans_PriorityQ::TimeSta..	0	0.00000	
TimeStampedTrans_PriorityQ::enqueue..	6	0.00005	
TimeStampedTrans_PriorityQ::dequeue..	12	0.00005	
TimeStampedTrans_PriorityQ::force_d..	0	0.00000	
Stock::Stock(char *,int the_id =-)	2	0.00011	
Stock::Stock()	0	0.00000	
Stock::Stock(const Stock &)	0	0.00000	
Stock::same_name(char *)	0	0.00000	
Stock::same_id(int)	0	0.00000	
Stock::get_id()	0	0.00000	
StockList::get_index(char *)	9	0.00019	

In Out Refresh

Figure 5.19. *The* **Function Usage View** *now reveals that the functions are within reasonable frequency and average time of execution.*

has slightly dropped compared with its execution time in **Implementation-2** (Figure 5.14). This can be explained by the fact that the improved time-stamp mechanism in **Implementation-3** has contributed to lowering the function execution time (i.e. the function also uses the time-stamp subsystem for system initialisation). *Ceteris paribus*, except for the changes in the transaction servers, the execution of **Implementation-3** is now reduced to 3.32 seconds. All the figures are snapshots of Visor++ views after 3.21 seconds of execution[27].

All the above implementations, **Implementation-1**, **Implementation-2**, and **Implementation-3** are implemented such that the customer database and the stock database are handled by the BLU. Using the databases, it is the job of the BLU to translate the intentions issued from the client into their associated transaction requests. As the number of transactions grow, the BLU may well become a bottleneck in terms of execution speed. The question is: *what would the effect be if the databases are replicated in the transaction servers instead? What would the performance increase be?* These questions are explored by accommodating the changes in **Implementation-4** and analysing it using Visor++.

5.1.4.4 Implementation-4

Implementation-4 is exactly the same as **Implementation-3**, except that the stock database and the customer database are replicated and placed in the transaction servers. This is done to reduce the bottleneck experienced by the BLU. This means that the intentions to sell and to buy from the client are distributed, without processing by the BLU, to the transaction servers.

Comparing the views of **Implementation-4** with those of **Implementation-3**, it is found that for slight differences in the frequency and average function call time, there

[27] In the analysis, the comparison of the average execution time of "init_TR_server()" in **Implementation-2** and **Implementation-3** is carried out at different visualisation snap-shot times. In other words, the average execution time of the function at 3.21 seconds of a total of 3.32 seconds for the execution of **Implementation-3** is compared with the same entity at 5.30 seconds of a total of 5.96 seconds execution time for **Implementation-2** (see Section 5.1.4.2). Superficial observation suggests that such comparison is invalid. However, the function is used only once in both implementations. At such a snap-shot in time, the invocation of the function has been completed in both implementations.

	Un-instrumented	**Instrumented**	**% change**
Implementation-3	4.456 secs	5.693 secs	22.6 %
Implementation-4	4.228 secs	5.185 secs	27.8 %
Improvement	100.54 %	109.79 %	–

Table 5.3. Timing information from Implementation-3 and Implementation-4.

seems to be no other significant difference (Figure 5.20). However, the total execution time is increased to approximately 3.83 seconds, which is more than the 3.32 seconds for **Implementation-3** (Section 5.1.4.3). Logically, **Implementation-4** should perform *slightly faster* than **Implementation-3**. This "anomaly" can be explained by the fact both programs are executed on a network of UNIX time-sharing system, in which both the network load and the machine load can differ unpredictably from time to time. The above anomaly can, perhaps, be attributed to this condition. To justify this hypothesis, a measurement is done on both **Implementation-3** and **Implementation-4**, by executing both programs 10 times, in an interleaved fashion. Both programs are executed under the same condition with the same inputs of approximately 300 transactions. The results are shown in Table 5.3.

Table 5.3 indicates that **Implementation-4** is indeed slightly better than **Implementation-3**. However, under certain conditions, **Implementation-3** may perform better than **Implementation-4**. For example, if the cost of maintaining a centralised database (as in **Implementation-3**) is relatively less expensive than maintaining a distributed one, then clearly **Implementation-3** will perform better, and vice versa. Another factor that should be taken into account is the interaction of the subsystem with its operating environments. For example, on a network of machines with a heavy operating load, the performance of the programs will be affected. In the case of fluxtuating load, measurements may suggest that one implementation is better than the other, whereas under different conditions the reverse may be the case.

Appendix B illustrates the code used in **Implementation-3** and **Implementation-4**. However, since both implementations are relatively large (approximately 1500 lines), only the common skeleton of both implementations is given.

Function Usage View

Action

FUNCTION NAME	FREQ	MEAN-TIME (sec)	MEAN-TIME (graph)
String::String(const String &)	0	0.00000	
String::String()	30	0.00070	
TransactionClass::TransactionClass()	0	0.00000	
TransactionClass::TransactionClass(..	0	0.00000	
TransactionClass::TransactionClass()	0	0.00000	
DBserver::DBserver()	1	0.00016	
DBserver::DBserver()	1	0.00013	
DBserver::init_TRservers(String)	1	3.38963	
DBserver::process_stream(String)	3	0.02019	
TRserver::TRserver(int,int,int)	2	0.02966	
TRserver::TRserver()	2	0.00008	
TRserver::process_transaction(int,S..	9	0.00389	
TimeStampedTrans_PriorityQ::TimeSta..	2	0.00006	
TimeStampedTrans_PriorityQ::TimeSta..	2	0.00005	
TimeStampedTrans_PriorityQ::enqueue..	9	0.00010	
TimeStampedTrans_PriorityQ::dequeue..	18	0.00010	
TimeStampedTrans_PriorityQ::force_d..	2	0.00005	
Stock::Stock(char *,int the_id =-)	2	0.00011	
Stock::Stock()	4	0.00009	
Stock::Stock(const Stock &)	0	0.00000	
Stock::same_name(char *)	0	0.00000	
Stock::same_id(int)	0	0.00000	
Stock::get_id()	0	0.00000	
StockList::get_index(char *)	9	0.00009	

In Out Refresh

Figure 5.20. *The* **Function Usage View** *reveals approximately similar function profiles to those in Implementation-3.*

5.2 Discussion

Visor++ has some notable merits. These include the ability of the tool to support program understanding, fine-tuning, and even refinement during program development. Both the merits and limitations[28] are discussed in the following two sections.

5.2.1 Merits

The results in the previous sections indicate that when properly used and interpreted, the views in Visor++ can help users to understand and fine-tune their programs. By using the views, once program understanding is achieved, fine-tuning can be performed. Some tuning does not require source code modification, but only a re-arrangement of task structures. This is demonstrated in Section 5.1.2. Others may require slight program modification, as exemplified in Section 5.1.3. Still, some others may require a major overhaul of program implementation, as demonstrated in Section 5.1.4. This means that Visor++ can be used in a variety of meaningful ways. Furthermore, the visualisation is provided on an automatic basis, without the need for the users to manually intervene.

As demonstrated in Section 5.1.4, Visor++ can also be used for refining design during program development. The full-scale version of the electronic transaction system could have a large number of subsystems. With the current implementation of Visor++, the visualisation of all the subsystems at once would pose difficulties to the user. The reason is that the user may be confused by the many entities in the views vying for attention. One possible solution is to design a higher level algorithmic view which can present the program in a more compact manner. Another solution is to divide the user's large system into several orthogonal subsystems, possibly independent of each other. Program development can then be carried out on these smaller subsystems. At this level, Visor++ can sensibly be used.

