The Performance Profiling of a Load Balancing Multicomputer

Paul Martin

ECS-CSG-3-94
Department of Computer Science
Edinburgh University
Edinburgh EH9 3JZ, Scotland

June 1, 1994

Abstract

This report introduces a message-passing multicomputer called the 'Testbed' and describes the operating system and hybrid monitoring support for load balancing. A series of experiments are reported in which detailed and accurate performance figures are established for the functions associated with task migration, thus parameterising some key properties of the Testbed. A number of different performance metrics are compared in terms of their costs versus their utility for load balancing. Finally, a sample load balancing strategy is outlined for the Testbed and speedup results obtained for a range of applications.

Contents

1 Introduction

This report introduces an experimental message-passing multicomputer called the 'Testbed' which was developed at Edinburgh University. The Testbed is unusual because it offers dynamic load balancing, i.e. the moment-by-moment automatic and transparent distribution of work amongst processors with the aim of minimising execution time. In order to support load balancing the Testbed has a specially extended operating system and dedicated performance monitoring hardware.

This report presents three sets of experiments which were performed using the monitoring hardware as part of the design, tuning and evaluation of the Testbed's load balancer. In the discussion below, the process of load balancing is considered to have the following three 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.

In the first set of experiments the latencies of various system functions associated with load balancing were measured in order to assess the intrinsic performance properties of the operating system and communication hardware. Knowledge of these properties is vital for the design of the decision-making phase where, for example, the benefits of executing two tasks on different processors must be balanced against the higher costs of inter-processor communication over local communication. Knowledge of these properties is also useful for the design of the execution phase, for instance when calculating the time that must be allowed for a task to complete migration.

The second set of experiments relate to the reconnaissance phase and are used to compare different methods for assessing the load. Experience shows that it is not trivial to find an aspect of the load which can be measured with minimal overhead by each processor and also processed with minimal overhead into a form suitable for the decision-maker.

In the final set of experiments the effectiveness of the Testbed's load balancer is evaluated against a range of different types of application. Two aspects are measured—the net change in execution time and the quality of the load balancing decisions made.

The rest of this report is organised as follows. In Section 2 the Testbed is introduced in terms of its conventional features, its special task migration functions and its dedicated monitoring hardware. Section 3 describes the first set of experiments on measuring the intrinsic properties of the Testbed. The first part of Section 4 describes the second set of experiments on finding the best way to measure the load. The evaluation of the load balancer is described in the second part of Section 4. Finally, the conclusions of the report are presented in Section 5.

2 Presenting the Testbed

This section introduces the Testbed—the multicomputer on which the practical experimentation for this report was performed. The section 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.

2.1 Conventional Features

The Testbed (also described in Imre [16]) 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 called Centrenet (a detailed description of which can be found in Ibbett *et al* [14]) 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 serial link to the local area network and hence access to a filestore. Figure 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. Tasks may be invoked on any operating system and migrated between operating systems transparently to the user. Figure 2 gives an overview of the Testbed system software.

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`.

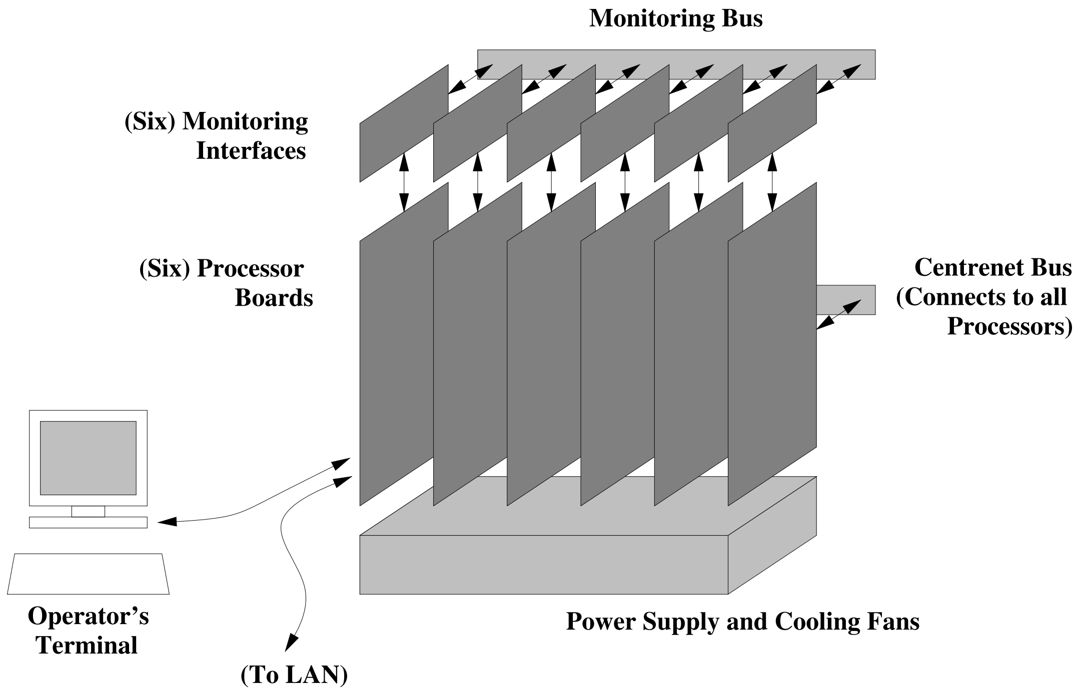

Figure 1: *An overview of the Testbed hardware.*

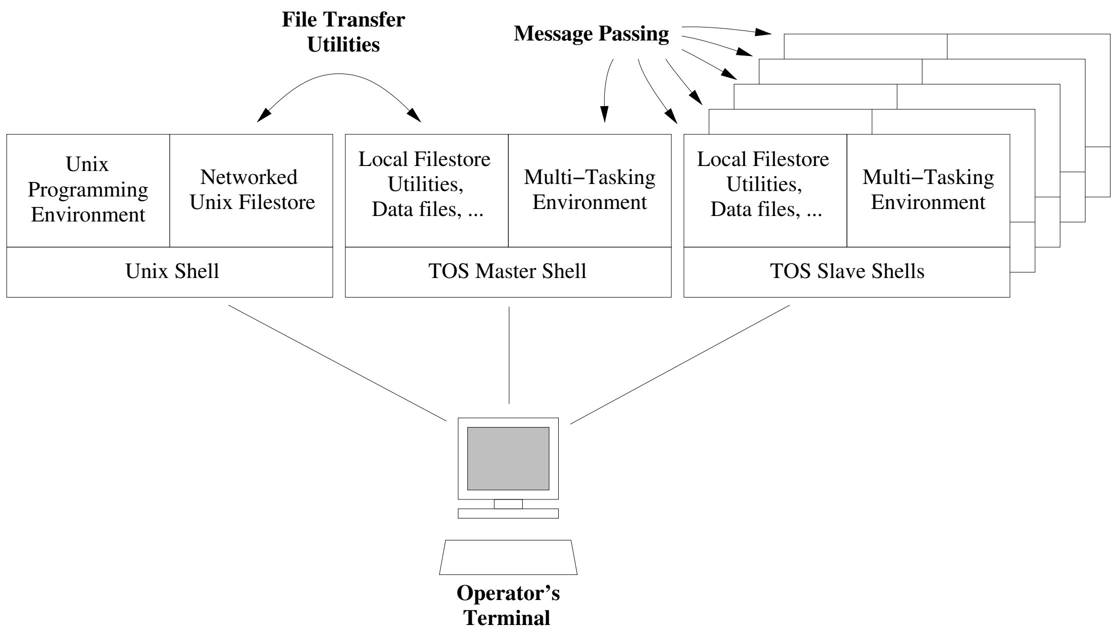

Figure 2: *An overview of the Testbed software.*

2.1.1 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 2.2.4 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 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 2.1.3) 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 report all experiments use the simplifying assumption that Testbed inter-processor communication requires at most one hop.

2.1.2 The Testbed operating system

The description of TOS support for load balancing is deferred until Section 2.2. 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 pre-emptive 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 2.1.3) 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`.

2.1.3 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 [18, 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 [32] and INMOS [22] and a discussion of the unique benefits of `occam` in Welch [36]. Thread control is an operating system function, accessed by means of the following library routines:

`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 error 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 [37].

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:

`send_block(chan, buffer, length)` A thread requests to send a message of `length` bytes beginning at the address given by `buffer` on channel `chan`.

`int receive(chan, buffer, length)` A thread requests to receive at most `length` bytes of data beginning at the address given by `buffer` on channel `chan`. The return value specifies the actual number of bytes received.

2.2 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.

2.2.1 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'. Data structures such as doubly-linked lists are necessary.

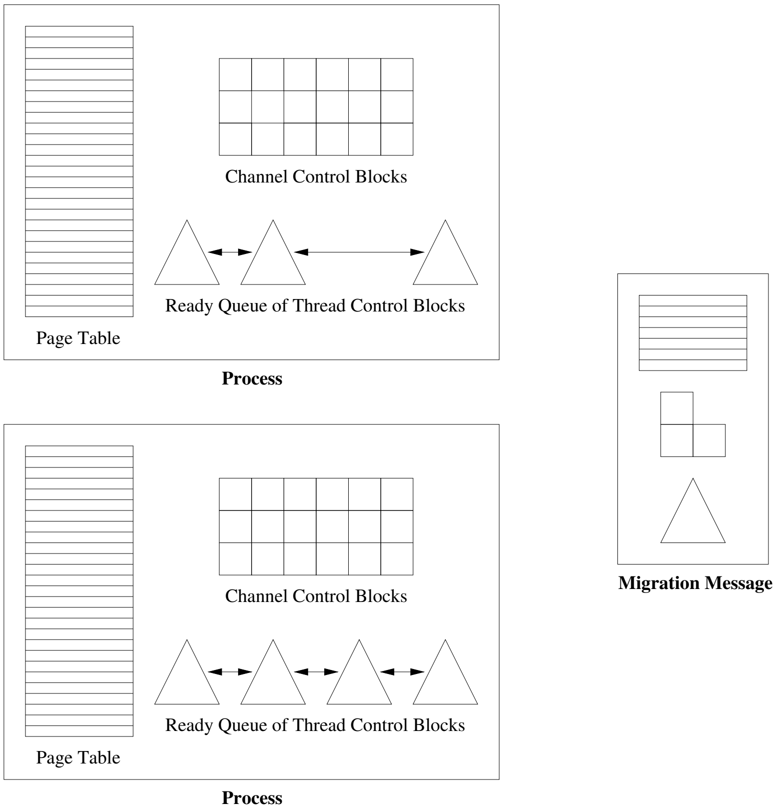

Figure 3: *A processor with two executing processes and a migrating thread waiting for transmission.*

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. 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.

2.2.2 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 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 companion report Martin [26]. 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' 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.

2.2.3 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 Section 3 and for the load balancing in Section 4 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% slowdown in the TOPSYS project (described by Bemmerl *et al* [4] and Bemmerl and Bode [2]) although it is rather more that the 0.1% claimed for the TMP monitor (Haban and Wybranietz [13] and Haban and Shin [12]). Detailed results about the exact number of events produced can be found in Section 3.3.7.

