The POSIE Project: Studies to Support the Design of Operating Systems for Multicomputers

Rosemary Candlin

June 12, 1995

Preface

This is a largely historical account of work that took place in the Computer Science Department from 1988 to 1994 under the general heading of the "POSIE Project". Many people were either directly or indirectly involved, and some who in no sense could be described as project members made valuable contributions and appear as co-authors on papers that arose out of the project. In fact, this was one of the interesting things about POSIE – no-one was in charge, and the project group had distinctly permeable boundaries. This situation contributed both to its strengths and weaknesses!

The purpose of this report is not to provide technical information, although brief excerpts of some of the main work appear here. A full bibliography of internal and external publications is given at the end, and those who want to find out more can refer to them. There has also been no attempt to provide a bibliography of relevant work that was done elsewhere. All the published papers have full bibliographies which will enable readers to track down references to work that interests them. It was thought worthwhile to write an "overview" of POSIE, because in many ways, the different aspects of the work add up to quite a substantial contribution to the basic knowledge that should underpin the design of operating systems for MIMD machines. It is not comprehensive in this respect, but provides a useful starting point.

There are very many people who participated in the project, or generously gave their time to talk about particular aspects of it. We must thank Roland Ibbett for his support, particularly in providing the funds for construction of the Testbed machine. We owe a special debt of thanks to Peter Lindsay who finally got the Testbed to work, and to the technical staff who helped to construct it. We also made use of machines in the Edinburgh Parallel Computing Centre, and thank Prof. David Wallace, the previous Director, for permission to use these facilities.

Chapter 1

An Overview of the Project

1.1 Introduction

The POSIE project took shape as a result of coffee-time conversations between Rob Pooley and Rosemary Candlin. The title was an acronym that involved the words "parallel", "operating system" and "environment", though it was never entirely clear how they related to the letters "POSIE"! The aim was to provide a focus for systems research in the area of operating systems for distributed-memory parallel machines, and as the project developed, this took in not only hardware and software construction, but simulation, formal methods and performance evaluation methodology. The first year report [31] identified these themes, and it is interesting to see, now that the project as such is coming to an end, how fruitfully they have been developed.

It seems worth while to provide an overview of research that has been done, either under the POSIE umbrella, or in closely related fields, and to draw these individual strands together in one short report. Most of this work has been published elsewhere, and a full list of papers and reports appears at the end, for those who want more details. The project has been a fascinating example of research with distributed control, and provides an interesting sociological case-study of the strengths and weaknesses of collaborative, pure academic research. One of the important characteristics of this project was that it was entirely funded internally, and that there were therefore no external constraints to push it in one direction rather than another. Another was that no one was "in charge", but a very large number of members of the department contributed to it at one stage or another.

We start with an historical overview, and describe the various contributions of staff and students. In many cases, interesting lines of research were identified that were much wider in their application than just to this project. The way in which the project developed was largely influenced by hardware implementation. This proved to be much slower than we had hoped, but the delay had certain positive effects, in that it diverted effort into software techniques which would perhaps not have been so thoroughly investigated if the machine had been ready earlier on.

The other chapters give an indication of the variety of the work done during the project: hardware monitoring, research on distributed operating systems, and performance evaluation. We finish with a critical evaluation of the project as a whole. What was good about it? In what ways was it disappointing? Could we have done it better with hindsight? What should we do next?

1.2 Background

Think back to 1988! Parallel machines were moving out into the public domain and Edinburgh was setting up the EPCC (Edinburgh Parallel Computing Centre), with the transputer-based Meiko Computing Surface as its main parallel platform. The EPCC developed considerable expertise in adapting applications programs to run efficiently on its particular machine, but the user environment was primitive. Computer Science made use of the Computing Surface for its MSc course in parallel systems, and a number of MSc projects were based on it, but it was immediately evident that it was quite difficult for inexperienced programmers to write correct and efficient parallel programs. There was virtually no operating system, the only programming language was occam and there were no profiling tools or software support for program development. Some of the early MSc projects attempted to address these questions, but it was clear that they required much more time than an MSc student could give to them.

At the same time, departmental staff who subsequently formed part of the now more identifiable *Computer Systems Group* were looking for a research area that might form a unifying theme for staff and students in the department. Obviously, we wanted this to develop out of our own interests, which at that point were primarily in parallel systems and performance evaluation. But we also wanted it to be at the practical end of "mainstream" computer science, involving the construction and evaluation of real systems. Edinburgh had in a earlier period been at the forefront of operating systems development, and we had a strong desire that our research should be applicable in practical situations. A run through the literature suggested that little progress had been been made on the question of establishing design principles for operating systems for distributed memory parallel machines. On the other hand, there were some areas of parallel systems research that were strongly represented: for example, static task mapping heuristics and parallelizing compilers. We did not want to compete in the areas where research was already mature: in particular, we did not want to compete with what some large, well-funded American groups were doing. This led us to identify the problem which our research would attempt to address: how to construct an operating system for a distributed-memory MIMD machine that should be both efficient and easy for a naive programmer to use. In other words, we wanted to make life easier for the applications programmer by giving responsibility for efficient execution to the systems programmer.

We then had to consider what research would be required to underpin the design of such an operating systems. We discuss this in more detail in Section 1.3. However, we must just mention here that we had to work under a number of practical constraints. Although several members of the staff were interested in this proposed project, it was not necessarily their primary interest, and the project aims had to be drawn rather loosely, in the sense of "contributing to understanding", rather than in terms of a definite set of "goals", "milestones" and "products", so dear to the hearts of grant-giving bodies. Lack of financial and technical resources meant that we could not be very ambitious in constructing large hardware or software systems. Finally, we homed in on the idea of working towards a better understanding of the execution-time behaviour of process-based parallel programs, and of the interaction between applications programs and the underlying operating system. We would build a small parallel machine with hardware monitoring facilities so that we could make reliable quantitative performance assessments. We would, concurrently, implement a simulation system and develop models to give good performance predictions, and we would look at the possibility of using formal methods to guarantee properties of our software. These themes fitted in well with the expertise in the department, and would draw in a large number of people to contribute towards the project.

1.3 Research Directions

1.3.1 Monitor Architecture

Hardware monitoring was seen as essential if we were to get reliable information about the run-time behaviour of parallel programs, because of the short time-scales for message transfer, and because instrumenting programs in software affects the programs that are actually being measured. However, it would not have been feasible to interface monitoring hardware to a commercial computer, even if we had been allowed to interfere with a service machine to make such experiments. It was therefore a necessity to have our own machine where we had complete control, and, in the absence of any suitable existing candidate machine, this meant that we had to do the construction ourselves. This in itself involved interesting research. There were only a few existing hardware monitoring systems, and it was not clear what were the most useful kinds of data to collect, how it was to be generated or what the data rates were likely to be. There was also the question of the interaction between the monitoring system and the operating system, because, from the beginning, we envisaged that we would close the control loop by driving load balancing algorithms from information supplied by the monitoring system. The design therefore had to allow for future flexibility, because the machine (the Testbed, as it became known) was seen as a vehicle for future research that was not necessarily defined at the time of its construction.

Since we wanted to get the machine up and running as quickly as possible, we based its construction on some existing designs. The processor board was based on a design by Russ Green [13] and the interconnect on the Centrenet bus [15]. The overall design was thrashed out by a group of interested people, amongst whom were Tim Hopkins, Archie Howitt, Rob Pooley, Roland Ibbett and, expecially, Kayhan Imre [16]. Tom Whigham did much of the actual construction and we owe a special debt to Peter Lindsay, whose patient disentangling of faults and careful redesign ensured that, finally, and much behind schedule, the machine actually worked as it should.

1.3.2 Software Systems

Our original intention was to measure user programs to a greater level of detail than could be done with software monitoring, by taking the programs that had been developed for the Meiko Computing Surface and running them on the Testbed. However, there were two reasons why our interests shifted over time. The first reason was that the hardware took much longer to build than we had expected, and by that time the Computing Surface was being phased out in favour of the Connection Machine CM2 with a completely different architecture. The second was that we found two other directions that seemed both more interesting and more original than what we had initially intended to do.

The first of these was pursued by Kayhan Imre [16]. He developed ways of interpreting event data, allowing the programmer to combine low-level events into higher-level macro-events that could be related more easily to the structure of the application program. Imre proposed a graphical language for processing event traces and constructed a tool with good visualization facilities. This work was directed towards helping the programmer to achieve faster performance by showing up poor resource utilization. In addition to this substantial piece of work, there were a number of smaller-scale CS4 and MSc projects [12, 17, 25, 28, 29], supervised by D.K. Arvind, Rob Pooley and Rosemary Candlin, that were largely directed towards the question of providing a more friendly environment for the applications programmer. In addition, various researchers like Mark Davoren, Qiangyi Luo and Zhou Ji made contributions in this area. A list of project reports and publications can be found at the end. A number of short internal papers are grouped together in the two POSIE reports [31, 32].

The second important software theme concerned the construction of the operating system itself. Originally, Brian Tompsett intended to adapt the existing Manchester MUSS operating system for the Testbed, but this idea was abandoned in favour of a purpose-built design. Our interests soon focused more closely on the requirements for supporting process migration, and we realized that the Testbed would provide an ideal environment for detailed measurements on operating systems functions, because the hardware monitoring facilities allowed costs to be measured very precisely. It therefore seemed to us that here was something that had not yet been done, that we were in a position to do, and that would be valuable in itself: to make precise measurements of the costs of all the operating system functions under a variety of different user loads. Our interest here was not so much in the actual figures obtained for the Testbed as in understanding how higher level costs (eg for a message transfer) related to lower level, hardware-based functions. We would then be in a position to estimate the overheads introduced by the operating system itself, and to look for ways of improving the effciciency of its code.

Since one of our aims was to relieve the applications programmer of the burden of handling process allocation, our operating system would have to implement run time load balancing. This was not a new idea, since several such systems existed, but more in the context of traditional distributed systems, with a number of independent users and little interprocess communication. This environment was quite different from that on a more tightly-coupled parallel system, with lightweight processes and high rates of interprocess messages. The timing characteristics were quite different too, since many of the existing operating systems were implemented on top of Unix, and had neither fast nor reliable communications. We therefore intended to study the suitability of various proposed load-balancing heuristics, to see how appropriate they would be in our context.