By using Visor++, many program facets can be uncovered with relative ease. The

[28]The limitations discussed in this section pertain to the *usage* of Visor++. In contrast, the limitations discussed in Section 4.5 pertain to the *implementation* of the tool.

multi-faceted perspective of the user's program is represented by highly inter-related views. These views incorporate a wide cross-section of program features, taking into account the important elements of task-parallel object-oriented programs. These include, among others, functions, threads, concurrent objects, and their interactions. These elements are represented in both high-level and low-level views which incorporate the static, dynamic, and auxiliary views. The views preserve some principles such as interface consistency, and the principles of least astonishment (Section 4.4.3.1). As a result, the views can closely link the user's mental model with the program itself. All this confirms the merits of Visor++. However, during experimentation, some limitations are also uncovered. These limitations are discussed below.

5.2.2 Limitations on Visor++ Usage

Through experimentation, it is clear that Visor++ is effective for small and medium-sized programs. This is approximately equal to 5,000 lines of code and less than 200,000 events. These numbers represent the *usage feasibility* of the tool. In other words, they represent the upper bound with which Visor++ can be sensibly used. They do not, however, represent the *technical feasibility* of the tool. The "limitation" on the usage feasibility is partly due to the tool design itself, and partly to the natural complexity of task-parallel object-oriented programs. With respect to the tool design, visualising a large program would instantiate views with many graphical entities vying for the user's attention. This can cause confusion and difficulty on the user's behalf to understand the program, or to pin-point problems. This problem can be alleviated by *vertical expansion* of the tool to include higher-level, preferably algorithmic, views. Such views can transcend the existing views, supplying the user with an overview of program activities at a higher level. However, due to the natural complexity of task-parallel object-oriented programs, such views may be difficult to construct. At the time of writing, there seems to be no tool which provides such high-level algorithmic views for large scale programs. This provides a fertile ground for further study. As a "short-cut" to work around this problem, the strategy of dividing a large program into smaller orthogonal subsystems can be used (Section 5.2.1).

To some extent, Visor++ preserves the principles of immediacy [131]. These principles state that the separation between cause and effect in terms of time, space, and program semantics has to be minimised. Visor++ supports *temporal immediacy* in that user control against the views are instantly effected. This is due to the fact that the post-mortem visualisation approach used by Visor++ can give greater control to the user. Visor++ also provides support for *semantic immediacy.* Semantic immediacy itself means that "the conceptual distance between semantically related pieces of information is kept to a minimum" [131]. In Visor++, this translates into colour and presentation consistencies (Section 4.4.3.1). Furthermore, relating similar information on the views is also relatively easy, particularly through close linkage among the views, and through the usage of auxiliary views. For example, obtaining further information, or opening a new view takes only one or two mouse-clicks away. However, Visor++ has a drawback in terms of *spatial immediacy.* Spatial immediacy states that "the physical distance between causally related events is kept to a minimum" [131]. In Visor++, this drawback is caused by the wide separation among the views. Further research into this area is, therefore, warranted.

Measurements in the experiments with Visor++ also verifies the fact that program instrumentation produces probe effects. This can be seen from the tables in Section 5.1. The exact effects of such probes depend on the program being instrumented. However, some measures have been taken in Visor++ to minimise them (Chapter 4). Observations have indicated that the additional amount of time needed by an instrumented program as opposed to its un-instrumented version varies consistently between 10% to 47%. This is still acceptable, taking into account that the resulting gain in execution time can be potentially much more than such perturbations. Even if this is not the case, the use of Visor++ can still facilitate better program design through refinement. In other words, the refined program will, generally, perform better than its predecessor. The final product is the associated un-instrumented program with better design or performance.

The data presented in the tables are, in fact, measurements of the degree of timing effects, *not* synchronisation-error effects, induced by Visor++ instrumentation.

However, in some cases, Visor++ can also be used to uncover synchronisation errors (Section 5.1.4.1)[29]. As shown by Gait [60], synchronisation-error effects may not be as predictable as the timing effects. Hence, this chapter has not attempted to measure these effects.

The experiments also uncover the fact that there are at least two types of information which, if present, can make the tool more effective. The first is information relating to the interleaving of thread execution. Source-code transformation, as used in Visor++, cannot capture such information during program execution. Having this information would enable the tool to more accurately present information, for instance, on how much time a particular thread is really idle, and how much blocking time is incurred on a particular RPC.

The second type of information is that of data structures and their contents. This is important for those programs which, on the greater part, manipulate different portions of a piece of data concurrently by using the same method. In other words, the information is important for data-parallel programs.

At the current stage of implementation, the types of information, as previously described, cannot be captured solely through source-code transformation. Compiler support, as in [13, 97], may be needed. A similar notion for the instrumentation of data-parallel programs has also been noted by Reed [111].

All the above limitations could become the agenda for further fruitful study.

[29]**Implementation-1** of the electronic transaction system contains a synchronisation error.

Chapter 6

Conclusions and Future Work

6.1 Summary

Applying software visualisation to task-parallel object-oriented programs poses interesting questions. The reason is that, typically, such programs exhibit complex behaviour as a result of the complex interaction among the program entities. Such interaction is, among others, caused by concurrency and distribution.

With the exception of several tools, such as AIMS and TAU, many existing tools only focus on a narrow selection of language features for visualisation. Observations indicate that the breadth of such selection varies in inverse proportion to the number of language features, and to the complexity of the language or programming environment of the visualised programs. For example, tools for inherently "simpler" paradigms, such as the sequential paradigm, use a wider selection of features for the associated paradigm. On the other hand, the majority of the tools for message-passing parallel programs, for instance, only focus on message exchanges.

The approaches used by the tools, as indicated by the above "trend", are viable. However, the reverse of the trend does not necessarily imply non-viability. Rather, a *wide selection* of language features can be used. Furthermore, based on the selected features, the visualisation can employ multiple coherent views depicting a visualised program from multiple angles. In other words, the wide selection of the features are visualised holistically. In such cases, the organisation of information to be displayed,

and how to display it, are of utmost importance. For the visualisation of *data-parallel* object-oriented programs, this proposition is exemplified by such tools as TAU. In such tools, a wide selection of language features, including concurrency and distribution, are visualised.

Perhaps, due to the complexity of *task-parallel* object-oriented programs, relatively few, if any, tools have been designed for the visualisation of such programs. Applying the wide-selection approach to visualising task-parallel object-oriented programs, therefore, poses an interesting challenge. This challenge is even greater, considering that the majority of programming languages do not provide support for producing the necessary visualisation. In other words, the majority of the languages are "visualisation-unconscious", i.e. visualisation is an after-thought. This thesis presents a framework with which visualisation of task-parallel object-oriented programs written in visualisation-unconscious languages can be realised, particularly for those languages derived from C++. The framework is embodied in the tool Visor++.

Visor++ is tailored for the visualisation of CC++ programs. It covers a wide selection of CC++ features, including sequentiality (such as functions and objects), concurrency (such as threads and atomic functions), and distribution (processor objects and RPCs). These features form the base on which the program visualisation is produced.

To make the framework and implementation more portable, Visor++ presupposes three conditions. Firstly, the target language allows an instrumentation method which can produce program traces that can depict program events and the relationships among program elements. Secondly, the visualisation (including the necessary instrumentation) can be produced by working on the source-code level. In other words, no low-level systems environment (such as operating systems instrumentation) are necessary. Finally, the target language employs the notion of guaranteed concurrency (Section 4.6) in its constructs. The last condition is essentially a minor requirement, but is nevertheless a desirable property.