The extra C instructions are of two forms:

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

and

```
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

Event Sequence	*Description*
`[0xaann]`	Global clock overflow
`[TID+0x4000]`	Thread (with ID `TID`) is being scheduled
`[CREATE+0x8000] [CTID]`	A child thread (`CTID`) is created
`[TRAP70]`	A thread is being blocked on communication

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

Event Sequence	*Description*
`[TID+0x4000]`	Thread joins ready queue
`[TID]`	Thread is removed from ready queue

Table 2: *An example showing some fixed length events used when measuring the size of the ready queue.*

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 1 and 2.

2.2.4 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 4 each of

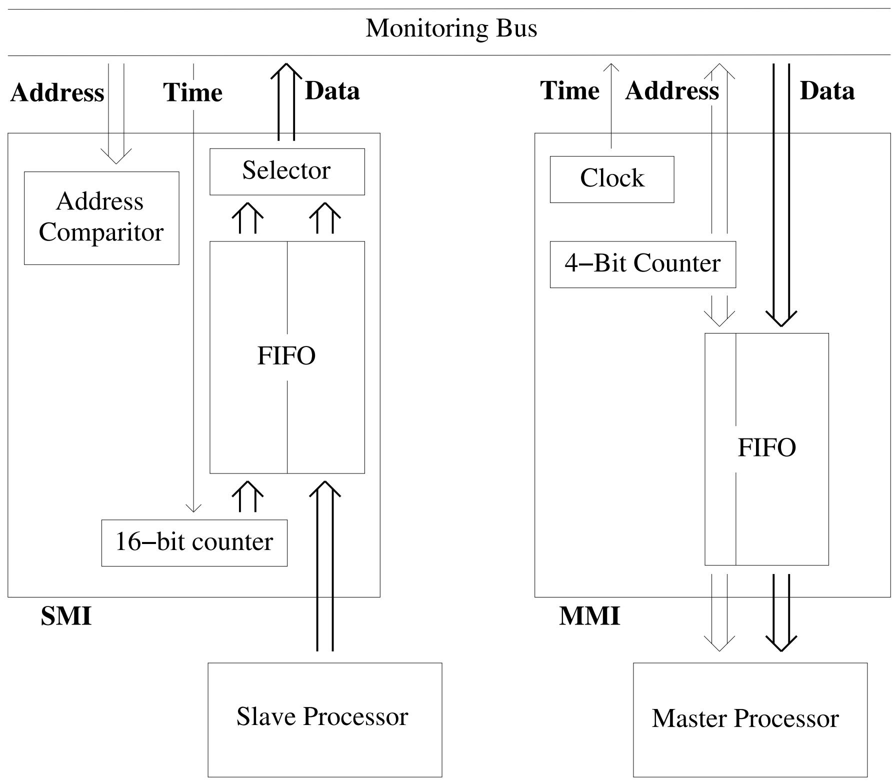

Figure 4: *An overview of the Testbed monitoring hardware showing the master monitoring interface and one of the slave monitoring interfaces.*

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.

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 [13] and Haban and Shin [12]. Different kinds of filtering are suggested in Mansouri-Samani and Sloman [25], 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 Section 4. 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 report 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.

3 Performance Profiling

This section 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 graph form. A series of finite state machines are also presented so that the results can be interpreted in terms of the formal specification presented in Martin [26].

The rest of this section 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 section concerns the experimental results and their interpretation. Finally, some conclusions are presented.

Terminology. In the following sections it is necessary to distinguish occam Channel Messages (**OCM**s) from Message Protocol Components (**MPC**s). 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.[1]

3.1 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 3.3.

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.

[1] Most remote channel communication, i.e. communications between threads on different processors, start with the sender's processor transmitting a *focus* token to the receiver's processor to notify the receiving thread that the sender is waiting. When the receiving thread is ready to communicate, an *rtr* is returned to the sender's processor causing the contents of the channel message to be transmitted (in a message of type *msg*) and the sender thread restarted. When the *msg* arrives, the data is copied into the receiver thread's memory area and the receiver is restarted. The *advert* token is used only during the first communication over a channel when the sender thread may have to advertise its presence to all processors in the system. Receiver threads are more passive than sender threads—it is generally the responsibility of the sender to locate the receiver when it wants to communicate.

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.

3.1.1 The test programs