It also appeared that little attention had been given to the question of eliminating the risk that the output of a user program might be affected by migration. We had thought at an early stage that formal methods might have a role to play in this project, and had had some discussion with Stuart Anderson about the suitability of the formal specification language **Z** for this purpose. Stephen Gilmore was subsequently co-opted as a supervisor for Paul Martin's PhD work [18, 22]. This addressed the problem of constructing a formal specification for that part of the operating system that was concerned with communication and process migration and proving that user programs were unaffected by migration. Martin then went on to measure operating systems overheads and to see how they depended on user programs of various types. An interesting, generally-applicable by-product of this research was the idea of relating performance metrics to schemas in the **Z** specification, and seeing where constraints in a schema, imposed to ensure a certain behaviour, impacted on performance [24].

1.4 Simulation

We were aware that construction of the Testbed might take some time, and that if we wanted to get actual results on a shorter time scale, we would have to provide an alternative way of performing experiments. To some extent, we were able to make use of the Computing Surface for preliminary performance measurements on communication patterns [2], but most of our serious experimentation was done by simulation. A prototype simulator based on Rob Pooley's ideas had been constructed by Thomas Guilfoy [14]. This was developed into a much more comprehensive tool by Neil Skilling in Chemical Engineering [35], and was used both by him and by Joe Phillips to investigate the effectiveness of static and dynamic process placement strategies (full details are provided in their PhD theses [33, 30]). MIMD was interesting in that it was an example of an object-oriented simulator. This enabled new classes for new types of program models and computer hardware to be introduced very easily, and gave great flexibility in handling new requirements. Its use was thus not restricted to simulating the execution of programs on the Testbed or the Computing Surface, though in practice it was set up to model the Computing Surface, which provided a means of validating simulator results.

Some interesting new developments came out of this work, which were much more generally applicable than just to the POSIE project. One of the problems in studying process allocation turned out to be the difficulty of finding suitable test programs: if one wants to find out whether a given allocation strategy is effective over the whole range of programs, it is necessary to have a large sample of programs with different characteristics. But, in fact, most programs had been parallellized with a particular platform in mind, and their original structure had been lost. There was therefore a strong argument for using synthetic programs with defined characteristics. This led on to the idea of parameterized models which could represent a very large range of different run-time behaviours. It was pointed out by Peter Fisk, then of the Statistics Department, that we should make use of standard techniques of experimental design, such as factorial experiments, for finding appropriate models and for making quantitative estimates of the influence of program parameters on performance. In fact, this proved an extremely helpful suggestion. Skilling implemented a tool [34] for the automatic generation of the large number of runs that were required for full factorial experiments. From the analysis we could get immediate indications of what program characteristics were important from the point of view of performance. We could thus relate performance to program characteristics, to allocation strategies and to the interaction between them, in a way that would have been much more laborious, or indeed impossible, without this systematic approach to experimentation. An overview of some of the statistical techniques that we found useful can be found in [30].

1.5 The Dynamics of Process Migration

The study of process migration was perhaps the main theme of the work done under POSIE. We have already mentioned Martin's work on the design of an operating system which allowed migration to take place in such a way that the user program was unaware that it was happening. This is obviously an essential requirement if the person who writes a program to run on a parallel machine does not have to take into account the fact that a process may move around during its execution. However, this is just a first step: process migration must also speed up execution, otherwise the overhead of moving processes will outweigh any benefit of better load balance. Martin himself made use of the Testbed monitor to compare the usefulness of a number of indicators that had been proposed in the literature. These had been used to select processors as donors or acceptors of processes. He showed that the length of the run queue was by far the most useful metric in selecting a donor processor, but that the best *process* to move was that with the highest external communication rate, rather than the one that imposed the highest load on the processor. He then showed in detail how the WATOR program (an ecological simulation program much used in load balancing studies) evolved over time and interacted with the operating system functions. Examples of Testbed output are shown in Chapter 2.1.

Phillips carried out a complementary study with a different intention. This was one of an extensive set of simulation studies based on synthetic programs with controlled parameter values. He was primarily interested in identifying those characteristics of a program that made it likely that process migration would improve performance. He found, for example, that for programs whose characteristics did not change appreciably over time, he could relate improvement under process migration to the value of the variance of process granularity, a property that could be obtained by preliminary profiling of the program. He also made a study of the detailed dynamic evolution of process movements for programs whose characteristics changed with time. He confirmed his results by measurements made on the Computing Surface. This work is described in detail in his thesis [30]. Unlike the Testbed, which is a bus-connected machine, processors on the Computing Surface are connected by a low-valence interconnect, and the process-migration decisions are made locally. Balance is thus achieved by a process of diffusion over the machine. Phillips was able to show that balance could be achieved very quickly on 16-processor domains, after just one or two migration attempts. The improvement he achieved was comparable to that obtained by Martin on a machine with a very different architecture and with a different algorithm.

Candlin was struck by the similarity between Phillips' graphs showing the evolution of process migration in time and those produced by impulse-driven linear dynamic systems. It was possible to describe overall system behaviour in terms of the *average* load imbalance over the processors, and to represent the actions of the process migration heuristic as that of a linear feedback controller. By considering a simple stochastic model, she was able to relate probabilistic changes on processors to the process variance identified as an important predictor by Phillips. This work confirmed previous results that indicated that certain, easily-obtained "macroscopic" properties of parallel systems could provide useful indications of their likely behaviour and performance [4, 3].

Chapter 2

The Testbed

2.1 Architecture

The Testbed is a distributed, message-passing multicomputer whose main features are its dedicated monitoring hardware and fine-grained, global clock. It is an example of a "hybrid monitor", where events are generated by software, but collected by special-purpose hardware. Such a monitor enables many detailed aspects of performance to be captured dynamically and with great precision, without appreciable disturbance of a running program. It is a single-user, diskless machine connected to a workstation acting as a file server on a LAN. The overall architecture is shown in Figure 2.1: the machine consists of six Motorola M68010 processor boards connected by a high bandwidth bus (Centrenet [15]). As can be seen from this specification, the Testbed was not exactly a massively parallel machine, even at the time when it was first designed. However, there are many examples where "modest" parallelism can be useful, in workstations for example, and the Testbed results would be immediately relevant on this scale. But, in fact, the small scale of the machine did not matter from the point of view of measuring operating systems functions, since link and processor load could easily be varied to simulate the effects of various environments.

Figure 2.2 gives a detailed view of part of the monitoring system. The board with the console and LAN connections is designated the "master"; the other boards are "slaves". Event records are collected from the slaves and delivered to the master where they can be aggregated and processed. The Slave Monitoring Interface (SMI) on the left accepts 16-bit words written by the associated slave processor, timestamps them from a global clock with a 2μs resolution and stores them in a buffer. The Master Monitoring Interface (MMI) polls each of the five SMIs in turn, collecting first a timestamp and then the associated event. The monitoring bus is only used by the Testbed operating system and not directly by the application threads.

It is the operating system's responsibility to generate events, and it has considerable flexibility in what it chooses to generate. It may generate events that

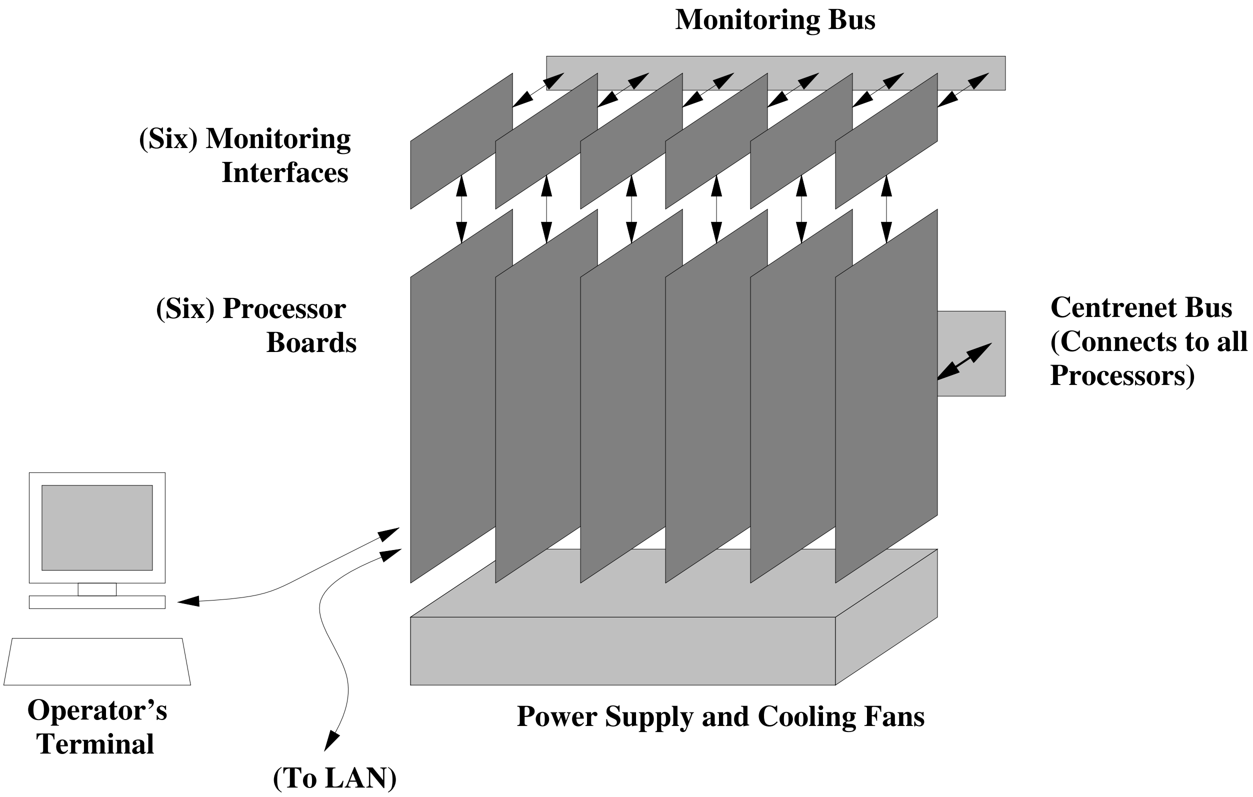

Figure 2.1: An overview of the Testbed.

