Adding Safe and Effective Load Balancing to Multicomputers

Paul Martin

Doctor of Philosophy

University of Edinburgh

1994
Dedication

To Joy.
Abstract

In the quest for ever greater cost-effectiveness researchers have begun to experiment with scalable, parallel architectures known as 'multicomputers'. The underlying assumption is that adding more processors to a computer is a cheap way to increase the problem size which it can tackle and/or decrease the execution time. However, results to date are less good than those hoped for, indicating that there are still a number of difficulties to be resolved. One problem in particular is felt by experienced multicomputer programmers looking for significant execution time speed-ups: much effort must be expended to tune a program for the underlying architecture if the work is to be evenly distributed between the processors. Fortunately, a solution to this problem can be found in dynamic load balancing, a mechanism for redistributing work between processors automatically and transparently, allowing the programmer to develop fast, portable programs without having to worry about performance tuning.

This thesis examines the many issues associated with adding load balancing to multicomputers and makes the following contributions. Firstly, a critical review of the literature on load balancing showing the techniques proposed and suggesting which are the most promising for future systems. Secondly, a detailed description of how a typical multicomputer operating system needs to be extended in order to be able to checkpoint a task on one processor and restart it on another. Thirdly, a study of the impact on performance of these extensions to the operating system functionality. Fourthly, an investigation into the use of formal methods for designing and verifying the protocols for exchange of tasks between processors. Lastly, a report on the best methods for detecting processor load imbalances and for deciding which user tasks to move. Thus, the thesis addresses all aspects of adding safe and effective load balancing to multicomputers.
Acknowledgements

Many thanks are due to my supervisors Dr Rosemary Candlin and Dr Stephen Gilmore for their encouragement and direction. I am indebted to Kayhan Imre, Tom Whigham and Peter Lindsay for the many hours they spent debugging the Testbed hardware. Finally, for their effort and care in reading my thesis, I am grateful to my internal examiner, Dr Gordon Brebner, and external examiner, Professor Maurice Clint of The Queen's University of Belfast.

This work was funded by a Research Studentship from the Science and Engineering Research Council.

Declaration

I declare that this thesis was composed by myself and that the work which it describes is my own. Some of the material has been published in Martin and Gilmore [65]. The Figure 8–22 on page 192 has been copied with permission from Sayle [91].

Paul Martin.
# Table of Contents

1. Introduction  
   1.1 Synopsis .................................. 3  

2. Previous Work as a Basis for Research  
   2.1 The Route to Higher Performance ............... 5  
       2.1.1 Terminology ................................ 6  
   2.2 The Problems with Concurrency .................. 7  
       2.2.1 Problem 1: Finding concurrency in the application .... 7  
       2.2.2 Problem 2: Synchronisation of concurrent tasks ....... 8  
       2.2.3 Problem 3: Machine dependencies ............... 9  
       2.2.4 Problem 4: Operating system overheads ............. 9  
       2.2.5 Problem 5: Scheduling concurrent tasks .......... 11  
   2.3 Load Balancing Schedulers ...................... 12  
       2.3.1 The reconnaissance phase .................. 14  
       2.3.2 The decision-making phase ................. 20  
       2.3.3 The execution phase ..................... 23  
   2.4 Review of Existing Systems Relevant to Load Balancing ...... 24  
   2.5 Conclusions and a Plan for Research .............. 29  
       2.5.1 Restrictions on the area of research .......... 29
Table of Contents

2.5.2 Outstanding questions ........................................ 30
2.5.3 A series of studies ............................................ 31
2.5.4 Selecting a methodology ....................................... 32
2.5.5 Selecting a multicomputer ..................................... 33
2.5.6 Selecting a formal method ..................................... 34

3. Introducing the Testbed ........................................... 38

3.1 Conventional Features ........................................... 38
3.1.1 The parallel processor boards ............................... 40
3.1.2 The Testbed operating system ............................... 41
3.1.3 Programming environment .................................... 43

3.2 Support for Load Balancing .................................... 45
3.2.1 Special data structures ....................................... 45
3.2.2 Migration protocols .......................................... 46
3.2.3 Software instrumentation .................................... 49
3.2.4 Hardware for event collection ................................ 52

4. Specification of Testbed Basics .................................. 54

4.1 A Short Introduction to the Z Language ...................... 55
4.1.1 Schemas ...................................................... 55
4.1.2 Sets .......................................................... 57
4.1.3 Relations and functions ...................................... 58
4.1.4 Miscellaneous Z notation ................................... 59
4.1.5 Checking the specification .................................. 59

4.2 Thread Design .................................................... 60
## 6.3 Transparency of Thread Migration

- 6.3.1 The receiver location problem
- 6.3.2 Lemmas
- 6.3.3 Migration proof

## 7. Performance Profiling

- 7.1 The Experiments to be Performed
- 7.1.1 The test programs
- 7.2 Execution Models
- 7.2.1 The TOS main loop
- 7.2.2 The communication and migration protocols
- 7.3 Results of the Experiments
- 7.3.1 Latency of local communication
- 7.3.2 Latency of remote communication
- 7.3.3 Thread migration latency
- 7.3.4 Remote page copy latency
- 7.3.5 The effects of a busy environment
- 7.3.6 Typical communication latencies
- 7.3.7 Event rates
- 7.4 Conclusions

## 8. Load Balancing

- 8.1 Assessing the Load
- 8.1.1 A characterisation for ‘balanced load’
- 8.1.2 Selection factors for load metrics
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1.3</td>
<td>Metric case studies</td>
<td>167</td>
</tr>
<tr>
<td>8.2</td>
<td>Strategy for Load Reconfiguration</td>
<td>177</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Centralised versus distributed balancing</td>
<td>179</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Establishing parameters for the decision strategy</td>
<td>180</td>
</tr>
<tr>
<td>8.3</td>
<td>Effectiveness of the Balancer</td>
<td>184</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Case study: synthetic programs</td>
<td>185</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Case study: asynchronous circuits</td>
<td>188</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Case study: population simulation</td>
<td>194</td>
</tr>
<tr>
<td>8.3.4</td>
<td>Summary</td>
<td>199</td>
</tr>
<tr>
<td>8.4</td>
<td>Conclusions and Recommendations</td>
<td>200</td>
</tr>
<tr>
<td>9.</td>
<td>Conclusions</td>
<td>203</td>
</tr>
<tr>
<td>9.1</td>
<td>Review of Results</td>
<td>203</td>
</tr>
<tr>
<td>9.2</td>
<td>Implications for Load Balancing</td>
<td>208</td>
</tr>
<tr>
<td>9.3</td>
<td>Future Work</td>
<td>209</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>211</td>
</tr>
<tr>
<td>A.</td>
<td>Test Program Listings</td>
<td>224</td>
</tr>
<tr>
<td>A.1</td>
<td>local</td>
<td>224</td>
</tr>
<tr>
<td>A.2</td>
<td>remote</td>
<td>225</td>
</tr>
<tr>
<td>A.3</td>
<td>migrate</td>
<td>226</td>
</tr>
<tr>
<td>A.4</td>
<td>multi-phase</td>
<td>227</td>
</tr>
<tr>
<td>A.5</td>
<td>overflow</td>
<td>228</td>
</tr>
<tr>
<td>A.6</td>
<td>synthetic</td>
<td>229</td>
</tr>
<tr>
<td>A.7</td>
<td>adder</td>
<td>231</td>
</tr>
</tbody>
</table>
B. Index of Z Terms
Chapter 1

Introduction

It seems that the cost-effectiveness of current computers is never good enough—or if it is then we want to execute new, more computationally-demanding programs on less expensive machines. One of the most promising approaches to feeding this hunger is the development of the scalable, parallel architectures known as multicomputers.

While the construction of large multicomputers is relatively straightforward using current hardware techniques, the difficulty of writing efficient parallel programs using current software techniques has significantly delayed the widespread use of multicomputers (except, perhaps, for certain problems with regular structure found in scientific computing). The key problem for the parallel programmer is how to create a program which will exploit the potential contribution of each processor. This is not easy when the program may be executed on multicomputers of different sizes, communication topologies or speeds; when the program may have very different dynamic behaviour according to its input data; or when the program may have to share the multicomputer with a variety of other programs.

The particular aspect of the parallel software problem studied in this thesis is the feasibility of transparent, dynamic load balancing, i.e. the automatic assignment by the multicomputer of program tasks to the most suitable processors. (Some systems assign tasks just once, but the emphasis in this thesis is on dynamic balancing which means that tasks can be reassigned as many times
as is necessary.) If load balancing is possible and the parallel program contains a sufficient number of semi-independent tasks then the parallel programmer is assured that a reasonable mapping of tasks to processors will always occur and that the program execution time will be close to the best achievable. Thus, the desirability of load balancing multicomputers is evident.

If load balancing is to be an operating system function (rather than being done on a per-program basis) then three new components are required: a mechanism for measuring the load currently being experienced by the processors, communication network and other valuable resources; a decision strategy which inputs the load measurements, identifies imbalances and generates reconfiguration instructions; and a set of routines which can be invoked to transparently exchange user tasks between processors.

Load balancing places certain requirements on the user programs that can be supported. Programs to be load balanced need to be structured as multiple but uniquely-identifiable tasks communicating with each other using message passing. Typically, imperative languages such as occam or C (with suitable extensions for concurrency) are used.

The specific questions addressed by this thesis are:

- What is the best way to measure the load?
- How can good load balancing strategies be found?
- What needs to be done to add load balancing to a typical operating system?
- How large are the costs of load balancing and what kinds of savings can be expected?

This thesis employs a range of methodologies to address these questions: a literature review to suggest possible techniques; formal methods to assist the design and verification of migration protocols involving multiple, autonomous agents; the use of specialised hardware able to measure in real time the detailed
activity of a multicomputer; and the development of a complete, functioning load balancing system to show proof of concepts.

The contributions of this thesis can be summarised as follows.

- A review of the literature on load balancing, introducing the problem area and comparing the techniques proposed and the systems implemented.
- A detailed description of how a typical multicomputer operating system needs to be extended in order to support load balancing.
- An investigation into the costs and benefits of applying formal methods to the design and verification of the task exchange protocols.
- A study of the performance costs associated with the additional operating system components required to support load balancing.
- An examination of the best methods for assessing the load as a basis for identifying imbalances.
- A series of case studies on the performance benefits obtained from load balancing in relation to the type of parallel program being executed.

In essence, therefore, this thesis addresses all aspects of adding safe and effective load balancing to multicomputers.

1.1 Synopsis

This thesis is organised as follows. Chapter 2 reviews the literature so as to introduce the problem domain and expound the solutions which have been proposed. The most promising approaches are then selected and used to develop the basic strategy which underlies the rest of the thesis.

Chapter 3 introduces the 'Testbed' multicomputer, its hardware architecture, operating system design and programming environment and gives a detailed
description of the system functions which support load balancing. The Test-
bed is used in many parts of the thesis because it allows various load balancing
topics to be illustrated and because it offers a realistic environment in which to
make measurements.

Chapters 4 and 5 introduce the formal language Z and show how a specifica-
tion of protocols for task synchronisation, communication and migration can
be developed. The subject of Chapter 6 is verification and in it the safety and
correctness of the specification are demonstrated.

The cost profiles of the main Testbed operating system functions are given in
Chapter 7. This work describes exactly how the experiments were set up and
relates the results to the mechanisms specified in Chapters 4 and 5.

Chapter 8 then enumerates methods for dynamic measurement of the load
experienced by multicomputers and shows how good metrics are selected. A
strategy for detecting load imbalances is developed and the effectiveness of a
sample load balancer is measured for a range of parallel programs.

Chapter 9 concludes the thesis, reviewing the contributions made to the subject
area.
Chapter 2

Previous Work as a Basis for Research

This chapter looks at the previous work on extending concurrent operating systems to support load balancing and develops a strategy for research into the subject of load balancing multicomputers. I begin by describing the motivation for concurrent computing, by classifying the key problems concomitant with concurrency and by identifying one of the greatest problems as being that of scheduling tasks on processors in a balanced way. The central part of this chapter presents a thorough review of the literature on scheduling and evaluates the costs and benefits of the various techniques that have been tried. Finally, I draw some important conclusions from the previous work and use them as a basis for developing a research plan on adding safe and effective load balancing to multicomputer operating systems.

2.1 The Route to Higher Performance

The story of computing over the last 40 years has been one of innovation in architecture design and improvement in technology. Computer performance has increased dramatically with the introduction of pipelining, multiple functional units, caches and optimising compilers while advances in silicon technology have
shrunk component sizes, reduced heat dissipation and increased speed. As documented by Hack [34], this has led to the startling rise in computer performance at all cost levels.

One recent development stands out from the rest. Concurrency in its many forms (Duncan [25] has a good summary) is widely acclaimed as the best source for future performance improvements because of its scalability. The relative ease with which large numbers of inexpensive processing elements can be connected together has led Fox et al [29, Chapter 2] to state that, '...there is no doubt that concurrency is the only route to high performance computation.' Indeed, the signs are that this view has been widely accepted because the concurrent computers which dominate the mid-range of the market-place are, according to Messina [67], beginning to assail even the supercomputer market.

2.1.1 Terminology

The term 'concurrent computer' has been used by various people to describe many different computer architectures. This thesis restricts the term to denote MIMD computer systems with interconnected processors and distributed memory. Concurrent computers are then divided into 'distributed computers', e.g. several Unix hosts connected by a local area network, and 'parallel computers', e.g. an array of Transputers. The key differences between distributed and parallel computers are:

- The computation to communication performance ratio is usually much higher for distributed computers than for parallel computers.

- Processor interconnections in distributed systems are typically less reliable and thus require more complex communication protocols that those used in parallel machines.

- Applications written for distributed machines tend to have a larger grain size and less frequent inter-grain communication than applications for parallel machines.
Distributed machines often support multiple users and their performance is typically measured in terms of response time. Parallel machines are more likely to support single users and be rated by amount of computation.

By further sub-division, the term 'multicomputer' is used to denote the class of homogeneous, parallel computers. The multicomputers are a particularly important type of concurrent computer because they offer the greatest scalability and cost-effectiveness at all levels of computing, from the humble workstation with only a handful of processors to the mighty supercomputers with hundreds or thousands of processors. As previously noted, the bulk of this thesis is concerned with load balancing multicomputers. However, Sections 2.2 to 2.4 also consider relevant techniques from research on the parallel and distributed classes of concurrent computer.

2.2 The Problems with Concurrency

Sadly, upgrading from sequential to concurrent computing is not proving easy because it requires the radical review of a number of well-established ideas. Problems with sequential computers which have been studied over many years now have to be tackled afresh for concurrent computers—hence the volume of new literature on debugging, performance evaluation, scheduling and resource usage modelling. Karp [47] states that the art of concurrent programming is in a 'sorry state' and Hack [34] says of concurrent computing that 'the potential for general-purpose performance improvements could be dramatically smaller than is currently believed'. This section looks at five of the biggest problems encountered when moving from sequential to concurrent computers.

2.2.1 Problem 1: Finding concurrency in the application

Suppose that a programmer has an application in mind and that for reasons of cost and performance s/he wishes to use a concurrent computer to execute the
application. The question is, can s/he decompose the application into enough concurrent parts to keep most of the processing elements in the concurrent computer busy most of the time?

Experience shows that a large amount of concurrency can be extracted from some applications and that for others it can be proved that the amount of concurrency is strictly limited. But, as Krishnamurthy [51] points out, there is no guaranteed way of finding, in general, the appropriate degree of concurrency in an application to match the number of processors.

It is obvious that as the number of processors in concurrent computers increases, then, to use the hardware efficiently, more concurrency must be extracted from the application. As Seitz [93] argues, the number of architectures upon which an application will execute efficiently increases as the grain size decreases.

However, if the grain size is decreased too much, a whole range of overheads may be introduced. Burkhart and Millen [16] describe such overheads: 'idle processor loss' occurs when one task waits to synchronise with another, 'access loss' occurs when tasks compete for shared data and 'conflict loss' occurs when tasks compete for hardware resources. Moser [75] adds to the list with 'garbage computation loss' which occurs when concurrent tasks recalculate values which, on a sequential machine, would be shared using global variables. Finally, the greater the number of tasks the more time is spent by the system creating new tasks, farming them out, combining partial results and deleting old tasks.

### 2.2.2 Problem 2: Synchronisation of concurrent tasks

Suppose now that the programmer has managed to decompose his or her application into a suitable number of concurrent tasks. These tasks will need, if only at the start and end of the computation, to synchronise and perhaps communicate data or results. Two problems result immediately: it is notoriously difficult to write correctly programs with synchronisation and communication; and once implemented, such programs often need to be tuned for performance because the synchronisation delays depend on the mapping of tasks to processors.
The first problem has been studied extensively in the literature on concurrent debugging. McDowell and Helmbold [66], for example, review the problems of debugging concurrent programs; LeBlanc and Robbins [54] present a post-mortem debugging tool that allows the execution of concurrent programs to be replayed; and Netzer and Miller [76] employ special selection techniques to enable the tracing of long program executions.

The tuning problem has also been the subject of much research. Marinescu and Rice [64], for example, propose techniques for minimising time lost due to synchronisation; Mohr [73] describes a tool environment called SIMPLE for performance evaluation; Burkhart and Millen[16] show how a family of tools can be integrated; and Böhm et al [10] illustrate the benefits of graphical tools.

2.2.3 Problem 3: Machine dependencies

As already noted, the optimal degree of application concurrency depends to some extent on the concurrency of the target architecture. But there are other dependencies too: Hwang and Briggs [41] explain that the execution time of the application also depends on the relative speeds and functionality of the processors, memory and communication systems.

All such dependencies make programming concurrent computers difficult—more so for those not directly involved in designing or researching concurrent machines. If, as Pancake [82] suggests, no one concurrent architecture dominates the market and programmers must learn for each architecture what concurrent operations are available and what the relative cost of each operation is, then it is not surprising that programmers find concurrent computers so difficult to use.

2.2.4 Problem 4: Operating system overheads

Operating systems for concurrent computers are more complicated than operating systems for their sequential cousins. In a sequential computer a single decision maker allocates resources and synchronises user activities, but in a concurrent
computer there are often multiple, independent decision makers operating simultaneously. A significant amount of negotiation must occur between these decision makers if resource allocation and user task synchronisation are to be performed safely and efficiently. Tanenbaum and Renesse [103] give a fair summary of the additional complexity of concurrent computer systems when they state, 'Building a 16-node distributed system that has a total computing power about equal to a single-node system is surprisingly easy.'

**Sources of overhead**

There are three main causes of additional overhead in concurrent operating systems: the complexity of communication between processors; the requirement for fault tolerance; and the need for careful control of object sharing.

All concurrent computers need some way of coordinating processor activity and exchanging data. This may be done by implementing synchronisation and communication primitives using shared memory or using message passing. Either way, additional computation and communication overheads are incurred as data is packaged up at the source processor, transferred, and then unpackaged at the destination processor. Tanenbaum and Renesse [103] say that inter-processor communication is the biggest single source of operating system overhead and Cheriton [18] makes high-performance communication the first requirement in his design philosophy for the distributed system 'V'.

A problem for distributed computers (rather than parallel computers) is that as the number of processors and communication links increases, so does the probability that some processor or link will fail. When this happens the operating system must be able to detect and contain the problem without halting the whole machine. Even in the absence of hardware failure, the fact that it is so difficult to write concurrent software correctly means that the operating system must protect itself against rogue or misguided user programs. The usual solution for both forms of failure is to add further layers to the communication protocols and better checking of parameters passed by user programs. If, more ambitiously,
failed computations are to be rolled back and restarted then additional state information must be kept. All these kinds of protection impose significant overheads. An extensive summary of fault tolerance in distributed systems can be found in Cristian [19].

Other significant overheads are introduced when objects such as physical memory, program text and data segments, files and devices are shared. Sharing necessitates a (possibly global) naming scheme and access and protection controls. The problems of sharing have been tackled in traditional multi-user machines, but in parallel machines sharing may be required at a much finer level of granularity. For instance, Finkel et al [27] report that it was very difficult to achieve efficiency in the implementation of the Charlotte operating system and DeWitt et al [21] describe how the Crystal Multicomputer (on which Charlotte runs) allows users to bypass the operating system and access the lowest level of system software directly.

2.2.5 Problem 5: Scheduling concurrent tasks

Suppose that all the problems previously mentioned have been solved: the application has been decomposed into a suitable number of concurrent tasks, its implementation is known to be correct and has no obvious performance bottlenecks. Suppose also that the operating system has been carefully optimised to impose minimum overheads consistent with safe and reliable operation. Experience shows that even now, the application may execute inefficiently if the scheduling is not performed carefully.

Poor scheduling can be usefully characterised as the scheduling which arises whenever some instances of a resource are much more congested than others. Examples of resources include processor (compute) cycles, physical memory space and communication or I/O bandwidth. Although most research in the literature is concerned with the optimal distribution of processor cycles, a schedule which, for example, resulted in stiff competition for I/O at one processor whilst other processors had spare I/O capacity can be equally undesirable.
Several factors may contribute to scheduling imbalance.

- The patterns of resource usage may be unpredictable because the dynamics of the program are very complicated or because the behaviour of the program depends heavily on its data.

- The details of the computing resources available and their costs may not be known in advance, or may be very difficult to measure.

- The application may have to execute on several different architectures, or it may be multi-tasked with different combinations of other programs.

These factors are unavoidable in many concurrent applications and thus it is very difficult for a software developer to guarantee that his or her program will always execute efficiently. These factors can be mitigated, however, by a computer with a load balancing scheduler that matches, moment by moment, the demands of the program with the available resources.

### 2.3 Load Balancing Schedulers

Of the five key concurrency problems outlined above, this thesis is most concerned with ways to improve the scheduling of concurrent tasks. The size of the improvement possible has been assessed by, amongst others, Livny and Melman [60], Eager et al [26], Bryant and Finkel [14] and Zhou [114] to be in the range of a 30% to 50% reduction in execution time. This section looks at how load balancing schedulers can use measurements of the workload to produce speed-ups consistently and without user intervention.

There is a substantial amount of research on load balancing schedulers, and some attempts have been made to classify the techniques proposed. The taxonomy presented by Casavant and Kuhl [17] (Figure 2-1) is an often quoted example, although the Jacqmot and Milgrom [45] classification is perhaps more intuitive and more closely concerned with the various stages in dynamic load
balancing. However, as there is no universally accepted categorisation, the scheduling techniques proposed are organised in this section according to the following three, common phases.

1. A reconnaissance phase during which the loads currently experienced at each processor are assessed. The loads reflect the usage and capacity of each critical resource at the processor.

2. A decision-making phase during which the load assessment is used to suggest a better distribution of the application on the processors.

3. An execution phase during which a mechanism is invoked to redistribute the application tasks.

Sections 2.3.1 to 2.3.3 describe each of the three phases in some detail, outline the various options and techniques which have been proposed and describe the relevant results.
2.3.1 The reconnaissance phase

Sources of load information

There are five main sources of information used when assessing the load.

- Data extracted statically: Bokhari [12] relies on data extracted during compilation to identify the concurrent tasks and to discover the synchronization dependencies between them. Seredyński [94] uses static data to identify tasks and their average computation and communication requirements.

- Data extracted dynamically: Ni et al [78] compute the load on each processor dynamically by measuring such quantities as resource demands or number of tasks. Barak and Shiloh [3] have each processor regularly estimate its average response time.

- Statistical data: Leland and Ott [56] measure the performance of large numbers of programs and derive a probability distribution for the CPU time used by an arbitrary task.

- Historical data: Osser [80] maintains, for individual programs, a statistical history of the resources used in the past and Ieumwananonthachai et al [43] analyse previous executions to improve the learning automata for scheduling the next execution.

- Architectural data: Ni and Hwang [77] look at the scheduling of tasks on a computer with a heterogeneous collection of processors. In their case, the schedule is heavily dependent on knowledge about the relative speeds of the processors.

These sources may be used individually, or combined. For instance, some information about the relative speeds of the processors and topology of the interconnection network is always needed.
Cost of collecting

The costs of collecting static, statistical and architectural data are generally regarded as being irrelevant since they are collected only once, but the costs of collecting dynamic and historical data can be significant. Many researchers, including Livny and Melman [60], have concluded that the costs of collecting dynamic information in particular can impose a serious overhead. Of course, the costs of collecting dynamic and historical data depend on exactly what is measured, and Xu and Hwang [112] and [113] experiment specifically on the tradeoffs between the measurement overheads and the quality of load balancing. Other researchers have tried combining statistical data with dynamic data (for example Ni and Hwang [77]), and collecting historical data at a coarse level so that the collection costs are submerged in the general system costs (for example Osser [80]).

Appropriateness of different sources

It is not possible to give a general ranking for the usefulness of different sources of data, but it is reasonable to point out that some sources of data are not appropriate in some situations. A static characterisation, for instance, of a program with very variable dynamic behaviour is bound to be inaccurate. Dynamic data is useless if the operating system does not support task migration or, as Douglish and Ousterhout [24] show, if tasks access files attached to a particular processor (even where remote file access is possible, it is generally much more expensive than local access).

Mechanisms for collecting

There are many papers which describe software or hardware mechanisms for making load measurements, e.g. Malony and Reed [62], Hofmann et al [39] and Mink et al [70]. There are also many papers which describe how, given the load measurements, balancing may be performed, e.g. Ni et al [78], Bryant and Finkel [14] and Gao et al [30]. Sadly, there are few papers in which the costs and
benefits of different mechanisms are related to the quality of the load balancing (Bemmerl et al [5] is an exception). Most papers on measurement mechanisms are aimed only at debugging or performance analysis.

When measurements are made by the same processing elements that execute user programs the term software monitoring is employed. When measurements are made by additional, dedicated hardware then the term hardware monitoring is employed. Software monitoring involves instrumenting, i.e. adding new instructions to, the operating system or user’s code in order to generate load measurements. Software monitoring has been used in the HMON and TOPSYS projects (described on pages 26 and 27 respectively). Hardware approaches are proposed by Liu and Parthasarathi [59] and Tsai et al [105].

Software monitoring conveniently allows new events to be introduced and the representation of old events to be changed. Software monitoring can generate higher level or more abstract information than hardware monitoring and, although hardware monitoring can extract information in great detail, the detail is often excessive to the point of obscuring the events of importance. However, pure software monitoring competes with the program being monitored for computational, memory and communication resources.

Hardware-only monitoring is not very flexible and proprietary hardware and internal caches limit the usefulness of that which can be observed by the monitor. It is also more expensive to add specialised hardware to a computer than to extend its software.

Most systems actually use hybrid monitoring to combine the benefits of software—flexibility, configurability, ease of implementation—with the benefits of hardware—minimal space and time overheads and minimal interference with the program being monitored. Typical slow-downs induced by software-only monitoring are in the range 5% to 20% whereas hybrid monitoring usually slows the computation down by less than 2%. Example hybrid systems are PEM, described by Burkhart and Millen [16], the VLSI design proposed in Mink et al [70] and the ZM4 (page 28).
When to collect

There are two rules for deciding when to make load measurements. With *sampling*, *periodic* or *time-driven* rules, measurements are taken at fixed time intervals. With *triggering* or *event-driven* rules, measurements are taken when some predetermined set of circumstances arise. Both techniques are described in Hofmann *et al* [39].

Sampling is simple to implement although care must be taken to sample at the appropriate interval: if the interval is too small the monitoring system may be swamped with data, or fooled by short-term transient behaviour into making unnecessary migrations; if the interval is too large the load assessment may be inaccurate or out of date. Out-of-date information may lead to *thrashing*, i.e. a task being moved repeatedly without progress, or *flooding*, i.e. too many tasks being moved simultaneously to an underloaded processor. Mirchandaney *et al* [71] give a mathematical treatment of the effects of delays in the monitoring system.

Triggering may be more complicated to set up and, in contrast to sampling, the quantity of data that the monitor has to deal with may vary quite dramatically over time. However, triggering can indicate when the system reaches interesting states whereas sampling only gives a statistical probability distribution of past states.

Xu and Hwang [113] suggest combining the two methods: a triggering mechanism is constrained to generate load measurements with a fixed minimum and maximum periodicity. The minimum period prevents rapid triggering from swamping the monitoring system. The maximum period comes into operation when triggering would otherwise be infrequent, and provides some fault tolerance or reassurance for the monitoring system.

Suen and Wong [102] describe a sampling system in which the sampling rate is varied according the size of the load imbalance, the rate being reduced (and hence reducing the collection costs) as the balance nears optimality. Xu and
Hwang [112] and [113] take this idea further and dynamically vary the sampling rate in proportion to the rate of change in the load.

**Intrusion**

One issue of some importance for designers of load measurement systems is *intrusion* (also called *invasion* and the *probe effect*). Instrumenting code can affect the program execution in a number of ways.

- At a very low level, the behaviour of machines with pipelining, look-ahead queues or caches may be subtly altered by the instrumentation instructions. These alterations may have a knock-on effect.

- A change in the instruction sequence is likely to change the pattern of memory accesses. This may be more noticeable in machines using memory interleaving or other memory management techniques.

- At a higher level, the order in which user tasks request to synchronise may be affected when the code is instrumented. This in turn may cause the observed behaviour of the program to change significantly.

Various strategies can be employed to ameliorate these problems. If a given program is always executed with the same instrumentation, then any effect on the non-determinism should be the same between executions. This is the approach of the INCAS project (described on page 27) and the HMON project (page 26). Alternatively, as Liu and Parthasarathi [59] suggest, if the intrusion of the monitor is known then it may be possible to subtract its effects. Finally, if only a small part of the code is instrumented, as Mink *et al* [70] recommend, then the intrusion will be limited.

**Using locality when collecting**

There has been a great deal of research on reducing the number of load measurements passed between processors, because this is seen as a major source of
overhead in load balancing systems. (Note that this is a slightly different problem from that of reducing the number of measurements made.) Examples of this work can be found in Lin and Keller [58], Ni et al [78] and Xu and Hwang [112] and [113]. It is particularly important to reduce the number of load messages in machines where the messages share communication bandwidth with users’ programs. Furthermore, not all parallel computers are fully interconnected—Transputer arrays, for example, are limited to four links per processor—so there are obvious benefits to a load measurement technique which limits the number of hops each message has to make.

If processors do not exchange load messages at all, then load balancing decisions are said to be non-cooperative and based only on direct state information. Most systems are, in fact, cooperative because processors incorporate indirect state information about some or all of the other processors in the system.

As Casavant and Kuhl [17] indicate, indirect information can easily become out of date and, after a certain time, should be discarded or updated. A common technique involves weighting indirect information to limit the deleterious effects of its becoming dated. However, weighting is complicated if the processors in the system are heterogeneous.

Barak and Shiloh [3] report success with a system in which processors select partners at random, exchange load information with them, and then compute an estimation of the entire system load. The research explores the effects of different rates of information exchange and the effects of retaining different amounts of indirect information on each processor.

Suen and Wong [102] point out that keeping indirect load information for a small subset of processors can leave heavily loaded processors and lightly loaded processors stranded from each other, and thus unable to exchange tasks. The solution they propose involves processors sending load messages to one half of the system and receiving load messages from the other. Then, if a lightly loaded processor receives a message from a heavily loaded processor it will request work, and if a heavily loaded processor receives a message from a lightly loaded processor it will request to give away work.
Lin and Keller [58] describe the gradient method in which each processor exchanges load messages with a limited number of other processors, typically its neighbours. Each message sent contains the sender's load and the sender's neighbours' previous loads. This allows underloaded and overloaded processors to locate each other even if they do not exchange load messages directly. Each processor builds up an image of the gradient over time, can detect when it is overloaded (or underloaded) and can exchange work with the most appropriate neighbour. Movement of tasks causes the gradient to change. The gradient method described by Boillat and Kropf [11] is particularly interesting because it involves one of the few load balancing schedulers which optimises the use of computation and communication resources. However, the gradient method does not have any direct mechanism to combat thrashing and has been found to respond slowly to some kinds of imbalance.

2.3.2 The decision-making phase

Once the load on the concurrent computer has been measured and those measurements distributed to the appropriate locations, what Xu and Hwang [113] call the control policy is brought into action. The control policy looks for significant imbalances in the load and allocates new tasks (or reallocates existing tasks) to processors. The agent which implements the control policy is sometimes known as the global scheduler—this is to be distinguished from the (local) scheduler which is present in all conventional preemptive operating systems.

If the control policy is implemented on a single processor then it is known as centralised global scheduling; otherwise it is known as (physically) distributed global scheduling. The general view is that distributed global scheduling is preferable because it is more scalable (although Zhou [114] concluded from his research that centralised global scheduling actually reduced the total overheads of load balancing).

There are two main strategies for allocating tasks to processors: optimal and heuristic. Optimal scheduling, finding an allocation of tasks to processors with
minimum total execution time, is known to be NP-complete unless the problem is severely restricted: Bruno et al[13] show that even scheduling identical processors with no synchronisation, i.e. no communication, is NP-complete in general and Ullman [106] shows NP-completeness for other formulations of the scheduling problem. However, if the application has several of the following properties:

- is time critical;
- has highly predictable performance;
- is to be executed many times;
- is composed of a small number of tasks;

then the benefits of optimal scheduling may be attractive enough to outweigh its very large space or time requirements. Optimal schedules are likely to be computed only once (during compilation) and are not usually associated with machines that perform dynamic task migration. Boglaev [9] uses polynomial-time algorithms for linear programming to find exact solutions for a restricted class of programs. Shen and Tsai [96] examine optimal scheduling using a graph matching approach. Ni and Hwang [77] propose a method of load balancing which is (probabilistically) optimal.

Heuristic scheduling methods are many and varied but, in contrast with optimal methods, they all attempt to use as little load information as possible in order to compute schedules quickly and simply. Heuristic methods rely on finding some measurement which can be made inexpensively but which is related, possibly in a very indirect way, to the load. The argument for heuristic scheduling given by Lin and Keller [58] can be summarised as follows: the overall aim is to improve response time which is an a posteriori measure; therefore the only practical solution is to try to improve some secondary measure (processor utilisation, for example) and hope that this will eventually lead to improved response time. While the low cost of heuristic methods makes them attractive, the lack of accepted theory of good secondary measures restricts their generality.
Kasahara and Narita [48] report some success with two heuristic algorithms: critical path with most immediate successors first (CP/MISF, an extension to the critical path method to deal with arbitrary task graphs and more than two processors) and a variant called depth first with implicit heuristic search (a depth first search method with a CP/MISF preprocessing stage to reduce the time and space requirements), although they make some simplifying assumptions about scheduling being non pre-emptive and communication taking zero time. Ieumwananonthachai et al [43] report on a system which iteratively refines its scheduling heuristics. The restrictions on their system are that the scheduling is static and that some a priori knowledge of task time is necessary.

Two more complex heuristic methods are simulated annealing (described in Fox et al [29, Chapter 13]) and genetic algorithms (described in Goldberg [31]). Both involve a randomising function and iterate towards optimal solutions, the difference being that genetic algorithms maintain sets of solutions. Bultan and Aykanat [15] offer a heuristic for static scheduling based on mean field annealing with the same quality of solution as simulated annealing but with improved performance. However, the computational requirements of both methods mean that they are less suitable for dynamic load balancing than simpler heuristic methods.

Damping is an important technique for limiting thrashing and flooding. For example, in adaptive contracting within a neighbourhood (Horton and Turner [40]) tasks maintain a count of the number of migrations that they have undergone since creation. Tasks are then migrated to the least loaded neighbour if they have travelled less than a certain number of hops, are not migrated at all if they have achieved a maximum number of hops (hence the damping) and, in between, are migrated depending on the relative difference in the loads of their host processor and its neighbours. The bounds on the minimum and maximum migration counts may vary during the program execution, usually inversely with the load.

Another form of damping depends on load thresholds: migration is only allowed between processors with a sufficiently large difference in load. Thresholds may
be varied during the program execution: if a threshold becomes too large, for example, then migration will be held up until the balance has become very poor. Some researchers, e.g. Suen and Wong [102] and Ni et al [78], use simplified or absolute thresholds: they characterise loads as low, normal or high and only permit migration between low and high processors.

### 2.3.3 The execution phase

Once the load has been measured and a decision to rebalance has been made, a mechanism for task migration is required. The mechanism is generally termed **sender-initiated** or **receiver-initiated** (after Xu and Hwang [113] and Eager et al [26]) according to whether it is the overloaded or underloaded processor which takes the active role in task migration. Receiver-initiated balancing is usually preferred because this places the migration overhead on the least busy processor and because the receiver is in the best position to prevent flooding. Receiver-initiated balancing is also more fault tolerant since non-functioning processors will not initiate migration.

The main argument for sender-initiated balancing is that only the sender can detect the arrival of new tasks and new tasks are particularly good candidates for migration. Kremien and Kramer [50] use simple sender-initiated migration when an overloaded processor discovers a new task appropriate for migration—at other times the overloaded processor initiates migration but uses negotiation (described next) to prevent flooding.