My experiments were performed using five small C programs: `local`, `remote`, `migrate`, `multi-phase`, and `overflow`. Program listings in pseudocode are given in Appendix A.

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.

3.2 Execution Models

3.2.1 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.

3.2.2 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 states 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

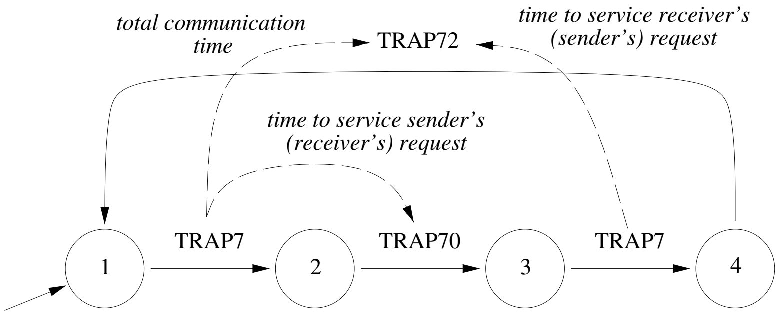

Figure 5: *The FSM for local communication.*

made explicit. The FSMs also have a direct interpretation in terms of the formal specification of the operating system and this is explained for each machine.

Local communication. Figure 5 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.

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 specification schemas. Assuming that the sender thread communicates

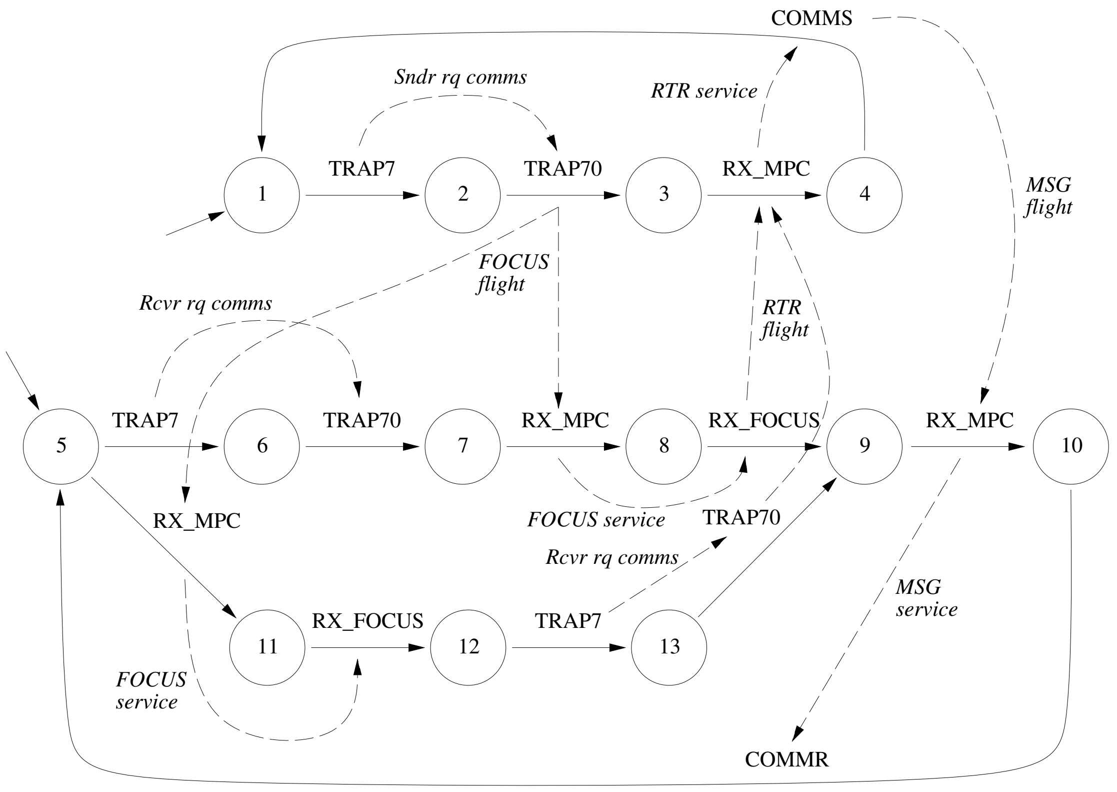

Figure 6: *The FSM for remote communication.*

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 6 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) show 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 3 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.

Event	*Interpretation*
TRAP7	Thread requests to communicate
TRAP70	Thread is blocked after request has been serviced
RX_MPC	Begin servicing an MPC (Message Protocol Component)
RX_FOCUS	MPC received was a $focus$ token
COMMS	Sender is returned to ready queue
COMMR	Receiver is returned to ready queue

Table 3: *Events emitted during the remote communication protocol.*

Transition	*Schema*	*Transition*	*Schema*
1 via 2 to 3	$FSend$	7 via 8 to 9	$FFocus$
5 via 6 to 7	$FReceive$	3 via 4 to 1	$FRtr$
12 via 13 to 9	$FReceive$	9 via 10 to 5	$FMsg$
5 via 11 to 12	$FFocus$		

Table 4: *Correspondence between state transitions in the FSM for remote communication and the specification schemas.*

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 specification schema, and since the FSM for remote communication has one transition per event, it follows that each specification schema corresponds to a *pair* of FSM transitions. These correspondences are shown in Table 4.

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

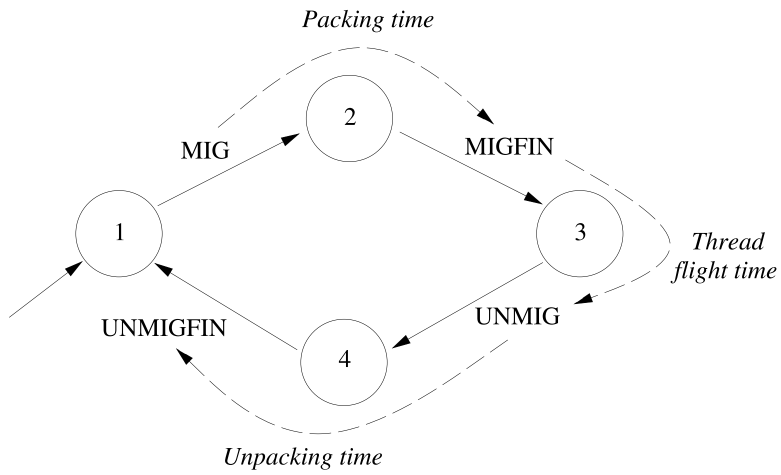

Figure 7: *The FSM for thread migration.*

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.

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 specification schemas are as follows: the FSM transition from 1 via 2 to 3 is modelled by the $MDisconnect$ schema; the transition from 3 via 4 to 1 is modelled by $MConnect$.

Copying remote pages. Figure 8 shows the protocol for copying remote pages. The times of interest are:

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.

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.

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 specification does not model the copying of pages between processors so there are no schemas relating to this FSM.

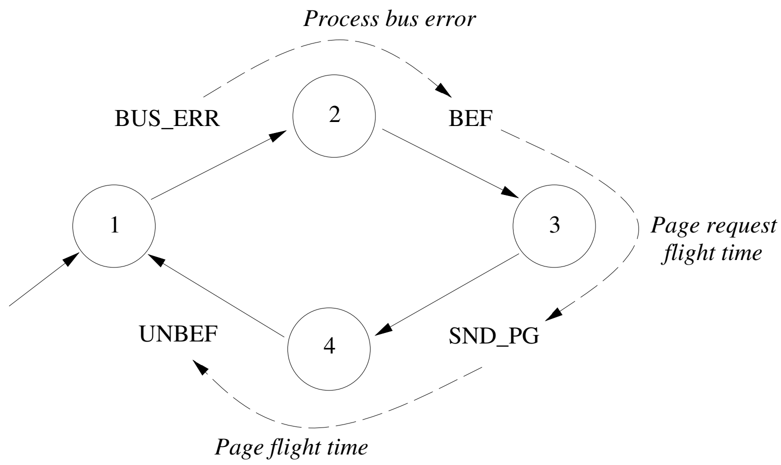

Figure 8: *The FSM for remote page copying.*

3.3 Results of the Experiments

3.3.1 Latency of local communication

Figure 9 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.