The views in Visor++ are presented by using some strong principles, namely: layout consistency and behavioural consistency (Section 4.4.3.1). To some degree, it also

exhibits the application of the immediacy principles [131], i.e. semantic and temporal immediacy (Section 5.2.2). The provision of these views also revolves around program source-code. In other words, Visor++ employs source code transformations to generate visualisation. This ensures greater portability of the concepts and implementation to other similar platforms. Additionally, Visor++ provides the static, dynamic, and auxiliary views of the programs. Such a configuration and application of the above-mentioned principles make Visor++ easy to use for helping users understand and fine-tune their programs. Experiments with the tool, as exemplified in Chapter 5, testify to this statement.

As with other similar tools, Visor++ also exhibit some (minor) limitations, particularly for the instrumentation and the visualisation subsystems. In particular, Visor++ produces some minor effects on the introduction of new scope regions and on the instrumentation methods for RPCs. Fortunately, both minor problems can easily be rectified (Section 4.2.3.5 and Section 4.2.3.7). Second, Visor++ is unable to instrument and visualise data structure information (Section 4.5). Such capability may well be useful for visualising data-parallel operations. Third, Visor++ also produces probe effects. However, measurements indicate that these effects only account for an additional timing factor of consistently less than 50% of the original program timing (Section 5.2.2). This, however, is relatively small, considering the gain in performance which can be obtained. Furthermore, these effects are only present in the instrumented versions of the programs. Fourth, the Visor++ views admittedly still have to be improved to provide better visual presentation. In particular, the views have some difficulties with spatial immediacy (Section 5.2.2). Wide separations among different windows on the screen could cause difficulty for users during visualisation. Fifth, Visor++ provides automatic visualisation without the need of user intervention. This is a desireable property. However, in some cases, customised visualisation may be more desireable. Visor++ does not provide customised visualisation in terms of instrumentation (Section 4.2.4) and visualisation (Section 4.5). In instrumentation, Visor++ does not provide the capability to control insertion and execution of instrumentation probes. In terms of visualisation, it does not provide customised view construction. Given the complexity of concurrent

object-oriented systems, incorporation of the above constructs warrants further investigation. Finally, Visor++ is effective for visualising small to medium-sized programs. This does not represent the inherent technical limitation of the tool, but, rather, its usage feasibility (Section 5.2.2). For visualising larger systems, the program may need to be decomposed into smaller orthogonal subsystems, then visualised and tuned separately (Section 5.2.1).

6.2 Conclusions

Visor++ embodies a unique approach in that it provides a framework for visualising task-parallel object-oriented programs written in languages based on C++. The approach is suitable for languages without (explicit) visualisation support.

For visualisation, Visor++ uses a wide selection of program features as the basis. Program events are then represented with a hierarchically structured views, which encompass static, dynamic, and auxiliary views. This approach proves to be useful for understanding, and subsequently fine-tuning users' programs.

In terms of the ideals as set out in Section 2.1.1, Visor++ is relatively portable and extendible. The reason is that the Visor++ framework assumes only the minimal source-code level capability of the language environment to produce visualisation. In terms of technical feasibility, Visor++ is scalable, but in terms of usage feasibility, it is only *partially* scalable, because it can effectively visualise only small to medium-sized programs. Visor++ also provides automatic visualisation without the need of user intervention. The experience with Visor++ shows that the visualisation, with minimal probe effects, is effective for conducting program analysis.

6.3 Future Work

The Visor++ framework can also be extended with additional capabilities. Such extensions are regarded as future work.

Ideally, the instrumentation subsystem is designed as part of the language system

design. In other words, the language is designed to be "visualisation-conscious" [97]. Another direction is to make the system more portable by using language-independent visualisation. One such system has been developed in which source code is tagged with language-independent markup code. The application of this method for the visualisation of source-code is described in [36]. The eligibility of such an approach for visualising task-parallel object-oriented programs is yet to be studied.

Although probe effects can hardly be eliminated, their reduction is desirable. Probes which are more light-weight must be designed. To make the visualisation more able to accurately reflect the real program execution, the probe effects can be "eliminated" by employing a probe-compensator. Using such an approach, timing delays due the execution of the probes are eliminated [61, 89, 115, 116, 142]. Other approaches, such as the logical clock approach [27, 28], the event-ordering analysis [50], and adaptive dynamic tracing [111] may also be adapted. Background processing load of the computers can also be taken into account for more effective probe-effect analysis and elimination [142]. This especially applies to time-sharing systems such as UNIX.

The spatial immediacy of Visor++ views needs to be addressed. One method is to provide intelligent support in the presentation and interpretation of views. Intelligent agents [125, 140] and expert systems [87] can be used for this purpose. Spatial immediacy can also be addressed through vertical expansion to include high-level algorithmic views. Given the complex nature of task-parallel object-oriented programs, automatic generation of such views is an interesting study.

Newly emerging paradigms, such as immersive environments, virtual reality, and multimedia can also be used for visualisation [107, 112]. In such environments, multimodal interfaces, such as sound, can enhance the presentation of the visualisation [19, 58, 118]. Another merging trend is the use of the Internet as the platform for visualisation. Some successful software visualisation has been constructed using the World Wide Web technology [21, 34, 42, 76]. It is worth noting that the tool POLKA, on which Visor++ is built, is being ported to the Java language [52, 120]. Of interest, the Nexus run-time system, on which CC++ is build, has also been implemented in Java [57]. In short, both POLKA and Nexus use Java as a common platform. It would

be interesting to explore this platform commonality to provide a better visualisation scheme than that currently implemented in Visor++.

As indicated in Section 6.1, customised visualisation, in terms of customised instrumentation and customised view construction, is another area worth investigating. Customised instrumentation support is important for users to determine the constructs to instrument and visualise. It also allows users to co control the amount of perturbation during program execution [111]. By the same token, customised view construction enables users to construct more meaningful views to cater to their own needs [77].

Finally, testing the framework of Visor++ by applying it to other systems and performing a comparison among the visualisation of such systems can also be done. It is interesting to note that the CC++ language (which is task-parallel) is currently being merged with pC++ (which is data-parallel) into a new language HPC++ [8]. It would, therefore, also be interesting to study how the framework of Visor++ can be applied to it.

Appendix A

An Instrumentation Example

This appendix contains the full source code of the master-slave program as described in Section 5.1.1. It consists of three source files:

1. `Slave.h`. This file contains the definition of the slave processor object.
2. `Slave.cc++`. This is the implementation of the slave processor object.
3. `Master.cc++`. This is the implementation of the master processor object.

Both the original and the instrumented versions are included. It is to be noted that the source files reproduced in this appendix are *exactly* those used in the experimentation. This example program is small. It is chosen as as an appendix precisely for this reason. This, however, does not hamper the description of the major features of the instrumentation subsystem.

For readability, the code in this appendix is reproduced with appropriate indentation and spacing.

A.1 The Original Code

This section contains the original code of the program.

1. **Slave.h**

```
#include <iostream.h>

global class Slave {
public:
   Slave() {}
   void say_hi (int id);
   ~Slave() {}
};
```

2. **Slave.cc++**

```
#include "Slave.h"

void Slave::say_hi (int id)
{
   cout << "Hello world from Processor Object #" << id << endl;
}
```

3. **Master.cc++**

```
#include<iostream.h>
#include "Slave.h"

int main (int argc, char **argv)
{
   int P=atoi(argv[1]);

   Slave *global G_slave[10];
   parfor (int p=0 ; p<P ; p++)  {
      proc_t placement = proc_t("Slave.out", argv[2+p]);
      G_slave[p] = new (placement) Slave;
      G_slave[p]->say_hi(p);
   }

   for (int p=0 ; p<P ; p++) {
      delete G_slave[p];
   }

   return 0;
}
```

A.2 The Instrumented Code

This section contains the instrumented version of the code in Section A.1. Several notes are relevant here.