Migration mechanisms may be by **simple request** or **negotiation**. Mechanisms of the former type do not allow the receiver processor to reject a new task; mechanisms of the latter type do. Negotiation is more expensive, because the sender must wait for confirmation, but it prevents senders acting upon dated information and flooding receivers. A simple form of negotiation is described in Stankovic [101]. Here, overloaded processors broadcast offers of work to their neighbours, the neighbours return bids reflecting their ability to take on new work and the work goes to the highest bidder.
Bidding is a simple and general negotiation technique but many researchers, including Osser [80] and Suen and Wong [102], have criticised bidding because of the number of messages that must be broadcast or multicast. Apart from the cost of sending multiple messages, most load balancers try to avoid broadcasting and multicasting, especially in architectures that are not fully connected.

Bryant and Finkel [14] simulate a more sophisticated form of negotiation called pairing. Each processor periodically attempts to form a pair with another processor and exchange load estimates. If the difference in load is sufficiently large then tasks are migrated. Bryant and Finkel concluded that the complexity of negotiation was worthwhile because it produced a more stable load.

Ni et al [78] propose a system called drafting in which underloaded processors may make bids to multiple sender processors but without the risk of flooding. Each processor stores the last known loads of its neighbours and bids only to neighbours it suspects of being heavily loaded (this limits the number of bid messages). An up-to-date load figure is returned for each bid, allowing the bidder to place the request-for-work in the most appropriate place. The receiver-initiated nature of the mechanism prevents flooding.

A different approach to the bidding-based methods is used by the stochastic learning automata in Stankovic [101]. Load balancing is sender-initiated and based on load information broadcast at regular intervals. After each migration the receiver returns a reward or penalty according to whether the load is now more or less balanced. Seredyński et al [95] employ a similar method derived from game theory.

2.4 Review of Existing Systems Relevant to Load Balancing

This section reviews a number of research projects in which issues associated with load balancing have been addressed. The reviews are intended to give an
impression of the breadth of the approaches attempted and the extent of success to date.

The Accent system, implemented at Carnegie Mellon University, is described by Rashid [88] and Rashid and Robertson [89]. Accent explores the problems of providing location and failure transparency. Process migration is possible in Accent using the normal communication mechanism, but the uses to which migration is put are not described in the references. Mach combines elements of Accent and Unix and is described by Black [8]. The Mach scheduler uses a task migration policy which can be varied between architectures (or computer sites) but it does not take account of the current load. Mach has been used on a variety of commercial computers.

Artsy and Finkel [1] and Finkel et al [27] describe Charlotte, which was designed at the University of Wisconsin-Madison. The system executes on twenty VAX-11/750 machines and tackles the problems of providing location and failure transparency. The scheduling implements load balancing and makes use of a variety of load information that is periodically emitted by each kernel. The task migration policy is implemented as a utility program so that it can be easily changed, but no specific policies are reported.

Cronus is a commercial system from BBN Labs Inc. designed to execute on a network of heterogeneous computers. As described by Schantz et al [92], an object-oriented programming environment is supported and changing loads are balanced by replicating server objects and/or migrating (some kinds of) objects. Location transparency ensures that migration does not adversely affect the user’s program.

The DEMOS/MP system from the University of California at Berkeley is implemented on a number of different architectures and offers a framework for experimentation with access, location and failure transparency. Powell and Miller [87] describe the migration mechanism but the work reported does not take account of the system load or suggest any reconfiguration strategy.

The University of Washington’s object-oriented Eden system is a portable
(but rather slow) software layer for execution on top of a standard operating system. Again, issues in access, location and fault tolerance are tackled. Also from Washington is the Emerald operating system for homogeneous, LAN-based networks of computers. According to Jul et al [46], Emerald tackles the problems of fine-grained migration (of data as well as of processes) in an object-oriented system by tightly coupling the functions of the compiler and kernel.

Dodd and Ravishankar [23] describe the HMON monitor for the HARTS distributed computer, a hexagonal mesh of shared-memory, multiprocessor nodes developed at the University of Michigan. HMON is a hybrid monitor which gathers information locally, orders it by time and then compresses it before sending it to a central monitor. Although the research does not involve load balancing, it does address the problems of gathering detailed performance information in real time with minimal overhead and minimal interference to the user's program.

Popek et al [85] and Smith [97] report on LOCUS, originally from UCLA but now a commercial venture. LOCUS connects stand-alone Unix systems into a reliable, distributed operating system with access and failure transparency. Remote process creation and process migration are possible, as is cooperation between processes on different machines. Being strongly based on Unix, it is not surprising that LOCUS achieves access transparency by using a modified file system that makes all files globally accessible.

Medusa (Ousterhout et al [81]), from CMU, executes on the Cm* multi-computer and is, because of the low performance processors, a kernel of small size. Despite this, location transparency is offered. Also, from the University of California at Berkeley is Sprite (Douglis and Ousterhout [24]). Sprite is an extension of Unix for diskless workstations on a LAN in which processes can be dynamically migrated to idle workstations transparently to the user.

Barak and Litman [2] and Barak and Shiloh [3] at the Hebrew University of Jerusalem describe a distributed load balancing system for the MOS operating system. Although MOS executes on a network of Unix machines rather than a parallel computer, the research looks in detail at several sophisticated load balancing control policies.
Chapter 2. Previous Work as a Basis for Research

Zitterbart [115] describes **NETMON-II**, a low-intrusion hybrid monitor for Transputer networks. Each Transputer processor executes an instrumented version of the operating system and is monitored by a PC-compatible with filtering software. A global clock is used to time-stamp the monitoring events, which are then transferred to a central site over a dedicated token ring.

Haban and Wybranietz [33] and Haban and Shin [32] describe a hierarchical system for monitoring. The INCAS operating system developed in Germany is based on Motorola microprocessors and a token bus. The replicated kernels are instrumented so that they generate monitoring events. These events are collected by the **Test and Measurement Processors (TMPs)**, at least one of which is connected to each node. The TMPs carry out preliminary processing of the monitoring data before transmitting a summarised version of the results over a global monitoring bus to one of the TMPs designated as master. The quantity of event data can be controlled by the master because it can re-program the slave TMPs.

Bemmerl *et al* [6] and Bemmerl and Bode [4] describe the **TOPSYS** project on performance monitoring tools and in particular the benefits of software versus hardware monitoring. TOPSYS executes on top of MMK, an iPSC/2 hypercube operating system for object-oriented programming in which remote creation of objects and migration of (certain) objects, allows for load balancing. Their hardware has local monitors attached to the processors and a dedicated connection for a limited form of communication between local monitors. The monitors watch the program counter, control flags and address bus of the target CPUs but the researchers note that this kind of monitoring is becoming increasingly difficult as more functionality is put on the same chip—a second phase of the project employs instrumentation of the operating system instead.

**V**, developed at Stanford University and reviewed by Cheriton [18] and Smith [97], is a system for multiple workstations that provides dynamic migration, access and location transparency but no failure transparency. One of the main issues addressed is how the environment of a migrating process may be maintained: **V** does this by supporting virtual memory and transparent file
access between machines and by prohibiting the migration of processes attached to particular devices. V is currently being used in several locations.

The ZM4 project, described by Hofmann et al [39] and Mohr [73] and [74], has a performance monitor attached to each parallel processor, the results being sent over a dedicated Ethernet to a master monitor. The monitoring hardware is powerful in comparison with the parallel processors—the slave monitors are PC-compatibles and the master monitor is a Unix workstation. The time-stamping uses local high-resolution clocks and a global synchronisation line.

Hanxleden and Scott [36] perform a series of rather specific experiments on an iPSC/2 message-passing hypercube. In order to make an accurate comparison of load balancing techniques they consider a single application—a Monte Carlo population simulation similar to that described in Section 8.3.3—and tailor their load monitoring to that application. The experiments performed compare static and dynamic load balancing (of communication and computation) and different kinds of problem decomposition.

The projects reviewed divide fairly neatly into two distinct classes. Those in the first class, e.g. LOCUS, Sprite and V, are concerned with location and failure transparency as motivations for task migration. The Cronus, Emerald and MMK projects also fall into this class, although they allow task migration at a finer granularity. The distributed computer is the typical architecture in this class and the rules which govern load balancing in these systems are usually simple and fixed.

Projects in the second class, e.g. HMON, TMP and ZM4, are more concerned with software and hardware components for performance monitoring. Although none of these projects supports task migration, their results on the issues of intrusion, collection overheads and control of the quantity of monitoring data are distinctly relevant to the load balancing reconnaissance phase.
2.5 Conclusions and a Plan for Research

At the beginning of this chapter concurrent computers were identified as one of the most promising approaches for achieving higher performance at reasonable cost. Subsequent discussion identified scheduling as a major problem with concurrent computers and reported the consensus that speedups of between 30% and 50% are possible if the scheduler can be made load balancing. Then, a whole range of techniques were described for the reconnaissance, decision-making and execution phases of load balancing. Lastly, a number of research projects were reviewed showing that much work had been done on task migration and on performance measurement but that few systems combined results from both areas.

This section starts by summarising the restrictions on the area of research assumed in the remainder of the thesis. A number of outstanding questions, not answered by the literature, are posed and a series of studies proposed to resolve them. Finally, I outline the approaches taken in this thesis to carry out the studies.

2.5.1 Restrictions on the area of research

The restrictions on the research described in this thesis are now reviewed.

1. The thesis is only concerned with multicomputers, i.e. distributed memory, MIMD computers with homogeneous processors and high bandwidth, reliable communications.

2. Only dynamic load balancing techniques are investigated, i.e. techniques where tasks may be reassigned to processors during execution but only on the basis of load information gathered during that execution. Other sources of information (such as those listed in Section 2.3.1) can be very valuable but their utilisation is beyond the scope of this thesis.
3. Hybrid or software monitoring facilities are assumed so that the loads on various system resources can be collected with accuracy but without imposing severe overheads on the users' programs. Without such monitoring, load balancing cannot be responsive enough to changing patterns in the system-wide load. Hardware-only monitoring is rejected for being inflexible and less cost-effective.

4. Only heuristic load balancing methods are considered. Optimal methods are an alternative but require static information (which is sometimes impossible to obtain even in principle), are deemed to be too computationally intensive to be performed dynamically and are more difficult to implement in a distributed fashion.

5. Medium grain, parallel programs are assumed such as those typically written in occam or in standard imperative languages such as C augmented with message-passing functions. Dataflow and parallel, functional languages are deemed to be too fine grain. The time required to measure the load, compute a better mapping and wait for task migration to complete, means that the dynamic behaviour of the program must not vary too quickly, otherwise load balancing will not be of use.

2.5.2 Outstanding questions

Multicomputer scheduling has been identified as an important problem and dynamic load balancing seems a promising approach to solving this problem, but the work reported in the literature falls short of providing a full solution in the following ways. Firstly, there has not been any detailed comparison of the utility of different ways of measuring the load. If effective load balancing is to be carried out then accurate representations of the load must be gathered with least cost. Secondly, although a number of decision-making strategies have been proposed, little detail is given on the extent to which the strategy depends on intrinsic properties of the computer on which it is being executed. This makes it difficult to give an objective comparison of the different strategies. Thirdly, little has
been written about the extra operating system functionality required to support load balancing. For instance, the protocols for ensuring that task migration is transparent are complex but must be efficient if they are not to impose severe overheads. Finally, few researchers have looked at how the complex migration protocols may be formally proven to be safe and transparent.

2.5.3 A series of studies

A series of studies was carried out, in order that this thesis may complement the existing research by addressing the outstanding questions listed above.

- In the first study, a multicomputer with hardware for hybrid monitoring is selected and an operating system with support for load balancing implemented. This has two benefits: the process of implementing the operating system gives deep insights into the extra functionality required for load balancing and thus directly addresses the third question raised above; and a realistic testbed is produced for later experimentation. Implementing an operating system is time-consuming but it is difficult to find existing software with the appropriate functionality and a documented way of accessing and adapting the low-level routines as required. Furthermore, having a full understanding of the operating system proves beneficial, when it comes to creating the specification and designing and interpreting the practical experiments.

- The second study uses formal methods to produce a specification of the parts of the operating system (developed in the first study) involved in task migration. The formal methods are then used to prove the specification safe and correct. This study addresses the fourth question raised above and (for very little additional effort) provides insights into the more general question of the utility of formal methods.

- The third study involves using hybrid monitoring to give an accurate and detailed parameterisation of the multicomputer. A series of experiments are performed to measure the costs of services associated with task migration.
These costs are related back to the specification provided in the second study, so that later results on the cost-effectiveness of different load metrics and decision-making strategies can be interpreted precisely in terms of the intrinsic properties of the machine. This addresses the second question raised above.

- In the fourth study, experiments are performed to measure the cost-effectiveness of some sample load metrics and of a sample load balancing decision strategy. This goes some way towards addressing the first outstanding question raised above, by showing how hybrid monitoring can be used to measure the cost of a load metric as compared to the reduction in the execution time.

The results of these studies can then be interpreted on two levels. On the first level, they provide accurate and detailed information on the costs and the benefits of carrying out load balancing on a particular multicomputer (the Testbed). Furthermore, if the intrinsic parameters of a second multicomputer are known, then because the parameters of the Testbed have been measured and related to a formal model it should be possible to extend the results presented in this thesis to the second multicomputer. On a higher level, the results of these studies can be interpreted as a case study in the use of hybrid monitoring for parameterising multicomputers and their load balancers, and as a case study in the use of formal methods for specifying migration protocols and proving their safety and correctness.

2.5.4 Selecting a methodology

It is not self-evident that experimentation with a particular multicomputer is the best way to obtain results about load balancing. Two obvious alternatives are modelling and simulation. This section lists the advantages of each of the three methodologies and explains why experimentation is deemed the most suitable in the context of this thesis.
Modelling has the whole power of mathematical and analytic techniques behind it. Models may be relatively easy to create and a number of models have already been developed for load balancing: see for example Kurose and Simha [53], Mirchandaney et al [72] and de Souza e Silva and Gerla [20]. However, only simple models are tractable—as Liu and Parthasarathi [59] state, 'Modeling a multiprocessor environment is an extremely complex task. It is much more efficient to design a proper monitor to measure performance'. Wang and Morris [107] have said that 'dynamic algorithms have been found much more difficult to analyze'. Unfortunately, this comment applies to most of the algorithms required for load balancing.

Simulation can deal with more complex systems than modelling. Example applications can be found in Lin and Keller [58] and Xu and Hwang [112] and Benmohammed-Mahieddine and Dew [7]. The problem with simulation, however, is that input data is required to drive it. This data may be generated synthetically from assumptions about the target multicomputer architecture and workload. Or it may be generated from measurements made of a real system. When using the former strategy, the realism of the simulation can be challenged. The latter strategy begs many of the questions addressed in this thesis such as, how should the load be measured?

Experimentation with a real system is undoubtedly more costly in terms of time and effort, but this cost is mitigated by the fact that a real system is also required for some of the other studies listed in Section 2.5.3. The overriding advantage of experimentation is that it is completely realistic and not subject to any simplifying assumptions.

2.5.5 Selecting a multicomputer

A multicomputer is needed with the following properties: the availability of the documentation (and access) necessary to install an experimental, load balancing operating system; the possibility of efficient access to the hardware for interprocessor communication and load monitoring; the provision of a high resolution
clock for performance monitoring; and the possibility of complete control over any background tasks (i.e. the multicomputer must be a single-user machine). Finally, the greater the hardware support for monitoring (and the less the intrusion) the better.

When the present research was begun the Testbed, described in Chapter 4 and then under construction, was deemed to offer the most appropriate environment since it best met the above criteria.

2.5.6 Selecting a formal method

Thomas [104] asserts that roughly 10% of the functions in programs for high-integrity equipment have minor deviations from the specification and that 0.5% of the functions have errors which are serious enough to cause gross, erroneous behaviour of the equipment. This section discusses the advantages of formal methods for verifying the safety and correctness of protocols, such as those used in task migration, and justifies the choice of the formal language Z for use in this thesis.

First, however, some notes on terminology. Verification involves the use of formal or rigorous proofs to show, for instance, that the specification is consistent or that the design is a refinement of the specification. In contrast, validation involves the demonstration that conjectures about the specification hold, typically to show a customer that the specified product will do what s/he wants. Finally, animation is a technique for validation (and not verification): see for instance the Prolog and Miranda animations presented in Diller [22].

Why use a formal method?

This section considers the alternative approaches to giving confidence in the safety and correctness of task migration protocols and justifies the choice of formal methods. According to Thomas [104] there are several different ways in which the correct functioning of computer systems can be ensured.
• Multiple, diverse or redundant, system components can be used so that the responsibility for an action (such as safe and correct task migration) lies with more than one agent. Thus, there is no single point of failure and, provided that a consensus can be achieved, occasional errors by one component are automatically corrected by the others.

• Fault tolerant protocols can be used to detect when an error or failure has occurred and to correct the situation. For instance, if a processor $p_1$ has sent a task to $p_2$ but received no acknowledgement (within a suitable time limit) then the migration may be assumed to have failed and the task is retransmitted.

• Instead of relying on the automatic correction of faults, the computer system may be subjected to a demanding test phase and released with the assurance that ‘it never goes wrong in normal circumstances’.

• Finally, formal methods can be used to give formal, rigorous, or machine-checked proofs that the computer system will always behave as specified.

While diversity and redundancy are valuable techniques, they are really more appropriate for hardware components than software. Most hardware components wear with age and become increasingly unreliable. Software does not ‘go out of adjustment’ in this way. Thus, the diversity and redundancy approach is rejected in this thesis.

Fault tolerance is also a valuable technique, but it will impose extra time and space overheads on task migration. If one in a million migrations fail then the system has suffered the cost of almost a million unnecessary fault tests. An additional problem is that task migration is complicated enough without the extra layers of protocol needed to provide recovery in all the possible failure scenarios.

Testing is an attractive option, provided that it is exhaustive. But testing is time-consuming and cannot be truly exhaustive for most non-trivial programs
Chapter 2. Previous Work as a Basis for Research

... (and especially for most non-deterministic programs). Furthermore, while testing may show the presence of bugs it cannot directly show their absence.

Hayes [38] gives a good summary of the advantages of formal techniques: a specification assists prototyping because it allows new ideas to be tried out quickly (at a higher level of abstraction than changing the code); the same specification forms the basis for the implementation and documentation and testing. Even if the specification phase is performed after the implementation is complete, then: it highlights omissions or mistakes in the design; it facilitates modifications including porting to another architecture or programming language; and for large development groups it aids communication.

The problems with formal methods are discussed in Chapter 9, where conclusions are drawn about the costs and benefits of specifying the Testbed in Z and the utility of formal methods in general. However, as none of these problems are as severe as the problems with the other three approaches for giving confidence in task migration, it is the formal methods approach which is adopted in this thesis.

Why use Z?

There are many different formal methods to choose from. Milnes [69] and Woodcock and Loomes [110] distinguish model-oriented languages such as Z, the Vienna Development Method (VDM) and the Hierarchical Development Method (HDM); algebraic systems such as CLEAR and Larch; process algebras such as the Calculus of Communicating Systems (CCS) and its later development into the Synchronous Calculus of Concurrent Systems (SCCS) and LOTOS; and the modal logics which use temporal operators such as □ (always) and ◊ (eventually).

The Z language was chosen for the specification and proofs in this thesis, because it had the following, desirable properties.

- An extensive library of basic definitions is already part of the Z language.

These permit shorter specifications which can be more easily assimilated by
other people who may use the specification either to predict the behaviour of the implementation or as the ultimate arbiter of its correct behaviour.

- The ability to easily typeset $Z$ specifications and the availability of indexing and cross-reference tools which allow the relationships between parts of the specification to be tracked mechanically.

- The availability of a checker for $Z$ specifications which enforces the syntax, scope and type rules of the language and promotes consistency across the specification. Additionally, the checker is integrated (fairly well) with the typesetting tools.

- The possibility of carrying out formal or rigorous proofs. Rigorous proofs are used when fully formal techniques are too time-consuming. They involve a formal statement of what is to be proved followed by an informal justification. (As there is no automatic theorem prover for $Z$, machine-checking of proofs is not possible.)

- The presence at the University of Edinburgh of local expertise in creating specifications and proving implementability, data refinement and operation decomposition results.

- The maturity of the language: $Z$ has been used in many projects around the world and there are many articles, case studies, books and taught courses on $Z$. 
Chapter 3

Introducing the Testbed

This chapter introduces the Testbed—the multicomputer on which the practical experimentation for this thesis was performed. The chapter is in two parts: a description of the conventional aspects of the Testbed is followed by a more in-depth description of the innovative features such as task migration and hybrid event monitoring.

The Testbed hardware (also described in Imre [44]) was designed and built before the research described in this thesis began. All the software, including the operating system, the user interface, the programming environment and the support for load balancing, was then written by me as part of the research.

3.1 Conventional Features

The Testbed is an experimental, distributed memory (message-passing) multicomputer constructed at the University of Edinburgh between 1988 and 1991. A free-standing cabinet about five feet high contains a power supply, cooling fans, six processor boards, a bus-based processor interconnect from the node architecture of Centrenet (a detailed description of which can be found in Ibbett et al [42]) and special hardware for hybrid event monitoring. A single-user machine, the Testbed has one RS-232 link to a terminal for operator control and a second
Chapter 3. Introducing the Testbed

Figure 3-1: An overview of the Testbed hardware.

serial link to the local area network and hence access to a Unix-hosted filestore. Figure 3–1 gives an overview of the Testbed hardware.

The Testbed operating system (TOS) is written in C and provides a time-sliced, multi-tasking environment. TOS has a built-in shell which offers a typical Unix interface to the user. TOS is replicated over all processor boards and the console may be switched (in software) to communicate with any of the six shells. The processor board with the serial link to the LAN is designated as the master board and executes the master shell—the other Testbed processor boards execute the slave shells. Tasks may be invoked on any operating system and migrated between operating systems transparently to the user. Figure 3–2 gives an overview of the Testbed system software and its relationship to a Unix host on the LAN.

Some small utility programs have been implemented on the Testbed offering similar features to the Unix programs more, grep, compress and wc. A disassembler and a version of the editor ue have also been ported. However, as no compiler has been implemented, all new programs must be cross-compiled on another machine and then up-loaded to the Testbed via the LAN connection.
A typical experimental session might be as follows: edit and compile the test program on a Unix workstation; up-load the binary to the Testbed; execute the program on the Testbed collecting the results in a file; down-load the file to the workstation; and analyse and display the results with, for example, perl and gnuplot.

3.1.1 The parallel processor boards

Each processor board has a Motorola 68010 processor, several megabytes of RAM, support for virtual memory, a Centrenet controller (including a Direct Memory Access capability) and a connection to the monitoring bus. The board designated as 'master' also has drivers for two serial links and reads event data from the monitoring bus—the other, 'slave' boards may only write to the bus. Section 3.2.4 describes the monitoring bus and its interfaces in more detail.

The number of processors and their 1.2 MIPS performance put the Testbed in the MIMD, medium granularity class, somewhere between the Cray-XMP approach, which uses a small number of very high performance processors, and the DAP approach, which uses a large number of simple processors. The
six processors are a compromise that enables the problems with concurrency described in Section 2.2 to be observed and also allows the use of a simple but flexible bus-based interconnection strategy for Centrenet and the monitoring bus.

The Testbed is designed as a message-passing architecture rather than a shared memory architecture for several reasons. The occam model of programming had already been selected (for reasons explained below in Section 3.1.3) and it explicitly communicates sequences of bytes over channels and only shares variables if they are read but not written. The interconnection network required for message passing is generally simpler to implement (and more scalable) than that required for shared memory since global memory updates are avoided.

The Centrenet interconnection network has a hierarchical design: nodes, which comprise up to sixteen processors sharing a bus, are connected by fibre optic cable into a tree. Communication time is minimal if the source and destination processors are part of the same node; otherwise the communication time is proportional to the number of nodes traversed. Since the Testbed has just six processors, its communication network requires only a single Centrenet node. The communication speed is approximately 10Mbytes/second. The Testbed architecture is scalable (with a reasonable performance penalty as more nodes are added to the tree) but in this thesis all experiments use the simplifying assumption that Testbed inter-processor communication requires at most one hop.

3.1.2 The Testbed operating system

The description of TOS support for load balancing is deferred until Section 3.2. Here, TOS and its built-in shell are considered in terms of the more conventional operating system features supported.

Each replication of TOS on each processor may host up to sixteen processes at any one time. In this context, a ‘process’ is created every time a new program is invoked by the user. Each process has a code and a data segment of up to half a megabyte. Strict rules prevent one process from modifying the data of another.
process, although multiple invocations of the same program may share a code segment and trusted programs (such as debuggers) may have read-only access to other processes' memory. Processes may be marked as 'foreground', 'background' or 'suspended' (for debugging purposes). The command `ps` lists the processes, their 'threads' (described next) and optionally the thread contexts, `kill` may be used to remove processes and `fg` and `bg` move processes between the foreground and background.

In TOS terminology, it is not processes which execute but 'threads'. Each thread has its own program counter and stack but shares code and global data with the other threads in the same process. Each process may own up to sixty-four threads, the threads being multi-tasked on an equal priority, round-robin basis. Threads have a maximum time-slice of 20ms, although they may be pre-empted if they call certain operating system services. TOS has been made preemptive so that interactive programs will operate correctly, but the time-slice has been set relatively high in order to reduce the number of context switches. This is necessary because the context switch takes a long time, as much as fifty times as long as a Transputer context switch. Threads communicate over occam-style channels (as described in Section 3.1.3) and each process may own up to 128 channels.

The Testbed does not have backing store on which to keep parts of the virtual memory that have been 'paged out' so Testbed programs must be conservative in their use of memory. Practical experience, however, shows that the available RAM is almost always sufficient.

Each replication of TOS maintains its own filestore in local memory. The filestore is effectively a one-level directory and holds program files, scripts, data files and configuration files. Up to sixty-four files are allowed, a maximum size of half a megabyte per file being imposed. Files may be copied between processor boards by the user (versions of the Unix commands `ls`, `cp`, `rm` and `mv` are available) and executables are automatically mounted as needed. File permissions may be set with `chmod` to specify execute, read or write.

Other assorted features include a real-time clock, simple script interpretation,
a form of environment variables, redirection of *stdout* and some terminal control via *stty*.

### 3.1.3 Programming environment

Programs to be executed on the Testbed are written in C, compiled and linked with Testbed-specific libraries. Most of these libraries are implementations of the standard C library functions described in Kernighan and Ritchie [49, Appendix B] and the rest are new extensions to the C programming language to allow control of multiple threads and channel communication. Here is a summary of the standard functions available on the Testbed.

- The *stdio* functions for opening, flushing, closing, seeking and unlinking files, formatted printing (*fprintf*), character reading and writing, block reading and writing.

- The *string* functions for copying, concatenating, comparing and searching strings and memory block copying.

- Some of the *stdlib* functions for string-to-integer conversion, memory allocation, *exit* and environment variable search operations.

- Some of the *time* functions for reading the real time clock.

The *ctype*, *assert*, *stdarg*, *setjmp*, *signal*, *limits*, *float* and *math* functions have not been implemented.

### The thread model

The first area in which new extensions to the C programming language have been made is that of thread control. Thread control is based on the simple yet powerful process model of *occam*, as embodied by the *PAR* statement: a parent spawns a number of children and is blocked until the children complete. Details of the *occam* language can be found in Pountain [86] and INMOS [57] and a
discussion of the unique benefits of occam in Welch [108]. Thread control is an operating system function, accessed by means of the following library routines:

```c
int create(id, n_pages, processor, entry, stack) A child thread with ID number derived from id is created; the child is allocated n_pages of stack space; its initial program counter and stack pointer are loaded from entry and stack; the child is queued for execution on the processor board specified by processor or chosen randomly if processor has a negative value. The value returned by create is zero if the child cannot be created, otherwise the ID number of the child.
```

wait Once the parent has called create for each new child, it calls the wait function and is suspended until its children have terminated.

exit(err) Threads terminate by calling the exit routine and passing an termination code. Two of these codes are reserved for the occam HALT and STOP conditions.

occam is a static language in terms of process creation and channel communication. In occam the identity of all processes and channels can be known at compile time. This has the advantage for architectures without large virtual address spaces (such as the Transputer) that all memory requirements are known before the program is executed and memory can be allocated statically. Fully dynamic process creation is possible on the Testbed but passing the thread id value is more in keeping with the occam philosophy and simplifies the use of debugging programs, such as that designed by Woods [111].

The communication model

The second area in which new extensions to the C programming language have been made is that of inter-thread communication. The model of communication is based on occam: a pair of threads wishing to communicate reserve (for the entire program execution) a unique, unidirectional channel. One thread performs
send operations on the channel, the other receive operations. Both threads are blocked while communication completes.

Communication on the Testbed is implemented in the operating system and accessed through the following library routines:

\[
\text{send\_block}(\text{chan}, \text{buffer}, \text{length}) \quad \text{A thread requests to send a message of length bytes beginning at the address given by buffer on channel chan.}
\]

\[
\text{mt receive}(\text{chan}, \text{buffer}, \text{length}) \quad \text{A thread requests to receive up to length bytes of data beginning at the address given by buffer on channel chan. The return value specifies the actual number of bytes received.}
\]

### 3.2 Support for Load Balancing

In addition to the functionality of conventional parallel computers, the Testbed offers task migration and load balancing. These advanced features require extensions to all parts of the Testbed: special operating system data structures for representing tasks; protocols for migrating tasks and their resources; instrumentation of the software; and dedicated hardware to enable the collection of events. The extensions made in each of these areas are described below.

#### 3.2.1 Special data structures

The unit of migration on the Testbed is the thread. To make the migration of a thread \( t \) from source processor \( sp \) to destination processor \( dp \) as efficient as possible it must be easy to 'disconnect' the data structure that represents \( t \) from its environment at \( sp \), pack \( t \) into a message and transmit the message to \( dp \). At \( dp \), the reverse process of unpacking and reconnecting \( t \) must also be made simple. Migration is complicated by the fact that threads use local resources, such as communication channels and pages of memory, and for transparent thread migration these must be moved or copied in a consistent way.
Experience with the Testbed shows that in addition to the usual considerations when designing data structures—i.e. minimal size, simplicity for ease of implementation and maintenance, and efficiency of access—the data structures used in TOS to represent threads and their resources must have the following properties.

1. All the relevant data structures must be kept near each other, both to assist allocation and release, and in order that they may be located speedily during migration.

2. The number of dependencies or links between data structures must be minimised to speed and simplify ‘disconnection’ and ‘reconnection’.

In TOS, a single record called the Thread Control Block (TCB) is used to represent a thread. Unused TCBs are stored on a free list and allocated when a thread is created or arrives during migration from another processor. TCBs are destroyed when a thread terminates or when it is migrated away. When a thread is to be migrated, the information to be transmitted is localised in three areas: the TCB which holds register context and other control values; the Channel Control Blocks (CCBs) which store the status of channels used by the thread; and the virtual memory page table which stores information on memory pages owned or shared by the thread. The ‘disconnection’ of a thread simply requires that it be unlinked from the process ready queue.

An example migration is illustrated in Figure 3-3. On one of the Testbed processors two processes are executing. Each process has a virtual memory page table, an array of CCBs and a doubly-linked list of ready threads. In addition, a message containing a migrating thread awaits transmission. This message contains the TCB and copies of the relevant CCBs and memory page table entries.

### 3.2.2 Migration protocols

The migration protocols define the rules for moving and copying threads, memory pages and channel information between processors. These rules are needed to
Figure 3–3: A processor with two executing processes and a migrating thread waiting for transmission.
ensure migration transparency and to prevent, for example, the creation of multiple copies of a thread occurring, updates to the same memory page at different sites happening or the loss of messages on channels used by a migrating thread.

The migration protocols are complicated, principally because they have to deal with concurrent interactions between multiple processors, and their design was greatly assisted by the development of the formal specification described in the Chapters 4 and 5. In fact, several major alterations to the initial implementation were made when the specification showed that problems might arise in certain unusual sets of circumstances.

Any thread may be migrated as many times as desired, it is possible for several migrations to occur at the same time and no destination processor may refuse to accept a migrating thread. However, a source processor can refuse to send a thread if the thread is in the wrong state. Threads may be in only one state at a time and examples of possible states are: in the ready queue, waiting to communicate, waiting for a page of virtual memory or, indeed, in the act of being migrated. The TOS protocol states that only threads currently in the ready queue may be migrated. Without this restriction, different 'disconnection' (and 'reconnection') procedures would be needed for threads in each state. The motivation for distinguishing the ready state is that ready threads are the most likely to consume valuable system resources in the near future, and hence are likely to be good candidates for migration.

The protocol for moving and copying pages of memory between processors is too complicated to state in full here, although the following demands are placed upon the protocol. For efficiency reasons, pages from the read-only code segment of a process may be freely copied around the system whenever a migrating thread requires them. Following the semantics of occam, if a parent thread \( pt \) creates a child thread \( ct \) then \( ct \) may read (but not modify) variables in \( pt\)'s stack area—such pages are copied if necessary. Pages holding the stack area of a thread that has just migrated may be copied if the thread uses them again. Finally,
the memory page protocol also has to know when to flush out-of-date copies of pages.

The third protocol, for communication, is the most complicated of all. For efficiency reasons, CCBs are distributed and changes due to thread migration are not made globally—much care was taken in protocol design to ensure that all copies of a process's CCBs stay in a consistent state. Without presenting the complete protocol here, the following remarks are made. Suppose that threads $t_1$ and $t_2$ communicate over channel $c$. When they are on the same processor then a single CCB is used to represent $c$. When they are on different processors then two CCBs are needed. If $t_1$ migrates away from $t_2$ then information must be extracted from the shared CCB and used to create a new CCB. If $t_1$ migrates onto the same processor as $t_2$ then the information from two CCBs needs to be combined in a shared CCB.

### 3.2.3 Software instrumentation

The Testbed uses dedicated hardware to implement hybrid monitoring and to provide a global clock for time-stamping monitoring events. Hardware assistance helps limit intrusion by reducing the overheads of collecting load data.

There are several options for where to apply software instrumentation. The simplest method is to add instructions to the user's program—either manually or using a modified compiler—or to generate an instrumented version of the shared library routines. The Testbed, however, is instrumented by adding instructions to the operating system because the events that are needed for the profiling in Chapter 7 and for the load balancing in Chapter 8 involve information which is known only to the operating system. Furthermore, care is taken to use only information that is readily available in the processor registers or local stack frame and to avoid expensive computation.

Less than 1.5% (approximately 150 lines of code) of the 11,000 lines of code in TOS are concerned with generating events. This is comparable with the 1% slow-down in the TOPSYS project (described on page 27) although it is rather
more that the 0.1% claimed for the TMP monitor (page 27). Detailed results about the exact number of events produced can be found in Section 7.3.7.

The extra C instructions are of two forms:

```c
if (dgl & DIAG) { ... }
```

and

```c
send_event(n);
```

The first construct is used to test whether various classes of events are to be emitted: the 'diagnostic level' variable `dgl` is set by the user, `&` is the bit-wise AND operator and the `DIAG` value is a string of bits corresponding to one or more classes of events. The second construct causes a 16-bit event to be written to the monitoring hardware. The first construct requires three machine instructions and the second construct requires just one.

The events are grouped into classes which can be individually enabled— with all five processors generating all classes of event the master processor becomes overwhelmed with event data which it does not have sufficient time to process. The different classes have been chosen to support different types of experiment. There are event classes associated with load balancing, including events for changes in ready queue length, threads being scheduled and descheduled and threads communicating; and event classes not associated with load balancing including events for profiling TOS performance and debugging system or user code.

New events can be added to TOS as desired by the experimenter. Firstly, the section of TOS code that is associated with the event is identified. Secondly, an existing or new class is selected for the event so that the event can be conveniently enabled or disabled. Thirdly, a representation for the event as a sequence of 16-bit numbers is determined so that the event and its associated information (such as thread or channel IDs) can be reconstructed by the event collection software. It is particularly important to ensure that the new event cannot be confused with existing events in the same, or simultaneously enabled, classes. The convention currently used for one of the larger classes is as follows.
1. All 16-bit numbers of the form $0xaann$ indicate that the monitoring hardware’s global clock has overflowed ($0x$ indicates that the following number is in hexadecimal, $n$ represents any hexadecimal digit).

2. All 16-bit numbers, other than those specified above, with bit 14 set are events indicating that a thread (with ID specified in bits 13 to 0) is being scheduled.

3. All 16-bit numbers, other than those specified above, use bit 15 to indicate whether they are followed by another number pertaining to the same event or not. This allows complex events to be signalled by means of a variable length sequence.

Another class of events uses the simpler convention that bit 15 set indicates a global clock overflow, otherwise bits 13-0 specify a thread ID and bit 14 specifies whether the thread is being added to the ready queue or removed from the ready queue. Examples of event sequences from different classes are presented in Tables 3-1 and 3-2.

<table>
<thead>
<tr>
<th>Event Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[0xaann]$</td>
<td>Global clock overflow</td>
</tr>
<tr>
<td>$[TID+0x4000]$</td>
<td>Thread (with ID TID) is being scheduled</td>
</tr>
<tr>
<td>$[CREATE+0x8000]$</td>
<td>A child thread (CTID) is created</td>
</tr>
<tr>
<td>$[TRAP70]$</td>
<td>A thread is being blocked on communication</td>
</tr>
</tbody>
</table>

Table 3-1: An example showing some variable length events used for profiling the performance of the Testbed.