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 10 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 10 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 messages size and therefore at most a constant factor of improvement could be made to the operating system efficiency.

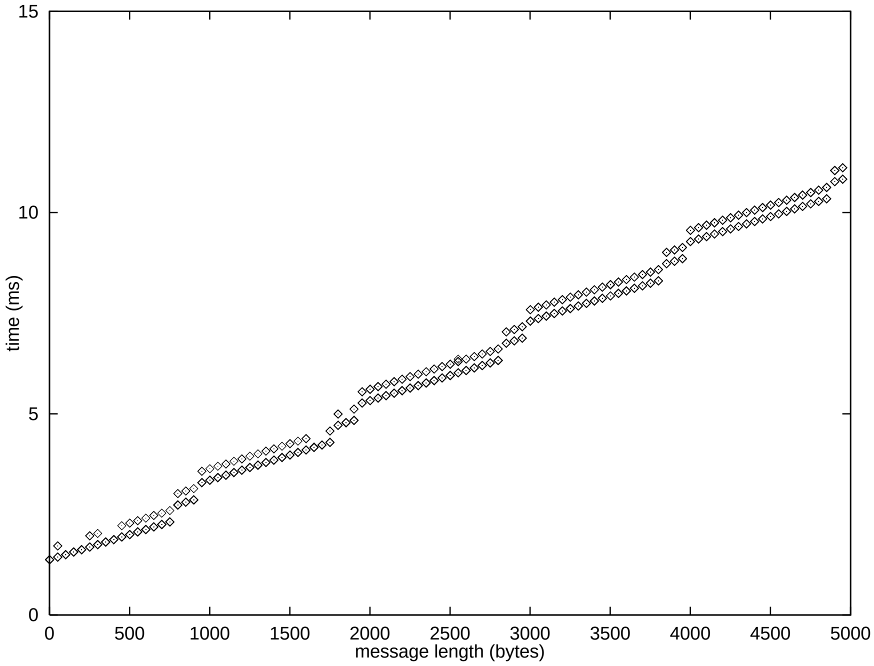

Figure 9: *Time taken to service a local send request, copy the message to the waiting receiver's memory area and restart both threads.*

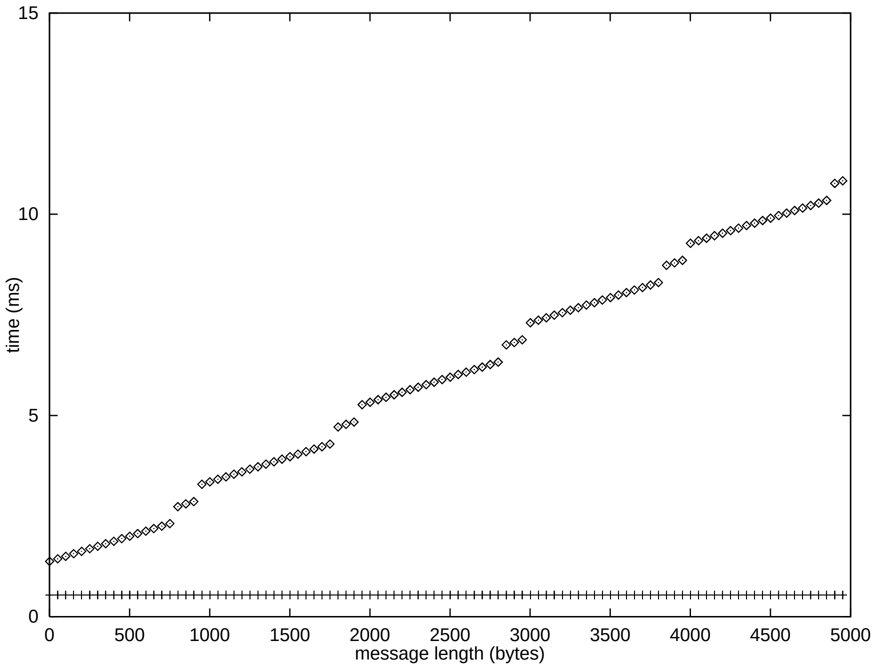

Figure 10: *Upper graph: time taken to service a local send request* ***corrected*** *by subtracting time spent servicing timer interrupts. Lower graph: time taken to service a receive request and block thread.*

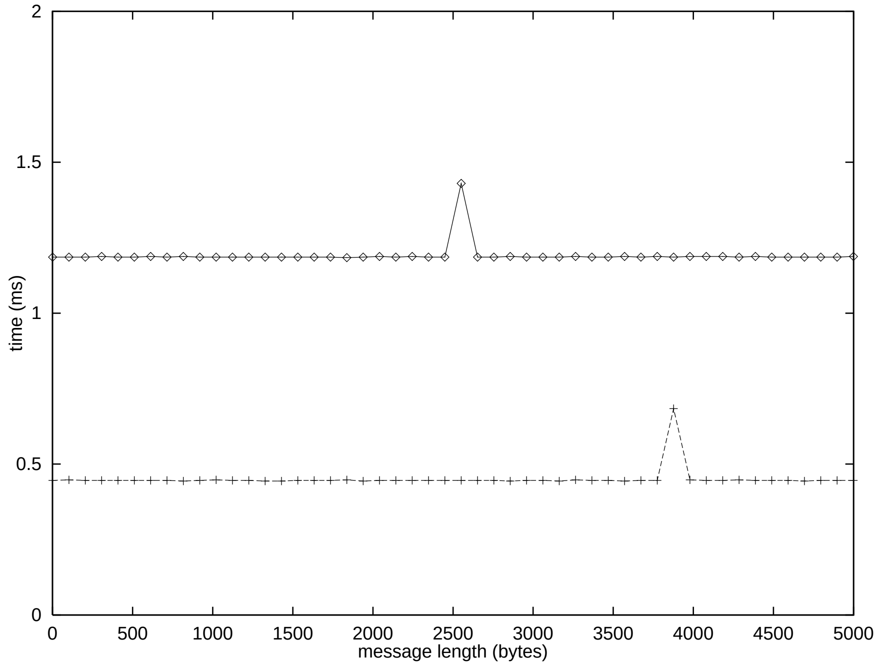

Figure 11: *Upper graph: time taken to service a send request over a remote channel. Lower graph: time taken to service a receive request over a remote channel.*

3.3.2 Latency of remote communication

Figure 11 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.

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 12 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

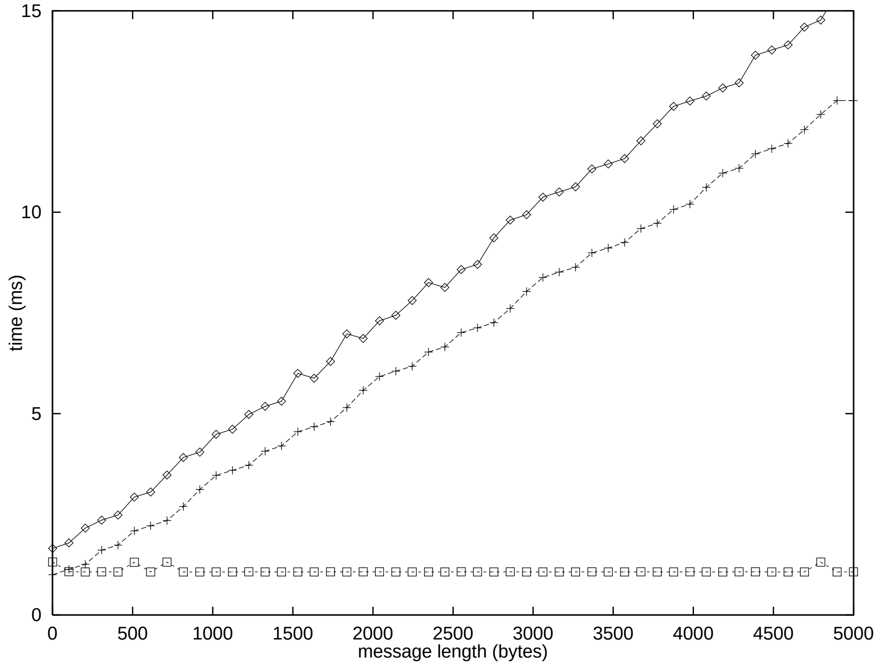

Figure 12: *Upper graph: time for the operating system to service an incoming rtr message. Middle graph: time to service a msg message. Lower graph: time to service a focus message.*

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 13 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.

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.

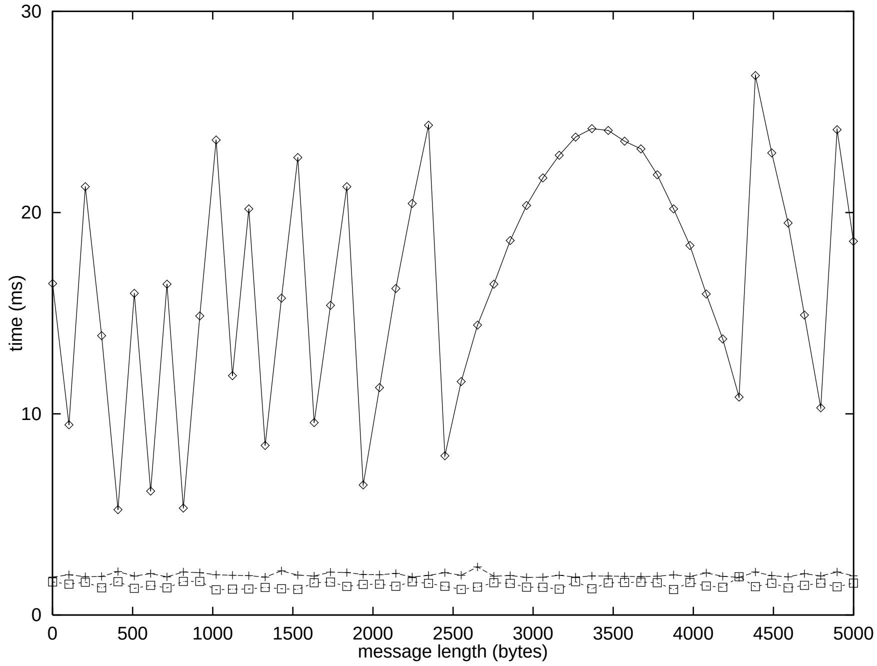

Figure 13: *Upper graph: the flight time for a msg being sent over Centrenet. Centre graph: rtr flight time. Lower graph: $focus$ flight time.*

2. With reference to Section 3.2.1, 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 14 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 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 15. 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 msg flight time, explained above.

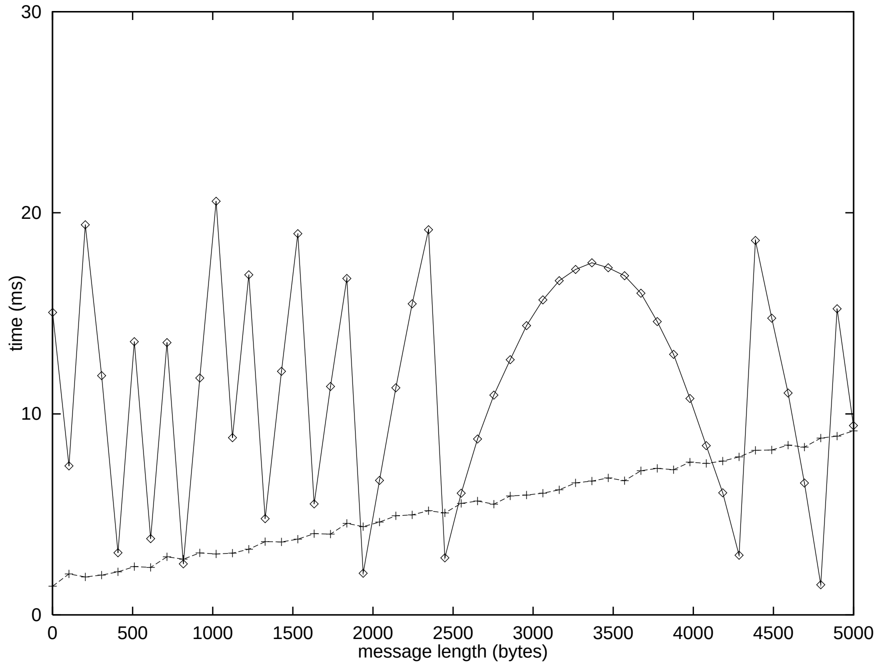

Figure 14: *Upper graph: the time between the operating system queuing the msg for transmission and the transmit queue being submitted to Centrenet. Lower graph: pure flight time, i.e. the difference between the msg flight time as shown in previous figure and the upper graph.*

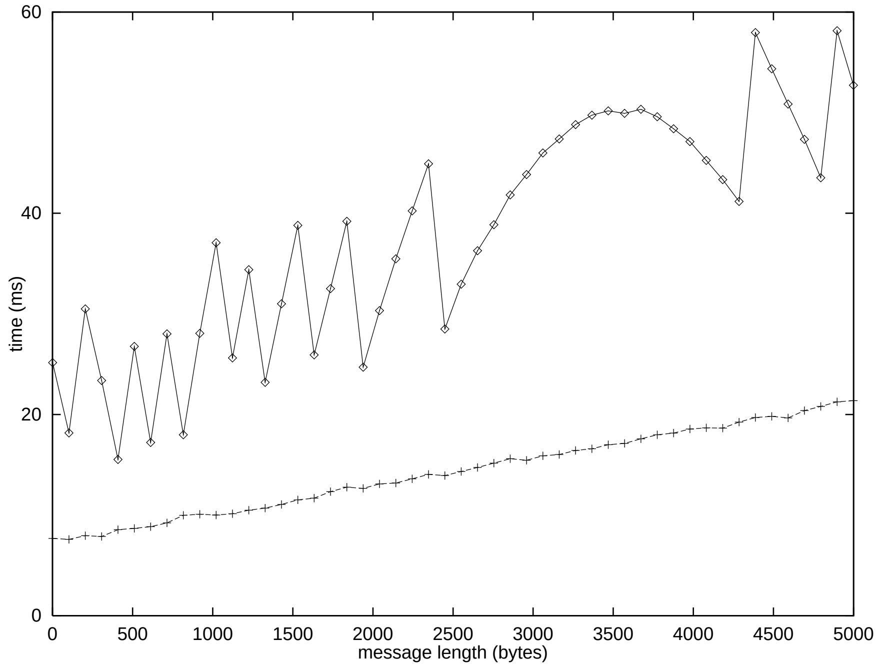

Figure 15: *Upper graph: the total time by which the receiver is delayed during communication. Lower graph: the total time by which the sender is delayed.*

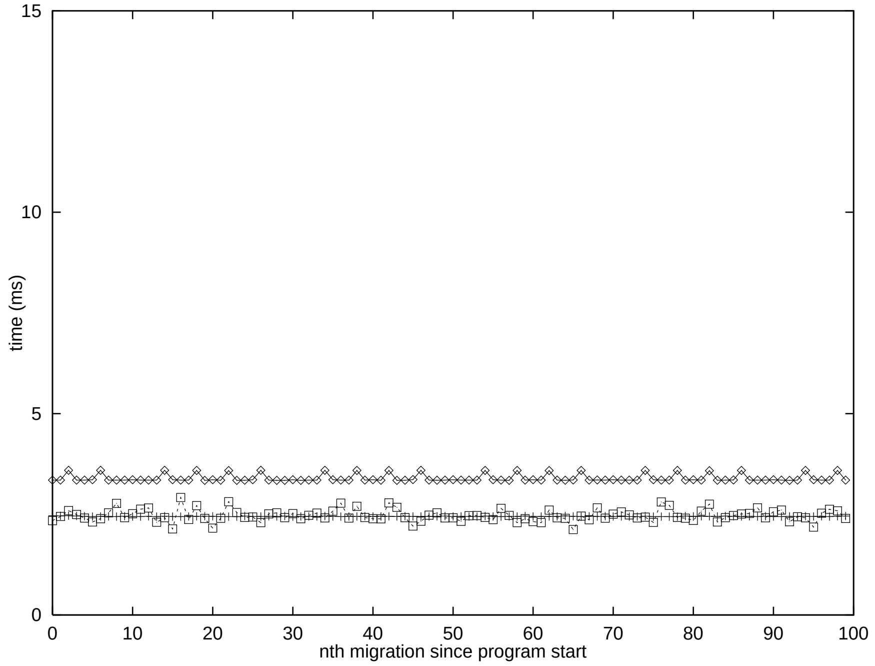

Figure 16: *Upper graph: time spent by operating system packing thread context into a Centrenet message buffer. Lower two graphs: flight time for thread message superimposed on the time spent by receiving operating system unpacking thread.*

3.3.3 Thread migration latency

In this section the latencies associated with thread migration are measured. Figure 16 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.

3.3.4 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 17.

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 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

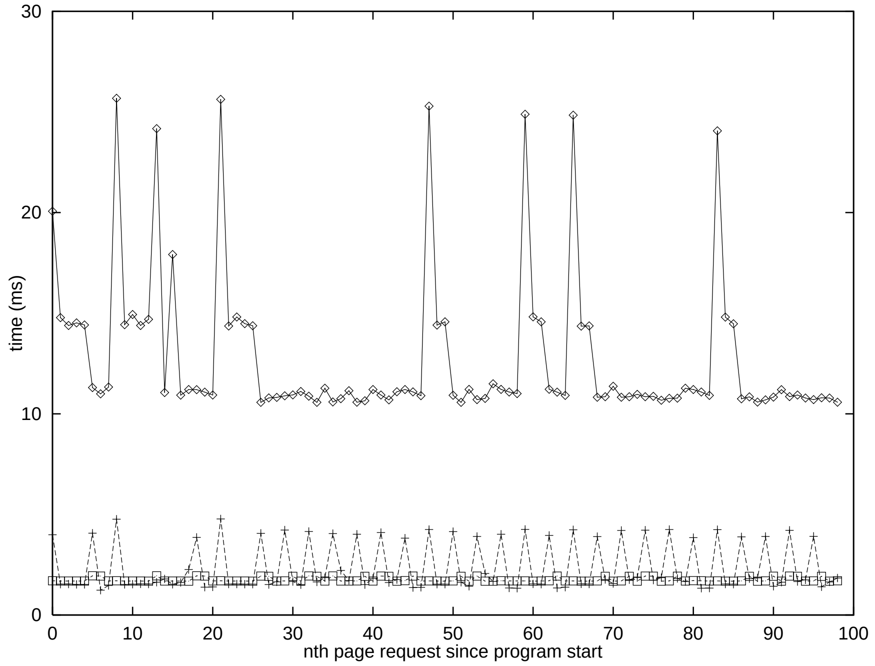