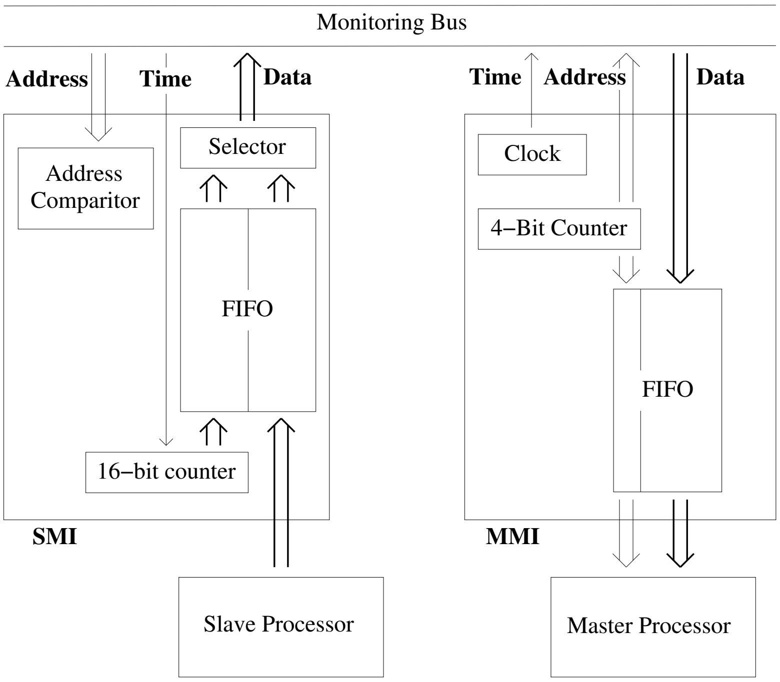

Figure 2.2: Design of the monitoring system

refer to its own operation, for example, recording when particular operating systems functions are called, or it may pass on information about a user program. With suitable compiler options, requests for event generation can be planted in the compiled code of a user program, and standard profiling operations conducted. In fact, we were less interested in this possibility than in collecting information about the low-level operation of the system. Typical events were therefore: sending a message, receiving a message, starting a process, or terminating a process. The master monitor could thus keep a record of overall system activity and see where contention for resources was highest.

2.2 Some Typical Measurements

Some typical experiments are presented here (more details can be found in [18, 21, 20]). Some metrics were simple to obtain, eg, the time required for the operating system to service a request for communication. Others were more complex—the latency of a remote communication, for instance, depends on the latencies of each of three phases of the inter-processor message transfer protocol. Figure 2.3 shows the latencies of local (intra-processor) and remote (inter-processor) communication for a range of message sizes. The graphs have several features which may be directly related to the implementation of the Testbed's communication routines. The communication latency is broadly linear with message length, as would be expected from the fact that the communication hardware is reliable and does not perform resends. The local send latency has pairs of steps at 1 Kbyte intervals which are caused by the block-copy routine crossing virtual memory page boundaries in the sender's and receiver's memory areas. Both types of communication have setup costs, and the two graphs diverge because remote communication involves three memory-block copies while local communication involves only one.

The next set of experiments shows how the system behaves under different types of user program. A synthetic test program was constructed to show the various combinations of light/heavy computation/communication loads that a program might impose on the system. On invocation, the test program creates several threads on each Testbed processor. These threads communicate with each other via a network of channels, some local and some remote. The test program passes through eight distinct phases, with a single, master thread controlling the transition between phases by means of barrier synchronization. The different phases are designed to explore the extremes of dynamic behaviour, and are summarized in Table 2.1.

Figure 2.4 shows how the length of the ready queue varies over five processors (named earth, fire, water, air and space) as the test program moves through its eight phases. These graphs show good agreement with the expected dynamic behaviour of the program. It is obvious that the average ready queue length is greater in the compute-bound phases 1 to 4 than in the other, communication-bound phases. However, while this gives a good indication about competition for

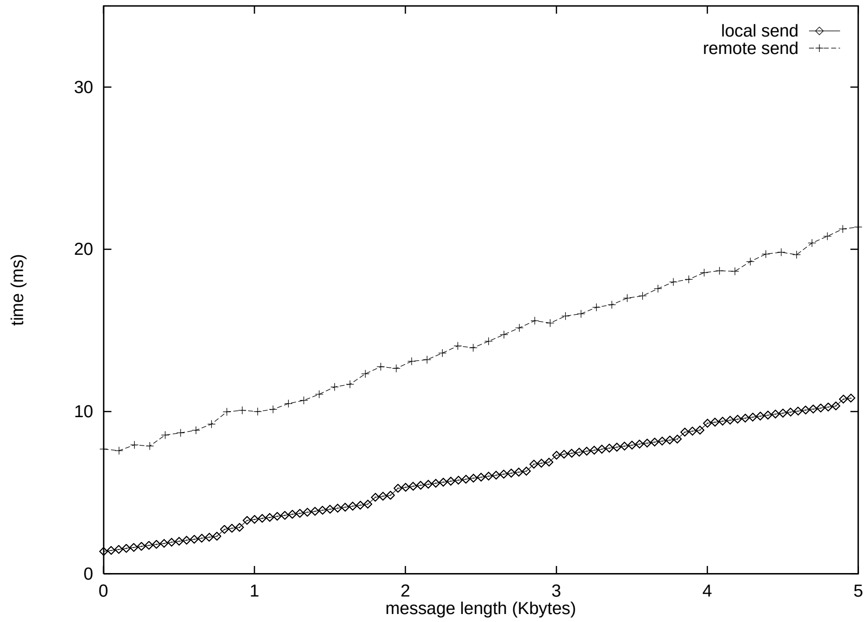

Figure 2.3: The latencies of two operating system communication services.

1	Compute	Local	Short
2	”	”	Long
3	”	Remote	Short
4	”	”	Long
5	Communication	Local	Short
6	”	”	Long
7	”	Remote	Short
8	”	”	Long
Phase	*...-bound*	*... communications*	*... messages*

Table 2.1: The eight phases of the test program (see Figs 2.4 and 2.5)

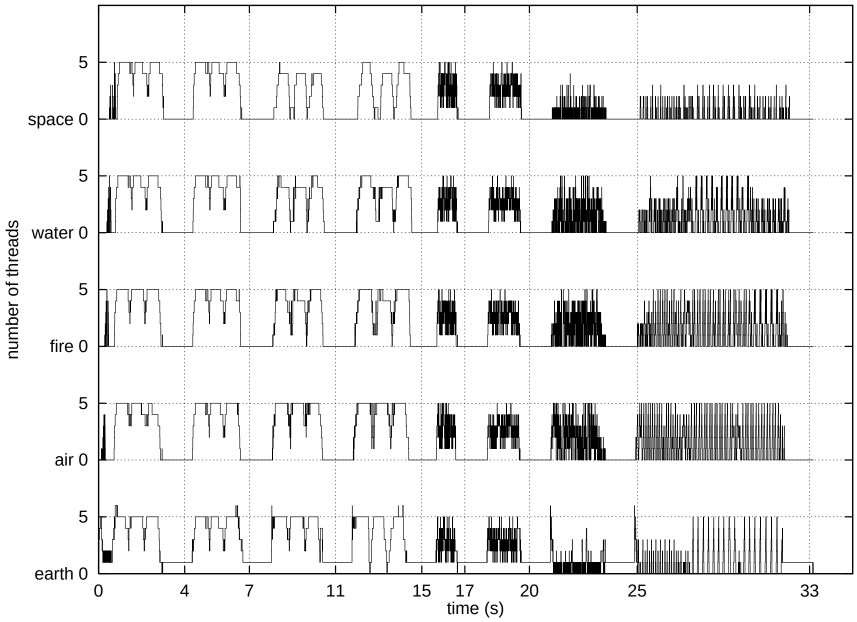

Figure 2.4: Length of the ready queue on each of 5 processors during the program execution. Time is measured in secs.

processor resources it says little about the load on communication resources—the short ready queues in phase 8 might, for example, have been caused by threads blocking on file input.

Figure 2.5 presents a better view of what is happening on the communication network. This is in fact a bus, but Figure 2.5 presents results in terms of pairwise communications between processors, which can more easily be interpreted in terms of the test program. As would be expected, the remote communication in phases 3, 4, 7 and 8 is particularly clear. The relatively heavy communication at the start of execution occurs as the threads are created on one processor and are then distributed over the others.

It can be seen from these two sets of results that an accurate picture of program execution, and of the demands that it places on the system resources, can be obtained from the monitoring data. It is worth emphasizing once again, that hybrid monitoring places a minimal additional load on the system, and that it would be impossible to obtain such accurate and up-to-date results by software monitoring.