<table>
<thead>
<tr>
<th>Event Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[TID+0x4000]$</td>
<td>Thread joins ready queue</td>
</tr>
<tr>
<td>$[TID]$</td>
<td>Thread is removed from ready queue</td>
</tr>
</tbody>
</table>

Table 3-2: An example showing some fixed length events used when measuring the size of the ready queue.
3.2.4 Hardware for event collection

The Testbed's monitoring hardware comprises five Slave Monitoring Interfaces (SMIs) and a Master Monitoring Interface (MMI). As shown in Figure 3–4 each of the five slave processors can write sequences of 16-bit event data to an associated SMI, addressing it as a memory mapped device. The SMIs add a 16-bit time-stamp from the global clock to the 16-bit numbers as they arrive and then store the time and data pairs in a 32-bit wide, 1024-entry queue.

![Diagram of the Testbed monitoring hardware](image)

**Figure 3–4:** An overview of the Testbed monitoring hardware showing the master monitoring interface and one of the slave monitoring interfaces.

The MMI polls each SMI in turn over the 16-bit wide monitoring bus and collects (unless the SMI is empty) first the time-stamp and then the event data. The time-stamps and data are stored by the MMI in a 20-bit wide, 512-entry queue. Each entry has four bits specifying the source SMI.

The master processor may poll the MMI (treating it as a memory mapped device) at any time, requesting data from its queue. This data is supplied as a 4-bit processor board number and 16-bits of time or event data. Software on
the master processor performs the task of re-uniting time-stamps and event data from the same SMI and outputs a time-ordered sequence of events.

The purpose of the MMI and SMI buffers is to cope with bursts of high event generation rates. If the burst lasts too long and the buffers fill then events must be discarded or the computation stopped.

The SMI boards have been designed with room for additional filtering logic to reduce event rates. The intention is that the logic will be programmed from the master processor and that the type of filtering applied can be changed during program execution as appropriate. This kind of filtering has already been done on the TMP and is described by Haban and Wybranietz [33] and Haban and Shin [32]. Different kinds of filtering are suggested in Mansouri-Samani and Sloman [63]. However, this additional functionality has not yet been implemented on the Testbed.

Load balancing software

Software for the reconnaissance and decision-making phases of load balancing is discussed in Chapter 8. It is worth noting at this point, however, that the design of the monitoring hardware requires load measurements to be centralised at the master processor board. For all practical experiments described in this thesis it has, in fact, proved convenient to use the master board exclusively for reconnaissance and decision-making and to ensure that the test programs execute only on the slave processors. While this decreases the Testbed's parallelism from six to five, it ensures that intrusion caused by the load balancer has minimal effect on the test programs.
Chapter 4

Specification of Testbed Basics

Z is a formal specification language developed by Oxford University’s Programming Research Group. Z is based on first-order logic and set theory and this makes it possible to express mathematical proofs in the language. Objects at the disposal of the Z specifier include sets, relations, functions, sequences and bags. Three good introductory articles on specification are: Meyer [68] in which a charming illustration is given of why formal methods are preferred to natural language for specification; Hall [35] in which some common myths about specifications are exploded; and Wing [109] in which the range of available formal methods is surveyed.

The next section introduces the parts of the Z language which I have employed in this thesis—as will be discussed in the concluding Chapter 9, it has proved beneficial to use only a limited range of Z features. The main body of this chapter is concerned with the specification of the task control and channel communication modules of the Testbed’s operating system (TOS) (the following chapter contains the specification of the migration protocols). The final section here explains in detail how the Z specification schemas are related to C functions in the implementation.

The specification was carried out with three aims in mind. Firstly, it should explain (at a suitable level of abstraction) the inner workings of the operating system. Secondly, it should form the basis for the proofs of correctness. Thirdly, it should help deal with the enormous complexity of adding task migration to
a multicomputer. Specification is often used for refinement purposes but this, unfortunately, is outside the scope of the thesis.

The specification does not attempt to model all parts of the operating system since this would require a considerable amount of time to complete. Instead, it concentrates on the functions related to thread synchronisation, channel communication and thread migration because, of all the functions involved in load balancing, these are the most difficult to implement correctly.

4.1 A Short Introduction to the Z Language

In this section I introduce enough Z notation and semantics for readers unfamiliar with the language but with some mathematical background to gain an understanding of the formal specification in Chapters 4 and 5. There are several papers and books with tutorials on Z, two good examples being Diller [22] and Spivey [98]. A detailed description of the Z language can be found in the reference manual by Spivey [100]. The specification style that I have used is fairly restricted, so it is by no means necessary to understand the whole of Z in order to understand this thesis.

4.1.1 Schemas

A Z specification is primarily a collection of schemas. Each schema is a grouping of declarations and predicates chosen to represent some part or aspect of the system being modelled. Declarations introduce new variables and assign types—variables must be declared before use and declarations are global. Predicates express relationships between variables. Schemas are generally used either to express the state associated with the system being modelled or operations on that state. A typical state schema contains a list of variables and some predicates expressing constraints or invariants on the state. A typical operation schema includes a state schema and, using a method for distinguishing the 'before' and 'after' state variables, the predicates show how the operation updates the state.
Consider the two example schemas $DBState$ and $AddMike$ which might be part of a specification modelling a database of company employees and their phone numbers.

$DBState$

| employees \( : \mathbb{P} \text{PERSON} \) |
|---|---|
| extensions \( : \text{PERSON} \leftrightarrow \mathbb{N} \) |
| $\text{dom extensions} \subseteq \text{employees}$ |

$AddMike$

$\Delta DBState$

| employees' \( = \text{employees} \cup \{\text{Mike}\} \) |
| extensions' \( = \text{extensions} \cup \{\text{Mike} \mapsto 1234\} \) |

The schema boxes group the declarations (above the horizontal line) and the predicates (below the horizontal line). The first schema specifies how the state of the database is represented: the $employees$ variable is a set of persons; the $extensions$ variable is a set of ordered pairs where each pair contains a person and a number. The second schema specifies the state-changing operation of adding a new employee Mike with extension number 1234 to the database. The $\Delta DBState$ schema declares the variables of $DBState$, $employees$ and $extensions$ and (because of the ‘$\Delta$’) also the primed (or ‘decorated’) variables $employees'$ and $extensions'$. These variables obey the relevant predicate. It is the convention that an undecorated variable represents a component of the state before the operation and a decorated variable represents a component of the state after the operation.

It is also the convention that an operation schema should define the value of every decorated state component declared. For reasons of clarity and parsimony the specification in this thesis deviates from this convention and readers should assume that where the value of a decorated state component is not defined it
 retains the value of the corresponding undecorated state component. A formal
treatment of this issue can be found in Pitt and Byers [84].

Multiple predicates in a schema are implicitly conjoined. Identifiers in the
predicates must be declared in the upper part of the schema or must be declared
as global variables. When a schema name $S_2$ appears as a declaration in another
schema $S_1$, the declarations of $S_2$ are merged with those of $S_1$ and the predicates
of $S_2$ are conjoined with the predicates of $S_1$. When a schema name $S_1$ appears
as part of a predicate $P$, the predicate part of $S_1$ may be substituted into $P$.

The AddPerson schema illustrates another feature of schemas: input and
output arguments. The $p?$ and $e?$ variables are not part of the state—they
are input variables to the operation. Output variables defined by the operation
are also possible—identifiers have an exclamation point appended.

AddPerson

$\Delta DBState$

$p?: PERSON$

$e?: N$

$employees' = employees \cup \{p?\}$

$extensions' = extensions \cup \{p? \mapsto e?\}$

4.1.2 Sets

Here is a summary of the Z notation for various sets and operations on sets:

Enumeration: The ::= notation is used when specifying a set by enumeration,
e.g. $INSTRUCTION ::= send | receive | stop$. (When a set is to be used as
a new type, it is usually given an identifier in upper case.)

Comprehension: Set comprehension uses the $\{signature | predicate \bullet term\}$
notation, e.g. $\{n : N | n \neq 0 \land n \mod 2 = 0 \bullet n\}$ specifies the set of positive,
even numbers as does $\{n : N | n \neq 0 \bullet 2 \ast n\}$. 
Equality: The $=$ symbol is used to express equality as in $\{1, 2, 3\} = \{3, 2, 1\}$.

Empty set: The symbol $\emptyset$ is the preferred way to denote the empty set.

Power set: The declaration $x : \mathbb{N}$ declares $x$ to be a natural number but the declaration $X : \mathbb{PN}$ declares $X$ to be a set of natural numbers.

Operators: Set difference is denoted by $\setminus$, domain anti-restriction by $\triangleleft$, e.g. $\{a\} \triangleleft \{a \mapsto 1, a \mapsto 2, b \mapsto 2, b \mapsto 3\} = \{b \mapsto 2, b \mapsto 3\}$, and function overriding by $\oplus$.

Domains and ranges: The $\text{dom}$ operation takes a relation and delivers the domain, ran delivers the range.

Sequences: The sequence is just a particular kind of function. There are some predefined operations on sequences: $\text{head}$ yields the first element in a sequence, $\text{tail}$ yields the sequence with the first element removed, $\text{last}$ yields the last element in a sequence and $\sim$ joins sequences. When a sequence is listed it should appear between sequence brackets ($\langle$).

4.1.3 Relations and functions

Relations and functions are a convenient way of associating different bits of data with each other:

Relations: If $R$ is a relation between sets $X$ and $Y$ then $R$ is a subset of the Cartesian product $X \times Y$ and this is denoted in $\mathbb{Z}$ by $R : X \leftrightarrow Y$. If, for example, the ordered pair $(x, y)$ is a member of $R$ then the $\mathbb{Z}$ notation $x \mapsto y \in R$ can be used.

Total functions: If $F$ is a total function from $X$ to $Y$ then $F : X \rightarrow Y$ is written meaning every member of $X$ maps to (exactly) one member of $Y$.

Partial functions: If $F$ is a partial function from $X$ to $Y$ $F : X \rightarrow Y$ is written to mean that a member of $X$ either does not map at all or maps to exactly one member of $Y$. 
4.1.4 Miscellaneous Z notation

Abbreviated definition: The == symbol is used to define the left hand side as an abbreviation for the right hand side, e.g. small_evens == {2,4,6,8}.

Defining types: Z has a few built-in types (N, the set of natural numbers, is the only one of these used in the specification), the ability to define new types based on existing types, by using × or → for example, and the ability to define types without saying exactly what they are. For example, the notation [PERSON, PHONE] defines two 'given types' representing, presumably, people and telephones.

Defining operators: Z contains many operators for manipulating relations, sequences, and other kinds of sets but it also has the ability to define new operators using 'axiomatic definitions', e.g.

\[
\text{square} : N \rightarrow N \quad \forall n : N \cdot \text{square}(n) = n \times n
\]

Quantification: An example of existential quantification is \( \exists x : N \mid x \leq 5 \bullet x = x \times x \) meaning that there exists a natural number \( x \) (which has the property of being less than 5) such that \( x \) is its own square. The \( \exists_1 \) notation is used where there is exactly one object which satisfies the property. An example of universal quantification is \( \forall x : N \bullet x \geq x \).

Tuples: An example pair could be written \( (1, x \mapsto y) \) and the first member of the tuple can be extracted with \( \text{first}(1, x \mapsto y) = 1 \).

4.1.5 Checking the specification

The Z specification presented in this thesis has been checked in six different ways.

1. The fuzz software package by Spivey [99] has been used to check that the specification complies with the Z rules for syntax, typing and scoping.
2. Simple indexing tools were used to help with locating declarations and dependencies.

3. Informal walk-throughs of the specification and the accompanying documentation helped to identify areas in which the model being proposed was inconsistent.

4. Informal comparison of the specification with the implementation code was used to justify the claim that the specification models the implementation.

5. The formulation and proof of statements about the specification enhanced confidence.

6. Executing the implementation (real-world testing) gave some confidence that gross errors had been eliminated, although the effort required to test under all possible conditions was prohibitively expensive.

It is worth noting that the use of fuzz, which requires declaration-before-use, in conjunction with the lack of modularity in Z (when compared, for instance, to VDM as in Hayes [37]) strongly influences the order of presentation in this chapter and produces a distinctly bottom-up style.

4.2 Thread Design

I begin this section by introducing the basic objects in the specification such as threads, processors, counters and the message types exchanged between the processors. I define that part of the Testbed state to do with thread control in terms of the relationships between parent and child threads, threads and counters, processors and threads and so on. The state changing schemas are introduced, first at a lower level showing the mechanisms for the task control and then at a higher level showing in what circumstances each state change may occur.
Appendix B contains an index of all globally declared schemas, relations and types used in the specification.

### 4.2.1 Basic objects

I declare the existence of two given sets, $\text{THREAD}$ and $\text{PE}$, which the type-checker interprets as user-defined types and the reader should interpret as the set of all threads that execute on the Testbed and the set of Testbed processing elements (processors).

$$[\text{THREAD, PE}]$$

Later on in the specification I will want to be able to use the idea of 'no thread' and to do this I distinguish a member of the set $\text{THREAD}$ by naming it $\text{null\_thread}$.

$$\text{null\_thread} : \text{THREAD}$$

I now model TOS 'counters' which are used to implement thread synchronisation. Parent threads have a counter associated with them which is incremented every time they create a child and decremented every time one of their children terminates. The parent can request to be suspended on the completion of its children, i.e. it can ask to be blocked until its counter has reached zero. The following piece of notation declares $\text{COUNTER}$ to be an abbreviation for the set of pairs of natural numbers and threads.

$$\text{COUNTER} == \mathbb{N} \times \text{THREAD}$$

The natural number part of variables of the $\text{COUNTER}$ type will be used to represent the number of children not yet terminated and the $\text{THREAD}$ part will be used to represent the parent thread blocked on the counter—where no parent is blocked the value $\text{null\_thread}$ will be used instead.

The Testbed processors synchronise their actions by communicating various kinds of control message over Centrenet. The type $\text{CNETMSG}$ is defined for these messages.
Chapter 4. Specification of Testbed Basics

\[
CNETMSG ::= focusm(\langle PE \times CHANNEL \times PE \rangle) \\
| advert(\langle PE \times CHANNEL \times PE \rangle) \\
| rtr(\langle PE \times CHANNEL \times PE \rangle) \\
| msg(\langle PE \times CHANNEL \times PE \times MEM\_BLOCK \rangle) \\
| term(\langle PE \times THREAD \rangle) \\
| thrd(\langle PE \times THREAD \times (P(CHANNEL \times PE)) \times (P(CHANNEL \times PE \times \mathbb{N})) \rangle)
\]

focusm The 'focus' message is the first of four kinds of message used during channel communication. When a thread requests to send and finds that the receiver thread is known to be on another processor it (the sender) sends a focus message to tell the receiver that it is waiting. The tuple components specify the source and destination processors and the channel involved.

advert The 'advert' message is used only during the first communication on a channel. When a thread requests to send and has no information about where the receiver is located it (the sender) sends an advertisement to each of the processors to tell them of its existence. The tuple components are interpreted in the same way as those for the focus message type.

rtr The 'ready to receive' message is returned by receiver threads when they receive a focus or advert message. Again, the tuple components are interpreted in the same way as those for the focus message type.

msg The 'message' message contains the actual data to be transferred during a remote channel communication. In addition to the expected tuple components, the value of type \( MEM\_BLOCK \) represents the actual data being transferred between the sender and receiver threads.

term The 'termination' message is not used in channel communication but in task synchronisation. If a parent thread creates children which subsequently migrate to a remote processor then when each child terminates it sends a message over Centrenet to the parent's processor to inform the parent. The tuple components specify the parent's processor and the parent thread.
The ‘thread’ message is used during thread migration. It contains various register values and control information.

The specification of the Testbed is primarily concerned with modelling the state and operations of TOS. In order to assist with the proofs, however, the specification defines a trace sequence which is built up as each state-changing operation is applied. The trace contains a concise summary of the operations applied (and their parameters) encoded as values of the type $OP$.

$$OP ::= \text{send}((PE \times \text{THREAD} \times \text{CHANNEL}))$$
$$| \text{receive}((PE \times \text{THREAD} \times \text{CHANNEL}))$$
$$| \text{advert}((PE \times \text{CHANNEL} \times PE))$$
$$| \text{focus}((PE \times \text{CHANNEL}))$$
$$| \text{rtr}((PE \times \text{CHANNEL}))$$
$$| \text{msg}((PE \times \text{CHANNEL}))$$
$$| \text{create}((\text{THREAD} \times \text{THREAD}))$$
$$| \text{sync}((\text{THREAD}))$$
$$| \text{terminate}((\text{THREAD}))$$
$$| \text{term}((\text{THREAD}))$$

Some of the $OP$ values have similar identifiers to those used in $TState$ components (defined next) and $CNETMSG$ types so the $OP$ values are printed in a sans serif font so that they can be distinguished. The use of the $OP$ types are not explained in detail but, for instance, the send $OP$ is used to extend the trace during the application of the channel send operation.

### 4.2.2 State

I declare a schema called $TState$ which represents part of the state that the Testbed may be in at any given moment. Formally, $parent$, $cntr$ and $par_{bd}$ are partial functions, e.g. $cntr$ is a partial function from threads to counters, $ready$ is a relation (between processors and threads) and $cnet$ is a set (of Centrenet messages).
Informally, *parent* gives for executing threads the parent thread that created them, *cntr* gives for each executing parent thread its counter, *par_bd* gives for every child the processor board on which its parent is executing, *ready* associates with each processor the set of threads that are ready to execute on it and *cnet* models the processor interconnect (sending a message is modelled by adding the message to *cnet* and receiving a message is modelled by removing a message from *cnet*).

$$
\begin{align*}
TState & \\
\text{parent} & : \text{THREAD} \rightarrow \text{THREAD} \\
\text{cntr} & : \text{THREAD} \rightarrow \text{COUNTER} \\
\text{par_bd} & : \text{THREAD} \rightarrow \text{PE} \\
\text{ready} & : \text{PE} \leftrightarrow \text{THREAD} \\
\text{cnet} & : \mathbb{P} \text{CNETMSG}
\end{align*}
$$

It is a convention in $\mathbf{Z}$ that each state schema should be followed by another schema which defines the initial value of the state schema, and this is what *TInit* does.

$$
\begin{align*}
TInit & \\
TState & \\
t? & : \text{THREAD} \\
p? & : \text{PE} \\
\text{ready} & = \{ p? \mapsto t? \} \\
\text{par_bd} & = \{ t? \mapsto p? \} \\
\text{parent} & = \{ t? \mapsto \text{null_thread} \} \\
\text{cntr} & = \{ \text{null_thread} \mapsto (1, \text{null_thread}) \} \\
\text{cnet} & = \emptyset
\end{align*}
$$

The reader may like to interpret this as follows. In the initial state execution of a user's program has just begun and there is just one thread $t?$ in the ready queue of one of the processors $p?$. The processor $p?$ is recorded as the location
of t?'s parent. For convenience, the null thread is allocated the only counter and recorded as being the parent for t?. Centrenet has no messages to deliver.

### 4.2.3 State-changing operations

Now that the basic entities have been defined (threads, processors and counters) and I have declared the relationships (parent, cntr, par_bd and so on) between entities that are to be recorded in the state $T_{State}$, the next part of the specification defines the operations which modify the state. The operations available to threads allow occam-like task control (as described in Section 3.1.3) to be implemented. Figure 4–1 shows an example where a parent thread creates three children and is suspended until they all terminate.

![Diagram](image)

**Figure 4–1:** An example of the occam-like task control on the Testbed.

The first operation specified is called $T_{Create}$ and models what happens when a parent thread calls the operating system and requests that a new child thread be created. The schema declaration $\Delta T_{State}$ indicates that this schema changes that part of the Testbed state modelled by $T_{State}$. The declarations $pt?, ct? : THREAD$ and $p? : PE$ are inputs to the operation and the reader should think of $pt?$ as the parent thread requesting the create service; $ct?$ as identifying the child thread to be created and $p?$ as being the processor on which the request is being made.
The first part of the schema predicate specifies how the counter part of \( TState \) is modified: if the parent does not have a counter \( (pt? \notin \text{dom}(cntr)) \) then a new counter \( (1, \text{null\_thread}) \) is associated with the parent thread; if the parent thread already has a counter \( (pt? \in \text{dom}(cntr)) \) then the counter is incremented to \( (\text{first}(cntr \ pt?) + 1, \text{null\_thread}) \). The \( \{ct?\} \triangleleft cntr \) ensures that initially no counter is associated with the new child.

### TCreate

\[
\Delta TState \\
pt?, ct?: \text{THREAD} \\
p?: \text{PE}
\]

\[
(\text{let } p == \begin{cases} \text{if } pt? \in \text{dom}(cntr) \text{ then } \text{first}(cntr \ pt?) \text{ else } 0 \end{cases} \cdot \\
\quad \text{cntr}' = (\{ct?\} \triangleleft cntr) \oplus \{pt? \mapsto (p + 1, \text{null\_thread})\} \\
\quad \text{parent}' = \text{parent} \cup \{ct? \mapsto pt?\} \\
\quad \text{ready}' = \text{ready} \cup \{p? \mapsto ct?\} \\
\quad \text{par\_bd}' = \text{par\_bd} \cup \{ct? \mapsto p?\}
\]

The identity of the parent is recorded by \( \text{parent}' = \text{parent} \cup \{ct? \mapsto pt?\} \). The new child is made ready to execute by adding it to the ready queue with \( \text{ready}' = \text{ready} \cup \{p? \mapsto ct?\} \) and the parent's processor board is recorded for the child by \( \text{par\_bd}' = \text{par\_bd} \cup \{ct? \mapsto p?\} \).

The next operation, \( TSync \), models what happens once a parent has created some children and wants to be blocked until its children have finished executing. The inputs to the schema are \( pt? \), the parent thread requesting to be blocked and \( p? \), the processor on which the request is made. The schema predicates specify that \( cntr \) and \( \text{ready} \) are updated if the counter associated with the parent has a value greater than zero \( (\text{first}(cntr \ pt?) > 0) \); otherwise all the children must have terminated and no action is taken.

If there are still children executing then the \( cntr \) function is updated by setting the parent's counter to the value \( (\text{first}(cntr \ pt?), pt?) \) showing that the number
of children is unchanged but that the parent thread $pt?$ is blocked on the counter. Simultaneously, the parent is removed from the ready queue by updating $ready'$ to $ready \setminus \{p? \mapsto pt?\}$.

\[
\begin{align*}
TSync
\Delta TState \\
pt?: \text{THREAD} \\
p?: \text{PE}
\end{align*}
\]

\[
\begin{align*}
\text{first}(\text{cntr } pt?) > 0 \Rightarrow \\
\text{cntr}' &= \text{cntr} \oplus \{pt? \mapsto (\text{first}(\text{cntr } pt?), pt?)\} \land \\
ready' &= ready \setminus \{p? \mapsto pt?\}
\end{align*}
\]

The next schema, $T\text{Terminate}$, models a child thread requesting to terminate. The inputs are $ct?$ and $p?$ representing the child thread and the processor on which the child is executing. This time, there is also an output argument, $msgs!$ which is defined as the messages to be sent over Centrenet (or the empty set if none). This output argument allows the lower-level schema $T\text{Terminate}$ to pass values back to the higher-level schema $F\text{Terminate}$ defined later on in Section 4.2.4.

The first two predicates delete any mappings involving the terminating thread from the $\text{parent}$ and $\text{par_bd}$ functions. The rest of the predicates are in two parts: the first part is applied when parent and child are executing on different processor boards ($\text{par_bd } ct? \neq p?$) and the second part is applied when they are on the same processor board ($\text{par_bd } ct? = p?$).

If the threads are on different processor boards then the output argument is set to inform the parent on the remote processor of its child's termination. The child is removed from the ready queue for the processor $p?$ and any counters which might be owned by the child are deleted (so that they can be reused by other threads).

If the threads are on the same processor board then the output argument $msgs!$ is set to $\emptyset$ to indicate that no messages need to be transmitted to remote parents.
For convenience the 'temporary variable' $c$ is created and set to the parent’s counter and the variable $r$ is created and set to the contents of the ready queue with the child removed. There are then two possible cases: $first\ c = 1$ meaning that the last child is terminating and $first\ c > 1$ meaning that there are still children executing. If the last child is terminating then it is removed from the ready queue, any counters owned by it are deleted and the parent’s counter is set to zero. If there was a parent waiting for the counter then the parent is returned to the ready queue. If the terminating child is not the last child then it is removed from the ready queue and the parent’s counter is decremented.

Finally, here is the $TTerm_{\text{msg}}$ schema which is responsible for receiving a 'termination message'. Termination messages are sent from terminating children to parents when the parents are executing on different processors. The input arguments are $pt?$, which indicates the parent, and $p?$ which gives the location of the parent. In the predicate part of $TTerm_{\text{msg}}$ a temporary variable is created and set to the parent’s counter. The value of this counter is tested and, as in the $TTerminate$ schema, if the counter indicates that the last child is terminating then the parent is returned to the ready queue and its counter reset; otherwise the parent’s counter is decremented. Despite the convention that state components not mentioned in an operation schema are assumed to retain their original value, the predicate $ready' = ready$ is stated here for reasons of clarity.

$$\begin{align*}
TState & \quad \Delta

\text{let } c == \text{cntr } pt? \quad \bullet \\

\text{first } c = 1 & \Rightarrow
\hspace{1cm} (\text{ready}' = \text{if second } c = pt? \ \text{then } \text{ready} \oplus \{p? \mapsto pt?\} \ \text{else } \text{ready} \land \\

\text{cntr}' = \text{cntr} \oplus \{p? \mapsto (0, \text{null\_thread})\}) \land
\end{align*}$$

$$\text{first } c > 1 \Rightarrow (\text{ready}' = \text{ready} \land \\
\text{cntr}' = \text{cntr} \oplus \{p? \mapsto (\text{first } c - 1, \text{second } c)\})$$
Chapter 4. Specification of Testbed Basics

Termination

\[ \Delta \text{State} \]

\( \text{ct?} : \text{THREAD} \)

\( \text{p?} : \text{PE} \)

\( \text{msgs!} : \mathbb{P} \text{CNETMSG} \)

\[ \text{parent'} = \{\text{ct?}\} \triangleleft \text{parent} \]

\[ \text{par_bd'} = \{\text{ct?}\} \triangleleft \text{par_bd} \]

\( (\text{let} \; pt == \text{parent ct?} \; \bullet \)

\[ \text{par_bd ct?} \neq \text{p?} \Rightarrow \]

\[ (\text{msgs!} = \{\text{term} (\text{par_bd ct?, pt})\} \land \]

\[ \text{ready'} = \text{ready} \setminus \{\text{p?} \rightarrow \text{ct?}\} \land \]

\[ \text{cntr'} = \{\text{ct?}\} \triangleleft \text{cntr} \land \]

\[ \text{par_bd ct?} = \text{p?} \Rightarrow \]

\[ (\text{msgs!} = \emptyset \land \]

\[ (\text{let} \; c == \text{cntr pt}; \]

\[ r == \text{ready} \setminus \{\text{p?} \rightarrow \text{ct?}\} \; \bullet \]

\[ \text{first c} = 1 \Rightarrow \]

\[ (\text{ready'} = \text{if second c} = \text{pt} \text{ then } r \oplus \{\text{p?} \rightarrow \text{pt}\} \text{ else } r \land \]

\[ \text{cntr'} = (\{\text{ct?}\} \triangleleft \text{cntr}) \oplus \{\text{pt} \rightarrow (0, \text{null_thread})\}) \land \]

\[ \text{first c} > 1 \Rightarrow \]

\[ (\text{ready'} = r \land \]

\[ \text{cntr'} = (\{\text{ct?}\} \triangleleft \text{cntr}) \oplus \{\text{pt} \rightarrow (\text{first c} - 1, \text{second c})\}) \]
4.2.4 Model of execution

So far in this chapter I have declared some basic entities for the model, defined a representation for the state and presented four operations which update the state. I now present four, higher-level schemas of a kind often termed 'framing schemas'. The framing schemas build on top of the preceding definitions and give an explanation of when, rather than what, operations on the state $TState$ are performed.

![Diagram of thread life-cycle](image)

**Figure 4-2: The life-cycle of threads on the Testbed.**

The life-cycle of Testbed threads is depicted in Figure 4-2. All threads begin in the unborn state and remain there until the create operation moves them into the ready state. In accordance with the scheduling procedure, threads are taken from the ready state one-by-one and allowed to execute on the CPU until: an error or termination request occurs; or the time-slice ends; or a request for an operating system service occurs. Terminating or erroneous threads make no further state changes, threads at the end of their time-slice return immediately to the ready state and threads requesting system services return to the ready state once the service has completed.
For simplicity, not all states and transitions are modelled. For the purposes of the occam task model I am concerned only with the ready and waiting-to-synchronise states and with the create, request-service, service-completed and terminate transitions.

The specification models the programs executed by threads as sequences of instructions where each instruction has a fixed type and, sometimes, parameters. The Z construct for free types is used to define `INSTRUCTION` as either: a create instruction parameterised on the child to be created; a synchronise instruction; a terminate instruction; or a send or receive parameterised by channel and memory block for the message. (The send and receive instructions will be used later on in Section 4.3.3 when I specify the schemas for channel communication.)

\[
\text{INSTRUCTION} ::= \text{create}(\text{THREAD})
\]

\[
| \text{sync} \\
| \text{terminate} \\
| \text{send}(\text{CHANNEL} \times \text{MEM BLOCK}) \\
| \text{receive}(\text{CHANNEL} \times \text{MEM BLOCK})
\]

I model more of the state of the Testbed by defining the schema `FState` as everything in `TState` plus a set of program counters (one per thread), programs (one per thread) and operation traces (one trace for all threads). The operation schemas defined in this section will modify `FState` by executing instructions from these programs. The function `pc` gives for each executing thread the number of the next instruction to be executed in its program and the `trace` is the sequence of all operations performed so far by the Testbed processors.

<table>
<thead>
<tr>
<th><code>FState</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TState</code></td>
</tr>
<tr>
<td><code>pc : THREAD \rightarrow \mathbb{N}</code></td>
</tr>
<tr>
<td><code>program : THREAD \rightarrow \text{seq INSTRUCTION}</code></td>
</tr>
<tr>
<td><code>trace : \text{seq OP}</code></td>
</tr>
</tbody>
</table>
The $FInit$ schema defines the initial value of $FState$ as all the predicates in $TInit$ plus the condition that the initial thread's program counter is set to 1 plus the condition that the null thread does not have a program. It is assumed that the programs associated with the other threads include sensible combinations of instructions—the proofs in Chapter 6 state exactly what is meant by a 'well-behaved program'. The creation of $t?$ is the only thing recorded in the trace. The application of $TInit[t/t?, p/p?]$ illustrates the use of substitution which is typical in framing schemas: the new variable $t$ declared in $FInit$ is to be substituted for the old $t?$ declared in $TInit$ and the new $p$ substituted for $p$.

\[
\begin{align*}
FInit \quad & \\
FState \quad & \\
\exists t : THREAD \setminus \{null\_thread\}; p : PE \cdot TInit[t/t?, p/p?] \land \\
pc = \{t \mapsto 1\} \land \text{trace} = (create(null\_thread, t)) \\
null\_thread \notin \text{dom(program)}
\end{align*}
\]

The framing schema, $FCreate$, for the create operation states that if a parent $pt$ is in the ready list of processor $p$ and the current instruction in its program is to create a child thread $ct$, then the $TCreate$ operation should be applied. Assuming that the preconditions of $TCreate$ are satisfied and that $TCreate$ updates $TState$, then the $FState$ is modified by incrementing the program counter for the parent, initialising the program counter for the child and augmenting the trace.

\[
\begin{align*}
FCreate \quad & \\
\Delta FState \quad & \\
\exists pt, ct : THREAD; p : PE \cdot (p, pt) \in \text{ready} \land \\
create(ct) = \text{program}_{pt}(pc_{pt}) \land \\
TCreate[pt/pt?, ct/ct?, p/p?] \land \\
pc' = pc \oplus \{pt \mapsto pc_{pt} + 1, ct \mapsto 1\} \land \\
\text{trace}' = \text{trace} \smallsetminus (create(pt, ct))
\end{align*}
\]
The \textit{FSync} schema follows a similar format to \textit{FCreate}. A parent thread \(pt\) in the ready state executes the instruction to synchronise with its children thus causing the \textit{TSync} schema to be applied, its program counter to be incremented and the trace to be augmented to record the synchronisation.

### FSync

\begin{align*}
\Delta FState \\
\exists pt : \text{THREAD}; p : \text{PE} \quad \bullet \\
(p, pt) \in \text{ready} \land \\
sync = \text{program } pt(pc pt) \land \\
\text{TSync}[pt/pt?, p/p?] \land \\
pc' = pc \oplus \{pt \mapsto pc pt + 1\} \land \\
trace' = \text{trace} \cup \text{sync}(pt)
\end{align*}

In \textit{FTerminate} the \textit{TTerminate} schema defines the output argument \(msgs!\). If the parent of the terminating child is on the same processor then \(msgs!\) is the empty set and the trace is augmented, otherwise the parent is on a different processor, a \textit{term} is transmitted and the trace is not changed.

### FTerminate

\begin{align*}
\Delta FState \\
\exists ct : \text{THREAD}; p : \text{PE}; msgs : \mathbb{P} \text{CNETMSG} \quad \bullet \\
(p, ct) \in \text{ready} \land \\
terminate = \text{program } ct(pc ct) \land \\
\text{TTerminate}[ct/ct?, p/p?, msgs/msgs!] \land \\
cnet' = cnet \cup msgs \land \\
pc' = pc \oplus \{ct \mapsto pc ct + 1\} \land \\
trace' = \text{if } msgs = \emptyset \text{ then } \text{trace} \cup \text{terminate}(ct) \text{ else trace}
\end{align*}

The final schema \textit{FTerm_msg} shows what happens at the remote site when the \textit{FTerminate} schema sends a \textit{term} message over Centrenet: \(\exists pp : \text{PE}; pt : \)
THREAD • term(pp, pt) ∈ cnet represents the situation where Centrenet delivers a terminate message for parent thread pt to processor pp. The effect of the schema is to apply the TTerm_msg schema and delete the term message from cnet to indicate that it has been obeyed. The trace is updated to record the termination.

\[
\text{FTerm_msg} \\
\Delta FState
\]

\[
\exists pp : PE; pt : THREAD \cdot \\
term(pp, pt) \in cnet \land \\
TTerm_msg[pp/?, pt/pt?] \land \\
cnet' = cnet \setminus \{\text{term}(pp, pt)\} \land \\
trace' = trace \setminus (\text{term}(pt))
\]

4.3 Channel Design

This section introduces two new objects (channels and memory blocks) which it uses to define that part of the Testbed state concerned with communication. Six channel operations are presented and six higher-level framing schemas.

4.3.1 Basic objects and state

I declare two more given sets, CHANNEL the set of all Testbed communication channels, and MEM_BLOCK the set of memory blocks. The memory blocks are used when modelling communications to give the idea of message transfer being effected by a memory-to-memory block copy.

\[\text{[CHANNEL, MEM_BLOCK]}\]
Here is the schema $CState$ which models the part of the Testbed state to do with channel communications. Actually, $CState$ includes everything in $TState$ as well, so it also models thread state.

\[
\begin{array}{l}
\begin{array}{l}
CState \\
TState \\
scontrol, \ rcontrol, \ sender, \ receiver : PE \leftrightarrow (CHANNEL \leftrightarrow THREAD) \\
sloc, \ rloc : PE \leftrightarrow (CHANNEL \leftrightarrow PE) \\
focus, \ used : PE \leftrightarrow CHANNEL \\
buffer : PE \leftrightarrow (CHANNEL \leftrightarrow MEM\_BLOCK)
\end{array}
\end{array}
\]

The functions used to model the channel state have quite complicated semantics, but at a reasonably abstract level they can be described as follows:

$scontrol$: The maplet $p \rightarrow \{c \mapsto t\}$ exists in $scontrol$ if and only if the processor $p$ has a thread $t$ blocked, waiting to send on $c$. The $scontrol$ function has two uses: it provides a place to store the thread while not in the ready queue and it can be checked to see if the channel is in use.

$rcontrol$: This function is used in the same way as $scontrol$ but for storing blocked receiver threads.

$sender$: The maplet $p \rightarrow \{c \mapsto t\}$ is present here if and only if the thread $t$ has, since the beginning of the program execution, requested to send on channel $c$ while executing on $p$ (or if it has communicated on $c$ and then been migrated to $p$). The $sender$ function is used during thread migration to check which channels a thread has sent on.

$receiver$: This function is similar to $sender$, except that it records the channels a thread has received on.