Figure 17: *Upper graph: flight time for a page being copied between processors. Middle graph: flight time for a page request. Lower graph: time between a user thread faulting on a page and the transmission of the page request over Centrenet.*

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 12. The 'pure' flight time for a remote page corresponds to the pure flight time for a message and is approximately 3ms (Figure 14). The reception and unpacking of a page corresponds to the processing of a *msg*, which takes about 3.5ms according to Figure 12. 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.

3.3.5 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 9 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. While 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.

3.3.6 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 5) to simulate compute and communication bound

1:	compute-bound	local comms	short messages
2:	”	”	long messages
3:	”	remote comms	short messages
4:	”	”	long messages
5:	communication-bound	local comms	short messages
6:	”	”	long messages
7:	”	remote comms	short messages
8:	”	”	long messages

Table 5: *Phases of the* `multi-phase` *test program.*

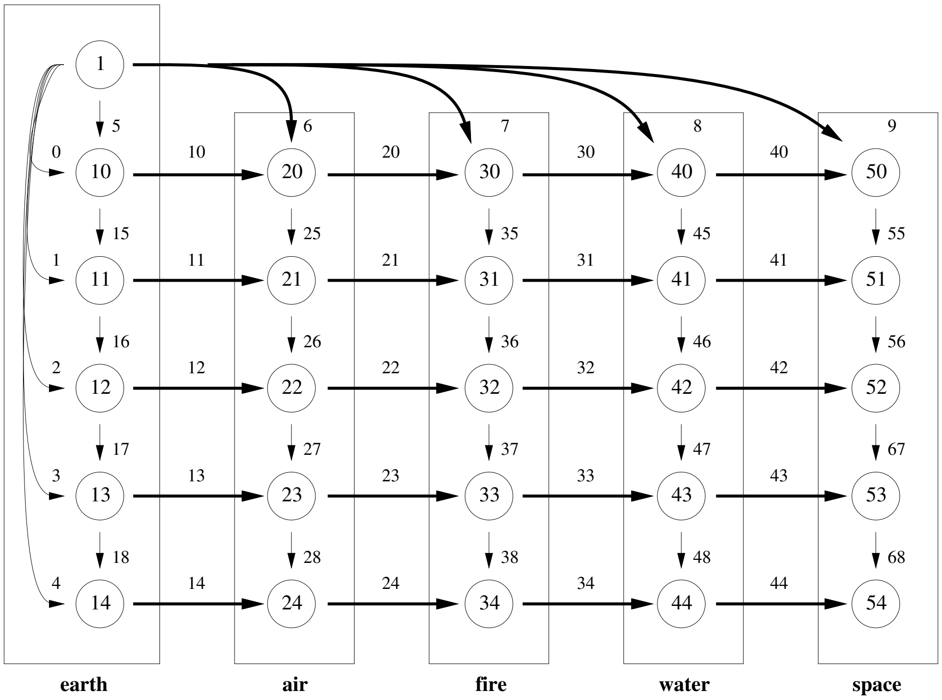

Figure 18: *Mapping of threads to processors and channels to pairs of threads in the* `multi-phase` *test program. The bold arrows represent channels for remote communication, the normal arrows represent channels for local communication.*

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.

The `multi-phase` program is listed in Appendix A and Figure 18 shows how threads are allocated to processors and the pattern of channels between threads. Note that the five slave processors are also know by the names `earth`, `air`, `fire`, `water` and `space`. Thread 1 controls the transition between phases by using barrier synchronisation as follows.

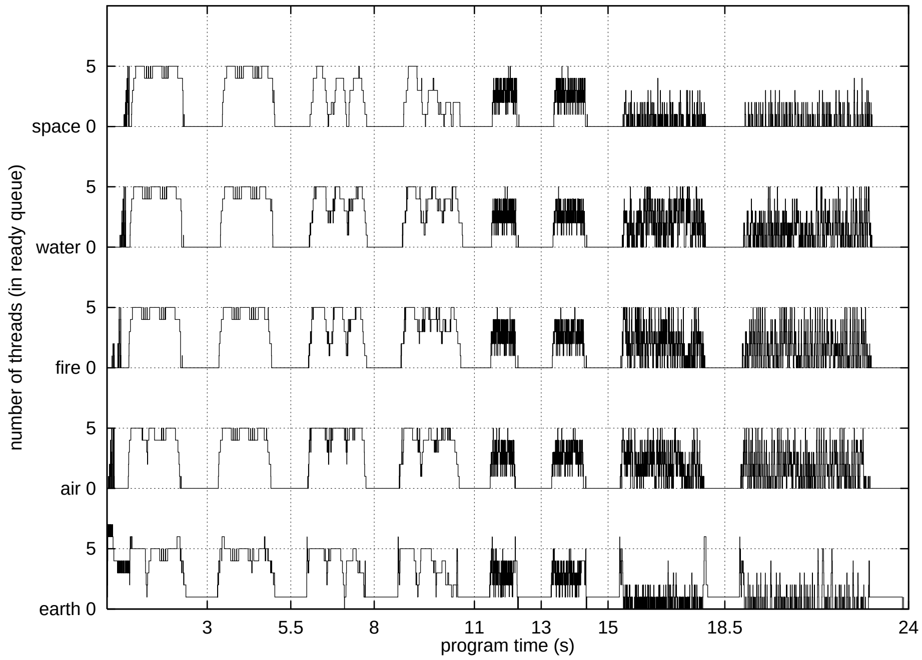

Figure 19: *Length of the ready queue on each of the five slave processors during an execution of the* `multi-phase` *program.*

- 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 19. 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 20). 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 21 (it is assumed that the receiver latencies are similar). The

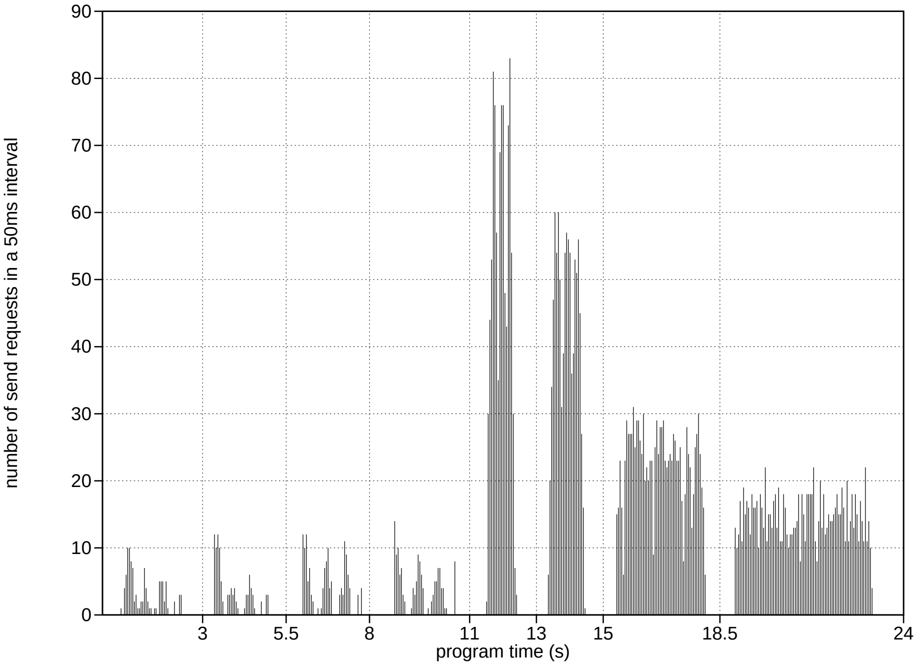

Figure 20: *Number of send requests per 50ms during the program execution.*

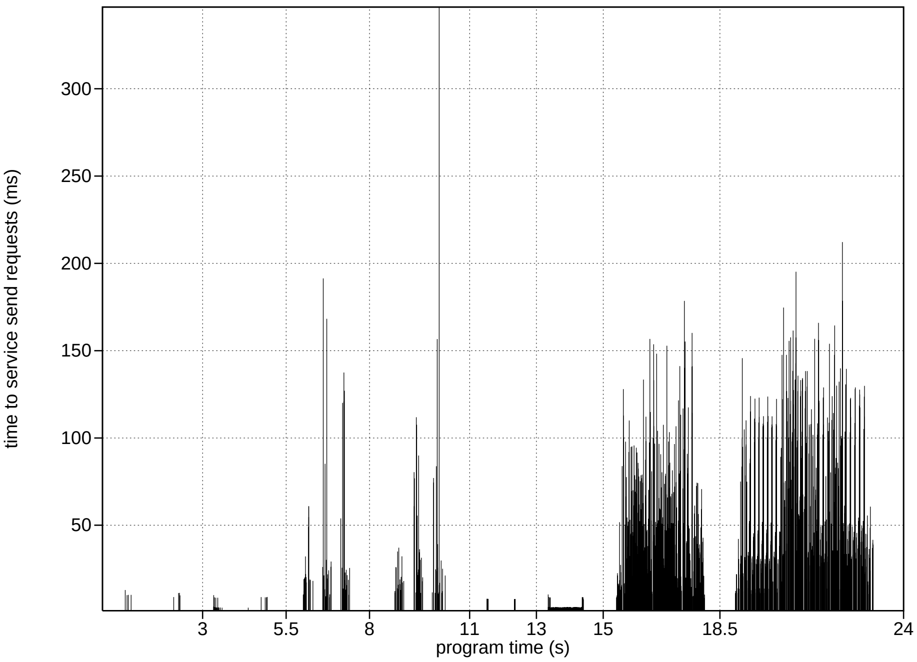

Figure 21: *Time to service a send request during the program execution.*

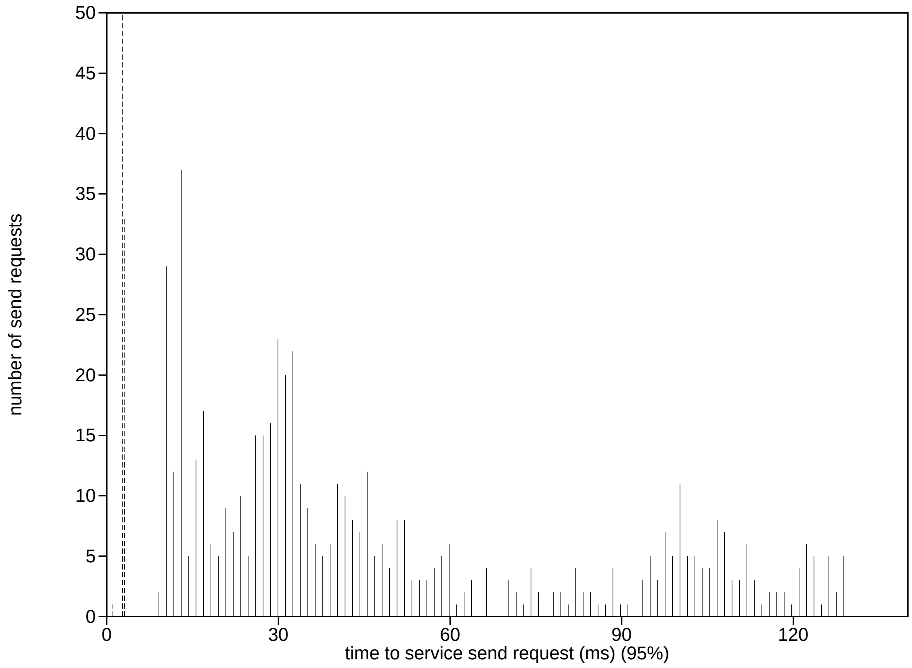

Figure 22: *The dashed lines (there are only three, between 0ms and 3ms) show the distribution of (local) communication latencies during phase 6 of the test program* `multi-phase`*. The un-dashed lines show 95% of the distribution of (remote) communication latencies during phase 8 of the test program.*

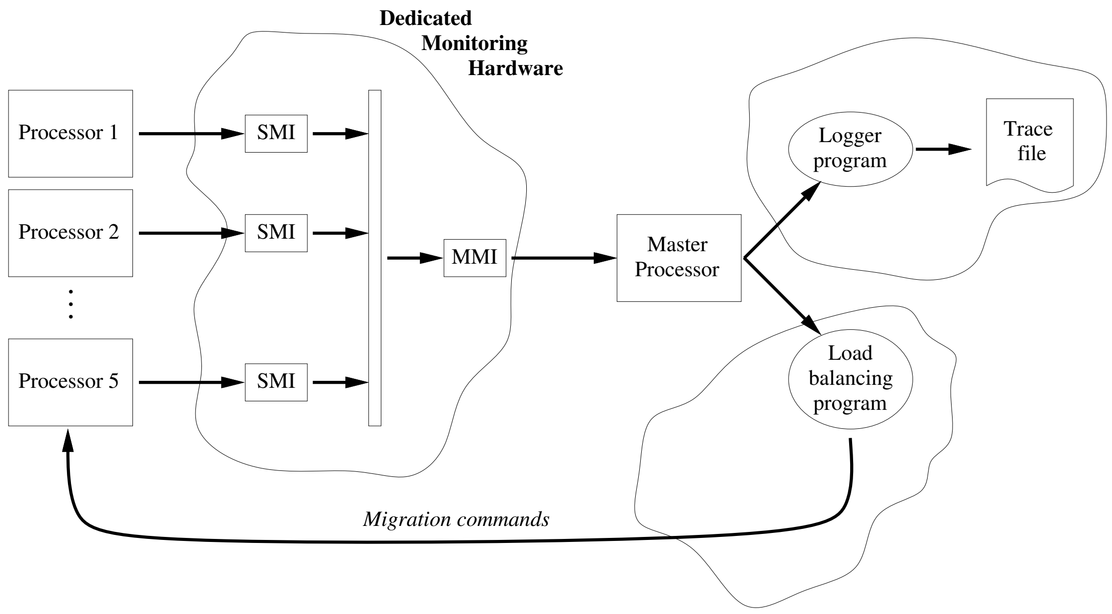

Figure 23: *The event collection and processing pathways.*

latencies are obviously longer in the remote communication phases as compared to the local communication phases.

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 22. 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).