- Firstly, the instrumented code is produced automatically by the instrumentation subsystem of Visor++. Such instrumented code contains a number of macros (for example: `ANN_GLOBAL_CLASS_PRIVATE_MEMBER`, and `ANN_FP_GLOBAL_CLASS_PARAM`). When the macros are expanded, the code is exactly as that described in Chapter 4.
- Secondly, some differences also exist between the methods as outlined in Section 4.2 and the real implementation. For example, in the implementation, each RPC is instrumented with a unique identifier which is used for the `EV_RPC_MARK_START` and `EV_RPC_MARK_FINISH` events. However, since such implementation differences are of little importance, they are not described in the body of this thesis.
- Thirdly, in addition to instrumenting user's code, the Visor++ instrumentation subsystem also adds code *as necessary*. For example, if the user's code does not contain a default constructor, one will be added and subsequently instrumented to the final code. The copy constructors and the destructor, are also added as needed.

The instrumented code is as follows.

1. **Slave.h**

```
#include "probe.h"   // File Id == 2
#include <iostream.h>

global class Slave {
private:
   ANN_GLOBAL_CLASS_PRIVATE_MEMBER;

public:
```

```
   virtual void Ann_delete (ANN_GLOBAL_CLASS_PARAM) {
      ANN_GLOBAL_CLASS_ANN_DELETE_HEADER;
      this->Slave::~Slave();
   }

public:
   Slave (const Slave &other) {
      ANN_GLOBAL_CLASS_COPY_CONSTRUCTOR_HEADER;
   }

public :
   Slave (ANN_GLOBAL_CLASS_PARAM_DEFAULT) {
      ANN_GLOBAL_CLASS_CONSTRUCTOR_HEADER;
      EntityProfiler Ann_FP (EV_GLOBAL_CONSTRUCTOR_START,
                             ANN_FP_GLOBAL_CLASS_PARAM,
                             0, Ann_RPC_Marker, 2, 5);

   }
   void say_hi (ANN_GLOBAL_CLASS_PARAM, int id);

   ~Slave () {
      ANN_GLOBAL_CLASS_MEMBER_FUNCTION_HEADER;
      EntityProfiler Ann_FP (EV_GLOBAL_DESTRUCTOR_START,
                             ANN_FP_GLOBAL_CLASS_PARAM, 2,
                             Ann_RPC_Marker, 2, 7);

   }
};
```

2. **Slave.cc++**

```
#include "probe.h"   // File Id == 1
#include "Slave.h"

void Slave::say_hi (ANN_GLOBAL_CLASS_PARAM, int id)
{
   ANN_GLOBAL_CLASS_MEMBER_FUNCTION_HEADER;
   EntityProfiler Ann_FP (EV_GLOBAL_MEMBER_START,
                          ANN_FP_GLOBAL_CLASS_PARAM,
                          1, Ann_RPC_Marker, 1, 3);
   cout << "Hello world from Processor Object #" << id << endl ;
}
```

3. **Master.cc++**

```
#include "probe.h"   // File Id == 0
#include <iostream.h>
#include "Slave.h"

int main (int argc, char ** argv)
{
   EntityProfiler Ann_FP (EV_MAIN_START,
                          ANN_FP_MAIN_CONSTRUCT_PARAM,
                          3, -1, 0, 4);
   int P = atoi (argv [1]) ;
   Slave *global G_slave [10] ;

   //--- PARFOR-BLOCK START
   Instr.synchronous_thread_block (EV_PARFOR_START,
                                   Ann_Thread_Id, 0, 9);
   parfor (int p=0 ; p<P ; p++) {
      IdType Ann_Parent_Id = Ann_Thread_Id;
      IdType Ann_Thread_Id = Instr.get_thread_id();
      EntityProfiler Ann_FP (EV_PARFOR_THREAD_START,
              Ann_Thread_Id, -1, -1, Ann_Parent_Id,
              -1 , -1, -1, -1, 0, 9);

      proc_t placement = proc_t ("Slave.out", argv [2+p]) ;

      {
         IdType Ann_RPC_Marker = Instr.get_RPC_mark();
         EntityProfiler Ann_FP (EV_RPC_MARK_START,
                                ANN_FP_RPC_MARKER_PARAM, 0, 11);
         G_slave [p] = new (placement)
                       Slave (ANN_GLOBAL_CLASS_CALL_PARAM);
      }

      {
         IdType Ann_RPC_Marker = Instr.get_RPC_mark();
         EntityProfiler Ann_FP (EV_RPC_MARK_START,
                                ANN_FP_RPC_MARKER_PARAM, 0, 12);
         G_slave [p]->say_hi (ANN_GLOBAL_CLASS_CALL_PARAM, p);
      }
   }
   Instr.synchronous_thread_block (EV_PARFOR_FINISH,
                                   Ann_Thread_Id, 0, 13);
   //----- PARFOR-BLOCK FINISH -----
```

```
   for (int p=0 ; p<P ; p++) {
      //----- Global-object delete -----
      {
         IdType Ann_RPC_Marker = Instr.get_RPC_mark();
         EntityProfiler Ann_FP (EV_RPC_MARK_START, Ann_Thread_Id,
                                -1, -1, -1, -1, -1, -1,
                                Ann_RPC_Marker, 0, 16);
         G_slave [p]->Ann_delete (ANN_GLOBAL_CLASS_CALL_PARAM);
      }
   }

   return 0 ;
}
```

Appendix B

Transaction Subsystem Code

This appendix provides only the common skeletal code for **Implementation-3** and **Implementation-4** of the experiments with the electronic transaction system (Section 5.1.4). In particular, the skeleton includes the code for the following:

1. the transaction requests,
2. the stock and customer databases,
3. the business logic unit,
4. the transaction servers, and
5. the transaction resolution subsystem.

Description of the code is also given as necessary.

B.1 The Transaction Requests

As described in Section 5.1.4, intentions from buyers and sellers are translated by the business logic unit into their associated transactions. In the system, a transaction is defined as a 4-tuple of `<Transaction_type, CustomerID, StockID, Quantity>`. The generic code of the transaction class is given in Figure B.1.

```
class TransactionClass {
   ...
public:
   TransactionClass (Id Transaction_type,
                     Id CustomerID, Id StockID,
                     Numeric Quantity);
   ~TransactionClass ();
   ...
};
```

Figure B.1. *Definition of the transaction request class.*

```
class CustomerDataBase {
private:
   [some internal data & functions]
   ...
public:
   CustomerDataBase ();
   ~CustomerDataBase ();
   Id get_DbaseEntry (CustomerData cd);
   ...
};
```

Figure B.2. *The customer database class.*

B.2 The Stock and Customer Databases

The implementation of the customer databases (holding the data of buyers and sellers) is straightforward. The databases are implemented as a class (`CustomerDataBase`), with the necessary functions to initialise, search, and update the databases, as shown in Figure B.2.