$sloc$: The maplet $p_1 \rightarrow \{c \mapsto p_2\}$ is defined in this function if and only if the processor $p_1$ is storing $p_2$ as the last known location for the sender for channel $c$. If undefined, then $p_1$ knows nothing about the sender's location.
rioc: This function is used in the same way as sloc, except that where defined it indicates the location of the receiver.

focus: Synchronisation during communication is enforced by having the sender produce a ‘focus’ (essentially, just a token) when it requests to send and requiring the receiver to consume that same focus before completing the communication. The focus for a channel $c$ is able to travel between processors (so that senders can synchronise with remote receivers) but when and only when it stops at a particular processor $p$ is the function focus defined for the maplet $p \rightarrow c$.

used: The function defines the maplet $p \rightarrow c$ if and only if a sender or receiver executing on processor $p$ has, since the beginning of the program execution, requested to communicate on channel $c$. The function is needed to distinguish the first communication over a channel when the sender and receiver still need to locate each other, from subsequent communications when each knows where the other is.

buffer: This is a place for blocked threads to store their messages or empty buffers while they wait for communication to complete.

The initial channel state $CInit$ is defined as the initial thread state extended to model the situation where no channel communication has occurred yet—hence all functions are undefined.

\[
\begin{align*}
CInit \\
TInit \\
CState
\end{align*}
\]

\[
\begin{align*}
scontrol & = \text{rcontrol} = \text{sender} = \text{receiver} = \emptyset \\
sloc & = \text{rioc} = \emptyset \land \text{focus} = \text{used} = \emptyset \land \\
buffer & = \emptyset
\end{align*}
\]
4.3.2 State-changing operations

There are six channel operation schemas. The first two are applied when threads request to send or receive on a channel, the other four are applied on reception of various kinds of Centrenet message involved in remote channel communication.

The \textit{CSend} schema models the situation where a sender thread \textit{st} executing on 'sender processor' \textit{sp} requests to send the message in its message buffer \textit{m} over channel \textit{c}. The schema also has an output argument \textit{msgs} which is a (possibly empty) set of messages to be transmitted over Centrenet.

The location of the sender is stored in \textit{sloc} for the use of the receiver (if it is local). The identity of the sender is stored in \textit{sender} so that during migration the Testbed can tell on which channels a given thread has sent messages. The rest of the predicates are in two parts depending on whether there is a local, receiver thread already waiting to receive on the channel \((c \in \text{dom} (\text{rcontrol} \text{ sp}))\) or not.

If there is a local, waiting receiver then a temporary variable \textit{rt} is defined to be the receiver thread and a variable \textit{b} to be the receiver's buffer. The detail of the transfer of the message is hidden inside the \textit{MEM\_COPY} schema (not defined in the specification). The receiver thread is unblocked \((\text{rcontrol}' = \text{rcontrol}\backslash \{\text{sp} \mapsto \{c \mapsto \text{rt}\}\})\) and returned to the ready queue \((\text{ready}' = \text{ready} \cup \{\text{sp} \mapsto \text{rt}\})\). The channel is marked as having been used. It is to be understood that a channel focus has been produced and immediately consumed.

If there is no local, waiting receiver then the sender's buffer is recorded in \textit{buffer} for ease of retrieval later and the sender is deleted from the ready queue and blocked on the channel. There are now two, alternative cases depending on whether there is a last known location for the receiver. If \(c \in \text{dom} (\text{rloc} \text{ sp})\) then the receiver was last at \textit{rloc} \text{ sp} \(c\) so the focus is sent there by putting \textit{focusm(rloc sp c?, c?, sp?)} in \textit{msgs}! or the state component \textit{focus} is updated locally if the receiver was last known on \textit{sp}.

If nothing is known about the location of the receiver \((c \notin \text{dom} (\text{rloc} \text{ sp}))\) then the focus is stored locally by \textit{focus}' = \textit{focus} \cup \{\text{sp} \mapsto c\} and a series of messages
**Chapter 4. Specification of Testbed Basics**

**CSend**

ΔCState

st? : THREAD

c? : CHANNEL

sp? : PE

m? : MEM_BLOCK

msgs! : PCNETMSG

\[ sloc' = sloc \oplus \{sp? \mapsto \{c? \mapsto sp?\}\} \]

\[ \text{sender}' = \text{sender} \oplus \{sp? \mapsto \{c? \mapsto st?\}\} \]

\[ c? \in \text{dom}(rcontrol sp?) \Rightarrow \]

\[ (\text{let } rt = rcontrol sp? c? \cdot \]

\[ (\text{let } b = buffer sp? c? \cdot \text{MEM\_COPY}[m?/from? , b/to?]) \land \]

\[ \text{ready}' = \text{ready} \cup \{sp? \mapsto rt\} \land \]

\[ rcontrol' = rcontrol \setminus \{sp? \mapsto \{c? \mapsto rt\}\} \land \]

\[ used' = used \cup \{sp? \mapsto c?\} \land \]

\[ msgs! = \emptyset \)

\[ c? \notin \text{dom}(rcontrol sp?) \Rightarrow \]

\[ (\text{buffer}' = buffer \oplus \{sp? \mapsto \{c? \mapsto m?\}\}) \land \]

\[ \text{ready}' = \text{ready} \setminus \{sp? \mapsto st?\} \land \]

\[ scontrol' = scontrol \oplus \{sp? \mapsto \{c? \mapsto st?\}\} \land \]

\[ c? \in \text{dom}(rloc sp?) \Rightarrow \]

\[ (\text{focus}', \text{msgs}!) = \text{if } rloc sp? c? = sp? \text{ then } (\text{focus} \cup \{sp? \mapsto c?\}, \emptyset) \]

\[ \text{else } (\text{focus}, \{\text{focusm}(rloc sp? c? , c?, sp?)\}) \land \]

\[ (c? \notin \text{dom}(rloc sp?) \Rightarrow \]

\[ (\text{focus}' = \text{focus} \cup \{sp? \mapsto c?\} \land \]

\[ \text{msgs!} = \{p : PE \setminus \{sp?\} \cdot advert(p, c?, sp?)\})\) \]
are sent advertising the presence of the waiting sender. These messages are sent to all processors except the one where the sender is waiting \((p : PE \setminus \{ sp? \})\).

The CReceive schema models the situation where a receiver thread \(rt?\) executing on `receiver processor` \(rp?\) requests to receive a message into its buffer \(b?\). The send and receive schemas are similar in the case where a thread requesting communication finds the other thread already waiting. In the case that the sender finds no receiver waiting, its actions are based on whether it knows the receiver's location and, if so, whether the receiver is local or remote. In the case that the receiver finds no waiting sender, it bases its actions on whether there is an indication of a waiting sender's focus or advertisement, or no such indication.

The location of the receiver is stored in \(rloc\) and the identity of the receiver in \(receiver\). The predicates are then in two parts, depending on whether there is a local, waiting sender. If there is a waiting sender then a temporary variable \(st\) is defined to be the sender and a variable \(m\) to be the sender's message. The message transfer is effected by \(MEM\_COPY\), the sender unblocked from the channel and returned to the ready queue. The used flag is set for the channel and the focus flag is forced to undefined regardless of its previous state. Again, it is to be understood that a focus was produced by the sender and immediately consumed by the receiver.

If there is no local, waiting sender then the receiver's buffer is recorded, the receiver is deleted from the ready queue and blocked on the channel. There are now two cases depending on whether a focus for the channel is present. If a focus is present then an \(rtr\) message is sent to the sender at the location specified in \(sloc\) \(rp?\) \(c?\) (I will prove later that this is indeed an appropriate destination for the message). The focus is consumed by the receiver.

If no focus is found then a decision is made whether to send an \(rtr\) or not. Initially, all channels have undefined sender location values and undefined used flags. A channel that has received an advertisement from a remote sender but has not been used, has a defined \(sloc\) but an undefined used flag—a receiver should send an \(rtr\) in this case. A channel that has not received an advertisement, has been used already, or which has a local sender, should not send an \(rtr\).
\begin{align*}
&CReceive
\Delta CState
rt\? : THREAD

\begin{array}{ll}
c\? : CHANNEL \\
rp\? : PE \\
b\? : MEM\_BLOCK \\
msgs! : \mathbb{P}\text{CNETMSG}
\end{array}

\begin{align*}
&\text{rloc'} = \text{rloc} \oplus \{rp\? \mapsto \{c\? \mapsto rp\?\}\} \\
&\text{receiver'} = \text{receiver} \oplus \{rp\? \mapsto \{c\? \mapsto rt\?\}\}
\end{align*}

\begin{align*}
c\? \in \text{dom}(\text{scontrol}(rp\?)) \Rightarrow \\
&(\text{let } st == \text{scontrol}(rp\? c\?) \cdot \\
&(\text{let } m == \text{buffer}(rp\? c\?) \cdot \text{MEM\_COPY}[m/\text{from}?, b?/\text{to}?]) \land \\
&\text{ready'} = \text{ready} \cup \{rp\? \mapsto st\} \land \\
&\text{scontrol'} = \text{scontrol} \setminus \{rp\? \mapsto \{c\? \mapsto st\}\} \land \\
&\text{used'} = \text{used} \cup \{rp\? \mapsto c\?\} \land \\
&\text{focus'} = \text{focus} \setminus \{rp\? \mapsto c\?\} \land \\
&\text{msgs!} = \emptyset)
\end{align*}

\begin{align*}
c\? \notin \text{dom}(\text{scontrol}(rp\?)) \Rightarrow \\
&(\text{buffer'} = \text{buffer} \oplus \{rp\? \mapsto \{c\? \mapsto b\?\}\} \land \\
&\text{ready'} = \text{ready} \setminus \{rp\? \mapsto rt\?\} \land \\
&\text{rcontrol'} = \text{rcontrol} \oplus \{rp\? \mapsto \{c\? \mapsto rt\?\}\} \land \\
&(\text{focus'}, \text{msgs!}) = \text{if } rp\? \mapsto c\? \in \text{focus} \\
&\text{then } (\text{focus} \setminus \{rp\? \mapsto c\?\}, \{\text{rtr}(\text{sloc}(rp\? c\?, c\?, rp\?)\}) \\
&\text{else } (\text{focus}, \\
&\text{if } c\? \in \text{dom}(\text{sloc}(rp\?)) \land \text{sloc}(rp\? c\?) \neq rp\? \land rp\? \mapsto c\? \notin \text{used} \\
&\text{then } \{\text{rtr}(\text{sloc}(rp\? c\?, c\?, rp\?)\} \text{ else } \emptyset))
\end{align*}
The \textit{CAdvert} schema models the situation on the Testbed where an advertisement for a channel sent by a sender on processor \textit{sp}\textsubscript{?} arrives at a processor \textit{rp}\textsubscript{?}. The predicates simply require that the sender location should be defined (or updated) and that an \textit{rtr} message be returned if there is a waiting receiver.

\begin{align*}
\text{\texttt{CAdvert}} \\
\Delta \text{CState} \\
\text{\textit{rp}\textsubscript{?}, \textit{sp}\textsubscript{?} : PE} \\
\text{\textit{c}\textsubscript{?} : CHANNEL} \\
\text{msg! : \mathbb{P} \text{CNETMSG}}
\end{align*}

\begin{align*}
\text{sloc'} &= \text{sloc} \oplus \{\text{\textit{rp}\textsubscript{?} \mapsto \{\text{\textit{c}\textsubscript{?} \mapsto \textit{sp}\textsubscript{?}\}\}}\} \\
\text{msg!} &= \text{if } \text{\textit{c}\textsubscript{?} \in \text{dom}(rcontrol \textit{rp}\textsubscript{?}) } \text{then } \{\text{\textit{rtr}(\textit{sp}\textsubscript{?}, \text{\textit{c}\textsubscript{?}, \textit{rp}\textsubscript{?})}\} \text{ else } \emptyset
\end{align*}

The \textit{CFocus} schema models the situation where a focus for a channel, originally from the \textit{sp}\textsubscript{?} processor, arrives at a new processor \textit{rp}\textsubscript{?}.

\begin{align*}
\text{\texttt{CFocus}} \\
\Delta \text{CState} \\
\text{\textit{rp}\textsubscript{?}, \textit{sp}\textsubscript{?} : PE} \\
\text{\textit{c}\textsubscript{?} : CHANNEL} \\
\text{msg! : \mathbb{P} \text{CNETMSG}}
\end{align*}

\begin{align*}
\text{sloc'} &= \text{sloc} \oplus \{\text{\textit{rp}\textsubscript{?} \mapsto \{\text{\textit{c}\textsubscript{?} \mapsto \textit{sp}\textsubscript{?}\}\}}\} \\
\text{\textit{c}\textsubscript{?} \in \text{dom}(rloc \textit{rp}\textsubscript{?})} \land \text{\textit{rloc} \textit{rp}\textsubscript{?} \text{\textit{c}\textsubscript{?} \neq \textit{rp}\textsubscript{?} } \Rightarrow \\
\text{\text{msg!} = \{\text{\textit{focusm}(\text{\textit{rloc} \textit{rp}\textsubscript{?} \text{\textit{c}\textsubscript{?}, \text{\textit{c}\textsubscript{?}, \textit{sp}\textsubscript{?})}\} } \\
\text{\textit{c}\textsubscript{?} \notin \text{dom}(\text{\textit{rloc} \textit{rp}\textsubscript{?})} \lor \text{\textit{rloc} \textit{rp}\textsubscript{?} \text{\textit{c}\textsubscript{?} = \textit{rp}\textsubscript{?})} \land \text{\textit{c}\textsubscript{?} \notin \text{dom}(\text{\textit{rcontrol} \textit{rp}\textsubscript{?}) } \Rightarrow \\
\text{\textit{\textit{focus}' = \text{\textit{focus} \cup \{\textit{rp}\textsubscript{?} \mapsto \text{\textit{c}\textsubscript{?}\}\} } \land \\
\text{\text{msg!} = \emptyset} \\
\text{\textit{c}\textsubscript{?} \notin \text{dom}(\text{\textit{rloc} \textit{rp}\textsubscript{?})} \lor \text{\textit{rloc} \textit{rp}\textsubscript{?} \text{\textit{c}\textsubscript{?} = \textit{rp}\textsubscript{?})} \land \text{\textit{c}\textsubscript{?} \in \text{dom}(\text{\textit{rcontrol} \textit{rp}\textsubscript{?}) } \Rightarrow \\
\text{\text{msg!} = \{\text{\textit{rtr}(\textit{sp}\textsubscript{?}, \text{\textit{c}\textsubscript{?}, \textit{rp}\textsubscript{?})}\} }
\end{align*}

The predicates specify that the sender location is updated and that the focus is either forwarded, stored, or consumed. Forwarding of the focus occurs if
the channel’s rloc points to another site—this would occur if the receiver had migrated away from rp?. The focus is stored if rloc is undefined (or points to rp?) and there is no waiting receiver. Otherwise, if rloc does not point to another site and there is a waiting sender then an rtr is returned to the sender and the focus is consumed.

The CRtr schema models the situation where an rtr message for channel c? arrives at a sender’s processor sp? from the receiver’s processor rp?. The receiver’s location is updated by overwriting rloc with \{sp? \mapsto \{c? \mapsto rp?\}\} and the output variable msgs! is set to contain the message data which was stored in buffer during the preceding application of CSend. The output argument st! is defined and the sender thread is unblocked by moving it from scontrol to ready. Regardless of their previous values, the flag focus is forced to be undefined for channel c? at processor sp? and the flag used is forced to be defined for channel c? at processor sp?.

\[
\begin{align*}
\Delta CState \\
sp?, rp? : PE \\
c? : CHANNEL \\
st! : THREAD \\
msgs! : \mathbb{P} CNETMSG \\
\end{align*}
\]

\[
\begin{align*}
\text{rloc}' &= \text{rloc} \oplus \{sp? \mapsto \{c? \mapsto rp?\}\} \\
\text{msgs}! &= \{\text{msg}(rp?, c?, sp?, \text{buffer sp? c?})\} \\
\text{st!} &= \text{scontrol sp? c?} \\
\text{ready}' &= \text{ready} \cup \{sp? \mapsto \text{st!}\} \\
\text{scontrol}' &= \text{scontrol} \setminus \{sp? \mapsto \{c? \mapsto \text{st!}\}\} \\
\text{focus}' &= \text{focus} \setminus \{sp? \mapsto \text{c?}\} \\
\text{used}' &= \text{used} \cup \{sp? \mapsto \text{c?}\}
\end{align*}
\]

The CMsg schema models the situation where a receiver on processor rp? receives a data message from sp?. The copying of the message from the input
argument \( m? \) into the receiver’s memory area is suggested by \texttt{MEM\_COPY} and the receiver, defined by the output argument \( rt! \), is unblocked by removing it from \texttt{rcontrol} and returned to the ready queue. The location of the sender \( st! \) is updated at the receiver’s processor and the \texttt{used} flag is set for the channel \( c? \) on the receiver’s processor.

\[
\begin{align*}
\Delta CState \\
\text{rp?, sp? : PE} \\
c? : \text{CHANNEL} \\
m? : \text{MEM\_BLOCK} \\
rt! : \text{THREAD} \\
\text{(let } b == \text{buffer } \text{rp? } c? \bullet \text{MEM\_COPY}[m?/from?, b/to?]) \\
sloc' = sloc \oplus \{\text{rp? } \mapsto \{c? \mapsto \text{sp?}\}\} \\
rt! = \text{rcontrol } \text{rp? } c? \\
ready' = \text{ready } \cup \{\text{rp? } \mapsto \text{rt!}\} \\
rcontrol' = \text{rcontrol } \setminus \{\text{rp? } \mapsto \{c? \mapsto \text{rt!}\}\} \\
\text{used'} = \text{used } \cup \{\text{rp? } \mapsto \text{c?}\}
\end{align*}
\]

4.3.3 Model of execution

I have now declared the basic channel communication entities, defined a representation for the channel state and presented six operations which update this state. In this section I present framing schemas for the operations to specify \textit{when} (rather than \textit{how}) the channel state is changed.

Figures 4–3 and 4–4 summarise the sequences of channel operations allowed. The transitions are labelled with the names of framing schemas presented later in this section and indexed by the number of the processor at which they are applied (\( i \) and \( j \) are two arbitrarily selected processors such that \( i \neq j \)). The two vertices in bold indicate states where there is no ongoing communication. The eight cases may be interpreted as follows:
Figure 4-3: Allowed sequences of local channel operations.

Figure 4-4: Allowed sequences of remote channel operations.
Case 1: Two threads executing on the same processor (number $i$) share a local channel. The sender communicates first (the $F_{Send}.i$ transition).

Case 2: This case is the same as Case 1, except that it is the receiver which communicates first (the $F_{Receive}.i$ transition).

Case 3: Two threads share a channel but the sender is executing on processor $i$ and the receiver on processor $j$. The sender communicates first.

Case 4: This case is the same as Case 3 except that it is the receiver which communicates first.

Case 5: Two threads communicate for a second or subsequent time over a local channel. The sender communicates first.

Case 6: As Case 5 except that the receiver communicates first.

Case 7: Two threads on different processors communicate for the second or subsequent time. The sender communicates first.

Case 8: As Case 7 except that the receiver communicates first.

For reasons of presentational simplicity, two simple extensions are now made to the $F_{State}$ and $F_{Init}$ schemas which were defined in Section 4.2.4 (pages 71 and 72). In new schemas given below, $F_{State}$ now includes the channel state $C_{State}$ and $F_{Init}$ now includes the initial channel state $C_{Init}$.

<table>
<thead>
<tr>
<th>$F_{State}$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>$T_{State}$</th>
</tr>
</thead>
</table>

| $pc : THREAD \rightarrow N$ |

| $program : THREAD \rightarrow \text{seq INSTRUCTION}$ |

| $trace : \text{seq OP}$ |

| $C_{State}$ |
The framing schema for the send operation, $F_{Send}$, states that if sender $st$ is in the ready queue of processor $sp$ and the first instruction in its program is to send a message $m$ over channel $c$, then the $C_{Send}$ operation should be applied. Assuming that $C_{Send}$ updates $C_{State}$, $F_{State}$ is modified by adding any messages in $msgs$ to $cnet$, by incrementing the sender’s program counter if the communication completes and by adding the tuple $(send, st, c)$ to the trace.

The framing schema $F_{Receive}$ states that if receiver $rt$ is in the ready queue of processor $rp$ and the first instruction in the program is to receive a message
into buffer $b$ over channel $c$, then the \textit{CReceive} operation should be applied, new messages submitted to Centrenet, the program counter incremented if the communication completes and the trace extended.

\[ \text{\textit{FReceive}} \]

\[ \text{\textit{FState}} \]

\[ \exists c: \text{CHANNEL}; \quad b: \text{MEM\_BLOCK}; \quad rt: \text{THREAD}; \quad rp: \text{PE}; \quad \]
\[ \text{msgs}: \mathbb{P} \text{CNETMSG} \cdot \]
\[ (rp, rt) \in \text{ready} \land \]
\[ \text{receive}(c, b) = \text{program}\ rt(pc\ rt) \land \]
\[ \text{CReceive}[rt/rt?, c/c?, rp/rp?, b/b?, \text{msgs}/\text{msgs}!] \land \]
\[ pc' = \text{if} (rp, rt) \in \text{ready}' \text{ then } pc \oplus \{rt \mapsto pc\ rt + 1\} \text{ else } pc \land \]
\[ \text{cnet}' = \text{cnet} \cup \text{msgs} \land \]
\[ \text{trace}' = \text{trace} \smallsetminus \text{receive}(rp, rt, c) \]

The rest of framing schemas, \textit{FAdvert}, \textit{FFocus}, \textit{FRtr} and \textit{FMsg} are similar in format to the framing schema \textit{FReceive} except that they require a particular message to be delivered by Centrenet rather than the next program instruction to be of a particular kind.

\[ \text{\textit{FAdvert}} \]

\[ \text{\textit{FState}} \]

\[ \exists c: \text{CHANNEL}; \quad sp, rp: \text{PE}; \quad st: \text{THREAD}; \quad \text{msgs}: \mathbb{P} \text{CNETMSG} \cdot \]
\[ \text{advert}(rp, c, sp) \in \text{cnet} \land \]
\[ \text{CAdvert}[rp/rp?, c/c?, sp/sp?, \text{msgs}/\text{msgs}!] \land \]
\[ \text{cnet}' = (\text{cnet} \setminus \{\text{advert}(rp, c, sp)\}) \cup \text{msgs} \land \]
\[ \text{trace}' = \text{trace} \smallsetminus \text{advert}(rp, c, sp) \]
\textbf{FFocus}

\textbf{ΔFState}

\[ \exists c : \text{CHANNEL}; sp, rp : \text{PE}; st : \text{THREAD}; \text{msgs} : \mathbb{P} \text{CNETMSG} \cdot \]
\[ \text{focusm}(rp, c, sp) \in \text{cnet} \land \]
\[ C\text{Focus}(rp/rp?, c/c?, sp/sp?, \text{msgs}/\text{msgs!}) \land \]
\[ \text{trace'} = \text{trace} \smallsetminus \langle \text{focus}(rp, c) \rangle \land \]
\[ \text{cnet'} = (\text{cnet} \setminus \{\text{focusm}(rp, c, sp)\}) \cup \text{msgs} \]

\textbf{FRtr}

\textbf{ΔFState}

\[ \exists c : \text{CHANNEL}; sp, rp : \text{PE}; \text{msgs} : \mathbb{P} \text{CNETMSG}; rt, st : \text{THREAD} \cdot \]
\[ rtr(sp, c, rp) \in \text{cnet} \land \]
\[ C\text{Rtr}(sp/sp?, rp/rp?, c/c?, \text{msgs}/\text{msgs!}, \text{st}/\text{st!}) \land \]
\[ \text{cnet'} = (\text{cnet} \setminus \{rtr(sp, c, rp)\}) \cup \text{msgs} \land \]
\[ pc' = pc \oplus \{st \mapsto pc \text{ st} + 1\} \land \text{trace'} = \text{trace} \smallsetminus \langle rtr(sp, c) \rangle \]

\textbf{FMsg}

\textbf{ΔFState}

\[ \exists rp, sp : \text{PE}; c : \text{CHANNEL}; m : \text{MEM\_BLOCK}; rt : \text{THREAD} \cdot \]
\[ \text{msg}(rp, c, sp, m) \in \text{cnet} \land \]
\[ C\text{Msg}(rp/rp?, sp/sp?, c/c?, \text{msgs}/\text{msgs!}, \text{rt}/\text{rt!}) \land \]
\[ \text{cnet'} = \text{cnet} \setminus \{\text{msg}(rp, c, sp, m)\} \land \]
\[ pc' = pc \oplus \{rt \mapsto pc \text{ rt} + 1\} \land \]
\[ \text{trace'} = \text{trace} \smallsetminus \langle \text{msg}(rp, c) \rangle \]

Bringing together the framing schemas for thread control and channel communication and using $\textbf{Z}$'s notation for flat schema definition, a computation step can now be defined as a \textit{Receive} step or a \textit{Schedule} step:
4.4 Relating Design and Implementation

If the specification and correctness proofs about the specification are to give confidence in the safety of the implementation of the Testbed's operating system, then a convincing demonstration must be given that the specification is a true representation of the implementation. In this section I give just such a demonstration by showing how to relate each component of the specification to a part of the operating system.

The reader should note that the specification is assumed to operate with correct input. It is possible to extend specifications to deal with erroneous or inconsistent input but, as Hayes [38] reports, is not easy to model error recovery. Furthermore, when modelling a system as complex as an operating system to do so would obscure the real features of interest.

4.4.1 Basic entities

The \textit{THREAD} and \textit{CHANNEL} objects are implemented as records named \texttt{tcb} ('thread control block') and \texttt{ccb} ('channel control block'). The \textit{COUNTER} objects are also realised as records, of the type \texttt{counter}. The set \texttt{PE} corresponds to the six processors of the Testbed and values from \texttt{PE} are represented by the natural numbers 1..6. The \textit{MEM\_BLOCK} type is not represented directly because the specification does not represent the details of the Testbed's memory organisation.

The \textit{CNETMSG} type is implemented as a record of predefined size. Certain fixed fields are used to mark the type of the message as \texttt{focusm}, \texttt{advert} or whatever and to contain the variable number of parameters associated with each message type. The \texttt{OP} type is not represented in the implementation because

\[
\text{Receive} \equiv FTerm\_msg \lor FAdvert \lor FFocus \lor FRtr \lor FMsg
\]

\[
\text{Schedule} \equiv FCreate \lor FSsync \lor FTerminate \lor FSend \lor FReceive
\]
it is used only for constructing traces and the traces are there only to assist the proofs.

### 4.4.2 Functions and relations

Functions are represented by records and pointers or records and values. Consider for example, the \textit{scontrol} function defined in \textit{CState} with domain \textit{PE} and range \textit{CHANNEL} $\rightarrow$ \textit{THREAD}. The range is represented in records of the type \textit{ccb} by a field called \textit{sctrl}. Each instance of a \textit{ccb} record represents a particular channel and the value of \textit{sctrl} is either \texttt{NULL}—representing a channel for which \textit{scontrol} is undefined—or it is a pointer to a \textit{tcb} record—representing a thread blocked on the channel. The fact that the domain of \textit{scontrol} is the set \textit{PE} is represented by giving each processor its own set of \textit{ccb} records.

The \textit{focus} function in \textit{CState} shows the use of records and values. The range of \textit{focus} is represented by a field of type \texttt{unsigned char} in the \textit{ccb} records (although in practice the only values used are 0 meaning undefined or focus not set, and 1 meaning defined or focus set). The domain of \textit{focus} is implemented, as before, by giving each processor its own set of \textit{ccb} records.

Relations can be represented in the same way as functions or as sets of records. The \textit{ready} relation between processors and threads is, for instance, implemented by giving each processor a pointer to a list of threads.

### 4.4.3 Initial states

The initial state \textit{TInit} is implemented on the Testbed by having the operating system zero all fields in the \textit{tcb} records before creating a new thread. The initialisation of \textit{ready} occurs when the first thread of a new program is created. Similarly, the initial state \textit{CInit} is implemented by zeroing the \textit{ccb} records before use.
4.4.4 Operation schemas

The \textit{TCreate} and \textit{TTerminate} schemas are typical of the operation schemas so I shall use them to illustrate the correspondence between specification and implementation.

The \textit{TCreate} schema corresponds to the system function \texttt{trap6} which can be called by the user via the processor's exception mechanism. The input variable \texttt{pt?} corresponds to the current value of \texttt{curr_tcb}, a global pointer maintained by the operating system and pointing to the currently executing thread; \texttt{ct?} is a parameter passed in a register by the calling thread; \texttt{p?} is stored in a static variable which simply holds the number of the processor on which the operating system is running.

The \texttt{par_bd ct? \neq p? \Rightarrow construction} (from the \textit{TTerminate} schema) is implemented using the C construction if (condition) \texttt{block1} else \texttt{block2} where the condition tests for set membership and the choice of the block to execute duplicates the implication. The definition of after state in terms of before state, e.g. \texttt{par_bd' = par_bd \cup \{ct? \mapsto p?\}} (from the \textit{TCreate} schema) models assignment in C.

Other operation schemas use the $<$ and $\backslash$ operators to remove maplets from sets. In the implementation, this corresponds to assigning a zero or NULL value. The \texttt{let pt == parent ct? \bullet construction} (found in \textit{TTerminate}, for example) corresponds to the declaration and initialisation of a local or automatic stack variable in a C function.

The framing schemas fall into two groups according to the way that they are 'driven'. Schemas \textit{FCreate}, \textit{FSync}, \textit{FTerminate}, \textit{FSend} and \textit{FReceive} depend upon the current program instruction being of the appropriate type while the other framing schemas have preconditions such as \texttt{term(pp, pt) \in cnet} and are applied, therefore, according to the messages that Centrenet is ready to deliver.
Chapter 5

Specification of Testbed Migration

In this chapter I formally specify the non-standard features of the Testbed operating system that implement the mechanism for task migration. Chapter 6, which follows, contains the proofs of correctness based on the specification presented here and in the preceding chapter. It is interesting to note that, given the specification of threads and channels in Chapter 4, the additional specification required for migration is relatively short.

I model the two operations of disconnecting a thread and reconnecting a thread. The thread migration protocol takes care of moving the thread register context and control information and updating the channel control blocks. The protocol which enables recently migrated threads to retrieve their code and data memory pages is not modelled in this thesis because the memory page protocol is independent of the thread migration protocol and to present it here would add little of new interest.

5.1 Migration Protocol

This section gives, by means of a simple example, an informal introduction to the migration protocol implemented in the Testbed operating system.

With reference to the left half of Figure 5-1, consider a scenario involving three processors $p_1$, $p_2$ and $p_3$ and three threads $t_1$, $t_2$ and $t_3$. Two threads are
executing on one of the processors, one on another and the third processor is idle. There are two communication channels in use: a local channel between \( t_1 \) and \( t_2 \) implemented by the shared channel control block (CCB) labelled \( c_1 \) on processor \( p_1 \) and a remote channel between \( t_2 \) and \( t_3 \) implemented by two CCBs on \( p_1 \) and \( p_3 \) respectively. Each CCB has two outgoing arcs, the dashed arc represents the \( rloc \) value and points to the last known location for the receiver and the undashed arc represents the \( sloc \) value and points to the last known location for the sender. For example, the CCB \( c_2 \) on processor \( p_1 \) thinks its sender is on \( p_1 \) and its receiver is on \( p_3 \).

A possible optimisation would be to migrate thread \( t_2 \) from processor \( p_1 \) to \( p_2 \). This removes the competition between \( t_1 \) and \( t_2 \) for CPU cycles on \( p_1 \) but it also changes the local (low latency) channel between \( t_1 \) and \( t_2 \) into a remote (high latency) channel. Whether the new configuration is more efficient or not is program dependent, i.e. it depends on the relative amounts of computation and communication that \( t_1 \) and \( t_2 \) perform.

The first phase of thread migration is modelled by the \textit{MDisconnect} schema (presented in the next section) and involves disconnecting thread \( t_2 \) from its environment on processor \( p_1 \). The CCBs used by \( t_2 \) are examined, a snapshot is...
taken of their current status and they are updated to reflect the fact that $t_2$ is soon going to be at $p_2$. Then, the CCB snapshot is put into a Centrenet message along with $t_2$'s register context (and some other control information) and the message queued for transmission to $p_2$.

The right half of Figure 5-1 shows the situation at this point. Thread $t_2$ is somewhere in transit between processors $p_1$ and $p_2$, CCB $c_1$ has been updated to have the (soon-to-be) correct location for its receiver $t_2$ and the other CCBs are unchanged.

Phase two of the migration occurs when the Centrenet message containing $t_2$ and its associated information arrives at $p_2$ and begins to be processed. As modelled by the $M$Connect schema, $t_2$ is unpacked from the message and reconnected in its new environment. The channel status from the CCB snapshot is integrated with the local CCBs and the thread is put into the processor ready queue.

The new system state is shown in the left half of Figure 5-2 — thread $t_2$ is in place and its local CCBs have been updated to locate the sender for channel 1 on processor $p_1$ and the receiver for channel 2 on $p_3$.

Although the CCBs for channel 2 on $p_1$ and $p_2$ are not updated with respect to their sender locations, this is done by the next communication on channel 2:
the \textit{CSend} operation updates the sender location, \textit{sloc}, on \textit{p}_2 and the reception of the \textit{focusm} message at \textit{p}_3 in the \textit{CFocus} operation will update \textit{sloc} at \textit{p}_3 — as shown in the right half of Figure 5–2. In general, it is the sender who will initiate communication and so it is important that the sender have an up-to-date idea of where the receiver is, but it is not necessary for the receiver to remember where the sender is until communication is under way.

### 5.2 Migration Operations

The \textit{MDisconnect} schema models the disconnecting of a ‘migrating thread’ \textit{mt}? from its environment on the ‘from processor’ \textit{fp}? and the submission of \textit{mt}?’s register context and channel snapshot to Centrenet for transmission to the ‘to processor’ \textit{tp}?.

\begin{verbatim}
\textbf{MDisconnect} \\
\textbf{\Delta FState} \\
mt?): THREAD \\
fp?, tp?: PE \\

(fp?, mt?) ∈ ready \\
mt? ∈ \text{dom}(cntr) ⇒ \text{first}(cntr \ mt?) = 0 \\
(\text{let } sc == \{c : CHANNEL | mt? = sender \ fp? \ c\}; \\
rc == \{c : CHANNEL | mt? = receiver \ fp? \ c\} • \\
\text{let } scl == \{c : sc • (c, rloc \ fp? \ c)\}; \\
rc1 == \{c : rc • (c, slc \ fp? \ c, if \ fp? \ ⇒ c \in focus \ then \ 1 \ else \ 0)\} • \\
focus' = focus \setminus \{c : rc • \ fp? \ ⇒ c\} \land \\
\text{rloc}' = \text{rloc} \oplus \{c : rc • \ fp? \ ⇒ \{c \mapsto \text{tp}\}\}\land \\
cnet' = cnet \cup \{thrd(\text{tp}, \ mt?, \ slc, \ rcl)\})
\end{verbatim}

The schema has the precondition that the migrating thread must be in the ready queue and not, for instance, blocked on channel communication. It is
deemed too complicated to migrate threads that are part-way through communication or some other activity.

If the precondition holds then the two temporary variables \( sc \) (‘sending channels’) and \( rc \) (‘receiving channels’) are defined holding, respectively, all the channels which \( mt \) has sent and received over. The sending channels are then used to construct the ‘sending channels list’ \( scl \) which gives for each sending channel the last known location for its receiver.

The receiving channels are used to construct the ‘receiving channels list’ \( rcl \) which gives for each receiving channel the last known location of the sender and an indication of whether the channel focus is present. The focus for all receiving channels is forced to undefined. What this is doing, in effect, is to migrate the foci which are present with the thread. Finally, the receiver location is updated to point to the new site and the migration message is queued for transmission by Centrenet.

The \( MConnect \) schema models the (re)connecting of a thread \( mt \) at the destination site \( tp \). This involves moving \( mt \) from the Centrenet message and putting it in the ready queue and integrating the channel snapshot with the information already present in the CCBs.

The precondition for the schema is that a message of the appropriate form is found in the set \( cnct \). If the precondition holds, then the state component \( rloc \) is updated so that for all \( mt \)’s sending channels \( rloc \) now points to the processor specified in \( scl \) and for all \( mt \)’s receiving channels \( rloc \) points to \( mt \)’s new location, \( tp \).