The final graphs show results for a realistic application, the WATOR program. This is a somewhat simplified ecological simulation based on a grid of ocean points where fish and sharks are in competition with one another. A process handles a fixed set of grid points throughout the simulation, which proceeds by a series of iterative steps. By the end of each iteration, the grid points have been updated, and boundary exchange takes place ready for the next round. Several processes are initially assigned to each processor, but process activity varies as fishes and

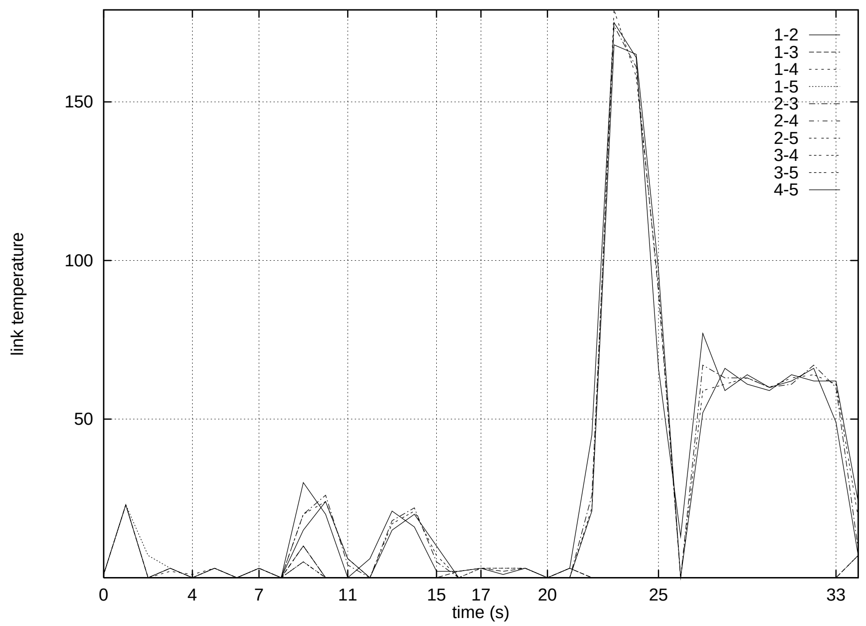

Figure 2.5: The number of pairwise remote communications per second

sharks breed and die off and unbalanced loads soon develop. Figure 2.6 shows the output during 100 simulation steps. The predominant feature is the succession of fish booms and busts. On the other hand, the communication load is very even over the network, because the amount of boundary data exchanged is the same for any pair of processes and all processors host the same number of processes (this rather uninteresting graph is not shown here).

When load balancing is enabled, the situation changes. Figure 2.8 shows that the ready queue lengths are now, on the average, shorter and more evenly shared between processors. This means that the load balancer is performing correctly and is distributing work sensibly. Furthermore, the execution time for 100 simulation steps is now 600s, a 30% reduction over the case with no load balancing. However, communication traffic has now increased. Figure 2.9 shows several peaks of almost 200 messages per second. While the load balancer does try to reduce communication traffic, it optimizes computational load first.

These results provide a justification for the assumption that the operating system can usefully manage run time load balancing and produce a noticeable improvement in execution speed. This looks promising from the user's point of view: it is extremely tedious to implement process migration within a user program, and a distraction from the main purpose of getting out results. However, if a user is to have confidence that the program will actually produce the *same* results as it would when processes stay where they are initially loaded, the operating system has to be able to offer a guarantee that this will be the case. The work in the next chapter describes an approach to solving this problem.

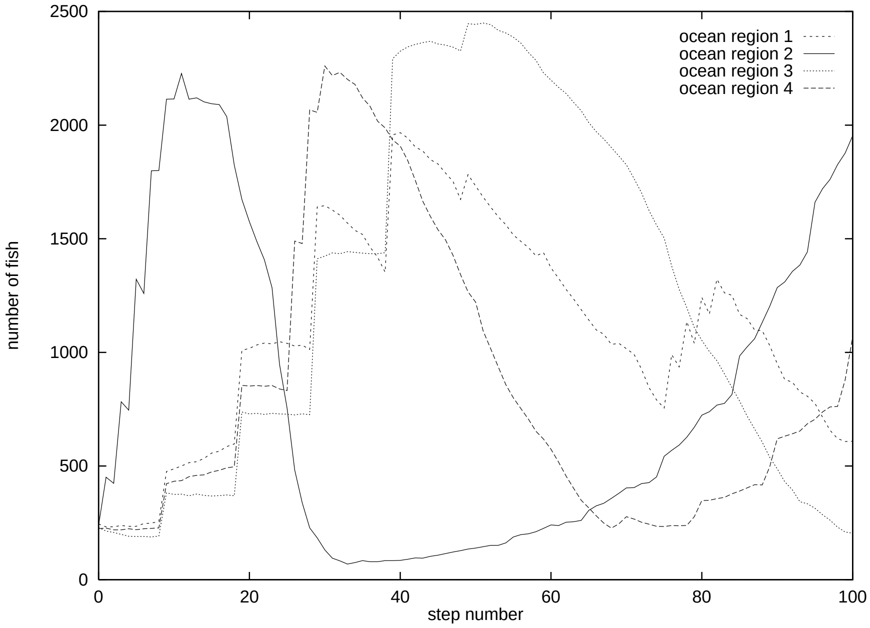

Figure 2.6: Variation in number of fish during 100 steps of the WaTor simulation.

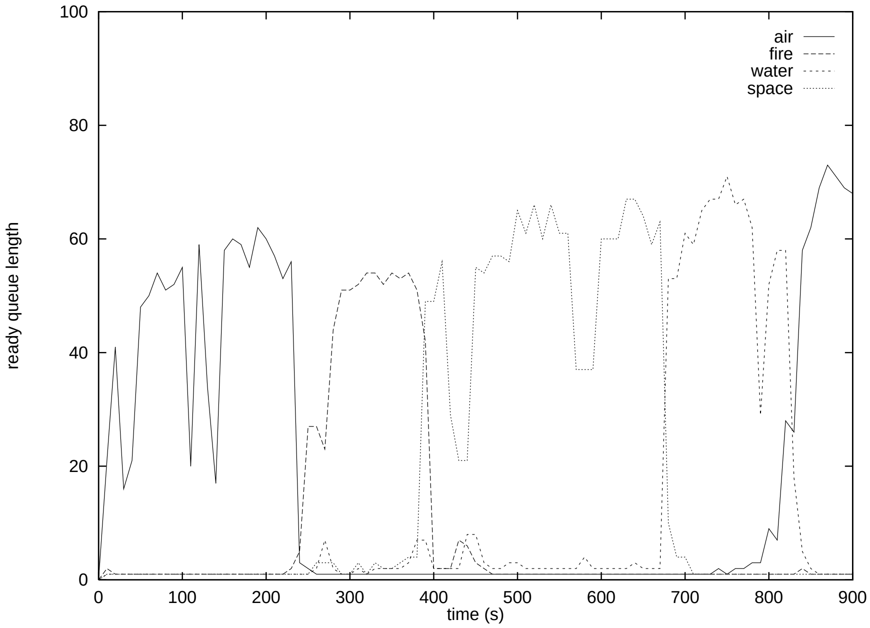

Figure 2.7: Ready queue variation during 100 steps of the WaTor simulation **without** load balancing.

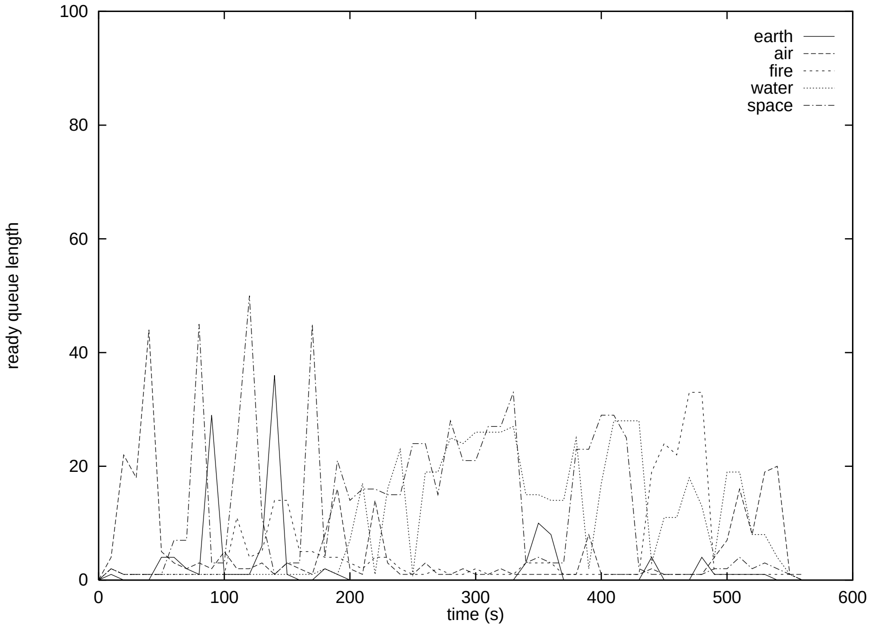

Figure 2.8: Ready queue variation during 100 steps of the WaTor simulation **with** load balancing.

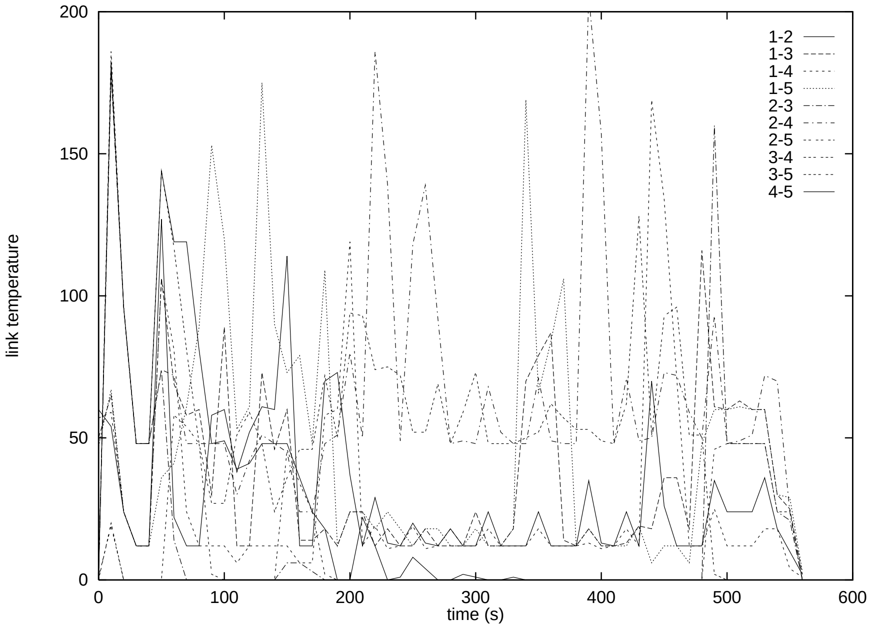

Figure 2.9: Variation in link load during 100 steps of the WaTor simulation **with** load balancing.