The implementation of the stock databases is a little different. The reason is that the databases are also used by the transaction resolution subsystem to match buyers and sellers. The code for the databases is given in Figure B.3.

```
class StockDataBase {
private:
   [some internal data & functions]
   ...
public:
   StockDataBase ();
   ~StockDataBase ();
   ...

   // Used by the BLU or trans-server to hold
   // portions of the stock database.
   atomic void prepare_StockDataBase
               (Id StockPartitionStart,
                Id StockPartitionEnd,
                Dbase &db);

   // Used by the transaction resolver to
   // match a buyer with a seller, and vice versa.
   atomic void do_transaction
               (Id transaction_type, Id customer_id,
                Id stock_id, Numeric quantity,
                TimeStamp t);
};
```

Figure B.3. *The stock database class.*

```
//--- The business logic unit
 global class DBserver {
    StockDataBase sdb;      // stock/commodity
    CustomerDataBase cdb;   // buyers and sellers
    ...
 public:
    DBserver ();
    ~DBserver ();
    ...

    // To initialise the transaction servers
    void init_TRservers (NodeName TRserver_NodeNames);

    // To translate intentions into transaction
    // requests, and regulate the flow of information
    void process_stream (Intention the_stream);
    ...
 };
```

Figure B.4. *Definition of the business logic unit.*

B.3 The Business Logic Unit

The business logic unit is responsible to regulate the flow of information between the client and the transaction servers. In the implementation, it is implemented as a processor object class (`DBserver`). The business logic unit, in turn, is also responsible for creating and initialising other POs which represent the transaction servers.

Figure B.4 is an outline of the code for the business logic unit in **Implementation-3**. The code for **Implementation-4** is similar, with some minor alterations.

B.4 The Transaction Servers

Transaction servers is responsible for matching potential buyers with potential sellers. This matching is carried out by using a transaction resolution subsystem, which is described in Section B.5.

The transaction servers are initialised by the business logic unit through the function `void DBserver::init_TRservers(NodeName TRserver_NodeNames)`. The skele-

```
global class TRserver {
private:
   [some internal data & functions]
   ...
public:
   TRserver (Id MyId, Id StockPartitionStart,
             Id StockPartitionEnd);
   ~TRserver ();
   ...

   // To process a transaction request, i.e. to queue
   // the request into the transaction resolver.
   // Later on, the trans-resolver will try to match
   // potential buyers with sellers.
   void process_transaction (TimeStamp t,
                             TransactionClass r);
};
```

Figure B.5. *Definition of the transaction server.*

tal code for the transaction server is given in Figure B.5.

B.5 The Transaction Resolution Subsystem

The transaction resolution subsystem is part of each transaction server. The subsystem contains two queues: the seller queue and the buyer queue. Both priority queues are used by the transaction servers to match buyers and sellers. The code is given in Figure B.6.

```
class TimeStampedTrans_PriorityQ {
private:
   [declarations for buyer & seller queues]
   ...
public:
   ...
   TimeStampedTrans_PriorityQ ();
   ~TimeStampedTrans_PriorityQ ();

   // Called by TRserver::process_transaction() to
   // put a transaction request.  The queue to be
   // used is determined automatically by the system,
   // based on the transaction type.
   int enqueue_trans (Id TransType, Id CustumerId,
                      Id StockId  , Numeric Quantity,
                      TimeStamp t);

   // Similar to enqueue_trans(), this function is
   // called by TRserver::process_transaction() whenever
   // a buyer can be matched with a server, i.e.
   // process_transaction() accesses both the seller
   // and the buyer queues.
   int dequeue_trans (Id &TransType, Id &CustumerId,
                      Id &StockId  , Numeric &Quantity,
                      TimeStamp &t);
   ...
};
```

Figure B.6. *Definition of the transaction resolution subsystem.*

Bibliography

[1] P.J. Asente, R.R. Swick, and J. McCormack. *X Window System Toolkit: the Complete Programmer's Guide and Specification.* Digital Press, 1990.

[2] P. Ashton. Algorithms of off-line clock synchronisation. Technical Report TR-COSC 12/95, Department of Computer Science, University of Canterbury, New Zealand, December 1995.

[3] P. Ashton. The Amoeba Interaction Network Monitor – Initial Results. Technical Report TR-COSC 09/95, Department of Computer Science, University of Canterbury, New Zealand, October 1995.

[4] R.A. Aydt. *An Informal Guide to Using Pablo.* Department of Computer Science, University of Illinois at Urbana-Champaign, January 1995. Available from ftp://www-pablo.cs.uiuc.edu/pub/Release/Documentation/PabloGuide.ps.Z.

[5] R. Baecker, C. DiGiano, and A. Marcus. Software Visualization for Debugging. *Communications of the ACM*, 40(4):44–54, April 1997.

[6] R.M. Baecker and A. Marcus. Design Principles for the Enhanced Presentation of Computer Program Source Text. In *Proceedings of Human Factors in Computing Systems (CHI'88), Washington D.C.*, pages 51–58. ACM Press, May 1988.

[7] T. Ball. Software Visualisation in the Large. *IEEE Computer*, 29(4):33–43, April 1996.

[8] P. Beckman, D. Gannon, and E. Johnson. Portable parallel programming in HPC++. In H.J. Siegel, editor, *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, pages 132–139. IEEE Computer Society Press, Augustus 1996.

[9] A. Beguelin, J. Dongarra, A. Geist, and V. Sunderam. Visualisation and Debugging in a Heterogeneous Environment. *IEEE Computer*, 26(6):88–95, June 1993.

[10] A. Beguelin and E. Seligman. Causality-Preserving Timestamps in Distributed Programs. Technical Report CMU-CS-93-167, School of Computer Science, Carnegie-Mellon University, 1993.

[11] B.N. Bershad, D. Lazowska, and H.M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software — Practice and Experience*, 18(8):713–732, August 1988.

[12] H. Bocker, G. Fischer, and H. Nieper. The Enhancement of Understanding through Visual Representations. In *Proceedings of the Computer Human Interaction, 1986 Conference, Human Factors in Computing Systems - III*, pages 44–50, 1986.

[13] F. Bodin. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In *Proceedings of the Supercomputing'93 Conference in Portland, Oregon*, November 1993.

[14] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *Proceedings of OONSKI'94, Oregon*, 1994.

[15] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In *Proceedings of the 1993 Supercomputing Conference, Portland, Oregon*, pages 588–597, 1993.

[16] G. Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley Longman, Inc., Reading, Massachusetts, second edition, 1994.

[17] D. Brown, S. Hackstadt, A. Malony, and A. Malony. Program Analysis Environments for Parallel Language Systems: The TAU Environment. In *Proceedings of the 2nd Workshop on Environments and Tools for Parallel Scientific Computing, Townsend, Tennessee, USA*, pages 162–171, 1994.

[18] M.H. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. Technical Report 75, Systems Research Center, Digital Equipment Corporation, February 1992.

[19] M.H. Brown and J. Hershberger. Colour and Sound in Algorithm Animation. Technical Report 76a, The Systems Research Centre, Digital Equipment Corporation, Palo Alto, California, August 1991.

[20] M.H. Brown, B.A. Myers, and E.P. Glinert. *Introduction to Visual Programming Environment.* ACM Press, New York, 1989. ACM SIGGRAPH '89 course notes/SIGGRAPH '89.

[21] M.H. Brown and M.A. Najork. Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom. Technical Report 142, The Systems Research Centre, Digital Equipment Corporation, Palo Alto, California, May 1996.

[22] M.H. Brown and R. Sedgewick. A System for Algorithm Animation. *Computer Graphics*, 18(3):177–186, July 1984.

[23] P.A. Buhr, G. Ditchfield, R.A. Stroobosscher, B.M. Younger, and C.R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software — Practice and Experience*, 22(2):137–172, February 1992.

[24] P.A. Buhr and M. Karsten. *μC++ Monitoring, Visualisation and Debugging, Annotated Reference Manual, Preliminary Draft.* Department of Computer Science, University of Waterloo, Waterloo, Canada, version 1.1 edition, December 1996. Also available from ftp://plg.uwaterloo.ca/pub/MVD/Visualization.ps.gz.