The state component \( sloc \) is also updated for all \( mt \)’s receiving channels to point to the processor \( p \) defined in \( rcl \). The foci migrated with the thread are unpacked and attached to the appropriate channel by \( focus' = focus \cup \{ c : CHANNEL; p : PE \mid (c, p, 1) \in rcl \bullet tp \Rightarrow c \} \) and the used flag is set for all \( mt \)’s receiving channels. Finally, \( sender \) is updated to record the channels that \( mt \) sends on and \( receiver \) to record the channels \( mt \) receives on.
Chapter 5. Specification of Testbed Migration

MConnect

ΔFState

\( tp? : PE \)

\[ \exists mt : THREAD; \text{ scl : } \mathbb{P}(\text{CHANNEL} \times PE); \text{ rcl : } \mathbb{P}(\text{CHANNEL} \times PE \times \mathbb{N}) \]

\[ \text{thrd}(tp?, mt, \text{ scl}, \text{ rcl}) \in \text{ cnet } \land \]

\[ rloc' = rloc \oplus \]

\[ \{ c : \text{CHANNEL}; p : PE \mid (c, p) \in \text{ scl } \bullet tp? \mapsto \{ c \mapsto p\} \} \oplus \]

\[ \{ c : \text{CHANNEL}; p : PE; n : \mathbb{N} \mid (c, p, n) \in \text{ rcl } \bullet tp? \mapsto \{ c \mapsto tp?\} \} \land \]

\[ slc' = slc \oplus \{ c : \text{CHANNEL}; p : PE \mid (c, p, 1) \in \text{ rcl } \bullet tp? \mapsto \{ c \mapsto p\} \} \land \]

\[ focus' = focus \cup \{ c : \text{CHANNEL}; p : PE \mid (c, p, 1) \in \text{ rcl } \bullet tp? \mapsto c \} \land \]

\[ used' = used \cup \{ c : \text{CHANNEL}; p : PE; n : \mathbb{N} \mid (c, p, n) \in \text{ rcl } \bullet tp? \mapsto c \} \land \]

\[ sender' = sender \oplus \]

\[ \{ c : \text{CHANNEL}; p : PE \mid (c, p) \in \text{ scl } \bullet tp? \mapsto \{ c \mapsto mt\} \} \land \]

\[ rceiver' = receiver \oplus \]

\[ \{ c : \text{CHANNEL}; p : PE; n : \mathbb{N} \mid (c, p, n) \in \text{ rcl } \bullet tp? \mapsto \{ c \mapsto mt\} \} \land \]

\[ cnet' = cnet \setminus \{ \text{thrd}(tp?, mt, \text{ scl}, \text{ rcl}) \} \]
Chapter 6

Verification of Specification

This chapter has three sections and three proofs. The first two proofs assume that thread migration is not allowed and show that the Testbed implements the occam semantics for, respectively, synchronisation between parent and children threads and synchronisation between sender and receiver threads during channel communication. The third proof shows that the semantics for channel communication are preserved even when one of the threads using a channel undergoes migration between consecutive communications.

I do not provide any proofs about the unidirectionality and point-to-point property of Testbed channels because these properties are best enforced by the programmer and/or compiler. For practical reasons, the operating system does implement some checking of channels but, because complete checking would require the broadcasting of control messages to ensure (amongst other things) the global uniqueness of channels, the specification and proofs use an assumption that user programs never attempt to violate these properties.
6.1 Thread Synchronisation

This section begins with three assumptions and a lemma which rule out certain kinds of erroneous behaviour by threads. The proof that the Testbed implements correct synchronisation between parents and children proceeds by showing that the first synchronisation is correct and that an arbitrary synchronisation is correct provided that the immediately preceding synchronisation was correct.

6.1.1 Assumptions and lemmas

There are a number of conditions which are necessary for the correct functioning of the Testbed but which cannot be derived from the specification. The first two assumptions relate to the proper operation of the compiler and the third assumption to the expected operation of the Centrenet communication system. The lemma demonstrates that the specification meets the intended semantics for functions \textit{parent} and \textit{par\_bd} and is stated here to simplify the synchronisation proof.

\textbf{Assumption 1} A given thread may not be created more than once: it is not possible for the \textit{create(ct)} instruction to belong to two different programs or to occur at two different places in the same program.

\[\forall \text{FState}; pt_1, pt_2, ct_1 : \text{THREAD}; n_1, n_2 : \mathbb{N} \bullet
\text{program} pt_1 n_1 = \text{create}(ct_1) \land
\text{program} pt_2 n_2 = \text{create}(ct_1) \Rightarrow pt_1 = pt_2 \land n_1 = n_2\]

\textbf{Assumption 2} All programs have a terminate instruction and that instruction is always the last instruction in the program. All synchronise instructions are preceded by at least one create instruction. All pairs of synchronise instructions have at least one intervening create instruction. (These constraints should be satisfied by any competent compiler.)
\( \forall FState \cdot \forall p : \text{ran}(\text{program}) \cdot \\
(\exists 1 \ n : \mathbb{N} \cdot p \ n = \text{terminate} \land n = \#p) \land \\
(\forall n_1 : \mathbb{N} \cdot p \ n_1 = \text{sync} \Rightarrow \\
(\exists n_2 : 1 \ldots n_1; \ ct : \text{THREAD} \cdot p \ n_2 = \text{create}(ct))) \land \\
(\forall n_1, n_2 : \mathbb{N} \cdot p \ n_1 = \text{sync} \land p \ n_2 = \text{sync} \land n_2 > n_1 \Rightarrow \\
(\exists n_3 : n_1 \ldots n_2; \ ct : \text{THREAD} \cdot p \ n_3 = \text{create}(ct))) \)

**Assumption 3** CentreNet does not introduce spurious messages, or duplicate or lose messages. Additionally, CentreNet will always deliver messages within a finite time. The first statement need not be represented as it is a direct consequence of the use of the set \( cnet \) to model CentreNet. The second statement cannot be stated in terms of the specification because I have not formally developed any notion of time.

**Lemma 1** Where the functions \( \text{parent} \) and \( \text{par\_bd} \) are defined for a child thread \( ct \), \( \text{parent} \ ct \) always gives the thread that created \( ct \) and \( \text{par\_bd} \ ct \) always gives the processor on which the parent resides.

**Proof of lemma** I note that only \( TCreate \) and \( TTerminate \) change \( \text{parent} \), the former associates the parent thread with the child upon creation of that child and the latter removes any mappings from the child upon termination of that child. Similarly, only \( TCreate \) and \( TTerminate \) modify \( \text{par\_bd} \), the former associates the parent’s processor with the child upon creation of that child and the latter removes any mappings from the child upon its termination. The migration schema \( MDisconnect \) prohibits parents with children from migrating so the child/processor association remains constant during the child’s lifetime.

### 6.1.2 Synchronisation proof

I prove that when a thread requests to be suspended on completion of its children it will be removed from the ready queue at least until all of its children have terminated.
Chapter 6. Verification of Specification

The proof proceeds by as follows: the base case considers all state transitions from the creation of the parent thread up to the unblocking of the parent after its first synchronisation and the step case considers an arbitrary, subsequent synchronisation under the assumption that the immediately preceding synchronisation was correct.

Note that for concision the entire proof is universally quantified over \( FState \) and threads \( pt \) and \( ct \).

Synchronisation proof base case:

\[(\exists n : \mathbb{N} \cdot \text{trace } n = \text{sync}(pt)) \land \text{last trace } = \text{create}(pt, ct) \Rightarrow \]
\[(pt \in \text{ran parent } \Rightarrow pt \notin \text{ran ready})\]

The base case can be stated informally as follows: after the first synchronisation request and before the creation of another child, if the parent still has children then it cannot be in the ready queue.

\[
\text{last trace } = \text{create}(pt, ct) \Rightarrow ct \notin \text{dom cntr}
\]

The first step in the argument is to show that threads begin with no counter at all. When the first thread of the program is defined by \( TInit \), only the null thread has a counter and for each new thread \( ct \), \( TCreate \) ensures \( ct \) has no counter.

\[
\text{last trace } \in \{\text{sync}(pt), \text{terminate}(ct), \text{term}(ct)\} \Rightarrow \text{create}(pt, ct) \in \text{ran}(\text{trace})
\]

The first counter-modifying operation to befall the parent thread must be its first application of \( FCreate \). None of the other counter-modifying operations are possible: \( FSync \) cannot occur until after the first \( FCreate \) by Assumption 2; \( FTerminate \) only modifies the counter belonging to the parent of the terminating thread as defined by \( \text{parent ct} \) and our thread does not participate in the \( \text{parent} \) function until after its first \( FCreate \); likewise \( FTerm_{msg} \) only modifies the counter of the thread specified in the \( \text{term} \) message, the \( \text{term} \) message must have been generated by a preceding \( FTerminate \) since Assumption 3 prohibits
Chapter 6. Verification of Specification

Centrenet from inventing messages, and that preceding \textit{F}Terminate again uses \textit{parent} to define the thread in the \textit{term} message.

\[ \text{last trace} = \text{create}(pt, ct) \Rightarrow \text{cntr pt} = (1, \text{null\_thread}) \land pt = \text{parent ct} \]

Now that I have argued that threads begin with no counter and the first counter-modifying operation they perform is \textit{F}Create, I conclude that the first time a thread becomes a parent it receives a counter with a value of \((1, \text{null\_thread})\).

\[ \text{create}(pt, ct) \in \text{ran trace} \land \text{sync}(pt) \notin \text{ran trace} \Rightarrow \]
\[ \text{cntr pt} = (\#\{ t : \text{THREAD} | pt = \text{parent t} \} + \#\{ p : \text{PE} | \text{term}(p, pt) \in \text{cnet} \cdot \text{term}(p, pt) \}, \text{null\_thread}) \]

Between creating the first thread and requesting to synchronise, our thread has a counter of value \((n, \text{null\_thread})\) where \(n\) is the number of children yet to terminate plus the number of relevant \textit{term} messages in Centrenet. To support this statement, consider all the counter-modifying operations which may be applied between the first thread creation and synchronisation:

\textit{F}Create increments the counter because the predicate \(pt? \in \text{dom(cntr)}\) is true.

The schema also creates a thread so the first part of the counter still correctly represents the number of threads in the system plus the relevant \textit{term} messages.

\textit{F}Terminate either terminates the child and decrements the counter, or terminates the child, adds a \textit{term} message to Centrenet and does not change the counter. In both cases the first part of the counter reflects the number of children plus relevant \textit{term} messages and the second part of the counter is assigned the \textit{null\_thread} or retains its value.

\textit{F}Term\_msg consumes a \textit{term} message and decrements the counter to give it the value \((0, \text{null\_thread})\) or \((\text{first c} - 1, \text{second c})\)—in both cases the first part of the counter reflects the number of children plus relevant \textit{term} messages and the second part of the counter is assigned the \textit{null\_thread} or retains its value.
Hence, I conclude that between creating the first thread and requesting to synchronise our thread has a counter of the appropriate value.

\[
\text{last trace} = \text{sync} (pt) \Rightarrow \\
(\text{first}(cntr \ pt) = 0 \land pt \in \text{ran ready}) \lor \\
(\text{second}(cntr \ pt) = pt \land pt \notin \text{ran ready})
\]

When our parent thread requests to synchronise it will be returned to the ready queue immediately if there are no child threads still active or relevant term messages; otherwise it will be suspended. From the argument above, the value of our thread's counter immediately prior to the application of \text{FSync} is the number of children plus the relevant term messages. From inspection of \text{TSync} it is obvious that if there are no children and no relevant term messages, i.e. \text{first}(cntr \ pt?) = 0, then no action is taken, the counter is unchanged and our parent remains in the ready queue. Alternatively, if there are children and/or relevant term messages then the predicate \text{first}(cntr \ pt?) > 0 will be true and the counter will be updated by setting the thread value to our parent \text{pt} and our parent will be deleted from the ready queue. I note that if our parent is not suspended then its counter is not changed and has the value \((0, \text{null_thread})\).

\[
\text{cntr} \ pt = (n, pt) \Rightarrow n = \# \{ t : \text{THREAD} \mid pt = \text{parent} \ t \} + \\
\# \{ p : \text{PE} \mid \text{term}(p, pt) \in \text{cnet} \bullet \text{term}(p, pt) \}
\]

If our parent thread is blocked on synchronisation then each time one of its children terminates our parent's counter will be decremented or an appropriate term message produced. Additionally, our parent's counter will also be decremented each time an appropriate term message is consumed. Consider an application of \text{FTerminate} when one of our parent's children terminates: if the child resides on the same processor \((\text{par_bd ct} = p?)\) and it is the last child of the parent \((\text{first c} = 1)\) then the counter is reduced to zero, if it is not the last child then the counter is decremented to obtain the value \((\text{first c} - 1, \text{second c})\). If the child does not reside on the same processor then a term message is submitted for transmission by Centrenet specifying the parent thread and its processor board (by Lemma 1 this term message is directed at the appropriate parent at the appropriate processor). Consider an application of \text{FTerm_msg} and the consumption of a
term message directed at our parent: if the present counter has a numeric value of 1 then the counter is updated to \((0, \text{null\_thread})\) and if the present numeric value is greater than 1 then the counter is updated to \((\text{first } c - 1, \text{second } c)\). In both cases the numeric part of the counter is decremented by 1.

When a counter with a thread blocked on it is decremented to zero the thread is returned to the ready queue. This is justified by examination of \(TTerminate\) and \(TTerm\_msg\). When \(TTerminate\) decrements a counter of value \((1, pt)\) it updates \(\text{ready}\) to the value \(\text{ready}\{p? \leftarrow ct?\}\oplus\{p? \leftarrow pt\}\). When \(TTerm\_msg\) decrements a counter of value \((1, pt)\) it updates \(\text{ready}\) to the value \(\text{ready}\oplus\{p? \leftarrow pt\}\). Note that in both cases our parent’s counter is set to \((0, \text{null\_thread})\).

This completes the base case of the proof of the correctness of thread synchronisation.

**Synchronisation proof step case:** I noted for the base case that the parent’s counter is set to \((0, \text{null\_thread})\) when the parent is restarted after being blocked and I also noted that if the parent requests synchronisation, but is not blocked because there are no active children or relevant term messages, then its counter must also have the value \((0, \text{null\_thread})\). This counter value may therefore be assumed as the initial counter value for the step case.

The same arguments that were given for the base case now apply to the step case except I note that in the first application of \(FCreate\) the predicate \(pt? \in \text{dom}(cntr)\) is true and our parent’s counter is assigned the value \((\text{first}(cntr\ pt?) + 1, \text{null\_thread})\). Since I have just shown that the initial counter value may be assumed to be \((0, \text{null\_thread})\) the value of the counter assigned by the first application of \(FCreate\) is therefore \((1, \text{null\_thread})\), exactly as argued for the base case. This completes the argument for the step case. \(\square\)

**Corollary** Terminating children executing on the same processor as their parent decrement the parent’s counter as part of the \(TTerminate\) operation. Terminating children not sharing a processor with their parent cause a term message to be submitted to Centrenet as part of the \(TTerminate\) operation. By
Assumption 3 this message will be delivered within a finite amount of time to the parent's processor where the $TTerm_{-msg}$ operation will cause the parent's counter to be decremented. Hence, the suspended parent will be returned to the ready queue within a finite time from the termination of the last child.

The theorem and corollary complete the proof that the Testbed implements the $occam$ semantics for thread control.

6.2 Channel Synchronisation

This section is in two parts. An assumption is stated which constrains the sequences of channel operations allowed and a series of lemmas are presented which give the detailed working of the proof. The inductive proof for correct channel synchronisation is based on the lemmas and is given in the latter half of the section.

6.2.1 Assumptions and lemmas

It is, unfortunately, beyond the scope of this thesis to specify the behaviour of the Testbed channel communication protocol when subjected to sequences of communication requests contravening the $occam$ channel semantics. The restrictions on the sequences of communication requests required here are as follows:

\[
\forall \text{FState}; t_1 : \text{THREAD}; c : \text{CHANNEL}; m_1, m_2 : \text{MEM\_BLOCK} \bullet \\
\text{send}(c, m_1) \in \text{ran}(\text{program } t_1) \Rightarrow \text{receive}(c, m_2) \notin \text{ran}(\text{program } t_1) \land \\
(\forall t_2 : \text{THREAD} \setminus \{t_1\} \bullet \\
\text{send}(c, m_1) \in \text{ran}(\text{program } t_1) \Rightarrow \text{send}(c, m_2) \notin \text{ran}(\text{program } t_2) \land \\
\text{receive}(c, m_1) \in \text{ran}(\text{program } t_1) \Rightarrow \text{receive}(c, m_2) \notin \text{ran}(\text{program } t_2))
\]

Assumption 4 If a thread $t_1$ sends on a channel $c$ then it cannot also receive on $c$. If $t_1$ sends on $c$ then no other thread may send on the same channel, and if $t_1$ receives on $c$ then no other thread may receive on the same channel.
Lemmas 2 to 20 all have the same format. A new schema $Opn$ is defined in terms of one or more channel operations applied to a previously defined state, e.g. $Op4$ is the request-to-receive operation applied to the state which models a channel that has never been used. Next, a state schema $Sn$ is defined in which the relevant consequences of $Opn$ are shown, e.g. $S4$ states that $Op4$ causes the receiver to be blocked on the channel.

The $Sn$ state schemas all take the same input and output arguments, so the arguments have been packaged up into a schema called $Args$. Informally, $c?$ is the channel being communicated over, $st?$ and $rt?$ are the sender and receiver threads, $sp?$ and $rp?$ the processors on which the sender and receiver are executing and $msgs!$ contains any tuples to be submitted to Centrenet for transmission.

\[
\text{Args} \\
\begin{array}{l}
c? : \text{CHANNEL} \\
st?, rt? : \text{THREAD} \\
sp?, rp? : \text{PE} \\
msgs! : \mathbb{PCNETMSG}
\end{array}
\]

The lemmas are divided into those which relate to the proof base case and those which relate to the proof step case. The initial state for the base case is characterised by $SBase$. This initial state is a subset of $FInit$ and it shows that there is no sender or receiver, at any processor, blocked on the channel of interest and that there is no focus set for the channel.

\[
\text{SBase} \\
\text{FState}; \text{Args} \\
\forall p : \text{PE} \cdot c? \notin \text{dom(scontrol } p) \land c? \notin \text{dom(rcontrol } p) \land c? \notin \text{dom(rloc } p) \\
c? \notin \text{ran focus}
\]

The state after the first (and any subsequent) communication has completed, is characterised by $SStep$: no sender or receiver is blocked on the channel at any
processor, the channel focus is not set anywhere, the channel is marked as used at the receiver’s processor, the receiver location is known at both the sender’s and receiver’s processors to be \( r_p \) and the sender and receiver identifications are stored at the sender’s and receiver’s processors respectively.

\[
\begin{align*}
S_{\text{Step}} \quad & F_{\text{State}}; \text{Args} \\
\forall p : PE \bullet c? \notin \text{dom}(s\text{control} p) \land c? \notin \text{dom}(r\text{control} p) \\
c? \notin \text{ran focus} \land (r_p?, c?) \in \text{used} \\
(c?, st?) \in \text{sender} sp? \land (c?, rt?) \in \text{receiver} r_p? 
\end{align*}
\]

**The base case lemmas**

**Lemma 2** The \( O_{\text{p2}} \) operation models a sender thread \( st? \) executing on \( sp? \) requesting to send over a channel \( c? \). The previously defined state \( S_{\text{Base}} \) implies that this is the first-ever use of the channel. The \( S_2 \) schema shows the important consequences of applying \( O_{\text{p2}} \).

\[ O_{\text{p2}} \equiv S_{\text{Base}} \cdot C_{\text{Send}} \]

\[
\begin{align*}
\Delta F_{\text{State}}; \text{Args} \\
O_{\text{p2}} \Rightarrow st? \notin \text{ran ready'} \land \\
(\exists p : PE \bullet c? \in \text{dom}(s\text{control'} p)) \land (c?, st?) \in s\text{control'} sp? \land \\
(\forall p : PE \bullet c? \notin \text{dom}(r\text{control'} p)) \land \\
(\exists p : PE \bullet (p, c?) \in \text{focus'} \land p = sp?) \land \\
(rp?, c?) \notin \text{used'} \land \\
(c?, st?) \in \text{sender'} sp? \land \\
(\forall p : PE \setminus \{sp\} \bullet \text{advert}(p, c?, sp?) \in \text{msgs}!) 
\end{align*}
\]
The sender is removed from the ready queue \((st? \notin \text{ran } \text{ready}')\), the sender is blocked on the channel at \(sp?\) and only at \(sp?\) \(((\exists_1 p : PE \cdot c? \in \text{dom}(s\text{control}' p)) \land (c?, st?) \in s\text{control}' sp?)\), no receiver thread is blocked anywhere on the channel \((\forall p : PE \cdot c? \notin \text{dom}(r\text{control}' p))\), the channel focus is set at \(sp?\) and only at \(sp?\) \(((\exists_1 p : PE \cdot (p, c?) \in \text{focus}' \land p = sp?)\), the channel-used flag is not set at \(rp?\) \(((rp?, c?) \notin \text{used}')\), the sender thread is recorded as the legal sender for the channel at \(sp?\) \(((c?, st?) \in \text{sender}' sp?)\) and \text{advert} messages are queued for transmission by Centrenet to all processors other than \(sp?\) \(((\forall p : PE \setminus \{sp?\} \cdot \text{advert}(p, c?, sp?) \in \text{msgs}!))\).

The functions \text{ready}, \text{scontrol}, \text{focus} and \text{sender} are updated by \text{CSend} as is the output argument \text{msgs}!, note that \text{SBase} implies that there is no waiting receiver and that \text{rloc } sp? c? is not defined. Functions \text{rcontrol} and \text{used} are carried over from \text{SBase}.

**Lemma 3** The schema \text{S3} shows what happens when the first-ever receive request on a channel \(c?\) finds a sender already waiting.

\(Op3 \equiv S2 \cdot CR\text{Receive}\)

\[\text{S3}\]

\[\Delta F\text{State}; \text{Args}\]

\[sp? = rp? \land Op3 \Rightarrow st? \in \text{ran } \text{ready}' \land\]

\[\forall p : PE \cdot c? \notin \text{dom}(s\text{control}' p) \land c? \notin \text{dom}(r\text{control}' p) \land\]

\[c? \notin \text{ran } \text{focus}' \land (rp?, c?) \in \text{used}' \land\]

\[\text{rloc}' sp? c? = rp? \land \text{rloc}' rp? c? = rp? \land\]

\[(c?, st?) \in \text{sender}' sp? \land (c?, rt?) \in \text{receiver}' rp?\]

The sender is unblocked from the channel and returned to the ready queue, the receiver consumes the channel focus and sets the channel used flag, the receiver location is updated at \(rp?\) and therefore at \(sp?\) since they are the same processor, finally the receiver identification \((\text{receiver})\) is set—the sender identification carries over from \text{S2}. 
Lemma 4 The S4 schema models the first-ever receive request on a channel $c\?$: the receiver is removed from the ready queue and blocked on the channel, the receiver location is set at $rp\?$ (and therefore $sp\?$) and the receiver identification is recorded. The values of $scontrol$ and $focus$ carry over from $SBase$.

$$Op4 \equiv SBase \land CReceive$$

\[
\begin{align*}
\Delta FState; \text{Args} \\
sp? = rp? \land Op4 \Rightarrow rt? \notin \text{ran ready}' \land \\
(\forall p : PE \bullet c? \notin \text{dom}(scontrol' p)) \land \\
(\exists_1 p : PE \bullet c? \in \text{dom}(rcontrol' p)) \land (c?, rt?) \in rcontrol' rp? \land \\
c? \notin \text{ran focus}' \land \\
(c?, rt?) \in \text{receiver}' rp?
\end{align*}
\]

Lemma 5 The S5 schema gives the implications of a send after the first-ever receive on the same processor and for a channel $c\?$: the receiver is unblocked from the channel and returned to the ready queue thus $scontrol$ and $rcontrol$ are completely undefined for $c\?$, the channel used flag is set and the sender identification is set. The values of $focus$ and $rloc$ carry over from $S4$.

$$Op5 \equiv S4 \land CSend$$

\[
\begin{align*}
\Delta FState; \text{Args} \\
sp? = rp? \land Op5 \Rightarrow rt? \in \text{ran ready}' \land \\
(\forall p : PE \bullet c? \notin \text{dom}(scontrol' p) \land c? \notin \text{dom}(rcontrol' p)) \land \\
c? \notin \text{ran focus}' \land (rp?, c?) \in \text{used}' \land \\
(c?, st?) \in \text{sender}' sp? \land (c?, rt?) \in \text{receiver}' rp?
\end{align*}
\]
Lemma 6 The schema $S_6$ shows what happens when an advert message, emitted by the first-ever send on channel $c?$ in $S_2$, arrives at another processor: the sender location for the channel is set. All the other functions carry their values over from $S_2$.

$$Op_6 \equiv S_2 \triangleleft C_{Advert}$$

\[
\begin{align*}
\Delta FState; \text{Arg} \\
sp? \neq rp? \land Op_6 \Rightarrow (c?, sp?) \in sloc' rp? \land \\
(\exists p : PE \cdot c? \in \text{dom}(scontrol' p) \land (p, c?) \in focus' \land p = sp?) \land \\
(\forall p : PE \cdot c? \notin \text{dom}(rcontrol' p)) \land \\
(rp?, c?) \notin used' \land (c?, st?) \in \text{sender'} sp?
\end{align*}
\]

Lemma 7 The $S_7$ schema models the first-ever receive operation on a channel $c?$ for which an advert message has already been received: the receiver is removed from the ready queue and blocked on the channel, the receiver location is set at $rp?$, the receiver identification is set at $rp?$ and an $rtr$ message is queued for transmission back to the sender. The values of $scontrol$, $focus$ and $sender$ carry over from $S_6$.

$$Op_7 \equiv S_6 \triangleleft C_{Receive}$$

\[
\begin{align*}
\Delta FState; \text{Arg} \\
sp? \neq rp? \land Op_7 \Rightarrow rt? \notin \text{ran ready'} \land \\
(\exists p : PE \cdot c? \in \text{dom}(scontrol' p)) \land scontrol' sp? c? = st? \land \\
(\exists p : PE \cdot c? \in \text{dom}(rcontrol' p)) \land rt? = rcontrol' rp? c? \land \\
(\exists p : PE \cdot (p, c?) \in focus' \land p = sp?) \land \\
rloc' rp? c? = rp? \land \\
(c?, st?) \in \text{sender'} sp? \land (c?, rt?) \in \text{receiver'} rp? \land \\
rtr(sloc rp? c?, c?, rp?) \in \text{msgs!}
\end{align*}
\]
Lemma 8 The schema S8 shows what happens when an advert message, emitted by a sender in S2 during the first-ever communication over c?, is processed at a site where there is a waiting receiver: an rtr message is returned to the sender’s processor. The values of scontrol, focus and sender carry over from the application of CSend and the values of rcontrol, rloc and receiver carry over from the application of CReceive.

\[
Op8 \equiv (\text{CSend} ; \text{CReceive} ; \text{CAdvert}) \lor (\text{CReceive} ; \text{CSend} ; \text{CAdvert})
\]

\[
\Delta FState; \ Args
S2 \land Op8 \Rightarrow (\exists_1 p : PE \cdot c? \in \text{dom}(scontrol' p)) \land scontrol' \ sp? \ c? = st? \land
(\exists_1 p : PE \cdot c? \in \text{dom}(rcontrol' p)) \land rt? = rcontrol' \ rp? \ c? \land
(\exists_1 p : PE \cdot (p, c?) \in \text{focus'} \land p = sp?) \land rloc' \ rp? \ c? = rp? \land
(c?, st?) \in \text{sender'} \ sp? \land (c?, rt?) \in \text{receiver'} \ rp? \land
rtr(sloc \ rp? \ c?, sp?, c?) \in \text{msgs}
\]

Lemma 9 The S9 schema models an rtr message being processed at the sender’s processor during the first-ever communication over c?.

\[
Op9 \equiv (S7 \lor S8) \lor \text{CRtr}
\]

\[
\Delta FState; \ Args
st! : \text{THREAD}
sp? \neq rp? \land Op9 \Rightarrow st? = st! = scontrol \ sp? \ c? \land st? \in \text{ran ready'} \land
(\forall p : PE \cdot c? \notin \text{dom}(scontrol' p)) \land
(\exists_1 p : PE \cdot c? \in \text{dom}(rcontrol' p)) \land rt? = rcontrol' \ rp? \ c? \land
c? \notin \text{ran focus'} \land
rloc' sp? \ c? = rp? \land rloc' \ rp? \ c? = rp? \land
(c?, st?) \in \text{sender'} \ sp? \land (c?, rt?) \in \text{receiver'} \ rp? \land
msg(rp?, c?, sp?, buffer sp? c?) \in \text{msgs}
\]
The sender is unblocked from the channel and returned to the ready queue, the focus is consumed on the receiver’s behalf, the receiver location is set at \( sp? \) and a data message is queued for transmission to the receiver’s processor. The values of \( rcontrol, rloc \, rp? \), \( sender \) and \( receiver \) carry over from \( S7 \) or \( S8 \).

**Lemma 10** The \( S10 \) schema gives the implications of a data message being processed at a receiver’s processor during the first-ever communication over \( c? \): the receiver is unblocked from the channel (so \( c? \) is not defined anywhere in \( rcontrol \)) and returned to the ready queue and the channel used flag is set. The values of \( scontrol, focus, sender \) and \( receiver \) carry over from \( S9 \).

\[
\text{Op10} \equiv S9 \sqcup CMsg
\]

\[
\begin{array}{l}
\text{S10} \\
\Delta FState; \text{Args} \\
st!, rt! : \text{THREAD} \\
sp? \neq rp? \land \text{Op10} \Rightarrow rt? = rt! = rcontrol \, rp? \, c? \land \\\nrt? \in \text{ran \, ready'} \land \\\n(\forall p : PE \cdot c? \notin \text{dom}(scontrol' \, p) \land c? \notin \text{dom}(rcontrol' \, p)) \land \\\nc? \notin \text{ran \, focus'} \land (rp?, c?) \in \text{used'} \land \\\nrloc' \, sp? \, c? = rp? \land rloc' \, rp? \, c? = rp? \land \\\n(c?, st?) \in \text{sender'} \, sp? \land (c?, rt?) \in \text{receiver'} \, rp?
\end{array}
\]

**The step case lemmas**

**Lemma 11** The schema \( S11 \) shows what happens when a send request is made for a channel that has been used before and is therefore known to be local and to have no waiting receiver: the sender is removed from the ready queue and blocked on the channel (so there is now exactly one sender blocked on the channel), the channel focus is set and the sender identification is overwritten but, by Assumption 4, the new value must be the same as the old value. The functions \( rcontrol, rloc \) and \( receiver \) are carried over from \( S\text{Step} \).
Chapter 6. Verification of Specification

\[ Op11 \equiv S\text{Step} \ ; \ C\text{Send} \]

\[
\Delta F\text{State} ; \text{Args} \\
sp? = rp? \land Op11 \Rightarrow st? \notin \text{ran ready}' \land \\
(\exists p : PE \bullet c? \in \text{dom}(s\text{control}'p)) \land (c?, st?) \in s\text{control}' sp? \land \\
(\forall p : PE \bullet c? \notin \text{dom}(r\text{control}'p)) \land \\
(sp?, c?) \in \text{focus}' \land \\
r\text{loc}' sp? c? = rp? \land r\text{loc}' rp? c? = rp? \land \\
(c?, st?) \in s\text{ender}' sp? \land (c?, rt?) \in r\text{ceiver}' rp? 
\]

**Lemma 12** The \( S12 \) schema models a receive request at a local channel where no sender is waiting: the receiver is removed from the ready queue and blocked on the channel, the receiver location and identification are overwritten but not changed. The other functions carry over from \( S\text{Step} \).

\[ Op12 \equiv S\text{Step} \ ; \ C\text{Receive} \]

\[
\Delta F\text{State} ; \text{Args} \\
sp? = rp? \land Op12 \Rightarrow rt? \notin \text{ran ready}' \land \\
(\forall p : PE \bullet c? \notin \text{dom}(s\text{control}'p)) \land \\
(\exists p : PE \bullet c? \in \text{dom}(r\text{control}'p)) \land (c?, rt?) \in r\text{control}' rp? \land \\
c? \notin \text{ran focus}' \land \\
r\text{loc}' sp? c? = rp? \land r\text{loc}' rp? c? = rp? \land \\
(c?, st?) \in s\text{ender}' sp? \land (c?, rt?) \in r\text{ceiver}' rp? 
\]

**Lemma 13** As \( S13 \) shows, the implications of a send applied at a processor where the receiver is already waiting are the same as those in \( S5 \). The functions \( s\text{control}, focus, r\text{loc} \) and \( \text{receiver} \) keep their values from \( S12 \).
Lemma 14 The schema S14 shows that when a receive request finds a sender already waiting then the implications are the same as those in S3.

Lemma 15 The S15 schema models a send request on a channel that has been used before and is known to have a remote receiver.

The sender is deleted from the ready queue and blocked on the channel, the sender identification is updated (but not changed) and a message is queued for
transmission to convey the channel focus to the receiver's processor. The functions \( rcontrol \), \( focus \), \( rloc \) and \( receiver \) keep their values from \( SStep \).

**Lemma 16** The schema \( S16 \) shows what happens when a focus message, transmitted by a sender on a channel which has been used at least once before, arrives at the receiver's processor: the focus for the channel at \( rp? \) is set and the sender location is set at \( rp? \). The predicates on \( ready \), \( scontrol \), \( rcontrol \), \( sender \) and \( receiver \) are all derived from \( S15 \).

\[
O_{16} \equiv S15 ; CFocus
\]

**Lemma 17** The \( S17 \) schema models a receive request being applied for a channel known to be remote and for which the focus has already arrived.

\[
O_{17} \equiv S16 ; CReceive
\]
The receiver thread is deleted from the ready queue and blocked on the channel, the focus is consumed, the receiver location and identification are updated (but not changed) and an \( rtr \) message is returned to the sender.

**Lemma 18** The \( S18 \) schema gives the implications of a send and receive, in either order, on different processors: the sender and receiver are removed from their ready queues and blocked on the channel, the receiver location is overwritten (but not changed) at the receiver's processor, the sender and receiver identifications likewise, a message containing the channel focus is submitted at \( sp? \) for transmission to \( rp? \) but \( focus' \) retains its value from the schemas \( S12 \) and \( S15 \).

\[ Op18 \equiv (S15 \sqcup S12) \lor (S12 \sqcup S15) \]

\[
S18
\begin{array}{l}
\Delta FState; \text{Args} \\
sp? \neq rp? \land Op18 \Rightarrow st? \notin \text{ran \( \text{ready}' \land rt? \notin \text{ran \( \text{ready}' \land} \\
(\exists_1 p : \text{PE} \cdot c? \in \text{dom}(scontrol'p)) \land (c?, st?) \in scontrol' sp? \land \\
(\exists_1 p : \text{PE} \cdot c? \in \text{dom}(rcontrol'p)) \land (c?, rt?) \in rcontrol' rp? \land \\
c? \notin \text{ran \( focus' \land} \\
\text{rloc'} rp? c? = rp? \land \\
(c?, st?) \in \text{sender'} sp? \land (c?, rt?) \in \text{receiver'} rp? \land \\
\text{focusm}(rloc sp? c?, c?, sp?) \in \text{msgs!} \\
\end{array}
\]

**Lemma 19** The schema \( S19 \) shows the consequences of processing a message containing the channel focus at the receiver's processor: an \( rtr \) message is submitted at the receiver's processor for transmission to the sender's processor. The values of the other functions are carried over from \( S18 \).

\[ Op19 \equiv S18 \sqcup CFocus \]
Lemma 20 The $S20$ schema models the processing of an $rtr$ message at a sender's processor: the sender is unblocked from the channel and returned to the ready queue, the receiver location is overwritten (but not changed) and the data message queued for transmission to the receiver's processor.

$Op20 \equiv (S17 \lor S19) \circ CRtr$

6.2.2 Communication proof

Having stated the assumptions about the sequences of communication operations that the Testbed may be required to execute, and having given a list of lemmas demonstrating the effects of applying various operations to various states, I now prove that the lemmas include all possible state transitions and that all transitions representing a communication involve synchronisation between sender and receiver.

The proof base case assumes that the channel has never been used before (as modelled by $SBase$) and the step case assumes that at least one communication
has been completed (correctly) on the channel (as modelled by \textit{SStep}). Since channels operate independently from each other, the proof considers an arbitrary channel without loss of generality.

\textbf{Communication proof base case:} There are four sub-cases depending on whether the sender and receiver share a processor or not and depending on whether the sender requests to communicate first or second.

1. If the sender and receiver are executing on the same processor and the sender communicates first then Lemma 2 shows that the sender will be blocked. When the receiver requests communication, the situation is described by Lemma 3 which shows that the sender is restarted. The suspension of the sender implements synchronisation.

2. If the sender and receiver share a processor but the receiver requests to communicate first, then Lemma 4 shows that the receiver will be blocked. When the sender requests communication, the situation is described by Lemma 5 which shows that the receiver is restarted. The suspension of the receiver implements synchronisation.

3. If the sender and receiver do not share a processor then it does not matter which requests to communicate first but it does matter whether the \textit{advert} message issued by the sender is processed at the receiver's site before or after the receiver requests to communicate.