Chapter 3

Operating System Design

3.1 The Formal Specification

Many more details can be found in Martin's thesis or in [18, 19, 23].

3.1.1 Constructing a Model

The mechanism of process migration is a complex procedure in which processors have to negotiate about the process to be transferred, and the process has to be stopped on one processor, copied to another and restarted. Outstanding and future messages then have to be delivered to the new address. There is scope for things to go wrong because of wrong interleavings of the messages between the operating systems on the sender and receiver processes. The first essential requirement, therefore, is to ensure that communication is handled correctly, because both the preliminary negotiation, and the subsequent process transfer depend on this. The formal specification therefore concentrates on the part of the system that handles communication and migration. The development of a complete specification is an extremely lengthy business, and it was not possible to underatake it in the time available. Nor was there time to do a formal refinement of the specification down to the code—that was developed from the specification by hand, according to informal principles.

The Testbed's programming environment is represented in Figure 3.1. In this example, four concurrent processes or "threads" of control execute on two slave processors. Scheduling is pre-emptive, with compute-bound threads limited to a 20ms timeslice. Each thread can be uniquely identified and threads communicate via unidirectional, non-buffered channels. Channels are also uniquely identifiable and are associated with the same pair of threads for the duration of the execution. When a channel connects threads executing on the same processor, communication is effected by a single memory-to-memory copy, but when a channel connects threads on different processors, a protocol with multiple phases is required, involving the transmission of typed messages over the Centrenet bus. Load balancing is performed by moving threads between processors. Formal methods are

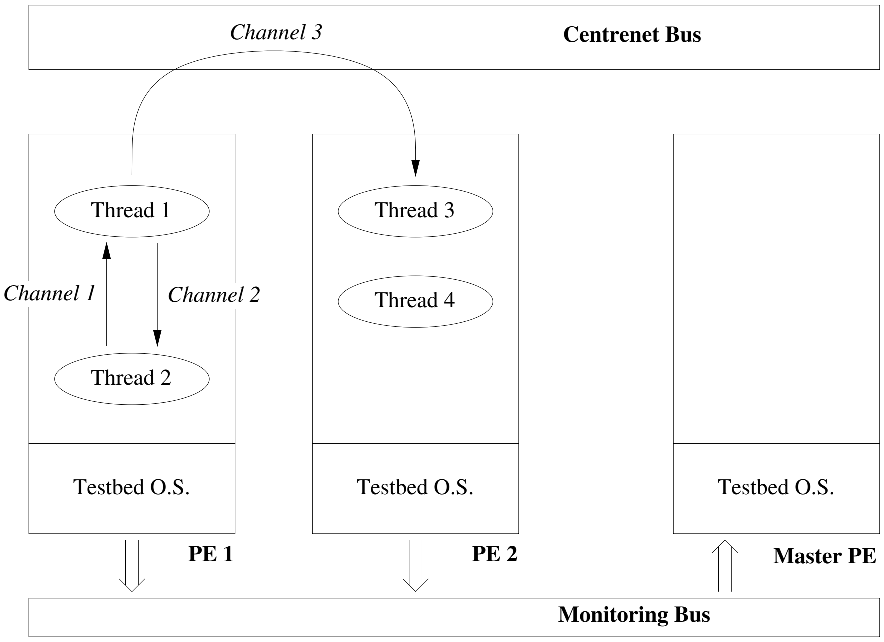

Figure 3.1: An example application executing on the Testbed.

used to show that thread migration is, in certain important respects, safe and correct.

Z was chosen as the formal specification language. In **Z**, we can build a model of the system in terms of sets and functions of *given types*, which have meaning to the designer of the system, but which are not further defined. The specification describes the states of the system in terms of these sets and functions and various predicates that define invariants. This information is bundled up into a unit called a *state schema*. In addition there are *state-changing schemas* which specify the before and after values of state variables under a given state-changing operation. Schemas provide a powerful, but not very easy-to-use, means of structuring a specification. In spite of this, **Z** had many good features from our point of view: we wanted to be able to describe the state before and after a communication, or before and after a migration, without going into details about *how* these operations were carried out.

3.1.2 The Specification

It is not possible to to go into the specification in great detail: just a small part of the **Z** model is presented to give the "flavour" of the specification. This is structured in terms of two conceptual levels. At the lower level, components of operating system functions are represented. An example is the schema $Send1$ which models the mechanics of a step in the inter-processor communication protocol. At the higher level, schemas deal with the question of *when* an operation is applied. For instance, the $Send2$ schema indicates that a $Send1$ schema is applied only when a process's next instruction is to send a message.

We start by declaring basic types to represent processing elements, executing threads and channels:

$$[PE, THREAD, CHANNEL]$$

Next we define the *state* schema (for simplicity, only part of this is shown). This forms the basis for the main part of the specification which consists of *state-modifying operations*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{State}1} \\
\hline
\mathit{ready_queue} : PE \leftrightarrow THREAD \\
\mathit{waiting_sender}, \mathit{waiting_receiver} : PE \nrightarrow \\
\qquad (CHANNEL \nrightarrow THREAD) \\
\mathit{location} : PE \nrightarrow (CHANNEL \nrightarrow PE) \\
\hline
\end{array}$$

The four identifiers—*ready_queue*, *waiting_sender*, *waiting_receiver* and *location* represent well-known operating system concepts: they model the physical distribution of threads which are waiting to communicate, either as senders or receivers, on given channels. This information is represented in terms of various mappings which specify the relations between threads, channels and locations. When a thread t is executing (or is in the queue waiting to execute) on the processing node modelled by p, the pair (t, p) will be included in the relation *ready_queue*. When a thread is blocked waiting to send a message over a communication channel c, a maplet $p \mapsto \{c \mapsto p\}$ is included in the *waiting_sender* function (*waiting_receiver* fulfils a similar role for receiver threads). When the Testbed has a channel connecting a thread executing on processing element p_1 to another thread executing on p_2, there will be a maplet $p_1 \mapsto d\{c \mapsto p_2\}$ in the function *location*.

Channel communication is implemented on the Testbed using three different kinds of message: "ready to send", "ready to receive", and "data". The first two are used for flow control, the third kind contains the actual data to be transferred between threads. This can be represented by the generic message type *MSG*

$$MSG ::= rts \mid rtr \mid data$$

Now we look at the first part of a state-modifying operation which describes the changes that take place when a thread requests to send data over a channel. The schema *Send*1 is parameterized by having input arguments (identifiers with question marks) and output arguments (identifiers with exclamation marks). The declaration $\Delta State1$ is a form of macro inclusion and introduces instances of the variables defined in *State*1 in both normal unprimed and primed forms. The unprimed variables represent components of the state *before* the operation. The primed variables represent the state *after* the operation. Thus, *Send*1 defines the primed variables of *State*1 and the output argument in terms of the unprimed variables of *State*1 and the input arguments.

$$
\begin{array}{|l}
\hline
\textit{Send}1 \\
\Delta \textit{State}1 \\
st? : \textit{THREAD} \\
c? : \textit{CHANNEL} \\
sp? : \textit{PE} \\
msgs! : \mathbb{P}\,\textit{MSG} \\
\hline
c? \notin \mathrm{dom}(\textit{waiting\_receiver}\ sp?) \Rightarrow \\
\qquad \textit{ready\_queue}' = \textit{ready\_queue} \setminus \{sp? \mapsto st?\} \wedge \\
\qquad \textit{waiting\_sender}' = \textit{waiting\_sender} \cup \\
\qquad\qquad \{sp? \mapsto \{c? \mapsto st?\}\} \wedge \\
\qquad msgs! = \mathbf{if}\ c? \in \mathrm{dom}(\textit{location}\ sp?)\ \mathbf{then}\ \{rts\} \\
\qquad\qquad \mathbf{else}\ \varnothing \\
\hline
\end{array}
$$

The interprestation of this schema is that a sending thread $st?$ executing on source processor $sp?$ is ready to send a message over channel $c?$. The output argument $msgs!$ is defined to be the (possibly empty) set of messages that are to be transmitted to another processor by the network interface.

The predicate section defines constraints on this operation, and says what its overall effect is to be. If there is no receiver thread currently blocked on the channel $c?$ at the sender's processor $sp?$, the sender thread $st?$ will be removed from the ready queue and placed in the wait queue. If the location of the other end of channel $c?$ is on a different processor then a "ready to send" message is added to the output variable $msgs!$, ready to be transmitted to the appropriate processor. Note that this schema only says what happens when the sender is ready before the receiver: the full specification contains another, similar schema to say what happens when the receiver is ready first.

At the higher level of the specification, the new type $REQUEST$ is introduced to represent "send" and "receive" requests on a particular channel.

$$
\begin{array}{rl}
\textit{REQUEST} ::= & \textit{send}\langle\!\langle \textit{CHANNEL} \rangle\!\rangle \\
| & \textit{receive}\langle\!\langle \textit{CHANNEL} \rangle\!\rangle
\end{array}
$$

The extended state schema ($State2$) encapsulates the notion of thread programs as sequences of requests and specifies that $Send1$ can only be applied if certain preconditions are met, ie, that the sender thread $st?$ is in the ready queue and that $st?$'s next instruction is to send.

$$
\begin{array}{|l}
\hline
\textit{State}2 \\
\textit{State}1 \\
\textit{program} : \textit{THREAD} \rightarrowtail\!\!\!\!\!\!\rightarrow \mathrm{seq}\,\textit{REQUEST} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Send2 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta State2 \\
st? : THREAD \\
sp? : PE \\
c? : CHANNEL \\
msgs! : \mathbb{P}\, MSG \\
\hline
(sp?, st?) \in ready\_queue \\
send(c?) = head(program\; st?) \\
Send1 \\
\hline
\end{array}
$$

3.1.3 Proofs

Proofs were in a "rigorous" style, and were lengthy, because a lot of different cases had to be considered. For example, only one instance of a communication operation has been shown here ($Send1$), but in the full specification there are five others: a "request to receive" and four operations for exchanging status and flow control messages between processing elements. So in this section we just give an idea of how the proofs were tackled. The approach was to prove first that, in the absence of migration, the communication protocol had certain desirable properties. Then it was proved that migration preserves these properties. Proofs proceeded by induction on the number of communications and the number of migrations respectively.