[25] P.A. Buhr, M. Karsten, and S. Jun. *KDB: Concurrent Debugger, Reference Manual.* Department of Computer Science, University of Waterloo, Waterloo, Canada, version 1.1 edition, February 1997. Also available from ftp://plg.uwaterloo.ca/pub/MVD/KDB.ps.gz.

[26] P.A. Buhr and R.A. Stroobosscher. *μC++ Annotated Reference Manual, Version 4.4.* Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, September 1995. Technical report unnumbered, available from ftp://plg.uwaterloo.ca/pub/uSystem/uC++.ps.gz.

[27] W.T. Cai and S.J. Turner. Process Scheduling and Program Monitoring on Transputers. In S. Atkins and A.S. Wagner, editors, *Transputer Research and Applications, NATUG-6, Proceedings of the 6th Conference of the North American Transputer Users Group*, pages 290–305. IOS Press, 1993.

[28] W.T. Cai and S.J. Turner. An Approach to the Run-Time Monitoring of Parallel Programs. *The Computer Journal*, 37(4):333–345, 1994.

[29] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. Medea: A Tool for Workload Characterisation of Parallel Systems. *IEEE Parallel and Distributed Technology*, 3(4):72–80, 1995.

[30] P. Carlin, K.M. Chandy, and C. Kesselman. The Compositional C++ Language Definition. Technical Report CS-TR-92-02, Department of Computer Science, California Insitute of Technology, Pasadena, California, 1993.

[31] J.M. Carroll and J.R. Olson. Mental Models in Human-Computer Interaction. In M. Helander, editor, *Handbook of Human-Computer Interaction*, chapter 2, pages 45–65. Elsevier Science Publishers B.V., North-Holland, 1990.

[32] CC++ Designer Team. *CC++ Tutorial.* Department of Computer Science, California Insitute of Technology, Pasadena, California, 1994.

[33] K.M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-Oriented Programming Notation. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 11, pages 282–313. The MIT Press, Cambridge, Massachusetts, 1993.

[34] P. Chen. Data Generation and Data Visualization Using Different Platform Computers in the World Wide Web Environment. In *Proceedings of the Conference of Visual Data Exploration and Analysis IV, IS&T/SPIE Symposium on Electronic Imaging: Science and Technology, San Jose, California, February 1997*, pages 2–13, San Jose, California, February 1997.

[35] A.A. Chien, U.S. Reddy, J. Plevyak, and J. Dolby. ICC++ – A C++ Dialect for High Performance Parallel Computing. In K. Futatsugi and S. Matsuoka, editors, *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software, Kanazawa, Japan, March 1996*, pages 76–95, Berlin, Germany, March 1996. Springer-Verlag.

[36] J.H. Cross and T.D. Hendrix. Language Independent Program Visualisation. In P. Eades and K. Zhang, editors, *Software Visualisation*, pages 27–45. World Scientific Publishings Co Pte. Ltd., Singapore, 1996.

[37] B.A. Delagi, N.P. Saraiya, and S. Nishimura. Monitoring Concurrent Object-Based Programs. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 15, pages 479–509. The MIT Press, Cambridge, Massachusetts, 1993.

[38] Department of Computer Science, California Institute of Technology. Summary of Workshop on Parallel Programming in C++, June 1993. Available from http://cs.caltech.edu/comp/papers/cppworkshop/cppworkshop.tex.

[39] W. DePauw, R. Helm, D. Kimelman, and J. Vlissides. Visualising the Behaviour of Object-Oriented Systems. *ACM SIGPLAN Notices*, 28(10):326–337,453–454, October 1993.

[40] W. DePauw, D. Kimelman, and J. Vlissides. Modeling Object-Oriented Program Execution. *Proceedings of the 8th European Conference on Object-Oriented Programming 1994*, pages 163–182, July 1994.

[41] K. Dincer and G.C. Fox. Using Java and JavaScript in the Virtual Programming Laboratory: a Web-based Parallel Programming Environment. *Concurrency: Practice and Experience*, 9(6):485–508, June 1997.

[42] J. Domingue and P. Mulholland. Fostering Debugging Communities on the Web. *Communications of the ACM*, 40(4):65–71, April 1997.

[43] T.H. Dunigan. Hypercube Clock Synchronisation. *Concurrency: Practice and Experience*, 4(3):257–268, May 1992.

[44] M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. Technical Report 21, Human Cognition Research Laboratory, The Open University, Milton Keynes, MK7 6AA, UK, 1986.

[45] S. Ellershaw and M.J. Oudshoorn. Program Visualisation — The State of the Art. Technical Report TR94-19, Department of Computer Science, University of Adelaide, November 1994.

[46] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1990. ANSI Base Document.

[47] R.F. Erbacher. Visual Debugging of Data and Operations for Concurrent Programs. In *Proceedings of the Conference of Visual Data Exploration and Analysis IV, IS&T/SPIE Symposium on Electronic Imaging: Science and Technology, San Jose, California, February 1997*, pages 120–128, San Jose, California, February 1997.

[48] C.H. Ferguson. Multiview: An Integrated Approach to Visualisation of Parallel Programs. Technical Report UCSC-CRL-90-20, Computer Research Laboratory, University of California at Santa Cruz, May 1990.

[49] C.J. Fidge. *Dynamic Analysis of Event Orderings in Message-Passing Systems.* PhD thesis, Department of Computer Science, The Australian National University, March 1989.

[50] C.J. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.

[51] V. Fix, S. Wiedenbeck, and J. Scholtz. Mental Representations of Programs by Novices and Experts. In S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel, and T. White, editors, *Proceedings of the Conference on Human Factors in Computing Systems, INTERACT'93 and CHI'93, The Netherlands*, pages 74–79. ACM, April 1993.

[52] D. Flanagan. *Java in a Nutshell.* O'Reilly & Associates, 2nd edition, May 1997.

[53] I. Foster. *Designing and Building Parallel Programs.* Prentice-Hall Publishing Company, 1st edition, 1995.

[54] I. Foster, J. Garnett, and S. Tuecke. Nexus User's Guide, August 1994. Available from ftp://ftp.mcs.anl.gov/pub/nexus/reports/nexus_iguide_v2.0.ps.Z.

[55] I. Foster, C. Kesselman, R. Olson, and S. Tuecke. Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems, August 1994. Available from ftp://ftp.mcs.anl.gov/pub/nexus/reports/nexus_spec_v2.0.ps.Z.

[56] I. Foster, C. Kesselman, and S. Tuecke. Nexus: Runtime Support for Task-Parallel Programming Languages. Available from ftp://ftp.mcs.anl.gov/pub/-nexus/reports/nexus_paper.ps.Z.

[57] I. Foster, G.K. Thiruvathukal, and S. Tuecke. Technologies for Ubiquitous Supercomputing: a Java Interface to the Nexus Communication System. *Concurrency: Practice and Experience*, 9(6):465–475, June 1997.

[58] J.M. Francioni and J.A. Jackson. Breaking the Silence: Auralisation of Parallel Program Behaviour. *Journal of Parallel and Distributed Computing*, 18:181–194, 1993.

[59] C. Fry. Programming on an Already Full Brain. *Communications of the ACM*, 40(4):55–64, April 1997.

[60] J. Gait. A Probe Effect in Concurrent Programs. *Software — Practice and Experience*, 16(3):2252–2330, 1986.

[61] J.A. Gannon, K.J. Williams, M.S. Andersland, T.L. Casavant, and J.E. Lumpp. Trace recovery in multi-processing systems: architectural considerations. In *Proceedings of the 1994 International Conference on Parallel Processing, St. Charles, Illinois*, volume II, pages 97–101, August 1994.

[62] E.R. Gansner, S.C. North, and K.P. Vo. DAG — A Program that Draws Directed Graphs. *Software — Practice and Experience*, 18(11):1047–1062, November 1988.