Suppose the advertisement arrives before the receiver requests to communicate. Lemma 6 shows that in processing the \textit{advert} message the location of the sender is stored so that in Lemma 7 the blocking receiver returns an \textit{rtr} to the sender's processor. Lemmas 9 and 10 then show the processing of the \textit{rtr} and the subsequent data message. Note that the sender is suspended until it receives the \textit{rtr} message and that the receiver is suspended until it receives the data message—this implements synchronisation.
4. Finally, suppose that the advertisement arrives after the receiver requests to communicate. Lemma 4 shows the receiver being blocked and Lemma 8 the processing of the *advert* message. The argument now continues as above with the processing of the *rtr* message.

**Communication proof step case:** I assume that all communications in the step case begin in the state characterised by *S*<sub>Step</sub> and I prove that they all end in state *S*<sub>Step</sub>. Note that all communications in the base case end in the *S*<sub>Step</sub> state (the post conditions of each of *S*<sub>3</sub>, *S*<sub>5</sub> and *S*<sub>10</sub> contain all the conditions of *S*<sub>Step</sub>). There are four sub-cases much as before.

1. If the sender and receiver are executing on the same processor and the sender communicates first then Lemma 11 shows that the sender will be blocked. When the receiver requests communication, the situation is described by Lemma 14 which shows that the sender is restarted. The suspension of the sender implements synchronisation. The post conditions of *S*<sub>14</sub> are a superset of *S*<sub>Step</sub>.

2. If the sender and receiver share a processor but the receiver requests to communicate first, then Lemma 12 shows that the receiver will be blocked. When the sender requests to communicate, the situation is described by Lemma 13 which shows that the receiver is restarted. The suspension of the receiver implements synchronisation. The postconditions of *S*<sub>13</sub> are a superset of *S*<sub>Step</sub>.

3. If the sender and receiver do not share a processor then it does not matter which requests to communicate first but it does matter whether the focus message issued by the sender is processed at the receiver's site before or after the receiver requests to communicate.

Suppose the focus arrives before the receiver requests to communicate. Lemma 16 shows that in processing the focus message the location of the sender is stored so that in Lemma 17 the blocking receiver returns an *rtr* to
the sender’s processor. Lemmas 20 and 10 then show the processing of the rtr and the subsequent data message. Note that the sender is suspended until it receives the rtr message and that the receiver is suspended until it receives the data message—this implements synchronisation. The postconditions of $S_{10}$ are a superset of $S_{Step}$.

4. Finally, suppose that the focus arrives after the receiver requests to communicate. Lemma 12 shows the receiver being blocked and Lemma 19 the processing of the focus message. The argument now continues as above with the processing of the rtr message.

This completes the proof. □

6.3 Transparency of Thread Migration

This section follows a similar format to the preceding one. A series of lemmas show, for different beginning states, what happens to channel communication when a thread migrates. The proof then demonstrates that all possible ‘before’ states have been considered and that in each case migration is transparent, i.e. the ‘after’ state of the operation is essentially the same as the before state.

The effects of migration are considered in relation to a single, arbitrarily chosen channel so the before states in the proof base case are a subset of those presented in Lemmas 2 to 20. In fact, the before states are a strict subset because migration can occur only if the migrating thread is in the ready queue (and not blocked on the channel). The before states are subject to schema renaming—the migration schemas presented below use the three input arguments $mt?$, $fp?$ and $tp?$ to represent the migrating thread, the ‘from processor’ and the ‘to processor’ respectively—so if the migrating thread is a sender for $c?$ then $mt?$ is identified with $st?$ and $fp?$ is identified with $sp?$, otherwise if the migrating thread is a receiver for $c?$ then $mt?$ is identified with $rt?$ and $fp?$ is identified with $rp?$.
6.3.1 The receiver location problem

The after states are subject to the same rules for renaming and, as will be shown, are identical in all relevant aspects to the corresponding before states. In most cases it is obvious that the functions in the after state have the same value as they did in the before state. The receiver location $rloc$, however, is updated in a more complicated fashion and therefore requires some further explanation.

For the first communication over a channel, senders locate the appropriate receiver by broadcasting their presence to all processors in the system. For reasons of efficiency, second and subsequent communications rely on the sender being able to transmit the channel focus more directly to the receiver. When migration is disallowed, the sender simply remembers the last location of the receiver and sends the focus there—this is how $rloc$ has been used in the lemmas up to this point.

However, when migration is allowed, a receiver may migrate from processor to processor between two communications and thus cause the sender to transmit its focus to the wrong place. The solution employed on the Testbed is to have receivers leave a forwarding address behind them every time that they migrate. Now, when a sender transmits a focus to an old address the focus can be forwarded as many times as is necessary until it catches up with the receiver. The receiver location $rloc$ is used to encode these forwarding addresses as shown in Figure 6-1.

To make it easier to compare $rloc$ functions in the before and after states I define a new predicate called $\text{linked}$. A focus message from a sender on processor $sp?$ for a channel $c?$ will reach a receiver on $rp?$ if the receiver’s processor is linked to the sender’s processor, i.e. if $rp? \in \text{linked}(rloc, sp?, c?)$.

\[
\text{linked} : ((PE \rightarrow (CHANNEL \rightarrow PE)) \times PE \times CHANNEL) \rightarrow \mathbb{P} PE
\]

\[
\forall r : PE \rightarrow (CHANNEL \rightarrow PE); p : PE; c : CHANNEL \bullet
\]

\[
\text{linked}(r, p, c) = \text{if } c \in \text{dom}(r p) \text{ then } \{r p c\} \cup \text{linked}(r, r p c, c) \text{ else } \emptyset
\]
The next group of lemmas follow the format already established: a new schema $Opn$ is defined in terms of $MConnect$ and $MDisconnect$ applied to a previously defined state and a state schema $Sn$ is defined in which the relevant consequences of $Opn$ are shown.

**Lemma 21** The $Op2i$ operation models the disconnection of a migrating thread $mt?_t$ at the from processor $fp?_t$ and its reconnection at the to processor $tp?_t$ in the situation where $mt?_t$ is a sender for $c?_t$ and the (local or remote) receiver thread has not yet requested to communicate.

$$Op21 \equiv SStep[mt?_t/st?_t,fp?_t/sp?_t] ; MDisconnect ; MConnect$$
\[ S21 \]

\[ \Delta FState; \text{Args} \]

\[ mt? : \text{THREAD} \]

\[ fp?, tp? : \text{PE} \]

\[ Op21 \Rightarrow (\forall p : \text{PE} \bullet c? \notin \text{dom}(scontrol' p) \land c? \notin \text{dom}(rcontrol' p)) \land \]
\[ c? \notin \text{ran focus'} \land (rp?, c?) \in \text{used'} \land \]
\[ rloc' tp? c? = rp? \land rloc' rp? c? = rp? \land \]
\[ (c?, mt?) \in \text{sender'} tp? \land (c?, rt?) \in \text{receiver'} rp? \]

Neither migrate schema in \( Op21 \) changes \( scontrol \) or \( rcontrol \), so they maintain their values from \( SStep \). The channel \( c? \) is in the sets named \( sc \) and \( scl \) but not in \( rc \) or \( rcl \) so the functions \( focus \) and \( used \) are not updated with regard to \( c? \). Function \( rloc \) is not updated at \( fp? \) (because \( M\text{Disconnect} \) does not find \( c? \) in \( rc \) but is updated at \( tp? \) (because \( M\text{Connect} \) finds \( c? \) in \( scl \) by \( tp? \mapsto \{ c \mapsto p \} \)). If \( tp? \) and \( rp? \) are the same processor then, as asserted by \( S21 \), \( rloc' tp? c? = rp? \) and (trivially) \( rloc' rp? c? = rp? \). Otherwise, if \( tp? \neq rp? \) the value of \( rloc rp? c? \) is not changed by application of \( M\text{Connect} \). The function \( \text{sender} \) is updated at \( tp? \) in the correct way, \( \text{receiver} \) retains its original value.

**Lemma 22** The \( Op22 \) operation models the migration of a thread \( mt? \) in the situation where \( mt? \) is a sender for \( c? \) and the receiver thread is currently blocked, waiting to receive.

The migrate operation \( Op22 \) does not update \( rcontrol \) or \( scontrol \) so their values are maintained from \( S12 \). The channel \( c? \) is in the sets \( sc \) and \( scl \) but not in \( rc \) or \( rcl \) so the function \( focus \) is not updated with regard to \( c? \). \( M\text{Connect} \) ensures that the sender’s new processor knows \( rloc tp? c? = rp? \) and, if \( rp? = tp? \) this ensures \( rloc rp? c? = rp? \) as well—otherwise, \( rloc rp? \) retains its value from \( S12 \). The function \( \text{sender} \) is updated at \( tp? \) in the correct way, \( \text{receiver} \) retains its original value.

\[ Op22 \equiv S12[mt?/st?, fp?/sp?] ; M\text{Disconnect} ; M\text{Connect} \]
Lemma 23 The $Op_{23}$ operation models the migration of a thread $mt?$ in the situation where $mt?$ is a receiver for $c?$ and a local sender is currently blocked, waiting to communicate on the channel.

$Op_{23} \equiv S_{11}[mt?/rt?, fp?/rp?] ; MDisconnect ; MConnect$

The migrate operation $Op_{23}$ does not update $rcontrol$ or $scontrol$ so their values are maintained from $S_{11}$. The channel $c?$ is in the sets $rc$ and $rcl$ but not in $sc$ or $scl$ so the focus is removed at processor $fp?$ and created again at $tp?$. The receiver location at $tp?$ is updated by $MConnect$ to be $tp?$. The function $receiver$ is updated at $tp?$ in the correct way, $sender$ retains its original value.
Lemma 24 The $Op24$ operation models the migration of a thread $mt?$ in the situation where $mt?$ is a receiver for $c?$, the remote sender has already requested to communicate but the focus message has not arrived.

$Op24 \equiv S15[mt?/rt?, fp?/rp?]; MDisconnect \; ; MConnect$

<table>
<thead>
<tr>
<th>$S24$</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Delta FState; Args</td>
</tr>
<tr>
<td>$mt?$ : THREAD</td>
</tr>
<tr>
<td>$fp?, tp?$ : PE</td>
</tr>
</tbody>
</table>

$Op24 \Rightarrow (\exists p : PE \bullet c? \in \text{dom}(scontrol' p)) \land (c?, st?) \in scontrol' sp? \land$

$(\forall p : PE \bullet c? \notin \text{dom}(rcontrol' p)) \land$

$rloc' tp? \; c? = tp? \land$

$(c?, st?) \in sender' sp? \land (c?, mt?) \in receiver' tp? \land$

$(\exists p : PE \bullet focusm(p, c?, sp?) \in msgs! \land fp? \in \text{linked}(rloc, p, c?))$

The migrate operation $Op24$ does not update $scontrol$ or $rcontrol$ so their values are maintained from $S15$. The channel $c?$ is in the sets $rc$ and $rcl$ but not in $sc$ or $scl$ so the receiver location recorded at $tp?$ is set to $tp?$ by $MConnect$. As before, $receiver$ is updated by the appropriate value at $tp?$ and $sender$ is left unchanged. The focus message is still waiting to be delivered, but to the processor that the receiver has just left. However, the $MDisconnect$ operation sets $rloc fp? c?$ to $tp?$ which, when $CFocus$ is applied at $fp?$, will cause the first implication to be true and the focus message to be forwarded to $tp?$.

Lemma 25 The $Op25$ operation models the migration of a thread $mt?$ in the situation where $mt?$ is a receiver for $c?$, the remote sender has already requested to communicate and the focus message has arrived.

$Op25 \equiv S16[mt?/rt?, fp?/rp?]; MDisconnect \; ; MConnect$
The migrate operation $Op25$ does not update $scontrol$ or $rcontrol$ so their values are maintained from $S16$. The channel $c?$ is in the sets $rc$ and $rcl$ but not in $sc$ or $scl$ so the focus is deleted at $fp?$ and restored at $tp?$. As before, $receiver$ is updated by the appropriate value at $tp?$ and $sender$ is left unchanged.

6.3.3 Migration proof

I now show the transparency of migration with respect to communication by demonstrating that the lemmas presented above exhaust all the situations in which migration may occur and by showing that the postconditions of the lemma state schemas are essentially the same as the before state of the lemma operations. The proof base case models the first migration after a communication on $c?$ and the step case models the second or subsequent migration after the last communication on $c?$.

It is assumed that no operations occur on the channel in question between the application of $MDisconnect$ and the application of $MConnect$—although this is not true for the Testbed the proof can be extended to cover this case relatively easily.

Migration base case: There are a number of sub-cases to consider depending on whether $mt?$ is a sender or a receiver for the channel and whether the other user of the channel has requested to communicate yet or not.
1. If $mt?$ is the sender for $c?$ then there are two situations depending on whether the receiver has requested to communicate yet.

(a) If the receiver has not yet requested to communicate then the state of $c?$ is described by $SS\text{tep}$ and the effect of the migration operation by $Op\text{21}$ in Lemma 21. It does not matter where the receiver is located. The postconditions of $S\text{21}$ contain the conditions of $SS\text{tep}[mt?/st?, tp?/sp?]$ so in this case migration is transparent.

(b) If $mt?$ is the sender for $c?$ and the receiver has already requested to communicate then the state of $c?$ is described by $S\text{12}$ and the effect of the migration operation $Op\text{22}$ by Lemma 22. It does not matter where the receiver is located. The postconditions of $S\text{22}$ contain the conditions of $S\text{12}$ so the migration is transparent.

2. If $mt?$ is the receiver for $c?$ then there are three situations depending on whether the sender is local or not and for remote senders whether the focus message has arrived or not.

(a) If the sender is local and has already requested to communicate then the state of $c?$ is described by $S\text{11}$ and the effect of the migration operation by $Op\text{23}$ in Lemma 23. The postconditions of $S\text{23}$ do not contain the conditions of $S\text{11}$ but they do contain the conditions of $S\text{16}$, and $S\text{16}$ represents essentially the same point in the communication protocol, so the migration is transparent. (In $S\text{16}$ there is a focus at the receiver's processor so the receiver can deduce that there is a waiting sender and the sender location points to the sender's processor so that an $rtr$ message can be sent to the correct location.)

(b) If $mt?$ is the receiver for $c?$ and the sender is not local, has already requested to communicate but its focus message has not arrived, then the state of $c?$ is described by $S\text{15}$ and the effect of the migration operation by $Op\text{24}$ in Lemma 24. The postconditions of $S\text{24}$ contain the conditions of $S\text{15}$ modified so that the focus message is being sent indirectly to the receiver.
(c) If \( mt? \) is the receiver for \( c? \) and the sender is not local, has already requested to communicate and its focus message has arrived, then the state of \( c? \) is described by \( S16 \) and the effect of the migration operation by \( Op25 \) in Lemma 25. The postconditions of \( S25 \) contain the conditions of \( S16 \) so migration is transparent.

**Migration step case:** The step case states that the above arguments also hold for the second or any subsequent migration. This completes the proof. □

**Conclusion**

These proofs have shown that the specification of the Testbed given in Chapter 4 correctly implements the **occam** semantics for thread synchronisation between parent and child threads, and between sending and receiving threads. Building on this foundation, it has been relatively easy to show that the extensions in Chapter 5 to support thread migration preserve the semantics for synchronisation between sending and receiving threads. It is relevant to note here that the specification and proofs would be much more complicated were it not possible to depend on the **static** nature of **occam** (augmented with some assumptions about the 'good behaviour' of user programs).

Now that most of the important mechanisms in the Testbed’s operating system have been formally specified and informally verified it is possible for measurements of the Testbed’s performance to be made and expressed in a meaningful and unambiguous way. As will be seen in the next chapter, not only can such measurements be related to particular stages of the synchronisation protocols but dependencies can be identified between the overall performance of the Testbed and certain key design requirements as expressed in the specification.
Chapter 7

Performance Profiling

This chapter describes a series of experiments in which the performance of the Testbed hardware and operating system is measured. The experiments are conducted using the Testbed's own monitoring hardware, for although the monitor is mainly intended for profiling user programs it is equally suitable for profiling hardware and system software. The aim of the experiments is threefold.

1. To establish some basic parameters for use in load balancing: the cost of local (intra-processor) communication, the cost of remote (inter-processor) communication and the cost of thread migration.

2. To examine the efficiency of the operating system's communication and migration protocols. Each phase of the protocols is considered independently and assessed according to the average time it takes to complete.

3. To parameterise the Testbed. This will provide the basis for future work on comparing the Testbed with other multicomputer systems and possibly input for models or simulations of Testbed-like systems. Considering the value of such parameterisations it is a pity that this kind of data is available for so few computers.

The experiments are carried out by constructing a number of small test programs, executing them on the Testbed and collecting their event traces from
the monitoring hardware. To ensure that the results are realistic, the experiments are repeated several times with different levels of background load. The experimental data is subjected to a variety of analyses using a range of Unix tools and the results are presented in graphical form. A series of finite state machines is also presented so that the results can be interpreted in terms of the formal specification presented in Chapters 4 and 5.

The rest of this chapter is organised as follows: after a brief note on terminology I list the experiments to be performed and give a short description of the test programs used. I then present a model of execution for the main loop of TOS and four finite state machine models of execution for the communication and migration protocols. The main part of this chapter concerns the experimental results and their interpretation. Finally, a section of conclusions is presented.

Terminology

In the following sections it is necessary to distinguish occam Channel Messages (OCMs) from Message Protocol Components (MPCs). OCMs are communicated over channels by occam processes—they are typed sequences of data to be shared. MPCs are operating system objects used to implement reliable communication of OCMs over Centrenet. All three kinds of MPC, the focus token, the ready-to-receive indication (rtr) and the actual message data (msg) are required to implement the communication of a single OCM.

7.1 The Experiments to be Performed

The experiments fall naturally into seven groups and are listed below in the same order that their results are presented in Section 7.3.

1. The minimum time required to complete the local communication of an OCM is measured for a range of message sizes.
2. The minimum time required to complete each step in the protocol for remote communication is measured for a range of message sizes. Remote communication is more complicated than local communication and each communication requires several different operating system services and the transfer of several different MPCs over Centrenet.

3. The minimum time required to complete each step in the protocol for thread migration. These times are independent of the particular thread and processors involved.

4. The minimum time required to complete each step in the protocol for the transfer of memory pages between processors. This protocol is invoked each time a migrated thread requires a page of memory resident on another processor.

5. Up to this point, all experiments are performed on a quiet machine and represent 'best case' results. Now, the effects of a busy environment on local and remote communication, thread migration and remote page copying are examined.

6. The range of behaviours that might be expected from a 'typical' user program are measured.

7. The final group of experiments are somewhat different in nature to the others and involve measuring the maximum throughput of the Testbed's monitoring system itself.

**7.1.1 The test programs**

My experiments were performed using five small C programs (the listings are in Appendix A): **local, remote, migrate, multi-phase, and overflow**.

The **local** and **remote** programs each have two communicating threads. During the execution of these programs, the size of the messages communicated increases linearly from one byte to 5,000 bytes. The difference between the
programs is that local's threads reside on the same processor whilst remote's threads execute in parallel on different processors. I use these programs to study the latency of different phases in the communication protocols.

The third program, migrate, comprises two threads which communicate intermittently over an occam channel. The first thread executes exclusively on one processor whilst the second migrates to a different processor after every communication. I use the migrate program to measure the latency of different phases of the migration protocol and to measure the time it takes to copy pages of memory.

Program four, multi-phase, comprises a number of threads which are distributed over the slave processors and which pass through a number of different phases to simulate a wide variety of dynamic behaviour.

Finally, the overflow program is used to measure the maximum throughput of the event monitoring hardware. A single thread executes on one of the slave processors and generates a large number of events. The program terminates after a fixed time period or when the monitor's event buffers overflow, whichever occurs first.

7.2 Execution Models

7.2.1 The TOS main loop

The main program loop of TOS is described in order to assist the interpretation of the experimental results later on. After reset or power-up each Testbed processor completes an initialisation sequence and then enters the main operating system loop. This loop has four phases.

1. In the send phase the transmit queue is checked for outgoing messages. Any messages found are passed to the Centrenet module for transmission.
2. During the *receive phase* the incoming message queue is checked to see whether Centrenet has received any messages from other processors—if it has then the new messages are processed.

3. The *schedule phase* involves scheduling the thread at the front of the ready queue. This thread then has complete control of the CPU until it makes a call to the operating system or until a timer interrupt signals the end of the time-slice.

4. In the *service phase* the operating system performs the service requested by the thread, assuming the thread was not pre-empted, and either returns the user thread to the ready queue or blocks it on the appropriate resource.

The loop now repeats from phase 1. The timer interrupts, which may occur during any phase, are used not just for pre-empting threads but for ensuring that important housekeeping tasks, such as updating the real time system clock, are performed regularly.

The phases in the main loop may be related to the *Receive* and *Schedule* computation steps described on page 88. Phase 2 corresponds to a series of *Receive* computation steps as both model the consumption of incoming messages. Phase 3 is not modelled by any computation step as the specification conveniently ignores the fact that user threads may perform their own computation—as far as the specification is concerned user threads only exist to call system services. Phase 4 corresponds to a single application of the *Schedule* computation step as the operating system meets the service request. Phase 1, in which messages waiting to be sent are passed to the Centrenet controller, does not have a corresponding computation step since the specification does not explicitly model the inter-processor communication subsystem.

### 7.2.2 The communication and migration protocols

Several steps are required to obtain the experimental results. In the first step a test program is executed on the Testbed. Then, the events are collected and
sorted into a single trace file. In the final step a range of tools are used to analyse the trace file and to extract the appropriate times of interest. This section explains how the last step is done by describing the relationship between the actions performed by the operating system during communication, thread migration or page copying, and the events found in the event trace.

To help with the explanation, four finite state machines (FSMs) are presented: two for the local and remote communication protocols and one each for the thread migration and page copying protocols. In these FSMs, vertices correspond to operating system states and arcs correspond to the events emitted as the operating system moves from state to state. The FSMs are decorated with dashed lines connecting the arcs and these lines are labelled with one of the phases of the communication, migration or page copying protocol depicted by the machine. Thus, the relationships between sequences of events and phases in the protocols are made explicit. The FSMs also have a direct interpretation in terms of the formal Z specification of the operating system and this is explained for each machine.

Local communication

![Figure 7-1: The FSM for local communication.]

Figure 7-1 shows how the local communication protocol can be divided into four phases. The vertices and arcs are informally interpreted as follows:

**Vertex 1:** Initially, no communication is in progress.
Chapter 7. Performance Profiling

Vertex 2: The operating system has been called because a sender thread (or alternatively a receiver thread) wishes to communicate. The TRAP7 event signals the transition from user to system mode.

Vertex 3: The emission of the TRAP70 event signals that the operating system has finished servicing the thread's communication request and has blocked the thread. Other user threads will now be executed.

Vertex 4: When the receiver thread (or alternatively the sender thread) also requests to communicate another TRAP7 is emitted as the operating system begins to service its request. Once the data has been copied from the sender to the receiver a TRAP72 is emitted and vertex 1 is reentered.

Now consider the dashed arcs in the FSM which represent the different phases in the protocol. The durations of each phase, or times of interest, are:

*Time to service the sender's (receiver's) request*: this is the time that the operating system spends processing the first thread's request.

*Time to service the receiver's (sender's) request*: the time the operating system spends processing the second thread's request, including the time to copy the message from the sender's memory area to the receiver's memory area.

*Total communication time*: this is the time between the first communication request and both threads being released back into the ready queue after a successful communication.

There is also a correspondence between transitions in the FSM and the application of the schemas presented in Chapter 4. (Appendix B contains an index of all globally declared schemas, relations and types used in the specification.) Assuming that the sender thread communicates first, the transition from vertex 1 via vertex 2 to vertex 3 is modelled by the application of the $FSend$ schema and the transition from vertex 3 via vertex 4 to vertex 1 is modelled by $FReceive$. Alternatively, assuming that the receiver communicates first, the transition from vertex 1 to vertex 3 is modelled by $FReceive$ and the transition from vertex 3 back to vertex 1 by $FSend$. 
Remote communication

The protocol for remote communication has more phases and so the FSM presented in Figure 7–2 is more complicated than the previous FSM. The upper half of the FSM (vertices 1 to 4) depicts state transitions of the processor on which the sender thread is executing. The lower half of the FSM (vertices 5 to 13) shows what happens at the receiver's processor. The dashed lines between the two halves correspond to Centrenet messages being exchanged between the two processors.

I will not describe the numbered vertices this time. Refer instead to Table 7–1 for an interpretation of the events. The times of interest indicated by the dashed lines in the FSM are described as follows:

*Sndr rq comms:* (arc 1/2 to arc 2/3) this is the time the operating system spent servicing the sender’s request to communicate.

*Rcvr rq comms:* (arc 5/6 to 6/7 and 12/13 to 13/9) the time the operating system spent servicing the receiver’s request to communicate.
<table>
<thead>
<tr>
<th>Event</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAP7</td>
<td>Thread requests to communicate</td>
</tr>
<tr>
<td>TRAP70</td>
<td>Thread is blocked after request has been serviced</td>
</tr>
<tr>
<td>RX_MPC</td>
<td>Begin servicing an MPC (Message Protocol Component)</td>
</tr>
<tr>
<td>RX_FOCUS</td>
<td>MPC received was a <strong>focus</strong> token</td>
</tr>
<tr>
<td>COMMS</td>
<td>Sender is returned to ready queue</td>
</tr>
<tr>
<td>COMMR</td>
<td>Receiver is returned to ready queue</td>
</tr>
</tbody>
</table>

Table 7-1: Events emitted during the remote communication protocol.

**FOCUS flight:** (arc 2/3 to 5/11 or 7/8) the time between the sender’s processor finishing with the sender’s request and the receiver’s processor beginning to deal with an incoming **focus**.

**RTR flight:** (arc 8/9 or 13/9 to 3/4) the time from the receiver’s processor having both a **focus** token and a waiting receiver, to the sender’s processor beginning to deal with the incoming **rtr**.

**MSG flight:** (arc 4/1 to 9/10) the time from the sender’s processor finishing with an incoming **rtr** to the receiver’s processor beginning to deal with an incoming **msg**.

**FOCUS service:** (arc 7/8 to 8/9 or 5/11 to 11/12) the time taken to service a **focus** token, including the time to generate an **rtr** message if appropriate.

**RTR service:** (arc 3/4 to 4/1) the time taken to service an **rtr** indication, including the time to package up a **msg** (this involves copying the message data) and return the sender to the ready queue.

**MSG service:** (arc 9/10 to 10/5) the time taken to unpack message data from the Centrenet buffers and release the receiver back into the ready queue.

When the Testbed executes an operating system service as part of the remote communication protocol, it emits one event before beginning the service and another event after finishing the service. Since each such service is modelled by a **Z** schema, and since the FSM for remote communication has one transition per
event, it follows that each Z schema corresponds to a pair of FSM transitions. These correspondences are shown in Table 7-2.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Schema</th>
<th>Transition</th>
<th>Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 via 2 to 3</td>
<td>FSend</td>
<td>7 via 8 to 9</td>
<td>FFocus</td>
</tr>
<tr>
<td>5 via 6 to 7</td>
<td>FReceive</td>
<td>3 via 4 to 1</td>
<td>FRtr</td>
</tr>
<tr>
<td>12 via 13 to 9</td>
<td>FReceive</td>
<td>9 via 10 to 5</td>
<td>FMsg</td>
</tr>
<tr>
<td>5 via 11 to 12</td>
<td>FFocus</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7-2: Correspondence between state transitions in the FSM for remote communication and the Z schemas in Chapter 4.

Thread migration

Figure 7-3: The FSM for thread migration.

Figure 7-3 shows the protocol for thread migration. The times of interest are:

**Packing time**: the time spent by the operating system locating the thread to be migrated, removing it from the ready queue, packing its context, channel and memory page information into a Centrenet message and deleting the local copy of the thread’s context.

**Thread flight time**: this is the time between the sender processor queuing the packed thread for transmission and the receiver processor starting to process the incoming thread message.
Chapter 7. Performance Profiling

Unpacking time: the time taken for a receiver processor to unpack the thread context, update the channel and memory page tables and prepare the thread for execution.

The correspondence between transitions in the FSM and the application of the schemas presented in Chapter 5 are as follows: the FSM transition from 1 via 2 to 3 is modelled by the \textit{MDisconnect} schema; the transition from 3 via 4 to 1 is modelled by \textit{MConnect}.

Copying remote pages

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fsm_remote_copying}
\caption{The FSM for remote page copying.}
\end{figure}

Figure 7–4 shows the protocol for copying remote pages. The times of interest are:

\textit{Process bus error}: the time for the operating system to service the bus error, locate the missing page, submit a remote page request to Centrenet and block the faulting thread.

\textit{Page request flight time}: this is the time between one processor blocking a thread on a page and another processor starting to process the page request.

\textit{Page flight time}: the time for the missing page to be located, packed into a message, transferred over Centrenet and the blocked thread restarted.

The formal \textit{Z} specification does not model the copying of pages between processors so there are no schemas relating to this FSM.
7.3 Results of the Experiments

7.3.1 Latency of local communication

Figure 7-5 shows the time TOS spends servicing the sender thread’s request to communicate. One hundred different message lengths are tried and each message length is sent several times. In this experiment the receiver always blocks before the sender, so copying of the message from the sender’s memory area to the receiver’s memory area occurs during the servicing of the sender’s request. This is why the service time increases almost linearly with the message length.

![Figure 7-5](image)

**Figure 7-5:** *Time taken to service a local send request, copy the message to the waiting receiver’s memory area and restart both threads.*

For most message lengths the times measured fall into two groups—hence the double-line effect—the longer times occurring when the operating system service is interrupted by the system timer. When the data is ‘corrected’ by subtracting time spent servicing the timer interrupt, Figure 7-6 is produced. This new figure shows an otherwise linear graph superimposed with a step function. A detailed
investigation of the relevant operating system routines shows that these steps occur when the message copying function has to cross the boundary between consecutive pages in the sender's memory area or the receiver's memory area, and indeed the graph shows pairs of steps occurring at intervals of 1024 bytes, the size of a memory page.

Figure 7-6: Upper graph: time taken to service a local send request corrected by subtracting time spent servicing timer interrupts. Lower graph: time taken to service a receive request and block thread.

Figure 7-6 also shows the time that the Testbed operating system spends servicing the receiver thread’s requests to communicate. As mentioned above, the receiver always requests communication first, there is no message copy involved, and thus servicing the request takes constant time.

It is a stated aim of these experiments to examine the efficiency of the operating system’s communication protocols. I observe that the time to service a communication request is either constant or linear with message size and therefore at most a constant factor of improvement could be made to the operating system efficiency.
7.3.2 Latency of remote communication

Figure 7–7 shows how long TOS spent servicing remote communication requests made by user threads. These requests are broken down into two types: requests made by sender threads and requests made by receivers. In the experiment, fifty different message sizes were tried, but each size was sent only once so the timer interrupts cause just two upward blips rather than the line-doubling effect seen before. With the small scale of this graph, it is now possible to see that the extra delay introduced by a timer interrupt is approximately 0.25ms.

![Graph showing latency of remote communication](image)

**Figure 7–7:** Upper graph: time taken to service a send request over a remote channel. Lower graph: time taken to service a receive request over a remote channel.

The time to service a remote send or receive request is constant for all OCM lengths at about 1.2ms for sends and 0.45ms for receives. This is because no copying of the message occurs during this phase of the remote communication protocol. As the service times are constant with message length it is seen that again, the operating system efficiency can be improved by, at most, a constant factor.
Servicing MPC messages

Figure 7-8 shows how long TOS spent servicing three of the different kinds of MPC involved in implementing the occam channel protocol: focus, rtr and msg (advert MPCs are ignored). These graphs also have timer interrupt blips although, because of the scale, these are less easy to see.

Servicing a focus is simply a matter of returning an rtr message to indicate that the receiver is ready; hence the zero gradient. Servicing an rtr involves packing up the OCM data to be sent so the time increases with OCM length. Likewise, servicing a msg involves unpacking the OCM data and the time is therefore proportional to the OCM length. This copying of the message twice is unavoidable on the current Testbed hardware. The rtr and msg graphs are different because they are implemented using different routines in the operating system.

Once again, all graphs are linear or constant.
Flight times

Figure 7-9 shows how long focus, rtr and msg MPCs spend ‘in flight’. Notionally, the flight time is the time spent by the MPC flying (with Centrenet’s help) from one processor to another but, as will be seen shortly, other factors may contribute significantly to the flight time. Considering the flight times for focus and rtr messages first, these are fairly constant at around 1.5ms. This is expected since focus and rtr MPCs always comprise six bytes, regardless of OCM length.

![Graph showing flight times](image)

**Figure 7-9:** Upper graph: the flight time for a msg being sent over Centrenet. Centre graph: rtr flight time. Lower graph: focus flight time.

The msg graph appears at first to suggest that some interesting interactions are occurring in the experiment. However, the curious behaviour is actually caused by a quite harmless relationship between the time at which the rtr message arrives and the time the subsequently-released sender computes before being preempted by the timer interrupt.

1. The time at which the rtr message is delivered to the sender’s processor is essentially asynchronous with that processor’s activity.
2. With reference to Section 7.2.1, the rtr is processed during the receive phase, a msg is queued for transmission and the sender is restarted after a constant delay. In all likelihood, the Centrenet transmission hardware will be idle and the msg will remain queued while the schedule phase occurs.

3. Any time up to 20ms later, the timer will interrupt and the sender thread, which in this experiment is compute-bound, will be preempted. The Centrenet transmission hardware will be restarted in the following send phase and at last the msg will commence its flight.

![Figure 7-10](image)

**Figure 7-10**: *Upper graph: the time between the operating system queuing the msg for transmission and the transmit queue being submitted to Centrenet. Lower graph: pure flight time, i.e. the difference between the msg flight time as shown in previous figure and the upper graph.*

Figure 7-10 summaries the situation: the upper graph shows the variability (of up to one 20ms time-slice) in the delay between steps 2 and 3 above; the lower graph shows that the pure msg flight time is much more predictable—in fact it is linear with the message size. Hence, in this experiment, the observed
behaviour is determined, quite innocently, by the delay between the transmission of the \textit{rtr} and the next timer interrupt on the sender's processor.

**Time for a complete remote communication**

The total delays experienced by the sending and receiving threads are shown in Figure 7-11. These times are derived from the preceding graphs and show the minimum time that a sender or receiver will be delayed on a quiet machine waiting for remote communication to complete. The variation in the receiver delay stems from the variation in the \textit{msg} flight time, explained above.

![Graph](image)

**Figure 7-11:** \textit{Upper graph: the total time by which the receiver is delayed during communication. Lower graph: the total time by which the sender is delayed.}

### 7.3.3 Thread migration latency

In this section the latencies associated with thread migration are measured. Figure 7-12 shows that the time spent packing a thread and its channel and memory page information into a Centrenet message is constant at around 3.4ms
and that the flight time and unpacking time are virtually the same as each other at approximately 2.5ms. As usual, the data is disturbed by 0.25ms blips caused by timer interrupts.

![Graph](image.png)

**Figure 7-12:** Upper graph: time spent by operating system packing thread context into a Centrenet message buffer. Lower two graphs: flight time for thread message superimposed on the time spent by receiving operating system unpacking thread.

### 7.3.4 Remote page copy latency

In addition to the packing, flight and unpacking latencies associated with thread migration, further costs are incurred each time the migrated thread faults on a remote memory page. The costs of copying a page of memory between processors are shown in Figure 7-13.

The time delay between a user thread faulting on a page and the page request being sent over Centrenet is constant at around 2ms. Page requests are queued for transmission during the operating system's service phase. This phase is closely
Figure 7-13: Upper graph: flight time for a page being copied between processors. Middle graph: flight time for a page request. Lower graph: time between a user thread faulting on a page and the transmission of the page request over Centrenet.

followed by the send phase, so page requests are transmitted by Centrenet fairly promptly.

The page request flight time is more variable because page requests may be delivered at any point in the operating system’s main loop and will not be processed (and the appropriate page queued) until the operating system has reached the receive phase.

The page flight time is the most variable of all because pages queued for transmission in the receive phase may very well not be sent until the next send phase and the time required for the intervening schedule phase may be as much as a time-slice. Additionally, when the page arrives at the requesting site some time may elapse before it is processed.

The page flight latency can be compared with the time required to send a 1K message by remote communication. The operation of locating a page and packing
it into a Centrenet message is similar to the operation of servicing an *rtr* message, roughly 4.5ms according to Figure 7-8. The 'pure' flight time for a remote page corresponds to the pure flight time for a message and is approximately 3ms (Figure 7-10). The reception and unpacking of a page corresponds to the processing of a *msg*, which takes about 3.5ms according to Figure 7-8. Thus, the total time required for these three stages of the remote communication protocol is 11ms, which is close to the average page flight time of 12ms.

7.3.5 The effects of a busy environment

This section describes how the times produced in the foregoing experiments are affected when the Testbed is executing background programs which compete for CPU cycles and Centrenet bandwidth.

Local communication

The time taken to service requests for local communication is not affected by increased background loading because the operating system cannot be preempted by user threads—Figure 7-5 is still an accurate representation. Similarly, local communication does not make any use of Centrenet and is therefore unaffected by increases in Centrenet traffic.