We limit our task by considering only those sequences of operations on one, arbitrarily chosen, channel—this is acceptable because channels are independent of each other. We then limit the number of sequences by ruling out illegal subsequences. For example, our model of communication has one-way channels so we disallow sequences involving a thread that performs a send request and then a receive request on the same channel. We can express a restriction like this formally, thus:

$$
\begin{array}{l}
\forall\, State2;\ t : THREAD;\ c : CHANNEL \bullet \\
\qquad send(c) \in \operatorname{ran}(program\; t) \Rightarrow \\
\qquad\qquad receive(c) \notin \operatorname{ran}(program\; t)
\end{array}
$$

We now enumerate possible subsequences of operations corresponding to a single message transfer on a given channel. Here we consider one of the possible subsequences for the first communication over a channel. We will assume that the state has already been correctly initialized, and that the next message transfer involves a send request followed by a receive request on the same processor. We define the state before the first communication as follows:

$$
\begin{array}{|l}
\hline
\textit{S0} \\
c? : \mathit{CHANNEL} \\
\mathit{State2} \\
\hline
\forall p : \mathit{PE} \bullet c? \notin \mathrm{dom}(\mathit{waiting\_sender}\ p) \land \\
\qquad c? \notin \mathrm{dom}(\mathit{waiting\_receiver}\ p) \land \\
\qquad c? \notin \mathrm{dom}(\mathit{location}\ p) \\
\hline
\end{array}
$$

Informally, this schema says that there are no senders blocked on channel $c?$, no receivers are blocked on channel $c?$, and nothing is known about the locations of the ends of channel $c?$.

We now define the consequences of applying the $\mathit{Send1}$ schema to $\mathit{S0}$ in a new schema $\mathit{S1}$:

$$
\begin{array}{|l}
\hline
\textit{S1} \\
\Delta \mathit{State2} \\
c? : \mathit{CHANNEL} \\
st? : \mathit{THREAD} \\
sp? : \mathit{PE} \\
msgs! : \mathbb{P}\,\mathit{MSG} \\
\hline
st? \notin \mathrm{ran}\ \mathit{ready\_queue}' \\
(\exists_1\, p : \mathit{PE} \bullet c? \in \mathrm{dom}(\mathit{waiting\_sender}'\ p)) \land \\
\qquad (c?, st?) \in \mathit{waiting\_sender}'\ sp? \\
(\forall p : \mathit{PE} \bullet c? \notin \mathrm{dom}(\mathit{waiting\_receiver}'\ p)) \\
\hline
\end{array}
$$

This schema says that the sending thread $st?$ has been removed from the ready queue and blocked on $\mathit{waiting\_sender}$ at exactly one ($\exists_1$) processing element p. Furthermore, there is no processing element with a receiver blocked on channel c.

The second part of the subsequence, the receive request, is applied in the same way to derive another state schema, $\mathit{S2}$ (not shown). We are now able to observe from inspection of $\mathit{S1}$ and $\mathit{S2}$ that the communication subsequence has certain desirable properties. For instance: $\mathit{S1}$ tells us that the sender was blocked, $\mathit{S2}$ tells us that data was communicated and the sender was unblocked, therefore we deduce that communication involved synchronisation—a desirable property for our model of communication.

The full proof proceeds by enumerating all other legal subsequences and by showing in each case that the desired properties hold for the first communication, and by induction for subsequent communications.

We now have to consider what happens when migration is allowed. A similar enumeration of communication subsequences is performed and it is shown for each subsequence that at whatever point a communicating thread migrates, the state of channel before the migration is, in all essential aspects (a definition that has to be spelt out), the same as the state of the channel after the migration. So we can show that migration preserves desired properties of the communication protocol. All in all, the proofs require some thirty pages of detailed (although somewhat repetitive) working.

3.2 Performance and the specification

By relating performance measurements to operation schemas, it is possible to to see how constraints imposed in the predicate section of a schema influence performance. If it turns out that an operation is apparently inefficient, one can then pinpoint the origin of the inefficiency and see if it is safe to relax the constraint.

However, communication involves the execution of several operating system functions, corresponding to several schemas. If threads on different processors are to communicate, for example, then the $Send1$ module, the receive request modules and three inter-processor message modules are needed—something which **Z** cannot represent easily. We *can* deduce legal sequences of schema applications, as we did in the proofs section, but this does not provide a convenient way to repesent actions at run time. We would like a more dynamic representation of sequences of legal schema applications.

Finite state machines (FSMs) proved to be a more appropriate representation. An FSM for a particular high-level operation can be derived from the specification by considering the constraints on the state-changing operations imposed by the predicates. The time required to execute each elementary operating system function can be extracted from the event traces collected during the execution of test programs on the Testbed. These times can then be attached to the arcs of the FSM and total times for composite operating system services found by summing along a path.

FSMs are a convenient and natural way to express performance results. Most event traces are large and complicated, typically containing interleaved streams of events from multiple pairs of communicating threads. The FSMs, however, made it easy to construct parsers to extract times for particular modules. It was then straightforward to identify the composite modules which imposed the greatest overheads and to focus in on particular modules (and hence specification schemas) which accounted for a large proportion of those overheads.

Chapter 4

Performance Experiments

Most of the experiments described in this chapter were simulation experiments, but a substantial amount of work was also carried out on the Meiko Computing Surface. More details can be found in [30, 33, 4, 3].

4.1 Experiments on the Meiko Computing Surface

The first studies were based on real programs running on the Meiko Computing Surface. This was a transputer (T800) array that was physically divided into a number of separate domains consisting of from 1 to 128 transputers. Most of our work was done on 16-tranputer domains, which had reasonable availability, with an occasional foray on to the 64-processor domain.

When we started the POSIE project, this was our only available "platform", either real or simulated. However, it was a service machine and we were limited in what we could do. We could only get restricted access to the machine, and got little technical support when we wanted to get down into its low-level operation. In addition, the unreliability of its hardware and software made program development unnecessarily laborious. So we tended to use the machine largely as a means of providing parameters for simulation models, and for validating results obtained by simulation. Nevertheless, our experience was valuable, and confirmed our feeling that the missing link in parallel computing was the systems software. Although new products like CS-Tools, which facilitated process placement, and Unix-like operating systems became available subsequently, it was clear that their design had not been based on any fundamental understanding of the run-time properties of parallel programs. We therefore added a new research topic to our list: to investigate the statistical properties of parallel programs. It seemed to us that it would be a hopeless task to attempt automatic process allocation unless parallel programs showed similarities in a statistical sense. On the other hand, if it turned out that we could identify classes of programs, all of which had a similar pattern of run-time behaviour, we would be able to look for a good allocation strategy which had a high probability of working well for *any* program of the class.

4.2 Simulation Experiments

Our aim here was to get general results which applied to the whole population of parallel programs. However, given that we were not sure when we started that we would indeed be able to find similarities between sets of programs, we thought we should be less ambitious and first study programs that were likely to be well-behaved, and have reasonably stable properties over the duration of the execution. We therefore fixed on occam-type programs for our investigations. In fact, there were other good reasons for choosing occam programs. We wanted to connect this work up with the POSIE operating system work, and it seemed to us that it would be very much easier to construct an operating system that handled process migration if we could assume that user programs conformed to the occam conventions. Just to mention some of the good points: dynamic process or channel creation is forbidden, there is no recursion and communications are synchronous. This considerably simplifies memory-handling because the sizes of process work spaces are known at compile-time. The one-to-one, synchronous communications means that there is never more than one outstanding message on a channel, and thus eliminates the need for buffer-handling. These restrictions are very helpful from the point of view of proving safety properties of the system.

To find results that were applicable to the whole class of these programs we wanted to work with random samples of synthetic programs, generated from from a parameterized model. So the first task was to find a suitable parameter set, in terms of which we could realistically characterize a program. It turned out that characterizing programs in terms of the moments of the distributions of rather obvious properties worked very well. Programs with the same parameter values produced very similar performance metrics, whereas programs with different parameters clearly had different behaviour. However, it also turned out that some of the parameters that we proposed had rather little effect on the resulting performance. Here for the first time was some quantitative evidence about the relative importance of various, easily-obtained program properties.

To obtain our set of parameters, we started with a weighted process graph. A node weight represents processing characteristics of the program, for example, the time average of the number of instructions executed by the process between channel I/O operations. An edge weight represents a communication property, for example,the time averaged message length on a channel. The next step is to compress the information in this graph, and summarize the weight distributions by a small number of statistics. This gives us a manageable number of macroscopic parameters for the graph as a whole.

By working backwards, we can generate synthetic programs that have preselected parameter values (details are given in [34]) and measure those performance metrics in which we are interested, either on a simulator, or, in principle, on a real computer. The great advantage of using this method is that we can choose parameter levels in advance, and set up factorial experiments. Basically, the idea here is that a relatively small number of experiments, each with a different combination of parameter (factor) values, can give the same amount of information about a system as a much larger number of experiments in which only one factor is allowed to vary at a time. Furthermore, factorial experiments give information about interactions between factors which is not otherwise obtainable. Factorial designs allow different models to be explored quickly, and the relative importance of the factors to be determined by an analysis of variance. In a two-level factorial design, each factor can appear at either a high or a low value, and each run takes a particular combination of high and low values of each parameter. A full factorial experiment allows the contributions to the variance of a response variable to be estimated for every factor, and for all interaction terms. A linear equation can easily be fitted from the results, giving an empirical, quantitative relation between a performance metric and factor values. This equation can then be used to predict the performance of an arbitrary program whose parameter values are known. This description refers to one of the simplest experimental designs—there are many others, including multilevel factorial designs which allow polynomial equations to be fitted.