[63] J. Garnett, May 1996. Private communication.

[64] N.H. Gehani and W.D. Roome. Concurrent C++: Concurrent Programming with Class(es). *Software — Practice and Experience*, 18(12):1157–1177, December 1988.

[65] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[66] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley. A User's Guide to PICL, A Portable Instrumented Communication Library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge National Laboratory, Mathematical Sciences Section, Oak Ridge, Tennessee, USA, October 1990.

[67] M.M. Gorman. *Enterprise Database in a Client/Server Environment.* John Wiley & Sons, Inc., New York, 1994.

[68] A.S. Grimshaw. An Introduction to Parallel Object-Oriented Programming with Mentat. Technical Report TR-91-07, Department of Computer Science, University of Virginia, April 1991.

[69] M.T. Heath and J.A. Etheridge. Visualising the Performance of Parallel Programs. *IEEE Software*, 8(9):29–39, September 1991.

[70] M.T. Heath, A.D. Malony, and D.T. Rover. Parallel Performance Visualisation: From Practice to Theory. *IEEE Parallel and Distributed Technology*, 3(4):44–60, 1995.

[71] M.T. Heath, A.D. Malony, and D.T. Rover. The Visual Display of Parallel Performance Data. *IEEE Computer*, 28(11):21–28, November 1995.

[72] D. Heller. *Motif Programming Manual for OSF/Motif Version 1.1*, volume six. O'Reilly and Associates, Inc., 1991.

[73] D.P. Helmbold, C.E. McDowell, and J.Z. Wang. TraceViewer: A Graphical Browser for Trace Analysis. Technical Report UCSC-CRL-90-59, University of California at Santa Cruz, California, October 1990.

[74] W.L. Hibbard, B.E. Paul, D.A. Santek, C.R. Dyer, A.L. Battaiola, and M.F. Voidrot-Martinez. Interactive Visualisation of Earth and Space Science Computations. *IEEE Computer*, 27(7):65–72, July 1994.

[75] M. Hsu, September 1996. Private communication.

[76] B. Ibrahim. World Wide Algorithm Animation. Available from http://www.-oac.uci.edu/indiv/franklin/doc/ibrahim/paper.html.

[77] KGT Inc. AVS/Express Viz: The Leading Multi-Platform Visualisation Solution for Scientific, Technical and Commercial Applications. Available from http://titan.kgt.co.jp/avs/Viz/viz_e.htm.

[78] S. Isoda, T. Shimomura, and Y. Ono. VIPS: A Visual Debugger. *IEEE Software*, 4(3):8–19, May 1987.

[79] D.F. Jerding and J.T. Stasko. Using Visualisation to Foster Object-Oriented Understanding. Technical Report GIT-GVU-94-33, Graphics, Visualisation and Usability Centre, College of Computing, Georgia Institute of Technology, July 1994.

[80] M.F. Kleyn and P.C. Gingrich. GraphTrace — Understanding Object-Oriented Systems Using Concurrently Animated Views. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications OOPSLA 1988*, pages 191–205, September 1988.

[81] J.A. Kohl and G.A. Geist. The PVM 3.4 Tracing Facility and XPVM 1.1. In H. El-Rewini and B.D. Shriver, editors, *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences, vol.1*, pages 290–299, 1996.

[82] E. Kraemer and J.T. Stasko. The Visualisation of Parallel Systems: An Overview. *Journal of Parallel and Distributed Computing*, 18:105–117, 1993.

[83] D. Kranzlmüller, S. Grabner, and J. Volkert. Program Analysis through Visualisation. In P. Eades and K. Zhang, editors, *Software Visualisation*, pages 183–202. World Scientific Publishings Co Pte. Ltd., Singapore, 1996.

[84] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[85] D.A. Lane. Visualisation of Numerical Unsteady Fluid Flows. Technical Report NAS-95-017, Computer Sciences Corporation, NASA Ames Research Centre, Moffett Field, California, 1995. Available from http://www.nas.nasa.-gov/NAS/TechReports/NASreports/NAS95017.html.

[86] T. Lehr, Z. Segall, D.F. Vrsalovic, E. Caplan, A.L. Chung, and C.E. Fineman. Visualising Performance Debugging. *IEEE Computer*, 22(10):38–51, October 1989.

[87] K.C. Li and K. Zhang. A Performance Advicer for the Development of Parallel Programs. *International Journal of High Speed Computing*, 8(8):658–669, 1995.

[88] T. Lin and P. Eades. Layout Creation Methods for Software Visualisation. In P. Eades and K. Zhang, editors, *Software Visualisation*, pages 61–82. World Scientific Publishings Co Pte. Ltd., Singapore, 1996.

[89] J.E. Lumpp, T.L. Casavant, J.A. Gannon, K.J. Williams, and M.S. Andersland. Trace Recovery for Debugging Parallel and Distributed Systems. *The 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego*, pages 208–210, May 1993.

[90] P. Lyons, C. Simmons, and M. Apperley. Hyperpascal: A Visual Language to Model Idea Space. In *Proceedings of the 13th New Zealand Computer Society Conference*, pages 492–508, New Zealand, August 1993.

[91] A.D. Malony, D.H. Hammerslag, and D.J. Jablonowski. Traceview: A Trace Visualisation Tool. *IEEE Software*, 8(9):19–28, September 1991.

[92] G. Marwaha and K. Zhang. Parallel Program Visualisation for a Message-Passing System. In *Proceedings of the 13th Annual IEEE International Conference on Computers and Communications, Phoenix, USA*, pages 200–205. IEEE Press, April 1994.

[93] D. McIntyre. Comp.Lang.Visual — Frequently-Asked Questions (FAQ) List. The Internet Newsgroup comp.lang.visual, April 1996. Updated weekly.

[94] S. Meyers. *Effective C++*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[95] G. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review*, 63:81–97, March 1956.

[96] B. Mohr. A portable dynamic profiler for c++ based languages. Available from ftp://ftp.extreme.indiana.edu/pub/sage/instr.ps.gz, September 1993.

[97] B. Mohr, D. Brown, and A. Malony. TAU: A Portable Parallel Program Analysis Environment for pC++. In B. Buchberger and J. Volkert, editors, *Lecture Notes in Computer Science, volume 854, Proceedings of the International Conference*

on Vector and Parallel Processing, CONPAR'94, pages 29–40. Springer-Verlag; Berlin, Germany, 1994.

[98] B. Mohr, A. Malony, and K. Shanmugam. Speedy: An Integrated Performance Extrapolation Tool for pC++ Programs. In H. Beilner and F. Bause, editors, *Proceedings of the Joint Conference PERFORMANCE TOOLS'95 and MMB'95, 20-22 September 1995, Heidelberg, Germany*, pages 254–268. Springer-Verlag, Berlin, Germany, 1995.

[99] G.M. Murch. Physiological Principles for the Effective Use of Colour. *IEEE Computer Graphics and Applications*, 4(11):49–54, November 1984.

[100] B. Nichols, D. Buttlar, and J.P. Farrell. *Pthreads Programming, A POSIX Standard for Better Multiprocessing.* O'Reilly and Associates, Inc., 1996.

[101] O. Nierstrasz. Composing Active Objects. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 5, pages 151–171. The MIT Press, Cambridge, Massachusetts, 1993.

[102] J. Oliver. *Introduction to the X Window System.* Prentice Hall, 1989.

[103] M.J. Oudshoorn, H.W. Widjaja, and S.K. Ellershaw. Aspects and Taxonomy of Program Visualisation. In P. Eades and K. Zhang, editors, *Software Visualisation*, pages 3–26. World Scientific Publishings Co Pte. Ltd., Singapore, 1996.