3.3.7 Event rates

This section seeks to establish the maximum data rates that can be supported by the monitoring system. While Imre [16] presents theoretical results, the experiments in this section demonstrate that the monitoring system suffers a bottleneck which reduces the maximum data rates dramatically. Figure 23 shows the pathways along which monitoring events flow and the components that process events.

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 Section 2.2.3, 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 1146 pairs per second.

The conclusion is that the logger program has a sustainable event rate of just over 1100 time-stamp and event data pairs per second, or 220 pairs per slave per second.

The Testbed's sustainable event rate is about one fifth of NETMON-II's (described by Zitterbart [38]) and about one tenth of the TMP's (Haban and Wybranietz [13] and Haban and Shin [12]). 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.

3.4 Conclusions

The latencies measured for the main operating system services are summarised in Table 6. 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 shows how the Testbed's performance compares with some other concurrent computer systems'. (Charlotte is described by Artsy and Finkel [1] and Finkel *et al* [10]. Emerald is described by Jul *et al* [17]. MMK is described by Bemmerl *et al* [4] and Bemmerl and Bode [2]. Sprite is described by Douglis and Ousterhout [8].)

The presentation of the Testbed's operating system latencies satisfies two of the three aims of this section—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 section 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. Thus, the only 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 formal specification of the Testbed strongly suggests that the communication protocols cannot be simplified without compromising the transparency of process migration.

Service	*Average time (ms)*	*Bounds (ms)*	*Figure*
Local communication	$1.95 * n + 1.025$	± 0.275	24
Remote communication	$2.77 * n + 7.35$	± 0.35	25
Thread migration	8.4	± 0.5	26
Remote page copy	16.9	$+15.3\ -3$	27

Table 6: *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).*

Project	*Service*	*Average time (ms)*
Charlotte	Local communication (1 byte)	10
"	Remote communication (1 byte)	23
"	Migrate process (32K)	240
Emerald	Migrate process (600 bytes)	40
MMK	Local communication (5K)	13
"	Remote communication (5K)	56
Sprite	Remote communication (5K)	10
"	Migrate Unix process	300

Table 7: *A selection of operating system latencies from other concurrent computer projects.*

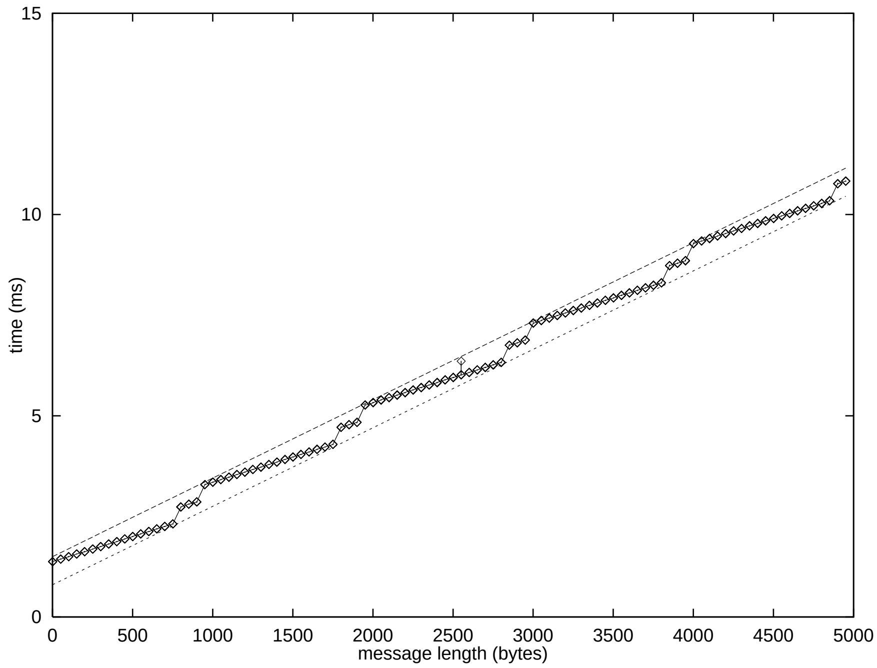

Figure 24: *The graph showing the latency for a single local send is trapped between lines $y = 1.95 * 10^{-3} * x + 1.5$ and $y = 1.95 * 10^{-3} * x + 0.8$.*

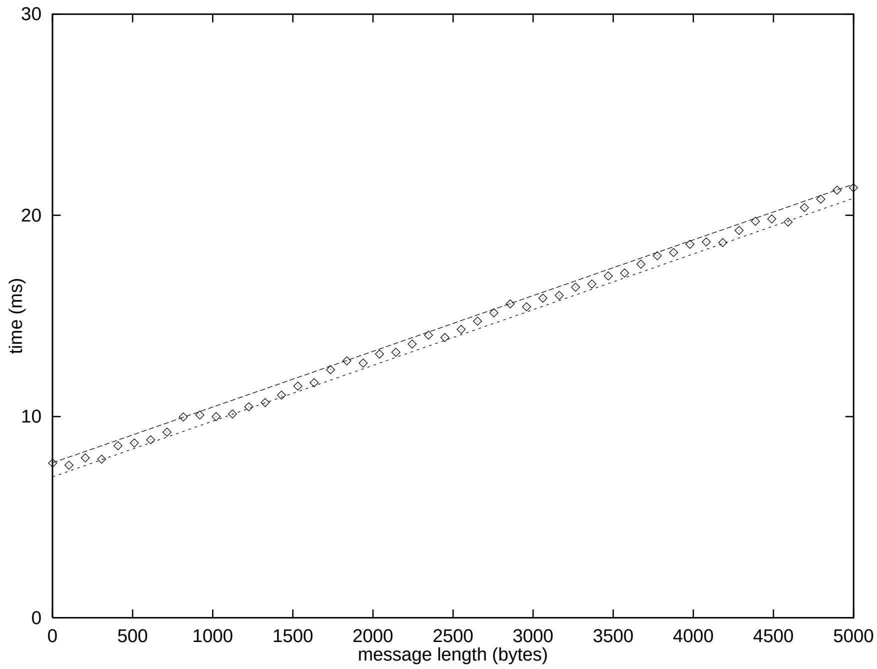

Figure 25: *The graph showing the latency for a single remote send is trapped between lines* $y = 2.77 * 10^{-3} * x + 7.7$ *and* $y = 2.77 * 10^{-3} * x + 7$.

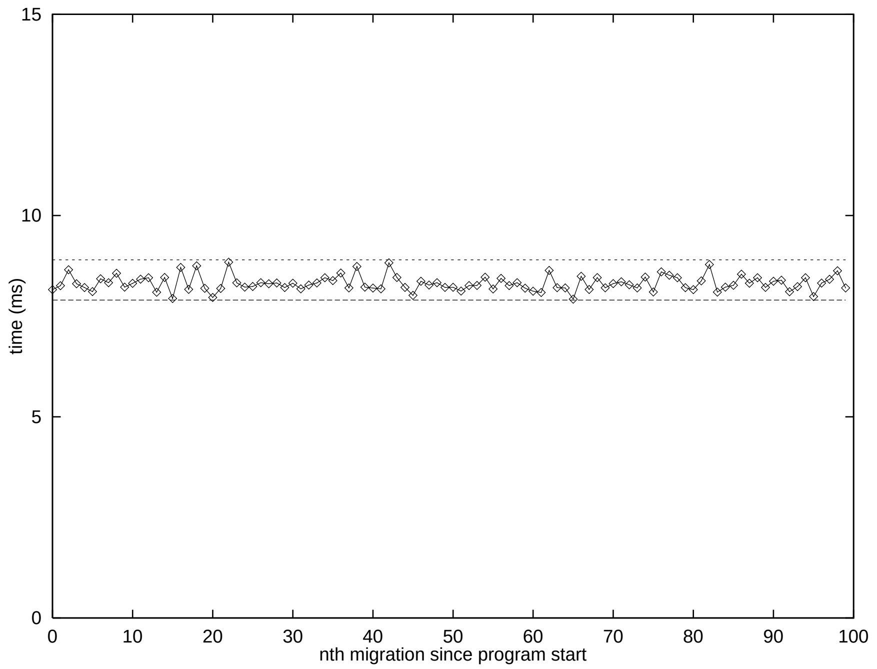

Figure 26: *The graph showing the latency for thread migration is trapped between lines* $y = 7.9$ *and* $y = 8.9$.