The total delay, however, as experienced by the first partner in any particular local communication, may be increased. This happens when other threads are scheduled after the first partner requests to communicate and before the second partner requests to communicate. The size of the delay depends on how many intervening threads are scheduled and the total number of CPU cycles that they consume.

The first partner in the communication will suffer additional delays because the amount of housekeeping that the operating system has to perform generally increases with the load.
Remote communication

Although the times to service communication requests and to process MPCs remain unchanged, the latencies of all other remote communication actions increase with background load. Threads which compete for CPU cycles extend the time between communication requests from the first and second communicating partners. Threads which compete for communication bandwidth extend the MPC flight times. If both processors hosting the communicating threads are compute bound, MPCs which arrive during the schedule phase may be held up for an additional time-slice (plus the service phase). In the worst case, since each OCM requires three MPCs, the remote communication protocol may be extended by more than three time-slices.

Thread migration and remote page copying

In the worst case, unpacking of a migrating thread which arrives during the schedule phase may be delayed by a time-slice plus the time required for the service phase plus the time required to process other outstanding Centrenet messages. When the background load is high, the new thread will also spend longer waiting for remote memory pages to be delivered.

7.3.6 Typical communication latencies

The experiments so far have measured the best case latencies for local and remote communication, thread migration and remote page copying. The qualitative effects of increasing the background load have also been discussed. In this section, a test program called multi-phase is executed to explore the range of typical communication latencies that might be experienced by a user program. The test program has eight distinct phases (Table 7-3) to simulate compute and communication bound programs, programs with different balances of local and remote communication and programs with different average message sizes. While multi-phase is not intended to model any particular parallel program, it does model the extremes of behaviour possible that bound the ‘typical’ case.
1: compute-bound local comms short messages
2: " " local comms long messages
3: " " remote comms short messages
4: " " remote comms long messages
5: communication-bound local comms short messages
6: " " local comms long messages
7: " " remote comms short messages
8: " " remote comms long messages

Table 7-3: Phases of the multi-phase test program.

Figure 7-14: Mapping of threads to processors and channels to pairs of threads in the multi-phase test program. The bold arrows represent channels for remote communication, the normal arrows represent channels for local communication.
The *multi-phase* program is listed in Appendix A and Figure 7-14 shows how threads are allocated to processors and the pattern of channels between threads. Note that the five slave processors are also known by the names *earth*, *air*, *fire*, *water* and *space*. Thread 1 controls the transition between phases by using barrier synchronisation as follows.

- Threads 10 to 14 require two communications on channels 0 to 4 before a remote communication phase can start using the 'horizontal' channels.
- Threads 10, 20, 30, 40 and 50 require two communications on channels 5 to 9 before a local communication phase can start using the 'vertical' channels.
- Thread 1 computes for a second between pairs of communications, thus ensuring that one second elapses between the end of each phase and the beginning of the next.

The effect of the phases on the ready queue lengths of the five slave processors can be seen in Figure 7-15. During each phase, the relative amounts of computation and communication govern the areas under the curves. Between phases (around 3, 5.5, 8, 11, 13, 15 and 18.5 seconds from the beginning of the execution) only the thread with ID 1 is executing.

The effect of the phases can also be seen in terms of the frequency with which local or remote communications are requested (Figure 7-16). The first four phases are predominantly compute-bound so the communication frequency is low. Of the communication-bound phases, 5 and 6 involve low-delay local communication and so have the highest frequency. The same number of communication requests are made in the last four phases, but since remote communication takes longer the phases are longer and the frequency is lower.

The communication latency for senders during each phase of the test program is shown in Figure 7-17 (it is assumed that the receiver latencies are similar). The latencies are obviously longer in the remote communication phases as compared to the local communication phases.
Figure 7-15: Length of the ready queue on each of the five slave processors during an execution of the multi-phase program.

Figure 7-16: Number of send requests per 50ms during the program execution.
Figure 7-17: Time to service a send request during the program execution.

Figure 7-18: The dashed lines (there are only three, between 0ms and 3ms) show the distribution of (local) communication latencies during phase 6 of the test program multi-phase. The un-dashed lines show 95% of the distribution of (remote) communication latencies during phase 8 of the test program.
To achieve a greater level of detail, local communication phase 6 and remote communication phase 8 are selected and the distribution of communication latencies calculated, the results being shown in Figure 7-18. Note that in order to obtain reasonable scaling factors the distribution for phase 6 actually goes off the top of the graph in one place and only 95% of the available data is shown for phase 8.

As might be expected, local communication is typically of low, predictable latency (between 1ms and 3ms) while remote communication takes a longer and much more variable time (between 10ms and 130ms).

7.3.7 Event rates

This section seeks to establish the maximum data rates that can be supported by the monitoring system. While Imre [44] presents theoretical results, the experiments in this section demonstrate that the monitoring system suffers a bottleneck which reduces the maximum data rates dramatically. Figure 7-19 shows the pathways along which monitoring events flow and the components that process events.

---

Figure 7-19: The event collection and processing pathways.
Events are generated by the five slave processors, collected and buffered by the dedicated monitoring hardware and delivered to the master processor. Software executing on the master processor interprets the events and either logs them to a file for later analysis or, when load balancing is enabled, forms an assessment of the load, identifies imbalances and issues migration instructions to the slave processors.

As indicated in Chapter 3, approximately 1.5% of the operating system is instrumented with event generating instructions. As the Slave Monitoring Interfaces (SMIs) are capable of handling at least one event every two processor clock cycles, the slave processor to SMI interface cannot be a bottleneck.

The Master Monitoring Interface (MMI) collects one event time-stamp and data pair from each SMI every 36\(\mu s\). If the performance of each slave processor is 1.2 MIPS and 1.5% of its instructions generate events then each slave can produce 18,000 events per second. Since the MMI can absorb about 28,000 events per SMI per second the SMI to MMI interface cannot be a bottleneck.

The rate at which the master processor reads events from the MMI is slightly more difficult to determine. However, the minimum set of activities that would have to be performed by a logger or load balancing program includes:

1. Polling the memory mapped hardware until a valid SMI number is returned, then reading the event time-stamp or data value.

2. Selecting and updating the appropriate data structures according to the number of the SMI and whether the value read was a time-stamp or event data.

3. In the case that both the time-stamp and event data are now available for a given SMI, carrying out some action such as logging the event to a file or checking for a load imbalance.

Obviously, the amount of work done by the master when it reads an event from the MMI is orders of magnitude greater than the single machine instruction
required by a slave to generate an event. The monitoring bottleneck is, therefore, located in the MMI to master processor interface.

The test program overflow (listed in Appendix A) is used to quantify the master processor's sustainable event rate. The program executes on a slave processor and causes events to be emitted at a rate set by the experimenter. A simple logger program executes on the master processor, collecting the events as they are generated. The test program is executed several times to find the event rate at which the monitoring hardware overflows and events are lost.

- In one experiment the test program was executed for 30,012ms without losing any events. Analysis of the event trace showed that 33900 time-stamp and event data pairs were collected. Therefore, on average the program had emitted 1129 pairs per second.

- In another experiment the test program was executed for 21,508ms after which events were lost because the monitoring hardware buffers were full. Analysis of the event trace showed that 33903 time-stamp and event data pairs were collected, showing that on average the program had emitted 1576 pairs per second.

The conclusion is that the logger program has a sustainable event rate in the range 1129 to 1576 time-stamp and event data pairs per second, or 226 to 315 pairs per slave per second.

The Testbed's sustainable event rate is about one fifth of NETMON-II's (described on page 27) and about one tenth of the TMP's (page 27). However, whereas the Testbed has one monitoring processor for all the slave processors, NETMON-II and TMP have one monitoring processor for each slave processor. The Testbed is also slowed down by the fact that the MMI delivers an interleaved stream of events from different SMIs—separating these streams out and finding a total time ordering of events proves to be quite a time-consuming task.
7.4 Conclusions

The latencies measured for the main operating system services are summarised in Table 7-4. The average best-case latency and bounds are given, along with the number of the figure in which average and bounds are depicted graphically. It should be noted that these accurate and detailed results could not have been obtained without the Testbed’s dedicated monitoring hardware and its global, high-precision clock. Although few performance results are given in the literature, and those that are given are not defined as precisely as those for the Testbed, Table 7-5 shows how the Testbed’s performance compares with some other concurrent computer systems’ performance.

The presentation of the Testbed’s operating system latencies satisfies two of the three aims of this chapter—to establish some basic parameters for use in load balancing and to enable future work simulating the Testbed or comparing it with other multicomputers. The other aim stated at the beginning of this chapter was to examine the efficiency of TOS’s communication and migration protocols. In all cases, the communication latencies increase linearly with message size and the migration and page copying latencies are constant. Two ways of reducing these latencies are:

• By optimising the software, e.g. through use of a better compiler or by hand-coding in assembler.

• By upgrading the hardware, e.g. to use a faster clock or to enable Direct Memory Access (DMA) in hardware to all areas of RAM rather than requiring messages to be copied in and out of Centrenet buffers.

The proofs presented in Chapter 6 strongly suggest that the communication protocols cannot be simplified without compromising the transparency of process migration.
### Table 7-4: Summary of Testbed latencies for communication, migration and remote paging services. (For communication, the latency is shown for the sender of an n-Kbyte message).

<table>
<thead>
<tr>
<th>Service</th>
<th>Average time (ms)</th>
<th>Bounds (ms)</th>
<th>Figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local communication</td>
<td>1.95 * n + 1.025</td>
<td>±0.275</td>
<td>7-20</td>
</tr>
<tr>
<td>Remote communication</td>
<td>2.77 * n + 7.35</td>
<td>±0.35</td>
<td>7-21</td>
</tr>
<tr>
<td>Thread migration</td>
<td>8.4</td>
<td>±0.5</td>
<td>7-22</td>
</tr>
<tr>
<td>Remote page copy</td>
<td>16.9</td>
<td>±15.3 −3</td>
<td>7-23</td>
</tr>
</tbody>
</table>

### Table 7-5: A selection of operating system latencies from other concurrent computer projects.

<table>
<thead>
<tr>
<th>Project</th>
<th>Service</th>
<th>Average time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charlotte (page 25)</td>
<td>Local communication (1 byte)</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Remote communication (1 byte)</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Migrate process (32K)</td>
<td>240</td>
</tr>
<tr>
<td>Emerald (page 26)</td>
<td>Migrate process (600 bytes)</td>
<td>40</td>
</tr>
<tr>
<td>MMK (page 27)</td>
<td>Local communication (5K)</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Remote communication (5K)</td>
<td>56</td>
</tr>
<tr>
<td>Sprite (page 26)</td>
<td>Remote communication (5K)</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Migrate Unix process</td>
<td>300</td>
</tr>
</tbody>
</table>
Figure 7–20: The graph showing the latency for a single local send is trapped between lines $y = 1.95 \times 10^{-3} \times x + 1.5$ and $y = 1.95 \times 10^{-3} \times x + 0.8$.

Figure 7–21: The graph showing the latency for a single remote send is trapped between lines $y = 2.77 \times 10^{-3} \times x + 7.7$ and $y = 2.77 \times 10^{-3} \times x + 7$. 
Figure 7–22: The graph showing the latency for thread migration is trapped between lines \( y = 7.9 \) and \( y = 8.9 \).

Figure 7–23: The graph showing the latency for remote page copy is trapped between lines \( y = 13.9 \) and \( y = 32.2 \) with an average of 16.9.
Chapter 8

Load Balancing

This chapter looks at the main issues in dynamic load balancing, describes the system implemented on the Testbed and reviews its effectiveness. I begin by proposing a characterisation of 'balanced' load and, by means of a case study, consider the best way of selecting a heuristic or 'load metric'. I give the rationale behind the Testbed's strategy for load reconfiguration, not relying on ad hoc assumptions but making use of the results obtained in Chapter 7. Then I measure the speedups obtained by balancing some typical parallel programs and, finally, I draw some general conclusions for the design of future dynamic load balancing systems.

8.1 Assessing the Load

As was described in Section 2.3, scheduling has three phases and the first, reconnaissance phase involves assessing the load currently experienced by each processor and forming this into a coherent summary for use in the second, decision-making phase. In this section I propose an informal characterisation of what it means for the Testbed's load to be balanced. This characterisation then guides the selection of a suitable 'coherent summary' which in turn suggests exactly what aspects of the load need to be measured. The system components are shown in Figure 8-1.
Chapter 8. Load Balancing

Figure 8–1: The key agents in a dynamic load balancing system and the information exchanged between them.

8.1.1 A characterisation for ‘balanced load’

Virtually all research to date has concentrated on making the best use of CPU resources—indeed for most this has been the exclusive aim. Consider the example depicted in Figure 8–2. If threads \( t_1 \) and \( t_2 \) are compute bound whilst \( t_3 \) and \( t_4 \) spend most of their time communicating remotely, then processor \( P_2 \) may have spare capacity while \( t_3 \) and \( t_4 \) wait for their message transfers to complete. A much better arrangement is depicted in Figure 8–3 where the compute-bound

Figure 8–2: Compute-bound threads \( t_1 \) and \( t_2 \) execute on processor \( p_1 \) while communication-bound threads \( t_3 \) and \( t_4 \) execute on processor \( p_2 \).
Figure 8–3: One compute-bound and one communication-bound thread execute on each processor.

threads will absorb all spare processor capacity whilst the communication-bound threads suffer at most by a delay of one time-slice per communication.

An example of load balancing to optimise processor resources can be found in Bryant and Finkel [14]. They estimate the remaining execution time for each candidate task, measure the response time for a notional average task on each processor and assign the candidate job to the processor which will give it the most appropriate service.

A small number of researchers have proposed that other system resources should be taken into account when balancing the load. Messina [67] indicates that memory subsystems are still very slow in comparison with processor speeds and that therefore memory use should be measured—indeed it can be imagined that a processor with ‘too many’ different user tasks may spend substantial amounts of time dealing with page faults. Bemmerl et al [5] suggest that communication latency is important and reason that user tasks will not be fully able to make use of processor resources if they are held up waiting for messages to be transferred. To illustrate this point, consider Figure 8–4. Suppose that $t_1$ is communicating with $t_3$ and $t_2$ is communicating with $t_4$: all threads suffer slow communication. If two threads are migrated to achieve the situation in Figure 8–5 then all threads are now communicating locally—a much better situation.

For the purposes of implementing a load balancer for the Testbed, I propose that a balanced load occurs when no processor has a ‘great deal’ more work in its ready queue than the average and when no pair of processors communicates over
the interconnect a 'great deal' more often than the average. Of course, quantifying a 'great deal' is not straightforward, as will be described later. This characterisation allows me to explore the difficult problem that occurs with particular programs where reducing inter-processor communication increases the imbalance in processor work and vice versa. Unfortunately, the simultaneous optimisation of a larger set of resources is beyond the scope of this thesis.

This notion of a balanced load depends on an commonly-used assumption which Casavant and Kuhl [17] summarise as, ‘...the philosophy that being fair to the hardware resources of the system is good for the users of that system’. The alternative to this philosophy is to seek to optimise each task—an attractive idea but for two drawbacks. Firstly, it presupposes that what is best for the individual task is best for all tasks collectively. Secondly, and more importantly with respect to an implementation on the Testbed, optimising individual tasks requires a much greater amount of monitoring data to be gathered.
8.1.2 Selection factors for load metrics

The next step is to select one or more heuristics or load metrics which will generate the raw data from which a coherent load summary can be produced. The characterisation of a balanced load, as provided by the preceding section, is necessary before this selection process can proceed, but it is not sufficient. This section therefore reviews other selection factors as suggested in the literature and, in a series of case studies, applies some of these factors to four candidate load metrics.

Bearing in mind what Ni et al [78] have to say on the matter, 'the estimation of processor load is a difficult problem for which no completely satisfactory solution exists', I have collected a number of selection factors from the literature and from Kremien and Kramer [50] in particular:

Cost —the additional time, space or hardware required to collect the metric and process it into the 'coherent summary'. Eager et al [26] state that, 'the value of a policy depends critically on the overhead required to administer it' and this will be seen shortly when I demonstrate that some load metrics can easily flood the monitoring system. A particular problem to be avoided is metrics with collection and/or processing costs which rise with load.

Relevance —the closeness of the relationship between the metric and the resource being optimised. Simpler relationships are better because they give more confidence that the load balancing will work under a range of conditions. A good metric should reflect any spare capacity as well as indicating the current load. Without sufficient care, relevant metrics are often costly to collect and inexpensive metrics often lack relevance.

Timeliness —the metric should be an up-to-date reflection of the load.

Effectiveness/hit ratio —the effectiveness is the observed change in performance and the hit ratio is the number of migrations which turn out to improve the balance versus the number of migrations which make the balance worse.
A bad decision not only fails to gain an improvement, it also imposes the additional costs of the migration.

**Scalability** — the effectiveness of the metric should not decrease too quickly as the number of processors being balanced increases. Metrics which are not tied to particular topologies or communication speeds are to be preferred.

**Generality and adaptability** — the metric should not be limited to balancing only a restricted set of programs and it should be effective over a wide range of rates of change in load balance.

These selection factors are considered, metric by metric, in the case studies presented in the next section.

### 8.1.3 Metric case studies

A wide range of load metrics are suggested in the literature as being of value. At a crude level, the ratio of 'system time' to 'user time' gives an indication of the nature of a processor's load. Research on 'program tuning', such as that in Haban and Wybranietz [33], suggests the time spent by tasks waiting for external events is important, e.g. the time spent waiting for synchronisation and/or communication to complete. More ambitiously, the time-varying patterns of communication may be revealing, whether gathered on a per-processor, per-task or per-channel basis. Wang and Morris [107] suggest the Q-factor which measures how closely the behaviour of a particular load balancing algorithm emulates that of an ideal first-come-first-serve global scheduler. Artsy and Finkel [1] collect a whole battery of statistics which they claim are, 'comprehensive enough to support most conceivable policies'.

I have chosen four metrics from the literature, metrics which can be measured conveniently on the Testbed and which would seem to give a good indication of the load. Each metric is made the subject of a case study in which it is tested against the selection factors presented above and in which use is made of practical experimentation on the Testbed.
N-THREADS The number of threads created on, or migrated to, a particular processor which have not terminated or migrated to another processor.

RQ-LENGTH The number of threads in the ready queue. Another per-processor metric.

CPU-TIME The proportion of time for which user threads hold the CPU. This is calculated per processor as an average over a fixed time period.

REM-COMM The amount of remote communications during a fixed time period per link (a link is an unordered pair of processors).

The four metrics were chosen to make collection on the Testbed architecture as simple as possible. Support for doing this comes from two sources: Kunz [52] and Eager et al [26], all of whom found that balancing policies based on single load metrics were as good as policies based on multiple, combined metrics. Research from the related field of profiling and visualisation, such as that reported by LeBlanc and Mellor-Crummey [55], suggests that program optimisations tend to be either very simple and automatable or very complex and require multiple program views to identify. The latter, complex optimisations are certainly beyond the scope of this thesis (and are not, perhaps, feasible in real time).

The test program

The Testbed's dedicated hardware monitoring system provides an accurate and detailed method for testing the load metrics against the cost, relevance, and generality and adaptability selection factors. The test program used for this purpose is multi-phase which was introduced in Section 7.3.6. The other selection factors (timeliness, effectiveness and scalability) are not measured experimentally but are discussed in detail.

The cost of a metric is determined objectively by counting the number of event packets emitted per time unit and, more subjectively, by an assessment of the ease with which a coherent load summary can be constructed. Execution of the
multi-phase program on an otherwise idle machine produces a load which is predictable and the relevance of the metric can be assessed by comparing the load it indicates with the expected load.

Generality and adaptability of each candidate metric is assessed by considering the aspects of program behaviour most likely to influence the effectiveness of the metric and by designing the test program multi-phase so that every combination of behaviours is generated. The aspects of behaviour most likely to be of significance are:

1. Whether the threads are predominantly compute or communication bound.
2. Whether most communication is remote, i.e. occurs between threads on different processors, or local, i.e. occurs between threads on the same processor.
3. Whether communications are of very large or very small messages.

Consequently, the test program has eight phases as illustrated in Table 7–3 (page 151). The same test program is used to test each of the four metrics but different sections of the instrumented operating system are enabled (as described in Section 3.2.3) to generate the appropriate events.

Results for N-THREADS metric

The number of threads on each processor is measured simply by instrumenting the appropriate operating system routines so that a 'create' event is emitted every time a new thread is created or arrives during migration and a 'delete' event is emitted every time a thread terminates or is migrated to another processor. The monitoring program keeps one counter per processor and increments or decrements them as appropriate. For the purposes of this experiment each change in a counter is logged to a file along with the time at which the event was issued and the file is then used to produce the graph shown in Figure 8–6.
Figure 8-6: N-THREADS—This graph shows, for each of the five processors, the number of threads executing on that processor during the execution of the test program multi-phase.

Measuring the number of threads on each processor seems attractively simple. However, when the selection factors listed in Section 8.1.2 are considered the metric is seen to have some serious defects which limit its usefulness for load balancing.

On the plus side, the measurement is inexpensive to make (examination of the log file showed that about 100 events were emitted during the 33 seconds of execution or 3 per second) since the Testbed threads are of medium grain and are generally not created, migrated or destroyed so frequently. The timeliness of the metric is very good since the events are issued immediately and require little processing. The metric is independent of processor topology and communication latency, so it is scalable.

On the minus side, the N-THREADS metric lacks relevance (and therefore is unlikely to be effective) because it does not indicate anything about competition for CPU cycles or communication bandwidth. In comparison with Figure 7-15 on page 153, for instance, it is impossible to identify the program phases or,
indeed, the periods between phases when only a single thread is executing on the earth processor.

Results for RQ-LENGTH metric

The lengths of the processor ready queues are determined by instrumenting the operating system routines that add and delete threads from the ready queue—an event sequence of unit length is sufficient to indicate whether the queue has grown by one or shrunk by one. Threads enter and leave the ready queue only when synchronising with external events, for example during communication—they do not leave the queue when they are scheduled on the CPU. As before, the monitoring program simply maintains one counter for each processor and records changes in a log file which is then used to produce Figure 8–7.

Figure 8–7: RQ-LENGTH—This graph shows, for each of the five processors, the length of the ready queue during the program execution.

The cost of collection is greater than before (the test program generated 16800 events during its execution time of 33 seconds, an average of about 500 packets a second) although the cost of processing each packet is no higher than before.
The event rate varies according to how frequently threads lose the CPU and are blocked on synchronisation, communication or memory pages and how frequently threads are pre-empted and remain in the ready queue.

The RQ-LENGTH metric is clearly more relevant than before, giving a picture of activity that corresponds closely to what would be expected from the test program's source code. In particular the greater competition for CPU cycles during the first four compute-bound phases is obvious. It is interesting to note that the metric does not indicate the speed at which the ready queue is moving—this may be important because, for example, a long but fast moving queue (i.e. most threads relinquish the CPU before being pre-empted) may still be acceptable for a thread that is not strongly compute bound.

Although the last four communication-bound phases appear different from the earlier, compute-bound phases, the metric does not give any direct information about the pattern of communication and it alone would not be of much use to a balancing strategy that attempted to reduce inter-processor communication.

As before, however, the metric has good timeliness and good scalability.

Results for CPU-TIME metric

The average time that user threads spend on the CPU can be calculated by issuing time-stamped events every time a user thread is given the CPU by the scheduler and every time a user thread loses the CPU because its time-slice is over or because it has requested a system service. The monitor program simply logs the time and nature of the events (schedule or deschedule). Calculation of the times for which the CPU was held by a user thread is carried out as a separate exercise after the test program has terminated because the event data rates are so high.

The metric can be defined as the percentage of a fixed time interval for which the CPU was held by a user thread, so in each fixed time period the individual times for which the CPU was held must be added up and the result divided by the time period itself. Selecting a time period of 20ms I have experimented
Figure 8–8: *CPU-TIME*—Graph is plotted every 20ms showing the percentage of user CPU time over the previous 20ms period.

Figure 8–9: *CPU-TIME*—Graph is plotted every 20ms showing the percentage of user CPU time over the previous ten 20ms periods.
Figure 8–10: CPU-TIME—Graph is plotted every 20ms showing the percentage of user CPU time over the previous hundred 20ms periods.

with different degrees of averaging: in Figure 8–8 the results are not averaged; in Figure 8–9 the results are computed using a sliding window of 200ms and in Figure 8–10 the greatest smoothing occurs as the percentage of user time at any point is calculated from the user times for the preceding two seconds.

The cost of this metric is the highest so far, because of the rate at which threads are scheduled and descheduled (about ten thousand schedulings occurred during the program execution and the total event rate was roughly 615 events per second), and has proved to be beyond the capabilities of the master processor to deal with in real time. Although it might be desirable (for reasons suggested in Section 8.2.1) to extend this system by passing a thread identifier with each event, it is obvious that the amount of processing required by the monitor to collect load information on a thread-by-thread basis is prohibitive.

The CPU-TIME metric is relevant to the CPU usage but irrelevant to the pattern of communication. It expresses the amount of computation being performed versus idling or execution of system services, but it cannot distinguish between a processor which, for example, always has one compute-bound thread
in its ready queue and another processor which has many more compute-bound threads in its ready queue—this is a definite disadvantage.

The metric also illustrates the problem of timeliness: if the minimal smoothing is increased to mask out the transients in Figure 8-8, then care has to be taken that the perceived load does not fall behind the real load—the peaks in Figure 8-10, for example, are about two seconds behind those in Figure 8-8.

**Results for REM-COMM metric**

The final metric indicates the amount of remote communication on links, i.e. between pairs of processors. In contrast to the previous three metrics which were collected in an event-driven way, this metric is collected by sampling. Each processor counts the link communications as they occur and, at fixed time intervals, transmits the totals to the central monitor and resets the counts. The system can be reconfigured to work with different time intervals so that the appropriate balance between level of detail and amount of event traffic can be found.

The measurements made with **multi-phase** are shown in Figure 8-11. The 'link temperature' is the number of messages transmitted over a link per sample interval. The key shows the ten links: '1-2' is the link between processors 1 and 2 for instance. The initial peak around one second is caused by the distribution of work from the processor where **multi-phase** was invoked to the other processors. The small peaks around 10 and 13 seconds are caused by phases 3 and 4 during which the compute-bound threads also perform a small amount of remote communication—note that the graph does not distinguish between the small and large message sizes. The peaks around 23 and 30 seconds are caused by phases 7 and 8 during which much larger numbers of remote messages are sent.

The cost of this metric depends on the sampling period. Sampling once per second, as in this experiment, requires four events per second to be collected from each processor (one for each link that processor may use) or twenty events
Figure 8–11: REM-COMM—This graph shows, for each of the ten processor-to-processor communication links, the number of communications that have occurred during the last sample period of one second.

per second in total. A minimum of processing is required to combine the events (processors increment their communication counts each time they send a message so the counts from both ends of each link must be added to compute the total number of messages sent over each link).

The REM-COMM metric provides a direct measure of the number of times each processor has to send a remote message, although it does not distinguish between the sending of large and small messages and Figure 7–8 (page 143) shows that the time required to service messages depends strongly on the length.

The timeliness of the metric depends on the length of the sampling period and although decreasing this period improves timeliness, it also increases the number of possibly misleading transients. The scalability of this metric is the best of all four metrics because the amount of monitoring data produced (and thus the amount of processing) can be controlled by changing the sample period. (In fact, the benefits of sampling are so great that, as will been seen shortly, it is worth considering sampling variants of the other metrics.)
Case study conclusions

<table>
<thead>
<tr>
<th>Metric</th>
<th>Event driven?</th>
<th>Cost</th>
<th>Relevant?</th>
<th>Timeliness</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-THREADS</td>
<td>yes</td>
<td>low</td>
<td>no</td>
<td>guaranteed</td>
</tr>
<tr>
<td>RQ-LENGTH</td>
<td>yes</td>
<td>high</td>
<td>yes</td>
<td>guaranteed</td>
</tr>
<tr>
<td>CPU-TIME</td>
<td>yes</td>
<td>very high</td>
<td>perhaps</td>
<td>guaranteed</td>
</tr>
<tr>
<td>REM-COMM</td>
<td>sampling</td>
<td>low</td>
<td>yes</td>
<td>depends</td>
</tr>
</tbody>
</table>

Table 8–1: Summary of results.

Referring to Table 8–1, the first metric (the number of threads extant on each processor) is satisfactory for most selection factors but is so lacking in relevance that it is not of any practical use. The second metric (ready queue length) is relevant but expensive to collect and requires some smoothing. The third metric (proportion of user time) is very expensive to collect and no more relevant than the ready queue length. The fourth metric (number of remote communications) is reasonably relevant and, with care, can be collected inexpensively and with sufficient timeliness.

8.2 Strategy for Load Reconfiguration

The first part of this chapter has considered the problems of defining a balanced load and selecting good load metrics. I now propose a decision-making strategy for reconfiguration of the Testbed load based on the two most promising metrics from the case studies. This strategy is then tested with a range of parallel programs to see how close it comes to achieving a balanced load.

The load metrics used are a variation of RQ-LENGTH (ready queue length) and REM-COMM (amount of remote communication). The Testbed optimises CPU load and communications load since, as Table 7–4 (page 159) shows, remote communication is about twice as expensive as local communication. In order to improve the quality of load reconfigurations, the responsibility for load balancing is shared between a centralised decision strategy and a distributed candidate
The load assessor is a software component which executes on the master processor, collects the raw load data transmitted by individual slave processors over the monitoring bus and forms it into a coherent load summary—the techniques for doing this were discussed in the preceding section. The decision maker is a new software component which also executes on the master processor. The decision maker inputs the load summary, identifies load imbalances and issues reconfiguration requests to the slaves.

Each reconfiguration request received by a slave processor contains an indication of whether a compute or communication overload has been detected, and the identity of the (presumably underloaded) processor to which a thread should
be migrated. When a communication overload is signalled, the identity of the overloaded link is also given. The job of identifying the best candidate thread for migration is left up to the slave. If a compute overload has been indicated then the slave will migrate the first thread it finds in the processor ready queue. If a communication link overload has been indicated then the ready queue is searched for a thread whose last communication used the overloaded link or, if no such thread can be found, the ready queue is searched for a thread whose last communication was not local. The benefits of sharing the responsibility for balancing in this way between master and slave processors are discussed next.

8.2.1 Centralised versus distributed balancing

The intention of the Testbed's designers was that the event monitoring system would collect load data for a centralised load balancer. The advantages of such an approach are that the intrusion suffered by the user's program on the slave processors is reduced to the minimum possible, the balancing strategy can have a global view of the entire machine and there is no risk of conflict between multiple, autonomous balancing agents.

However, experience gained during this research has confirmed that centralised systems cannot be scaled up as easily as distributed systems, even when the system in question has only six processors. If the Testbed monitoring were to be completely centralised, i.e. the six slave processors emitted monitoring events each time a thread was scheduled, preempted, or otherwise changed state and the load assessor combined these event into a model of the load imposed by each thread, then the load assessor would require more compute-power than the master processor can possibly provide.

The semi-centralised, semi-distributed alternative employed on the Testbed is to have each slave processor maintain its own load statistics and to transmit these statistics to the load assessor at intervals. This reduces the amount of work the load assessor must perform to manageable levels. However, now that the load assessor models processor and link loads rather than the activity of
individual threads, the slave processors must accept the additional responsibility of selecting candidate threads during migration.

Delegating candidate selection to the slave processors increases the interference between the monitoring system and the user's program. However, the quality of load balancing can be improved greatly because the slaves have access to local information about the candidate threads which would otherwise have to be copied to the load assessor on the master processor. Indeed, the idea that slave processors should regularly submit a thumbnail sketch to the master processor of each of the many threads they might be hosting just in case the balancer might want to migrate some of them seems extremely wasteful.

The algorithm employed by the slave processors to select a candidate is described in Figure 8–13. Threads with children are rejected because the specification in Chapter 5 requires that parent threads do not migrate. Threads not in the ready queue are also rejected because migrating them will not (at least in the short term) change the balance of the computational load. The last_comm(t) function returns the number of the link used during thread t's last communication, or NULL if the last communication was local or if the thread has never communicated at all.

8.2.2 Establishing parameters for the decision strategy

There are two time periods to be determined for the Testbed's load balancer. The first is the sample period, i.e. the interval at which the load assessor is to furnish a load summary. The second is the migration period, i.e. the minimum interval between the issuing of reconfiguration requests. The reason for having a fixed sampling period—in order to balance the quantity and the timeliness of load information against the degree of smoothing—was discussed above with reference to the REM-COMM load metric.

The classical reason for having a minimum migration period is to reduce the possibility of flooding or thrashing, although this minimum must not be set too high otherwise it will take a long time to syphon work away from overloaded
Chapter 8. Load Balancing

thread *select(l)             /*Return best candidate or NULL*/
   link *l;               /*Overloaded link or NULL if no such*/
{
   thread *ct=NULL,       /*Best candidate thread so far*/
   *t;                /*Variable to range over all.*/
   for (t=first_thread; t<last_thread; t++) { /*threads*/
      if (has_children(t) || !in_ready_Q(t))
         skip-this-loop;     /*Ignore parents and blocked threads*/
      if (l) {
         if (last_comm(t)==l) /*and thread used link.*/
            return t;     /*then return thread*/
         else if (!ct ||
            !last_comm(ct)) /*cand’s last comm not remote.*/
            ct=t;         /*then remember new candidate*/
      }
      else return t;    /*If not optimising any link.*/
   }
   return ct;          /*Return best candidate*/
}

Figure 8-13: The candidate selection algorithm, in a C-like pseudocode.

processors. The Testbed is additionally restricted since the proofs in Chapter 6 do not guarantee safety when two threads sharing a channel migrate together. The experience gained by performing the proofs suggests that the migration protocols would need to be significantly, but not impossibly, more complex if safe, concurrent migrations were required.

One way to ensure that migrations are not overlapped is to make the migration period greater than the maximum time required to complete a reconfiguration request. This latter time can be computed from the results presented in Chapter 7 as follows.
1. The reconfiguration command is issued on the master processor and sent to the appropriate slave processor over Centrenet. The Centrenet message is small (16 bytes) and its flight time will be the same as the flight time of a *focus* or *rtr* message—approximately 1.5ms (Figure 7–9). In the worst-case scenario (as described in Section 7.3.5) the reconfiguration command is queued at the beginning of the schedule phase and is not sent until the send phase, approximately one time-slice later, and the reconfiguration command arrives during the schedule phase and is not processed until the receive phase, during which another time-slice may have elapsed. Thus, at least 41.5ms must be allowed for the issuing of a reconfiguration request and its processing at the slave processor.

2. Provided that a candidate thread is found immediately, the time required to pack a thread, send it over Centrenet and unpack it at the other end is approximately 8.4ms (Table 7–4). In the worst case, the thread may arrive during the schedule phase and not be unpacked until the receive phase, approximately one time-slice later. Thus, at least 28.4ms should be allowed for the thread migration.

3. With some applications it is not always possible to find a candidate thread immediately on arrival of the reconfiguration request. This situation arises most often with communication-bound applications where most of the threads are blocked most of the time and cannot, therefore, be considered candidates for migration.

   If a reconfiguration request arrives at time \( t \) and cannot be satisfied immediately then it is satisfied the next time a thread is inserted into the processor ready queue. If the request is still unsatisfied after \( n \) timer interrupts then it is discarded.

Therefore, the total time that must be allowed for a reconfiguration command to complete is \( 41.5 + 28.4 + n \times 20 \)ms. Practical experimentation has shown that a value for \( n \) of 25 is appropriate, hence a migration period of 569.9ms will ensure that migrations do not overlap.
Theoretically, the average migration period can be reduced significantly if the slave processor is made to return a 'success' indication to the master processor on completion of the migration. For instance, if candidate threads can always be found immediately (as is likely with a compute-bound application) then an average migration period of \((69.9/2)\)ms is possible. However, the migration period is further constrained on the Testbed by its close relationship to the sample period, as described next.

The Testbed's decision strategy is relatively simple and is based on the intuition that if a significantly high load is detected on a processor or link then an appropriate reconfiguration request is issued. Rather than assess the size of the overload and issue a combination of requests intended to completely redistribute the overload, the load balancer issues a single request and waits to observe the effect of the migration in the load summary before proceeding with further migration requests. Thus, the migration period must always be greater than the sampling period and, conversely, the sampling period has an upper bound determined by the frequency with which migration is desired.

The actual values assigned to the sampling and migration periods are specified in the case studies in Section 8.3. While these values obey the constraints described in this section, there is still some scope for optimising them with respect to the application being balanced.

Once the sample and migration periods have been set, thresholds need to be determined for overloaded (and underloaded) processors and links. If the thresholds are set too low then the system may squander more time in migration than it saves by improving the balance. In the worst case, the system begins to flood underloaded processors, or thrash. If the threshold is too high, then the load may become very poorly balanced before the load balancer takes any action. Thresholds for the Testbed's balancer were determined by practical experimentation, as is reported in the next section.
8.3 Effectiveness of the Balancer

This section considers a range of parallel programs, discusses the characteristics of those that do and do not benefit from load balancing and presents three case studies showing the sort of benefit which can be obtained from the Testbed's load balancer.