This systematic approach, of proposing an empirical model, running factorial experiments and analysing the results by standard statistical procedures proved to be extremely useful. It allowed us to make quantitative statements of the form "When process migration is implemented it improves such and such a metric by so much, at the 95% confidence level", rather than saying "These results suggest that process migration improves performance".

The examples below show the results of some typical experiments, where the factors have been taken as

$$\{N, c, \mu_{cg}, \sigma_{cg}, \mu_{mg}, \sigma_{mg}\}$$

Here N is the number of process nodes and c the average node degree. These two factors reflect static, structural properties of the program. The parameters μ_{cg} and σ_{cg} summarise the computational properties in terms of the mean and standard deviation of the distribution over the graph of the time-averaged process granularity (the number of instructions between successive channel communications). The corresponding message length parameters are μ_{mg} and σ_{mg}. These parameters indicate the level and variation in demand for resources over the program graph averaged over the whole execution time. However, we make no attempt to represent the detailed instruction sequences, but generate random delays when we come to simulate the program.

Our first experiment is designed to show whether we get similar results for programs with the same parameter sets (and different instruction sequences), and different results when the parameter values are different. The performance metric which is studied here is closely related to the percentage processor utilization,

Source	D o F	Sum of Squares	Mean Squares	F Ratio	F Prob.
Between Replicates	2	0.00124	0.000621	1.013	0.366
Between Treatments	63	1.96	0.0311	50.273	0.001
Residual	126	0.0773	0.000613		
Total	191	2.038			

Table 4.1: Analysis of Variance Table for Percentage Utilization

and is a measure of how effectively the processors are kept working, and hence of the expected execution time. A two-level full factorial experiment was carried out with factor levels set to include those of a large number of real programs and with several replicated runs for each combination of parameter values. Some typical results are given, based on an analysis of the simulation experiments: they consist of an *analysis of variance table* and an *effects table*. A certain amount of explanation will now be given, so that the tables can be interpreted! Any elementary statistics book will provide a fuller description.

An analysis of variance (ANOVA) is a way of processing experimental data so as to be able to make formal statistical tests of significance. Do different parameter values lead to different results, or can differences in results be attributed to experimental error? An ANOVA is based on the idea of partitioning the variance in different ways. We first of all find the mean from the results for all the combinations and all the replicates (the *grand mean*). We also find the group means for each of the sets of replicates with the same parameter values. We then calculate the sum of squared deviations of the replicates from their own mean, and of the group means from the grand mean. To bring everything on to the same scale, we have to divide by the *degrees of freedom*, the number of independent observations in each set. The results are the *mean squares*, which will be about the same if the different parameters have no effect. The *F ratio* is to be compared with an F distribution and gives the *F probability* that this result would have arisen by chance.

The analysis of variance table for this experiment is given in Table 4.2. The differences in mean values for replicates with the same parameter values are very small whereas there are large differences when parameter values are different. More formal hypothesis testing on the F values shows that it can be accepted at a very high confidence level that replicates produce essentially the same result, whereas there is a highly significant difference between results for different factor levels. Now that this has been established, we can go on to find the size of the effects due to the different factors. The results of the analysis are shown in Table 4.2. The lefthand column lists those factors that were set at their high values. The second column shows the value that should be added to the overall mean (gm) when those factors are high, the t-value has to be tested against a t-distribution in order to find the confidence interval for the effect, and the last column indicates the percentage

Effect	Estimate	t-value	% Var
(gm)	0.1557	87.0936	
N	0.0505	28.27	24.05
μ_{cg}	0.0128	7.15	1.54
$c\,\mu_{cg}$	-0.0114	-6.39	1.23
σ_{cg}	-0.0783	-43.83	57.80
$N\sigma_{cg}$	-0.0155	-8.65	2.25
$\mu_{cg}\,\sigma_{cg}$	-0.0140	-7.81	1.84
μ_{mg}	-0.0122	-6.80	1.39
	Standard Error = 0.00179		

Table 4.2: Estimates of Effects Contributing at Least 1% to the Variability of the Response

of the variation that can be attributed to the effect. Positive effects correspond to an improved utilization relative to the mean, and negative to reduced utilization.

We can draw some interesting conclusions from the effects table, and relate them to our knowledge of the type of program and the machine environment. One of the most interesting features is the lack of sensitivity to message length. This may well be a consequence of the particular communications harness used and the fact that transputers can overlap computation and communication, as we also found the same thing on bigger domains of 50-60 tranputers. This identifies one set of program characteristics that we do not need to bother about– at least in this particualr environment. The other interesting characteristic is the predominant importance of σ_{cg} as a predictor of performance degradation. One would indeed expect processor utilization to drop as the value of σ_{cg} increased, because this corresponds to a mismatch in process compute-times and hence increased synchronization delays. In fact σ_{cg} also turned out to provide a useful indication of how likely it was that a program's performance could be improved by process migration. In a sense this is not surprising. We have already seen that to a first approximation we can forget about message transfer overheads, and that performance is largely determined by good load balance. If we take a set of processes and allocate them at random, but in equal numbers, to a set of processors, we shall get good load balance if σ_{cg} is small. As it becomes bigger, however, there is an increased probability of a variable load over the processors, and there is more room for improvement by rearranging the processes. Experiment confirmed that process migration did actually improve performance, increasingly as σ_{cg} increased.

We were then able to go further, and fit equations relating performance metrics to parameter values. It was interesting that we could do this sensibly, because the parameters proposed above refer only to the *program*, and not at all to its interaction with the machine. A comparison of observed and predicted results showed that that the agreement was very reasonable, given the simplicity of the model— about 8%.

Phillips also tried to find parameters that expressed the interaction between program and machine and succeeded in getting an agreement within 5% between observed random program executions and the predictions from his fitted equations. The additional parameters could only be measured during the run of the program, and would therefore not be so useful for initial predictions as those based purely on program characteristics. They were however used to improve the effectiveness of the process migration algorithm itself, by choosing migrating processes in such a way as to improve the values of the parameters.

The experiments described above were some of the simplest, and are primarily intended to give an idea of the methodology. The same approach was used in other investigations, including an extensive set of experiments to study the ways in which program parameters affected the usefulness of various static and dynamic process allocation strategies. It should be pointed out that the actual numbers we obtained from our experiments are specific to a particular machine environment, and that they depend on such factors as the machine topology, the computation to communication speeds, and the fixed overheads of message transfer. Also, of course, we have only looked at one type program model, although it is a commonly used one, and, with suitable parameterization, can represent a large variety of run-time behaviour. However, the experimental approach is quite general, and could be used to explore different program models (of functional or object-oriented languages, for example) and different machine architectures.

A better understanding of the reasons why certain parameters were predominantly important came out of Candlin's probabilistic modelling of of the migration procedure. This showed how local changes on individual processes were related to certain global properties like total number of processes moved. Earlier work had shown that a "restricted random" mapping, where as far as possible equal numbers of processes were loaded onto each processor, led to good performance for most programs. If we assume that initally all processors have the same number of processes and that the process weights are normally distributed, it is easy to find the distribution of total load over the processors. Then we can show (with certain simplifying assumptions!) that the variance of processor load decreases approximately exponentially with each successive invocation of the migration algorithm, and returns the system stably to a balanced state. This "systems theory" approach leads to an intepretion of the dynamics of process migration. The system as a whole (user program plus migration algorithm) can be thought of as the convolution of the migration system with the instantaneous impulses produced by changes in the user program which result in changes in the average load imbalance. This leads to a possible way of predicting the performance of *specific* programs. In principle, one could obtain an empirical time series model for a given program and predict its output with a previously characterized, underlying operating system.

One of the interesting things that came out of this work was the usefulness of working in terms of time and program graph averages. We had been doubtful about whether we would get meaningful results by looking at programs at such a coarse level of detail, but it turned out that useful predictions could be made. Most of our results were obtained with random programs, which allowed us to make inferences about the whole population of programs in the class that we studied. Fortunately, the particular examples of real programs for which we obtained measurements also fell within the same range of behaviour!

Chapter 5

Evaluation

5.1 Achievements

Since POSIE was a unifying theme, rather than a project with definite goals, it is not really possible to evaluate it against a set of predefined criteria. However, there were a number of concrete outcomes.

- **A monitoring environment**

  The construction of the Testbed permitted performance measurements to be made to a level of accuracy that would have been impossible on existing commercial machines. In particular, accurate estimates of operating system overheads could be made.

- **Formal methods in operating systems design**

  The operating system for the Testbed was formally specified, and had certain guaranteed properties. The formal specification provided a means of guiding performance experimentation, and of interpreting results. This was a new and original development in the software engineering of distributed systems.

  In addition, the formal specification defined a parameterization of operating systems functions. Higher level functions like migration could be interpreted in terms of lower level functions like message send, which were themselves closely related to the fundamental hardware properties of the machine. This would permit prediction of costs on a machine with different hardware characteristics.

- **A framework for simulation experiments**

  A methodology based on parameterized program models and factorial experiments was established and a tool implemented to support systematic simulation experiments. Some of the purposes for which it was used were: to find which program characteristics most influenced performance; to compare heuristics for static process allocation; to find the best values for various "tuning" parameters in a process migration heuristic; to find the characteristics of programs whose performance could most be improved by process migration.

- **Performance models of parallel programs**

  Based on the experiments mentioned above, it was shown that it was possible to find empirical performance models with good predictive power. The particular numerical values were specific to our given environment, but the method used to obtain the results was quite general, and could be applied to other machine environments.

  Experimental results agreed with those from a simple analytical model, which showed that parameters which were predominantly important experimentally were also those that appeared in the analytical model.

The result of all this work was that we developed a very much better understanding of the execution behaviour of parallel programs, and of the relation between that behaviour and observable macroscopic program properties. We could see how this information could be usefully deployed in a load-balancing operating system. In addition, we knew that it was possible to construct a system that supported process migration correctly. It was rather a disappointment that we did not have an opportunity to put these ideas into practice by implementing an operating system for a commercial parallel machine. This was because of the difficulty in getting access for systems development and because it would not really have been a suitable activity for graduate students. All the same, we were convinced that we could have produced such an operating system, based on sound principles, rather than on *ad hoc* testing and tuning.