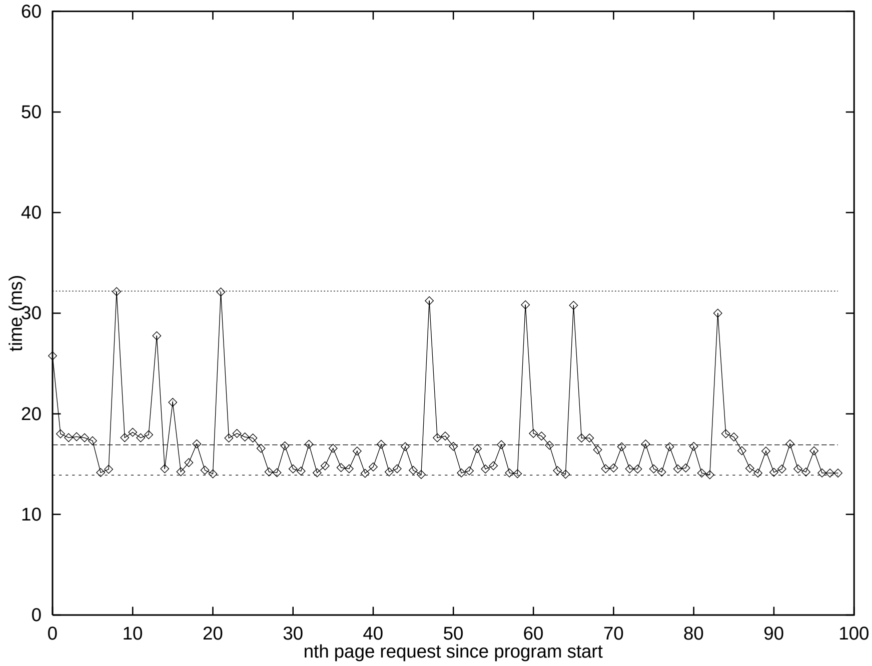

Figure 27: *The graph showing the latency for remote page copy is trapped between lines $y = 13.9$ and $y = 32.2$ with an average of 16.9.*

4 Load Balancing

This section 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 '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 Section 3. 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.

4.1 Assessing the Load

As was described in the introduction, 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 28.

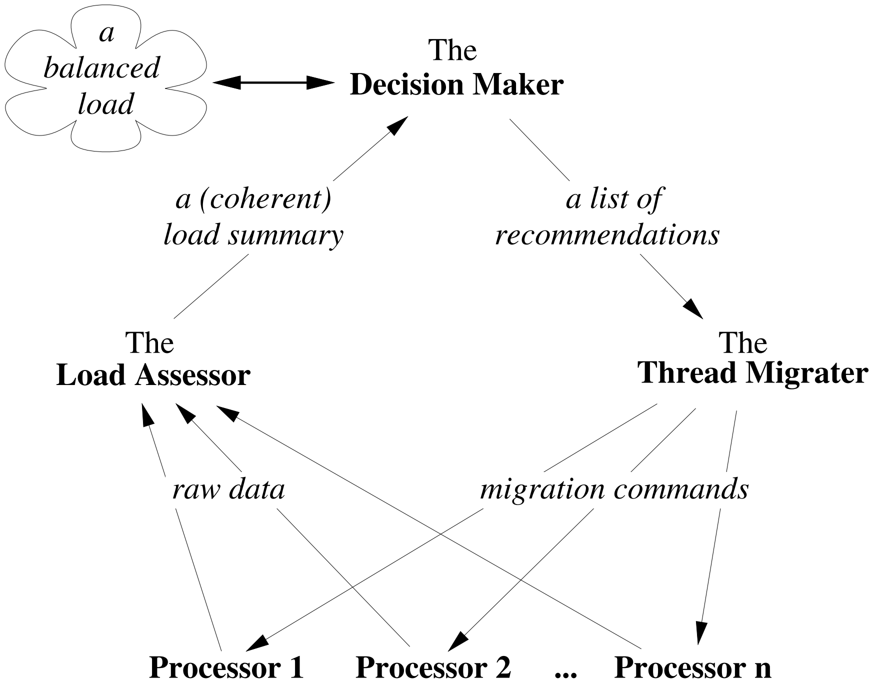

Figure 28: *The key agents in a dynamic load balancing system and the information exchanged between them.*

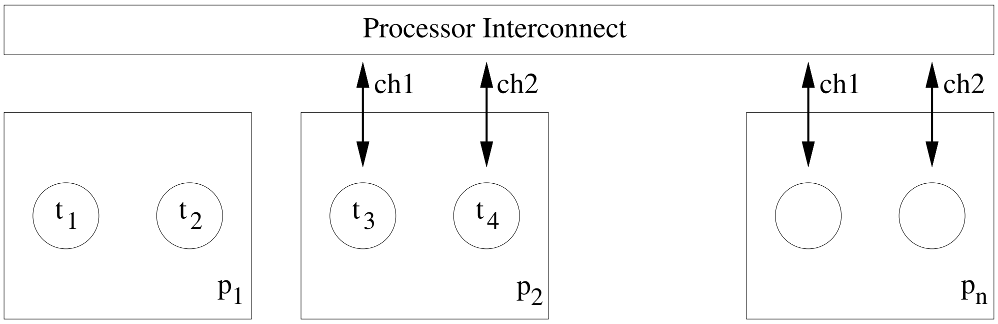

Figure 29: *Compute-bound threads t_1 and t_2 execute on processor p_1 while communication-bound threads t_3 and t_4 execute on processor p_2.*

4.1.1 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 29. 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 30 where the compute-bound threads will absorb all spare processor capacity whilst the communication-bound threads suffer at most by a delay of one time-slice per communication.

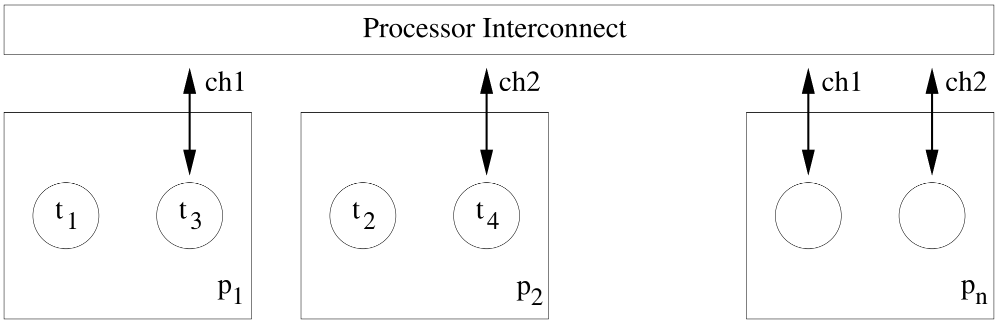

Figure 30: *One compute-bound and one communication-bound thread execute on each processor.*

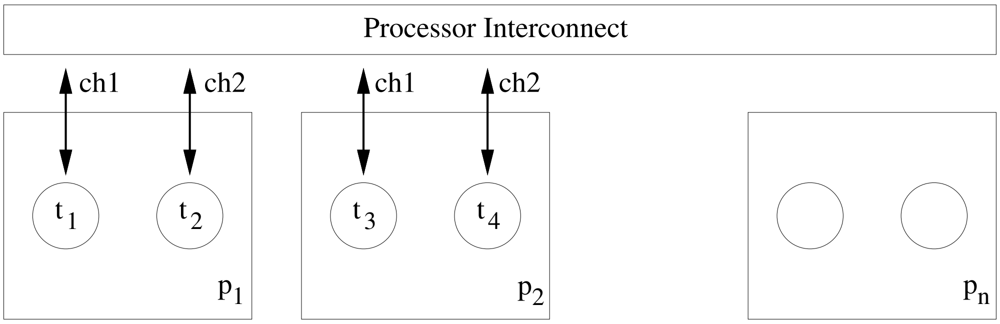

Figure 31: *All threads spend roughly half their time computing and half their time communication remotely over the network.*

An example of load balancing to optimise processor resources can be found in Bryant and Finkel [6]. 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 [27] 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* [3] 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 31. 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 32 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

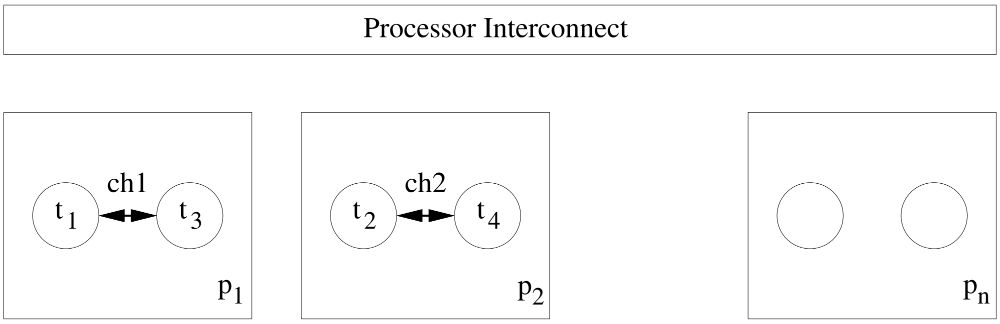

Figure 32: *All threads spend most of their time computing and a small proportion communicating locally.*

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 report.

This notion of a balanced load depends on an commonly-used assumption which Casavant and Kuhl [7] 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.

4.1.2 Selection factors for load metrics

The next step is to select one or more 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 above, 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* [28] 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 [19] in particular:

Cost —the additional time, space or hardware required to collect the metric and process it into the 'coherent summary'. Eager *et al* [9] 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 next.

4.1.3 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 [13], 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 [35] 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 [20] and Eager *et al* [9], 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-Crummy [21], 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 report (and perhaps not 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 3.3.6. 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.

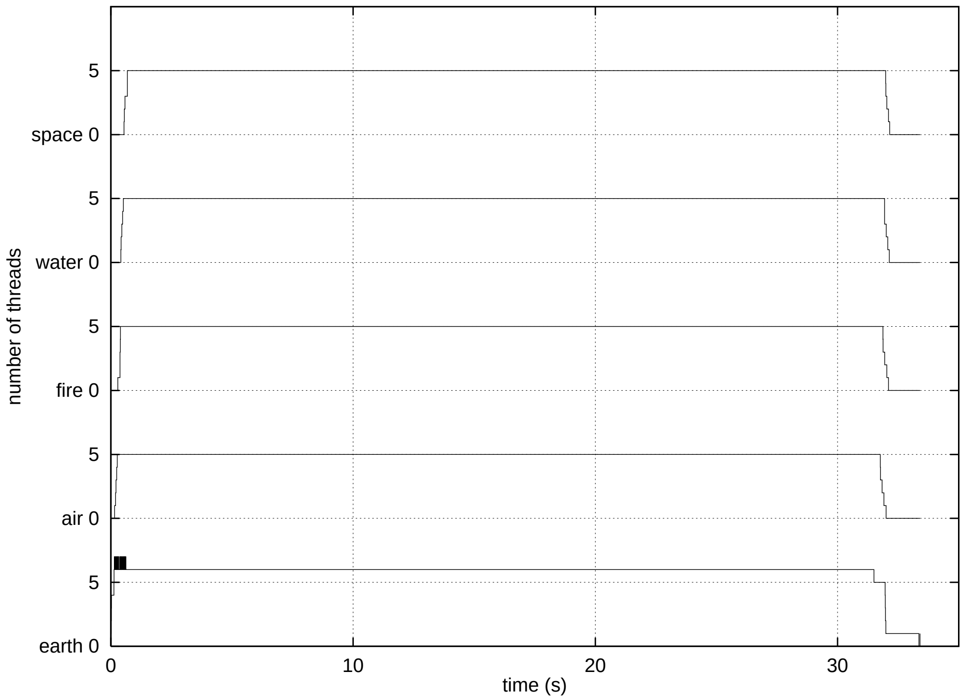