At the simplest level, dynamic load balancing can be beneficial when the resources required by individual tasks are not, or cannot, be known when the program is written. This may be due to fundamental unpredictability in the program, because the program behaviour is highly dependent on its input data, or simply because it is not worth the programmer's effort in finding out. (Code that is to be executed many times, such as the inner loop of a sorting function, is usually worth optimising but for the many, less critical sections of code, it is not cost-effective to perform optimisations.) Furthermore, the desire to achieve reusability or portability at an algorithmic level is usually at odds with the desire to achieve efficient programs.

On the other hand, dynamic load balancing is of little value when the computation requirements are well known beforehand and when the program can be optimised for a particular computer architecture.

There are two experimental approaches in the literature to testing load balancing systems (an example of a formal approach can be found in Rommel [90]). The first approach takes a simple program whose behaviour and optimal assignment to processors is obvious: Boillat and Kropf [11], for example, use a test program comprising a two-dimensional array of identical tasks where each task communicates with two or four of its neighbours; Lin and Keller [58] use a ten-line divide and conquer algorithm for binary tree traversal.

The second approach attempts more realism and uses much larger programs with unpredictable behaviour. The effectiveness of the balancer is found by repeating the experiment, once with the balancer enabled and once with the
balancer disabled. Osser [80], for example, reports on the execution times of common Unix utilities such as \LaTeX{} and \texttt{ls} and Ogle et al [79] simulate a war game. This thesis uses both approaches.

### 8.3.1 Case study: synthetic programs

The first case study explores the effectiveness of load balancing when applied to synthetic programs with predictable behaviour. The test program \texttt{synthetic} has a main loop in which a variable number of compute-bound, communication-bound or 'mixed' (half compute, half communication-bound) threads are created. These threads execute indefinitely and can be migrated between slave processors as the load balancer sees fit. The number and type of threads are determined by arguments passed on the command line.

Three experiments were performed to measure the effectiveness of load balancing a program with first, 20 compute-bound threads, then 20 communication-bound threads and finally 20 mixed threads. In all cases the load balancing sample and migration periods were one second (a reasonable value as determined by trial and error) and the \texttt{synthetic} program was executed for one minute. The experimental results are computed from a log file produced by the load balancer. The log file reports the load summary received during each sample period and the load reconfigurations as they occur, a sample is given in Figure 8-14: the \texttt{TIM} field gives the time in ms at which the load summary was obtained; the \texttt{LNK} field gives the number of communications on each of the ten links; the \texttt{RQL} field gives the length of the processor ready queues for each of the slave processors.

Three graphs are produced for each experiment. In the first graph the lengths of the ready queues of each processor are plotted against program time. In the second graph the amount of remote communication on each of the ten processor-to-processor links is plotted against program time. In the third graph the thread migrations induced by the load balancer are reported using an invented notation whereby a line between two processors indicates a single migration and the slope of the line indicates the direction of the migration. For instance, a line joining
TIM: 60258
LNK: 127 106 106 36 74 74 0 0 144 144 (av 81)
RQL: 4 5 1 1 1 (av 2)
move work from 3 to 5 to quieten link 9/3-5 (mig #6)
mig #6: 1-00-003 arrives at 5, time 60880
TIM: 70240
LNK: 22 69 81 41 14 72 58 0 117 119 (av 59)
RQL: 4 3 1 1 2 (av 2)
move work from 4 to 5 to quieten link 10/4-5 (mig #7)
mig #7: 1-15-004 arrives at 5, time 70581
TIM: 80241
LNK: 12 58 24 61 0 48 54 0 81 119 (av 46)
RQL: 2 2 1 1 3 (av 2)

**Figure 8-14: Extract from the Testbed load balancer’s log file.**

earth and fire with fire’s point offset to the right indicates that the migration was from earth to fire.

The results of the first, compute-bound experiment are given in Figures 8-15 to 8-17. The test program is invoked on earth and it is seen that the ready queue builds up rapidly and is then reduced as the load balancer syphons off one thread each second. After about 17 seconds the load is equalised across all processors. The numbers of link communications are minimal—the initial burst between 2 and 18 seconds of program time being due to the thread migrations and consequent remote paging.

The individual migrations are much as expected: all migrations move work away from earth and towards the other processors, different destinations being selected in turn. The hit ratio is perfect—no ‘backwards’ migrations are seen—and the program quickly settles down into a balanced mode.

The success of the load balancer is not quite so clear cut in the second experiment with communication-bound threads (Figures 8-18 to 8-20). The computa-
Figure 8-15: Length of the ready queue on each of the five slave processors while executing 20 compute-bound threads.

Figure 8-16: Number of communications on each link while executing 20 compute-bound threads.
Figure 8–17: Thread migrations while executing 20 compute-bound threads.

tional load does become balanced, although not until after 30 seconds. The link loads are much more significant than before but are also balanced, within a range of about 20 messages per (one second) sample. Several of the links (1-3, 1-4, 1-5 and 2-3) end up with no traffic at all although this is somewhat difficult to see on the crowded graph. The migrations shown in Figure 8–20 mostly occur between 0 and 30 seconds, program time. There are more of them than in the previous experiment because this time there are 20 pairs of threads, i.e. 40 threads in all. The hit ratio is less good than before with several 'back' migrations moving threads off space and air.

The effectiveness of the load balancer with the mixed threads is very similar to that with the communication-bound threads, so only the ready queue length graph is shown (Figure 8–21).

8.3.2 Case study: asynchronous circuits

The second case study looks at the effectiveness of load balancing applied to a realistic class of programs: simulations of asynchronous circuits. Conventional
Figure 8-18: Length of the ready queue on each of the five slave processors while executing 20 pairs of communication-bound threads.

Figure 8-19: Number of communications on each link while executing 20 pairs of communication-bound threads.
Figure 8–20: Thread migrations while executing 20 pairs of communication-bound threads.

Figure 8–21: Length of the ready queue on each of the five slave processors while executing 20 mixed threads.
design relies on clocks to control and synchronise the movement of data through digital logic. Innovations such as parallel or pipelined data paths help increase the speed at which such circuits can work. Asynchronous logic, however, has no clock: the basic components compute their functions as soon as their inputs are ready—an in-depth discussion can be found in Sayle [91].

Simulation of asynchronous circuits can be easily achieved on the Testbed. The adder program, which simulates the full adder circuit shown in Figure 8–22, has one thread for each element in the circuit (plus one to simulate the environment). The threads wait for messages on their input channels, compute the appropriate function and output a message or messages. If a record, with one field for the delay time induced by each type of element, is communicated between threads and each thread updates the appropriate field of the record according to its type, then the output of the simulation is a list of delay times giving the total time the circuit took to compute its answer. The simulation can be expected to have similar dynamic properties to the communication-bound, synthetic experiment described in the preceding section. The reason for carrying out the simulation is that the delay induced by a circuit is often a complex function of its inputs.

The results of load balancing the program adder are shown in Figures 8–23 to 8–25. As before, experimentation showed that sample and migration periods of one second were appropriate. The total amount of computation is much lower than before and it should be noted that the maximum value on the y-scale of Figure 8–23 is one tenth that of the corresponding figures presented earlier in this section. This is due to a minimal amount of computation that each of the 15 threads has to perform in updating and passing on the delay record. In fact, the computation is so limited as to fall below the load balancer's threshold.

In contrast, the degree of communication is high and the migrations carried out between 1 and 12 seconds of program time are an attempt by the load balancer to correct the situation. This is to some extent successful as from 15 seconds onwards all but four links have had their traffic reduced to zero. The high amount of traffic on links 1-2, 2-5, 3-5 and in particular 1-5 points to the limitations of the load balancer in achieving a perfect load. In fact, the balancer’s log file
Figure 8–22: An asynchronous adder circuit.

Figure 8–23: Length of the ready queue on each of the five slave processors while executing `adder`.
Figure 8-24: Number of communications on each link while executing adder.

Figure 8-25: Thread migrations while executing adder.
Chapter 8. Load Balancing

shows that reconfiguration requests were issued to correct the problem but that the slave processors could not find appropriate threads for migration.

8.3.3 Case study: population simulation

The third case study also involves the balancing of a simulation, although this version of the population simulation known as 'WaTor' and described in Fox et al [29, Chapter 17] is much larger and more prone to dynamic behaviour than the adder simulation.

The simulation concerns the populations of sharks and fish in a 100-by-100 cell ocean. Each cell may be occupied by a fish or shark and each animal is updated at each iteration with respect to its position, death through starvation or regeneration by birth of a new animal. A novel addition is made to the basic simulation: seasonal temperature variations occur across the ocean regions and these affect the fish's food sources, promoting or restricting the reproduction rate as appropriate. This addition has the effect of inducing large changes in the load balance during the simulation.

The wator program comprises multiple threads, each being responsible for a region of the ocean. The threads iterate, exchanging boundary conditions with their neighbours, updating the fish and sharks within their region and then performing conflict resolution with their neighbours. In all experiments the wator test program is executed for 100 simulation steps.

The output of the simulation is the total number of fish and sharks in the regions after each step: Figure 8-26 illustrates the predominant feature, which is the booms and busts experienced by the fish as they quickly build up their numbers by reproduction only to become a target for a more slowly swelling number of sharks. Once most of the fish have been eaten, the shark numbers fall due to starvation, the numbers of fish take off again, and the cycle repeats.

The effectiveness of load balancing the WaTor simulation is demonstrated by showing first the graphs for ready queue and link loads without load balancing
Figure 8-26: Variation in number of fish during 100 steps of the WaTor simulation.

Figure 8-27: Ready queue variation during 100 steps of the WaTor simulation without load balancing.
Figure 8–28: Variation in link load during 100 steps of the WaTor simulation without load balancing.

(Figures 8–27 and 8–28) and then by showing the same loads with load balancing in action (Figures 8–29 to 8–31).

The nature of the simulation algorithm is such that a period of intensive computation on all processors is followed by a period of intensive communication between processors. The amount of computation and communication varies as the number of fish and sharks in the ocean. Without load balancing, the average time required by a simulation step is around 10 seconds and this produces problems for the load balancer. If, for instance, the load is sampled at the one second intervals used in the preceding case studies, then the load balancer will collect several consecutive samples suggesting that the Testbed is compute bound, followed by several consecutive samples suggesting that the Testbed is communication bound. To overcome this problem, the load balancing sampling period is increased to 10 seconds.

Under the assumption that the processor air is responsible for region 2, fire for region 4, space for region 3 and water for region 1, the graph in Figure 8–26 can be approximately matched against the graph in Figure 8–27 to show
that the region with the most fish matches the processor with the longest ready queue. The match is only approximate because the x-coordinate in Figure 8–26 is the simulation step and not all simulation steps take the same amount of time. The fact that the exchange of boundaries between neighbours enforces a kind of synchronisation serves to accentuate the differences between processor ready queue lengths. (Note that the processor earth is not shown in Figure 8–26 as it does not have any region associated with it.)

In the preceding two case studies the sampling period was much larger than the basic period of the program being balanced. This was beneficial since the load summary presented to the load balancer was a smoothed version of the real load. Now, as the sampling period is of the same size as the basic period of the simulation, the load summary received by the balancer is unsmoothed—and this can be seen particularly easily in the peaks and troughs of Figure 8–28.

The effect of load balancing the wator simulation is seen in Figure 8–29 where the average ready queue length is reduced (cf. Figure 8–27). This strategy leads to an increase in remote communication—compare Figures 8–28 and 8–30—but
Figure 8-30: Variation in link load during 100 steps of the WaTor simulation with load balancing.

Figure 8-31: Thread migrations while executing 100 steps of the WaTor simulation.
since the time required to complete the 100 simulation steps is reduced from 900 seconds to under 600 this strategy is beneficial. The benefit derives from the fact that without load balancing one, busy processor is holding the others up whereas with load balancing several processors are kept busy—compare Figures 8–27 and 8–29. Figure 8–31 indicates the highly dynamic nature of the simulation: in contrast to the previous case studies no steady state can be reached and the load balancer must continually reconfigure the load in order to maintain balance.

### 8.3.4 Summary

<table>
<thead>
<tr>
<th>Application</th>
<th>Sequential</th>
<th>Random</th>
<th>Balancing</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 compute-bound threads</td>
<td>54 (100%)</td>
<td>15 (28%)</td>
<td>21 (39%)</td>
</tr>
<tr>
<td>20 communication-bound threads</td>
<td>72 (100%)</td>
<td>41 (57%)</td>
<td>35 (49%)</td>
</tr>
<tr>
<td>adder simulation</td>
<td>93 (100%)</td>
<td>183 (197%)</td>
<td>126 (135%)</td>
</tr>
<tr>
<td>wator simulation</td>
<td>900 (100%)</td>
<td>550 (61%)</td>
<td></td>
</tr>
</tbody>
</table>

Table 8–2: A comparison of the times (in seconds) required by each case study application to: execute sequentially; execute in parallel, assigned at random; and execute in parallel, assigned by the load balancer.

The effectiveness of the Testbed’s load balancer is summarised in Table 8–2. Each of the applications in the case studies presented above is made to perform a fixed amount of work and is executed three times: once with all threads on the same processor (wator excepted); once with threads assigned to processors at random; and once with threads assigned to processors by the load balancer.

For the compute-bound application random allocation, which assigns equal numbers of threads to each processor, is optimal and therefore out-performs the load balancer. However, if the total amount of computation is increased then the load balancing time will asymptotically approach the random allocation time. For the communication-bound application, the load balancer out-performs random allocation because it can move threads to convert remote channels into local channels. If the total number of communications is increased, then the load balancer’s performance will show even greater improvements.
The adder simulation takes longer to execute in parallel, although the load balancer is still better than random allocation. As the size of the circuit being simulated is increased, however, the time required for sequential execution will become longer than the time required for parallel execution, and the load balancer can be more effective. While it is possible to retune the balancing decision maker for the current simulation size so that it optimises links more aggressively, this is likely to worsen the balancer's performance with more 'typical' programs.

The wator simulation also shows that load balancing is better than random allocation. In this case, it is the ability of the load balancer to react to load changes during the execution that produces the improvement. As noted earlier, such gains will increase the longer the simulation is executed.

8.4 Conclusions and Recommendations

I have discussed the important issues in load balancing and proposed an implementation for the Testbed based on a detailed knowledge of the costs of various operating system functions, and experimental results indicating the effectiveness of different load metrics. I have presented selection criteria for load metrics and illustrated their use in several case studies. I have also informally characterised a 'balanced load' and presented detailed data showing the behaviour of the Testbed load balancer, indicating its strengths and weaknesses. Although the Testbed environment is specifically designed for collecting event trace data, I now discuss the extent to which the results obtained on the Testbed are relevant for other types of parallel computer systems.

Although Douglos and Ousterhout [24], for example, argue that the additional overheads of load collection make dynamic migration prohibitively expensive and Ieumwananonthachai et al [43] say the information collected will usually be incomplete and out of date, the work presented in this chapter (and that of Livny and Melman [60] and others mentioned in Section 2.3) shows that the usefulness of load balancing should not be doubted. Whilst I do not claim that all
kinds of parallel program will benefit from being balanced, it does seem plausible that many of the programs in common use will. With migration times in the order of milliseconds, any programs that execute for seconds or minutes become candidates for balancing.

The results of the case studies on load metrics should be broadly applicable to other types of multicomputer despite the fact that the results are partially dependent on the relative costs of computation and communication on the target architecture. However, the general problem of finding load metrics that are inexpensive, timely and relevant remains. Moreover it seems likely that the possibility of obtaining a particular load metric will be greatly determined by the target computer.

The other major problem area is the tuning of the balancing system. It is not easy to determine sensible sample periods or to set thresholds above which a load is considered to require rebalancing. In fact, it may prove that such tuning needs to be performed dynamically according to the program(s) being balanced.

The main limitation of the Testbed load balancer is that it only works with five processors and the ultimate aim is to write balancing systems that will work with much larger collections of processors. However, the following three-step procedure has proved valuable in the design of the Testbed balancer and will apply to larger systems, even those not fully connected.

1. Local agents collect and submit just enough information to one or more centralising agents for the centralising agents to establish a global view.

2. On the basis of their global view, the centralising agents then indicate to a subset of the local agents where imbalances are occurring.

3. The local agents use their (extensive) local knowledge to decide the best way to remedy the imbalance.

The only difficulty in applying this procedure is in the selection of centralising agents with a wide enough global view whilst maintaining a suitably low communication delay between each centralising agent and associated local agents.
Chapter 8. Load Balancing

It is a conclusion of this thesis that software-only monitoring and hybrid monitoring with limited hardware support are the most cost-effective methods for gathering load data. Three sets of results are combined to justify this assertion.

- The case studies on the effectiveness of the Testbed’s load balancer in Section 8.3 showed that significant speed-ups could be achieved with the RQ-LENGTH and REM-COMM metrics.

- The case studies on load metrics in Section 8.1.3 showed that such metrics could be gathered relatively cheaply, in terms of event rates. These metrics could also be gathered by purely software instrumentation.

- Section 7.3.7 showed the Testbed’s sustainable event rate is far below the theoretical event rate because the software components that produce the load summary have a much lower throughput than the dedicated monitoring hardware.

Therefore, most of the load balancing effort is made by the software and not by the hardware. A similar conclusion was reached by Phillips [83]—he found that a network of Transputers could be load balanced without dedicated monitoring hardware with an overhead of only 2%.

Experience in designing the operating system for the Testbed and in instrumenting the code indicates that it would not be cost-effective to enhance the capabilities of the hardware so that it performed a greater proportion of the load balancing activities. The necessary load information is very much more difficult to extract with hardware than with software. However, there is one significant benefit to be gained by having hardware support for communicating load data between processors: the load data is much more timely, i.e. it is not subject to contention for the main processor interconnect.
Chapter 9

Conclusions

In this final chapter I review the results presented in the thesis, summarise the implications for the next generation of load balancing systems and list some interesting directions for future research.

9.1 Review of Results

In Chapter 1, it was claimed that this thesis makes contributions to six areas of research in multicomputer load balancing. I now summarise my results in each of these areas.

*A review of the literature on load balancing, introducing the problem area and comparing the techniques proposed and the systems implemented.*

Chapter 2 showed that concurrent computers are important, because they can be very cost-effective, but that for several reasons they are not always straightforward to use efficiently. In particular, the scheduling problem was highlighted and load balancing proposed as a solution.

Load balancing makes a complicated topic because it has been applied in a variety of different environments, each application employing a combination of
techniques. Chapter 2 structured its description of load balancing techniques according to the three common phases of reconnaissance, decision-making and execution. Reconnaissance techniques explained and contrasted include: the extraction of load information statically, dynamically, statistically, historically and architecturally; software, hardware and hybrid mechanisms for collecting load information; sampling and triggering rules; ways of limiting intrusion; and the use of direct and indirect load information. Decision-making techniques described centre around the choice of control policy; centralised and distributed global scheduling; and optimal and heuristic global scheduling. Execution techniques listed involve the choice of sender-initiated, receiver-initiated or negotiation migration mechanisms.

Finally, a range of research and commercial systems were reviewed. All the systems chosen address some of the problems of load balancing—either the requirements for transparent migration of processes between Unix-style hosts or the requirements for gathering detailed load measurements—although few offer full support for the fine-grain, dynamic load balancing that is the main subject of this thesis.

*A detailed description of how a typical multicomputer operating system needs to be extended in order to support load balancing.*

Chapter 3 introduced the Testbed, the multicomputer on which the practical experimentation reported in this thesis was carried out. After the conventional features of the Testbed were described—its hardware, operating system and programming environment—the novel features specifically associated with load balancing were explained.

The special features needed to support load balancing were identified as: optimised data structures, migration protocols, software instrumentation and an event collection mechanism. As explained in Chapter 3, the optimised data structures are required to make the 'disconnection' of tasks at one processor and their 'reconnection' at another as efficient as possible. The migration protocols are
necessary to achieve safe sharing of tasks, communication channels and memory
pages between processors. Finally, instrumentation of the software produces the
load information which is then collected by, in the Testbed's case, the hybrid
event monitor for communication to the decision-making strategy.

**An investigation into the costs and benefits of applying formal methods
to the design and verification of the task exchange protocols.**

A formal specification in Z was developed in Chapters 4 and 5, focusing on those
parts of the Testbed operating system concerned with task exchange, i.e. thread
synchronisation, channel communication and thread migration. Framing schemas
were developed to show how sequences of operations are built up. Chapter 6 then
used the specification to demonstrate properties of safety and correctness—an
assurance that, provided certain constraints are met, user tasks can be migrated
as many times as is desired without changing the results of their computation.

A critical assessment of the Z language is presented next, after the advantages
and disadvantages of formal specification in general are discussed.

The formal specification was carried out in tandem with the development of
the Testbed's operating system. This was beneficial in several ways. Firstly,
the specification provided a much faster way of prototyping procedures for the
implementation than the traditional code, test and debug cycle. Secondly, the
specification coped well with changes in the requirements for the implementation
and was invaluable for indicating, albeit in a non-automatic way, ambiguities
and inconsistencies in those requirements. Thirdly, the specification was useful
as a tool for documentation, producing a clear description of the protocols being
implemented. This meant not only that walk-throughs of the code were simplified
but that the answers to various 'what if' questions could be given easily without
having to spend time writing and executing test programs. Most importantly, the
specification provided a precise way of presenting and interpreting performance
measurements made of the intrinsic properties of the Testbed. Finally, the proofs
showed the implementation to be safe and correct (provided the implementation
is a correct reification of the specification) to a degree that could never have been determined by enumerating all the possible interactions between the multiple processors.

A single caveat in the use of formal specification: it is not always easy to find the most appropriate level of abstraction in preliminary versions of a specification. The work in Chapters 4 and 5 went through a number of revisions before an acceptable balance was found between specifying the design principles clearly and specifying those features of the implementation important for reasons of efficiency. However, the verification process was helpful in indicating beneficial modifications to the specification.

I am still convinced that the Z language is one of the best for specifying load balancing protocols. However, there are a number of areas in which Z needs to be used carefully. Z is a rich language, it has most of the operations a specifier needs already defined. Arbitrary use of different operations can be confusing for the reader of a specification and the work in this thesis has shown that it is very beneficial to impose a 'house style' covering choice of operators, order of presentation and naming of user-defined objects. The value of having a house style is also emphasised in Macdonald [61] where a lengthy list of good practices is given.

The Z language has associated tools for syntax and type checking and for typesetting. These allow a high degree of consistency and evenness of appearance in the final specification which aid the reader in navigating the document. Prototyping and refinement were not attempted in this thesis, so the lack of tools to assist these processes was irrelevant. The lack of a theorem prover for Z would only become a problem if the rigorous proofs presented in Chapter 6 were to be made formal proofs.

In conclusion, formal methods provide an invaluable way of proving safety and correctness of complicated protocols, such as those for thread migration. Formal methods also provide a host of other benefits: faster implementation, better documentation and a precise basis for presenting and interpreting performance measurements.
Chapter 9. Conclusions

A study of the performance costs associated with the additional operating system components required to support load balancing.

In Chapter 7 a detailed and accurate profile was made of the performance of the Testbed operations involved in task migration. Finite state machines were used to explain exactly what was being measured in terms of the Z specification. Results were presented for the time required to send and receive messages of varying lengths over local and remote channels. The average times required to migrate a thread between processors and to copy a page of memory were also given. The effects of increased background load were discussed, with reference to the Testbed operating system's main program loop. Finally, a test program was executed to simulate combinations of extreme behaviours and thus establish the bounds on the performance expected from typical user programs.

An examination of the best methods for assessing the load as a basis for identifying imbalances.

Chapter 8 proposed an informal characterisation of a balanced load and used this characterisation as the basis for four case studies on different load metrics. A number of selection factors, e.g. relevance, cost, timeliness and scalability, were discussed and the metrics analysed accordingly. The results of the case studies showed that, with care, load metrics could be selected which gave a good reflection of the true load, did not flood the monitoring systems and were obtainable in a timely manner. Several 'obvious' load metrics were deemed to be highly unsatisfactory.

An example strategy for load reconfiguration on the Testbed was then given, based on variations of the two most successful load metrics RQ-LENGTH and REM-COMM. The division of responsibility between a centralised decision maker and distributed candidate selectors was justified. The determination of suitable sample and migration periods was discussed at some length, in particular the way in which the reconfiguration mechanism imposes a lower limit on the migration period.
A series of case studies on the performance benefits obtained from load balancing in relation to the type of parallel program being executed.

In the final part of Chapter 8, a series of case studies were used to explore the speedups that could be achieved by the Testbed's example load balancer. A range of test programs were used and the results produced showed that with some types of programs (particularly compute-bound, dynamic programs) the load balancer was extremely beneficial while with others (particularly communication-bound programs) the load balancer was less useful. The tuning of the sample and migration periods to the application was also discussed.

9.2 Implications for Load Balancing

The Testbed, around which the practical work of this thesis has centred, is a typical multicomputer both in terms of its architecture (replicated inexpensive microprocessors with local memory and a simple communication interconnect) and its programming environment (imperative programming style, augmented with message-passing functions). This section considers the relevance of the results presented in the thesis to designers of load balancing systems for other, typical multicomputers.

As was discussed in Chapter 3, the kind of fine-grain migration appropriate for multicomputers requires that the migration functions are designed into the operating system from the beginning. This is because migration requires many parts of the operating system to share data and to coordinate their activity. If this sharing and coordination is not done efficiently then the overheads of load balancing will be higher and it will be more difficult to achieve the desired speedups.

The formal specification of the Testbed's migration protocols are, of course, particular to the Testbed. However, the application of formal methods to load balancing is novel and has been shown to be of immense value for the purposes
of verification and the expression of performance measurements in terms of the underlying system design. occam is a fairly restrictive language in terms of the way channels and memory may be shared between concurrent tasks. Sharing objects in a system with a less restrictive programming model will, in fact, require more complicated protocols for sharing and thus the design of such a system will require the help of formal methods even more. Likewise, if the tuning of future load balancing systems is not to be done ad hoc but in a systematic way, then performance of the target multicomputer will need to be expressed in terms of a formal model.

The work presented in Chapter 8 has shown that load balancing can be effective even when only a small amount of load information is gathered per unit time. Chapter 7 demonstrated that the monitoring bottleneck is not in the event collection pathway, but in the load assessor. This strongly suggests the applicability of the methods proposed in this thesis to multicomputers without dedicated monitoring hardware. For multicomputers that are not fully connected, the hierarchical division of responsibilities between local and one or more centralising agents, as proposed in Section 8.4, is suggested to exploit the global view available to centralising agents whilst keeping as much information localised as possible.

9.3 Future Work

In this final section I list some interesting ways in which the research reported in this thesis might be developed.

The first development might be to improve the channel communication protocol so that simultaneous migration of threads was possible, without the risk of shared channels reaching an inconsistent state. This would allow experimentation with a more sophisticated load balancer which computed the current imbalance and then issued a list of reconfiguration requests which would correct
the balance completely. At the moment the Testbed cannot cope with rapidly changing workloads because its migration period is so long.

A more ambitious development might involve implementing load balancing on a system where the processors are not fully connected. This would be valuable in exploring the quality of balancing possible when no single agent can have a view of the entire machine. It would also allow experimentation on the maximum distance (in terms of processor-to-processor hops) allowable between local agents and centralising agents before reconfiguration commands are already out of date on arrival.

With respect to the formal specification of load balancing systems, it might be worthwhile investigating developments of $\mathbf{Z}$, e.g. Object-Z (Fogg [28]), which enable concurrent actions to be modelled explicitly. Currently, the $\mathbf{Z}$ language lacks any built-in operations for constructing models of execution. Other $\mathbf{Z}$ projects, e.g. Milnes [69], had to resort to using finite state machines as I did in Chapter 7. Another desirable extension to $\mathbf{Z}$ would be some way of defining a hierarchy in the specification. At the moment all objects must be defined before use and can be accessed globally.


Appendix A

Test Program Listings

The programs are given in a C-like pseudocode.

A.1 local

#define MAX_MSG 5000

main()
{
    char buffer[MAX_MSG];
    create_thread(id2, thread2, on_this_processor);
    for (msg_len=1; msg_len<MAX_MSG; msg_len+=MAX_MSG/100) {
        compute(approx_50_ms);
        for (rep=0; rep<10; rep++) {
            compute(approx_10_ms);
            send_message(channel1, buffer, msg_len);
        }
    }
    send_message(channel1, buffer, 0);
    wait(for_child);
    exit();
}
thread2()
{
    char buffer[MAX_MSG];
    while (receive_message(channel1, buffer, MAX_MSG));
    exit();
}

A.2 remote

#define MAX_MSG 5000

main()
{
    char buffer[MAX_MSG];
    create_thread(id2, thread2, on_another_processor);
    for (msg_len=1; msg_len<=MAX_MSG; msg_len+=MAX_MSG/100) {
        compute(approx.50_ms);
        for (rep=0; rep<10; rep++) {
            compute(approx.10_ms);
            send_message(channel1, buffer, msg_len);
        }
    }
    send_message(channel1, buffer, 0);
    wait(for_child);
    exit();
}

thread2()
{
    char buffer[MAX_MSG];
    while (receive_message(channel1, buffer, MAX_MSG));
    exit();
}
A.3 migrate

#define MAX_MSG 5000

main()
{
    char buffer[MAX_MSG];
    create_thread(id2, thread2, on_this_processor);
    for (many_loops) {
        send_message(channel1, buffer, MAX_MSG);
        compute(approx_100_ms);
    }
    wait(for_child);
    exit();
}

thread2()
{
    char buffer[MAX_MSG];
    for (many_loops) {
        receive_message(channel1, buffer, MAX_MSG);
        migrate(me, from_here, to_anywhere);
    }
    exit();
}
A.4 multi-phase

main()
{
    int i, id=2, p, phase;
    for (p=1; p<6; p++)
        for (i=0; i<5; i++)
            create_thread(id++, thread, on_processor(p));
    for (phase=0; phase<8; phase++)
    {
        barrier_send();
        barrier_send();
        compute(for_1000_ms);
    }
    wait(for_children);
    exit();
}

thread()
{
    int big_msg,
    comm_bound,
    comm_remote,
    i;
    for (comm_bound=FALSE; comm_bound<2; comm_bound++)
        for (comm_remote=FALSE; comm_remote<2; comm_remote++)
            for (big_msg=FALSE; big_msg<2; big_msg++)
            {
                barrier_receive();
                for (i=0; i<10; i++)
                {
                    compute(for_1_ms);
                    if (comm_bound)
                        if (comm_remote)
do_remote_comms(big_msg?MAX_MSG:1);
else
do_local_comms(big_msg?MAX_MSG:1);
else
compute(for_9_ms);
}
barrier.receive();
}
exit();

A.5 overflow

main(compute_time)
    int compute_time;
{
    int stop_time=clock()+30 SECONDS;
    while (stop_time>clock())
        if (send_event(0)==0) {
            printf("send event fails: monitoring bus full\n");
            break;
        }
    else compute(compute_time);
}
A.6 synthetic

main()
{
    int id = 2;
    for (number_of_communicating_pairs) {
        create_thread(id++, thread1, on_any_processor);
        create_thread(id++, thread2, on_any_processor);
    }
    for (number_of_computing_threads)
        create_thread(id++, thread3, on_any_processor);
    for (number_of_mixed_pairs) {
        create_thread(id++, thread4, on_any_processor);
        create_thread(id++, thread5, on_any_processor);
    }
    wait(for_children);
}

thread1()
{
    char buffer[MAX_MSG];
    for (loop_indefinitely)
        send_message(channel(get_pid()), buffer, MAX_MSG);
}

thread2()
{
    char buffer[MAX_MSG];
    for (loop_indefinitely)
        receive_message(channel(get_pid() - 1), buffer, MAX_MSG);
}
thread3()
{
    for (loop_indefinitely)
        compute(approx_1.ms);
}

thread4()
{
    char buffer[MAX_MSG];
    for (loop_indefinitely) {
        compute(approx_10.ms);
        send_message(channel(get_pid()), buffer, MAX_MSG);
    }
}

thread5()
{
    char buffer[MAX_MSG];
    for (loop_indefinitely) {
        compute(approx_10.ms);
        receive_message(channel(get_pid()-1), buffer, MAX_MSG);
    }
}
A.7 adder

main()
{
    create_keller(id++, input_channels, output_channels);
    create_call(id++, input_channels, output_channels);
    create_muller(id++, input_channels, output_channels);
    create_merge(id++, input_channels, output_channels);
    /* ...and so on for the other 10 circuit elements */
    for (opl=0; opl<256; opl++)
        for (op2=0; op2<256; op2++) {
            delay_record r;
            int n;
            zero_delays(&r);
            inject(op1, op2, &r);
            n=receive_answer(&r);
            printf("%d + %d is %d, op1, op2, n);
            print_delays(r);
            puchar('n');
        }
    exitO;
}

ekeller()
{
    int state=0;
    for (loop_forever) {
        delay_record r;
        n=alt_receive(&r, input_channels);
        r.keller++;
        if (set_channel(n)) {
            ...
        }
    }
}
state = 1;
send_output(&r, S);
}
else if (reset_channel(n)) {
    state = 0;
    send_output(&r, R);
}
else if (test(n) {
    if (state) send_output(&r, T1);
}
}
}

call()
{
    int state = 0;
    for (loop_forever) {
        delay_record r;
        n = alt_receive(&r, input_channels);
        r.call++;
        if (caller1(n)) {
            state = 1;
            send_output(&r, R);
        }
        else if (caller2(n)) {
            state = 1;
            send_output(&r, R);
        }
        else if (acknowledge(n)) {
            send_output(&r, state ? R2 : R1);
        }
    }
}
Appendix B

Index of Z Terms

advert, 62                  FAAdvert, 87
Args, 106                   FCAdvert, 87
advert, 63                   FCreate, 72

buffer, 75                  FFFocus, 88

CAdvert, 81                   FInit, 72, 86
CFocus, 81                  FMsg, 88
CNlnt, 76                     focus, 63
CMMsg, 83                 focus, 75

COUTNER, 61                   focusm, 62
create, 63                     FReceive, 87
CReceive, 80                 FRtr, 88
CRtr, 82                      FSend, 86
CSEnd, 78                       FState, 71, 85
CSTATE, 75                   INSTRUCTION, 71

MConnect, 97                   linked, 121

234
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDisconnect</td>
<td>95</td>
</tr>
<tr>
<td>MEM_BLOCK</td>
<td>74</td>
</tr>
<tr>
<td>msg</td>
<td>63</td>
</tr>
<tr>
<td>msg</td>
<td>62</td>
</tr>
<tr>
<td>null_thread</td>
<td>61</td>
</tr>
<tr>
<td>OP</td>
<td>63</td>
</tr>
<tr>
<td>par_bd</td>
<td>64</td>
</tr>
<tr>
<td>parent</td>
<td>64</td>
</tr>
<tr>
<td>pc</td>
<td>71, 85</td>
</tr>
<tr>
<td>PE</td>
<td>61</td>
</tr>
<tr>
<td>program</td>
<td>71, 85</td>
</tr>
<tr>
<td>rcontrol</td>
<td>75</td>
</tr>
<tr>
<td>ready</td>
<td>64</td>
</tr>
<tr>
<td>Receive</td>
<td>89</td>
</tr>
<tr>
<td>receive</td>
<td>63</td>
</tr>
<tr>
<td>receiver</td>
<td>75</td>
</tr>
<tr>
<td>rloc</td>
<td>75</td>
</tr>
<tr>
<td>rtr</td>
<td>63</td>
</tr>
<tr>
<td>rtr</td>
<td>62</td>
</tr>
<tr>
<td>SBase</td>
<td>106</td>
</tr>
<tr>
<td>Schedule</td>
<td>89</td>
</tr>
<tr>
<td>scontrol</td>
<td>75</td>
</tr>
<tr>
<td>send</td>
<td>63</td>
</tr>
<tr>
<td>sender</td>
<td>75</td>
</tr>
<tr>
<td>sloc</td>
<td>75</td>
</tr>
<tr>
<td>SStep</td>
<td>107</td>
</tr>
<tr>
<td>sync</td>
<td>63</td>
</tr>
<tr>
<td>TCreate</td>
<td>66</td>
</tr>
<tr>
<td>term</td>
<td>62</td>
</tr>
<tr>
<td>terminate</td>
<td>63</td>
</tr>
<tr>
<td>term</td>
<td>63</td>
</tr>
<tr>
<td>thrd</td>
<td>62</td>
</tr>
<tr>
<td>THREAD</td>
<td>61</td>
</tr>
<tr>
<td>TInit</td>
<td>64</td>
</tr>
<tr>
<td>trace</td>
<td>71, 85</td>
</tr>
<tr>
<td>TState</td>
<td>64</td>
</tr>
<tr>
<td>TSync</td>
<td>67</td>
</tr>
<tr>
<td>TTerm_msg</td>
<td>68</td>
</tr>
<tr>
<td>TTerminate</td>
<td>69</td>
</tr>
<tr>
<td>used</td>
<td>75</td>
</tr>
</tbody>
</table>