[104] C.M. Pancake, M.L. Simmons, and J.C. Yan. Performance Evaluation Tools for Parallel and Distributed Systems. *IEEE Computer*, 28(11):16–19, 1995.

[105] C.M. Pancake and S. Utter. A Bibliography of Parallel Debuggers, 1990 Edition. *SIGPLAN Notices*, 26(1):21–37, January 1991.

[106] N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19:295–341, 1987.

[107] L.D.S. Perry, C.M. Smith, and S. Yang. Current Virtual Reality Interfaces. *CROSSROADS, The ACM Student Magazine*, pages 23–28, 1997. Spring edition.

[108] P.A. Pierce. A Concurrent File System for a Highly Parallel Mass Storage Subsystem. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications (Monterey, California)*, pages 459–467. Association for Computing Machinery, May 1989.

[109] B.A. Price, R.M. Baecker, and I.S. Small. A Principled Taxonomy of Software Visualisation. *Journal of Visual Languages and Computing*, 4(3):211–266, September 1993.

[110] P. Ramanathan, G.S. Kang, and R.W. Butler. Fault-Tolerant Clock Synchronisation in Distributed Systems. *IEEE Computer*, 23:33–42, 1990.

[111] D.A. Reed. Performance Instrumentation Techniques for Parallel Systems. In L. Donatiello and R. Nelson, editors, *Lecture Notes in Computer Science, LNCS volume 729*, pages 463–490. Springer-Verlag, 1993.

[112] D.A. Reed, K.A. Shields, W.H. Scullin, L.F. Tavera, and C.L. Elford. Virtual Reality and Parallel Systems Performance Analysis. *IEEE Computer*, 28(11):57–67, November 1995.

[113] G. Roman and K.C. Cox. Program Visualisation: The Art of Mapping Programs to Pictures. In *Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia*, pages 412–420, May 1992.

[114] S.R. Sarukkai and D. Gannon. SIEVE: A Performance Debugging Environment for Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:147–168, 1993.

[115] S.R. Sarukkai and A.D. Malony. Perturbation Analysis of High Level Instrumentation for SPMD Programs. *SIGPLAN Notices*, 28(7):44–53, July 1993.

[116] T.E. Scheetz, T.A. Braun, and T.L. Casavant. Effectiveness of Software Trace Recovery Techniques for Current Parallel Architectures. In *Proceedings of the 1995 International Conference on High-Performance Computing, New Delhi, India*, 1995. To appear.

[117] T. Shimomura and S. Isoda. Linked-List Visualisation for Debugging. *IEEE Software*, 8(3):44–51, May 1991.

[118] C.M. Smith. Human Factors in Haptic Interfaces. *CROSSROADS, The ACM Student Magazine*, pages 14–16, 1997. Spring edition.

[119] D. Socha, M.L. Bailey, and D. Notkin. Voyeur: Graphical Views of Parallel Programs. *ACM SIGPLAN Notices*, 24(1):206–215, January 1988. Proceedings of the Workshop on Parallel and Distributed Debugging.

[120] J. Stasko, February 1997. Private communication.

[121] J.T. Stasko. POLKA Animation Designer's Package, August 1995. Available from ftp://ftp.cc.gatech.edu/pub/people/stasko/polka.tar.Z.

[122] J.T. Stasko and E. Kraemer. A Methodology for Building Application-Specific Visualisations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:258–264, 1993.

[123] M.D. Storey, H.A. Müller, and K. Wong. Manipulating and Documenting Software Structures. In P. Eades and K. Zhang, editors, *Software Visualisation*, pages 244–263. World Scientific Publishings Co Pte. Ltd., Singapore, 1996.

[124] B. Stroustroup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1993.

[125] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed Intelligent Agents. *IEEE Expert*, 11(6), 1996.

[126] A.S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall Publishing Company, Englewood Cliffs, N.J, 1992.

[127] D. Taylor and P.A. Buhr. *POET with μC++, Reference Manual.* Department of Computer Science, University of Waterloo, Waterloo, Canada, February 1997. Also available from ftp://plg.uwaterloo.ca/pub/MVD/Poet.ps.gz.

[128] The Nexus Development Team. Nexus source code. Available from ftp://ftp.-mcs.anl.gov/pub/nexus.

[129] K. Tödter and C. Hammer. PARC++: A Parallel C++. *Software — Practice and Experience*, 25(6):623–636, June 1995.

[130] S. Tuecke, October 1996. Private communication.

[131] D. Ungar, H. Lieberman, and C. Fry. Debugging and the Experience of Immediacy. *Communications of the ACM*, 40(4):38–43, April 1997.

[132] J.Y. Vion-Dury and M. Santana. Virtual Images: Interactive Visualisation of Distributed Object-Oriented Systems. *ACM SIGPLAN Notices*, 29(10):65–84, October 1994. Proceedings of Object-Oriented Programming Systems, Languages, and Applications 1994.

[133] A. West. Making a Case for Animating C++ Programs. *Dr. Dobb's Journal*, 19(11):54–60, October 1994.

[134] H. Widjaja and M.J. Oudshoorn. Devising a Program Visualisation Tool for Concurrent and Object-Oriented Programs: A Survey. Technical Report TR95-14, Department of Computer Science, University of Adelaide, Australia, December 1995.

[135] H. Widjaja and M.J. Oudshoorn. Visualisation of Concurrent and Object-Oriented Systems. In *Proceedings of the Eighth International Conference on Computing and Information (ICCI'96), University of Waterloo, Waterloo, Canada*, pages 518–535, Waterloo, Canada, June 1996.

[136] H. Widjaja and M.J. Oudshoorn. Concurrent Object-Oriented Programming — A Visualisation Challenge. In *Proceedings of the Conference of Visual Data Exploration and Analysis IV, IS&T/SPIE Symposium on Electronic Imaging: Science and Technology, San Jose, California, February 1997*, pages 310–321, San Jose, California, February 1997.

[137] H. Widjaja and M.J. Oudshoorn. Design and Use of a Visualisation Tool for Concurrent Object-Oriented Programs. Technical Report TR97-08, Department of Computer Science, University of Adelaide, Australia, September 1997.

[138] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

[139] L.D. Wittie. Debugging Distributed C Programs by Real Time Replay. *ACM SIGPLAN Notices*, 24(1):57–67, January 1989. Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging.

[140] M. Wooldridge and N.R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[141] P. Wright, D. Mosser-Wooley, and B. Wooley. Using Colour in Computer Interface Design. *CROSSROADS, The ACM Student Magazine*, pages 3–6, 1997. Spring edition.

[142] J. Yan, S. Sarukkai, and P. Mehra. Performance Measurement, Visualisation and Modelling of Parallel and Distributed Programs using the AIMS Toolkit. *Software — Practice and Experience*, 25(4):429–461, April 1995.

[143] A. Zeller and Lütkehaus. DDD — A Free Graphical Front-End for UNIX Debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, January 1996. Also available from ftp://ftp.ips.cs.tu-bs.de/pub/local/softech/papers/tr-95-07.ps.gz.

[144] D. Zernik, M. Snir, and D. Malki. Using Visualisation Tools to Understand Concurrency. *IEEE Software*, 9(3):87–92, May 1992.

[145] Q.A. Zhao and J.T. Stasko. Visualising the Execution of Thread-based Parallel Programs. Technical Report GIT-GVU-95-01, Graphics, Visualisation and Usability Centre, College of Computing, Georgia Institute of Technology, 1995.

[146] G. Zukav. *The Dancing Wu Li Masters.* Rider Books, London, 1995.