5.2 A Bit of Sociology

One of our original intentions was that POSIE should get staff tallking to each other. It was very successful in this respect, as can be seen from the large number of people whose names are mentioned in the introduction. It also provided a group for new PhD students to join, and permitted MSc and even CS4 students to feel that they were integrated into a departmental project. This was enormously beneficial during the first few years of the project. However, it was noticeable that the group structure tended to dissolve as time went on. There were a number of reasons for this. The healthiest, and perhaps the most natural, was that individuals developed rather specialized research directions, and that they became more interested in these than in POSIE itself. Since we had hoped that interesting research would arise from the project once we got started, we were pleased with this development. However, the other reasons were more worrying. The Testbed hardware took a very long time to implement satisfactorily, and it soon became evident that it was not safe to ask MSc or CS4 students to undertake projects that depended on the reliability of the Testbed. Since we had hoped to provide a departmental facility for short projects, this was one respect in which we failed to achieve our aims. In addition, the Testbed itself began to seem out of date, and not suitable as a platform for serious new research work.

The trends outlined above meant that POSIE did not continuously regenerate itself, and that it came to an end when the original participants had completed their work. However, it performed a very useful role in encouraging people to talk to each other and to exchange ideas, at a time when systems research in the department was fragmented. It also generated a useful number of publications, as the bibliography shows.

It might be asked whether hardware design and construction were necessary to this project. Could perhaps as much have been achieved, with much less anxiety, and more quickly, if we had stuck to simulation? Certainly, a lot of useful results were obtained by simulation, but, all the same, experiments on real hardware have an authority that unsupported simulation lacks. There would have been some things that we could not have done—the detailed performance measurements of operating system overheads, for example—and that would have left a gap. In addition, the construction of the Testbed allowed people with hardware skills to participate in the project, and drew in members of the department whose interests were in this aspect of systems development.

There is also the question of whether the project should have had stronger leadership. This was actually more or less impossible in an academic department, where people have to be persuaded to do things, rather than being told to do them! In these circumstances, perhaps things turned out quite well. The various research topics were mapped out by agreement at an early stage, and provided a useful framework for subsequent work. Where it *would* have been useful to have had a project leader was in the Testbed design and implementation work. With hindsight, it was probably a mistake to base the machine on a number of untested, inadequately documented, existing components. It would have been more satisfactory, and probably quicker, to have gone for a purpose-built design and a more robust construction method. Our experiences here suggest, that if we undertake a project like this again, we should not try to cut corners, and that we should be prepared to allocate a professional to oversee the project.

5.3 Future Work

As was mentioned above, it would be valuable to implement an operating system for one or more modern, commercial distributed memory machines. Before doing this we need more comprehensive performance models, which represent not only program characteristics, but also the effects of machine architectures and run time software. There seems no reason why this should not be done in the same way as the experiments described in this report. However, it would be a substantial undertaking and would require good simulation and analysis tools. We also need a lot more measurements on the performance of user programs. We were not able to do for POSIE because of lack of resources, but for an operating system for serious use, we would need appropriate data for tuning mapping and load balancing algorithms to suit the bulk of the applications programs.

As far as more speculative research is concerned, it would be interesting to see if the results reported here would also apply to machines with a much larger number of processors. It would be expected that communications properties would become more important as domains got bigger, and there might be non-linear effects with local hot-spots developing. There is scope for some interesting projects concerned with the measurement and modelling of really large systems.

Bibliography

[1] R. Candlin, "Black Box Models of Parallel Programs", in *Proceedings of Transputers '94*, Arc et Senans, IOS Press, 1994.

[2] R. Candlin and Q. Luo, "Communications Patterns in Occam Programs", in *Parallel Computing 89*, Amsterdam, North Holland, 1989.

[3] R. Candlin and J.G. Phillips, " The Dynamic Behaviour of Parallel Programs under Process Migration", to appear in *Concurrency: Practice and Experience, Special Issue on Dynamic Scheduling*, 1995.

[4] R. Candlin and J.G. Phillips, "Statistical Modelling as a Tool for Studying the Performance of Parallel Systems", Leeds Workshop on Abstract Machine Models, 1993.

[5] R. Candlin and J.G. Phillips, "A Statistical Study of the Factors that Affect the Performance of a Class of Parallel Programs on an MIMD Computer", *International Conference on Decentralized and Distributed Systems ICDDS '93*, Palma de Mallorca, 1993.

[6] R. Candlin and N. Skilling, " A Modelling System for the Investigation of Parallel Program Performance" in *Computer Performance Evaluation: Proceedings of Tools '91*, Turin, Elsevier, 1991.

[7] R. Candlin, T. Guilfoy and N. Skilling, "A Modelling System for Process-based Programs" in *Proceedings of the European Simulation Congress*, Edinburgh 1989.

[8] R. Candlin, Q. Luo and N. Skilling, "The Investigation of Communications Patterns in Occam Programs", *Developing Transputer Applications*, Proceedings of the 11th Occam Users' Group, IOS Press, 1989.

[9] R.Candlin, P.R. Fisk and N. Skilling, "A Statistical Approach to finding Performance Models of Parallel Programs" in *Proceedings 7th UK Computer and Telecommunications Performance Engineering Workshop*, Springer-Verlag Workshop Series 1991.

[10] R. Candlin, P.R. Fisk, J.G. Phillips and N. Skilling, "A Statistical Approach to Predicting the Performance of Concurrent Programs", in *Parallel Computing: from Theory to Sound Practice, Proceedings of the European Workshop on Parallel Computing*, Barcelona, IOS Press, 1992.

[11] R. Candlin, P. Fisk, J. Phillips and N. Skilling, "Studying the Performance Properties of Concurrent Programs by Simulation Experiments on Synthetic Programs", *Proceedings of ACM SIGMETRICS and Performance*, Newport RI, 1992.

[12] D. Christie, *Virtual Channels on the Meiko Computing Surface*, M.Sc. Dissertation, University of Edinburgh 1988.

[13] R. Green, *The Design of an MC68010 Microprocessor System*, M.Sc. Dissertation, University of Edinburgh, 1986.

[14] T. Guilfoy *A Modelling System for POSIE*, M.Sc. Dissertation, University of Edinburgh 1988

[15] R. N. Ibbett, D. A. Edwards, T. P. Hopkins, C. K. Cadogan and D. A. Train, "Centrenet— A High Performance Local Area Network", *Computer Journal* 28(3) 1985.

[16] K. Imre, *A Performance Monitoring and Analysis Environment for Distributed memory MIMD Programs*, Ph.D. thesis, University of Edinburgh, 1993.

[17] P. Marshall, *A Tool for Parallel Program Design*, M.Sc. dissertation, University of Edinburgh, 1989.

[18] P. Martin, *Adding Safe and Effective Load Balancing to Multicomputers*, Ph.D. thesis, University of Edinburgh, 1994.

[19] P. Martin, "The Formal Specification in Z of Task Migration on the Testbed Multicomputer", Computer Systems Group Report ECS-CSG-2-94, Department of Computer Science, University of Edinburgh, 1994.

[20] P. Martin, "The Performance Profiling of a Load Balancing Multicomputer", Computer Systems Group Report ECS-CSG-3-94, Department of Computer Science, Edinburgh, 1994.

[21] P. Martin and R. Candlin, "The Benefits of Dedicated Monitoring Hardware for Evaluating Load Balancing", *UKPEW '94, Proceedings of the 10th UK Workshop on Computer Performance Evaluation* , Edinburgh, 1994.

[22] P. Martin and S. T. Gilmore, "Pragmatic Experience of the Formal Specification of a Distributed Operating System", *Proceedings of the 5th International Conference on "Putting into Practice Methods and Tools for Information System Design"*, Nantes, 1992.

[23] P. Martin and S. T. Gilmore, "Correctness Issues in the Communication Problems of a Task Migration Algorithm", *UKPEW '94, Proceedings of the Tenth UK Computer and Telecommunications Performance Engineering Workshop*, Edinburgh, 1994.

[24] P. Martin, R. Candlin and S. Gilmore, "Correctness and Performance of a Multicomputer Operating System", *Proceedings of the IEEE International Computer Performance and Dependability Symposium IPDS '95*, Erlangen, 1995.

[25] M. Melachrinidis, *Position Independent Occam Programs*, M.Sc. dissertation, University of Edinburgh, 1989.

[26] J. Phillips and N. Skilling, "A Modelling Environment for Studying the Performance of Parallel Programs", in *Proceedings of the Seventh UK Computer and Telecommunications Workshop*, Edinburgh, 1991.

[27] J. Phillips and R. Candlin, "An Environment for Investigating the Effectiveness of Process Migration Strategies on Transputer-Based Machines, in *Proceedings of the World Occam and Transputer Users' Group Meeting 15-Ongoing Research*, Aberdeen, IOS Press, 1992.

[28] V. Rebello, *A Tracing Tool for POSIE*, Honours project report, Department of Computer Science, Edinburgh, 1989.

[29] P. Roberts, *An Occam 2 Compiler for POSIE*, Honours project report, Department of Computer Science, Edinburgh, 1989.

[30] J.G. Phillips, *A Statistical Investigation of the Factors Influencing the Performance of Parallel Programs, with an Application to the Study of Process Migration Strategies* . Ph.D. thesis, University Edinburgh. 1994.

[31] R.J. Pooley and R.Candlin (eds) *The POSIE Project Annual Report*, Computer Science Department Report CSR-284-88, Edinburgh, 1988.

[32] R.J. Pooley (ed) *The Second POSIE Report (misleadingly called "Introduction to the second Posie report")*, Computer Science Department Report CSR-6-90), Edinburgh, 1990.

[33] N. Skilling, *A Statistical Approach to Performance Evaluation of Parallel Systems with Reference to Chemical Engineering*, submitted for the degree of Ph.D., University of Edinburgh.

[34] N. Skilling, **eg**, University of Edinburgh, Department of Chemical Engineering, Internal Report.

[35] N. Skilling, MIMD, University of Edinburgh, Department of Chemical Engineering, Internal Report.