Figure 33: *N-THREADS—This graph shows, for each of the five processors, the number of threads executing on that processor during the execution of the test program* `multi-phase`.

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 5 (page 32). 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 2.2.3) 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 33.

Measuring the number of threads on each processor seems attractively simple. However, when the selection factors listed in Section 4.1.2 are considered the metric is seen to have a couple of serious problems 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 19 on page 33, 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 34.

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 suitable 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

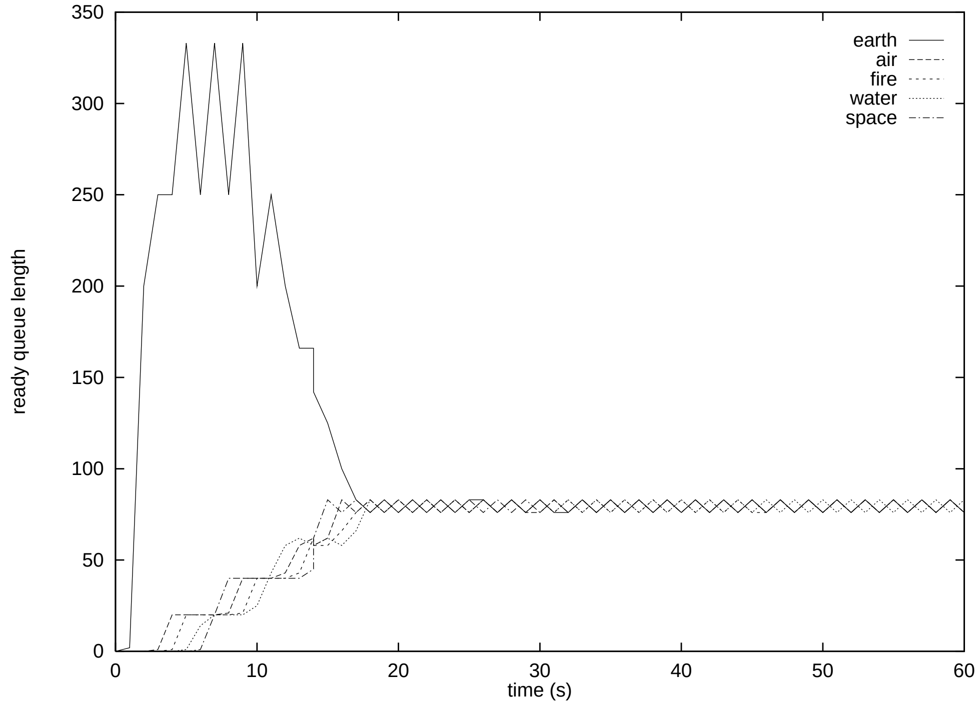

Figure 34: *RQ-LENGTH—This graph shows, for each of the five processors, the length of the ready queue during the program execution.*

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 with different degrees of averaging: in Figure 35 the results are not averaged; in Figure 36 the results are computed using a sliding window of 200ms and in Figure 37 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

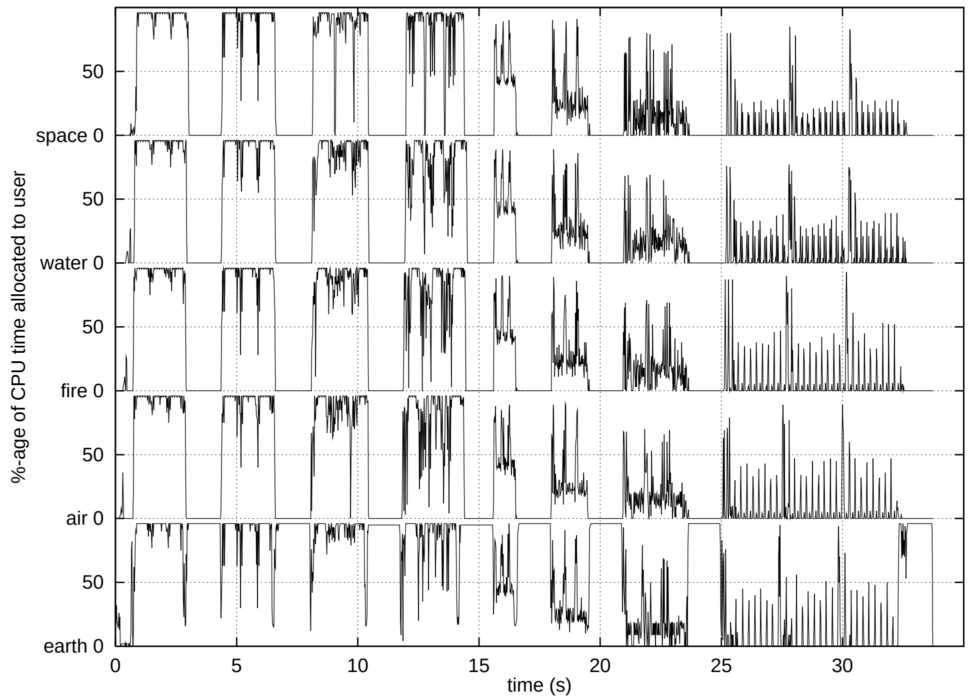

Figure 35: *CPU-TIME—Graph is plotted every 20ms showing the percentage of user CPU time over the previous 20ms period.*

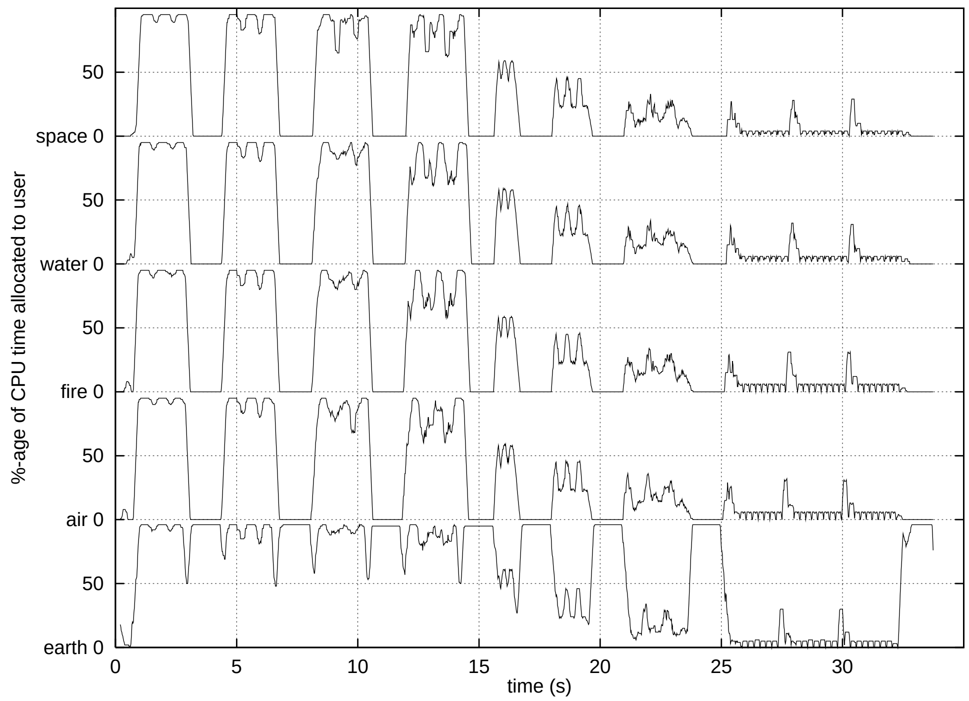

Figure 36: *CPU-TIME—Graph is plotted every 20ms showing the percentage of user CPU time over the previous* ***ten*** *20ms periods.*

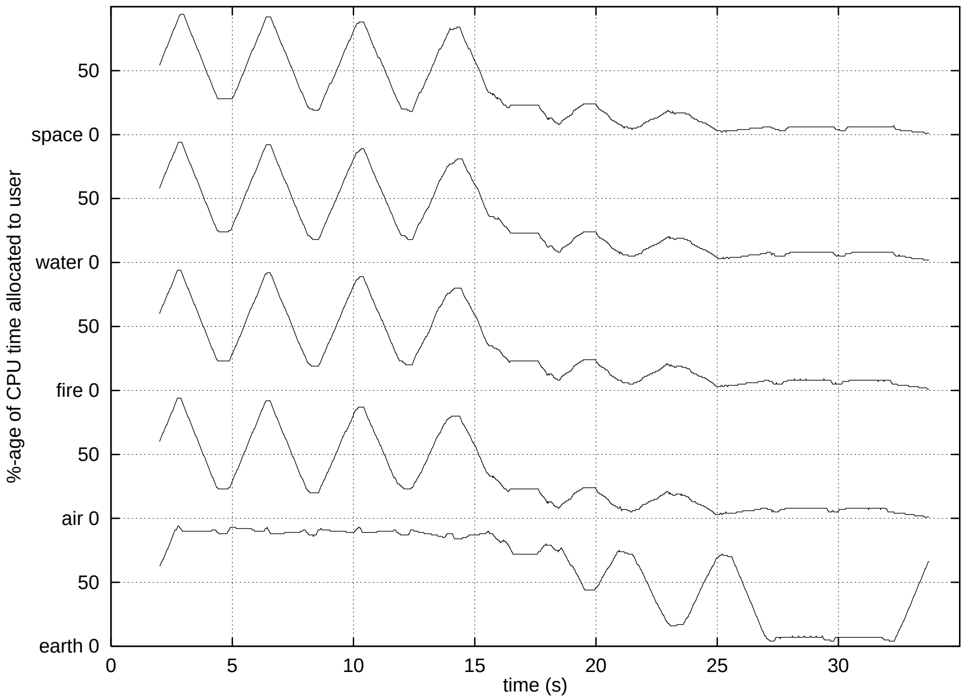

Figure 37: *CPU-TIME—Graph is plotted every 20ms showing the percentage of user CPU time over the previous* ***hundred*** *20ms periods.*

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 4.2.1) 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 35, then care has to be taken that the perceived load does not fall behind the real load—the peaks in Figure 37, for example, are about two seconds behind those in Figure 35.

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

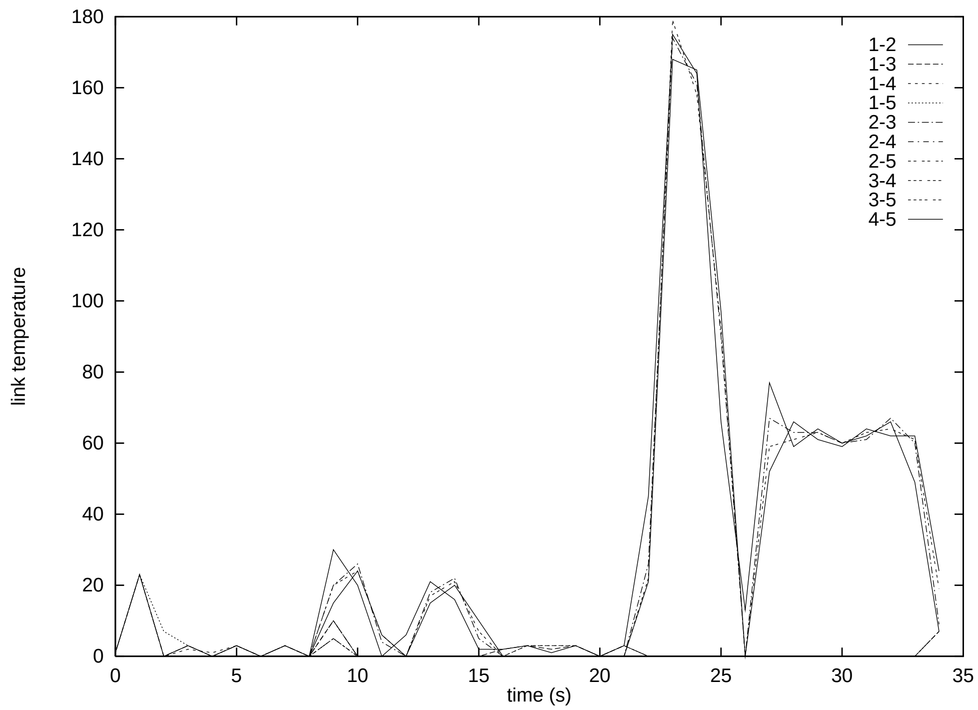

Figure 38: *REM-COMM—This graph shows, for each of the ten processor-to-processor communication links, the number of communications that have occurred during the last sample period of one second.*

is collected by sampling[2]. 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 38. 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

[2]There are two techniques 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.

Metric	*Event driven?*	*Cost*	*Relevant?*	*Timeliness*
N-THREADS	yes	low	no	guaranteed
RQ-LENGTH	yes	high	yes	guaranteed
CPU-TIME	yes	very high	perhaps	guaranteed
REM-COMM	sampling	low	yes	depends

Table 8: *Summary of results.*

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 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 12 (page 26) 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. Referring to Table 8, 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.

4.2 Strategy for Load Reconfiguration

The first part of this section 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

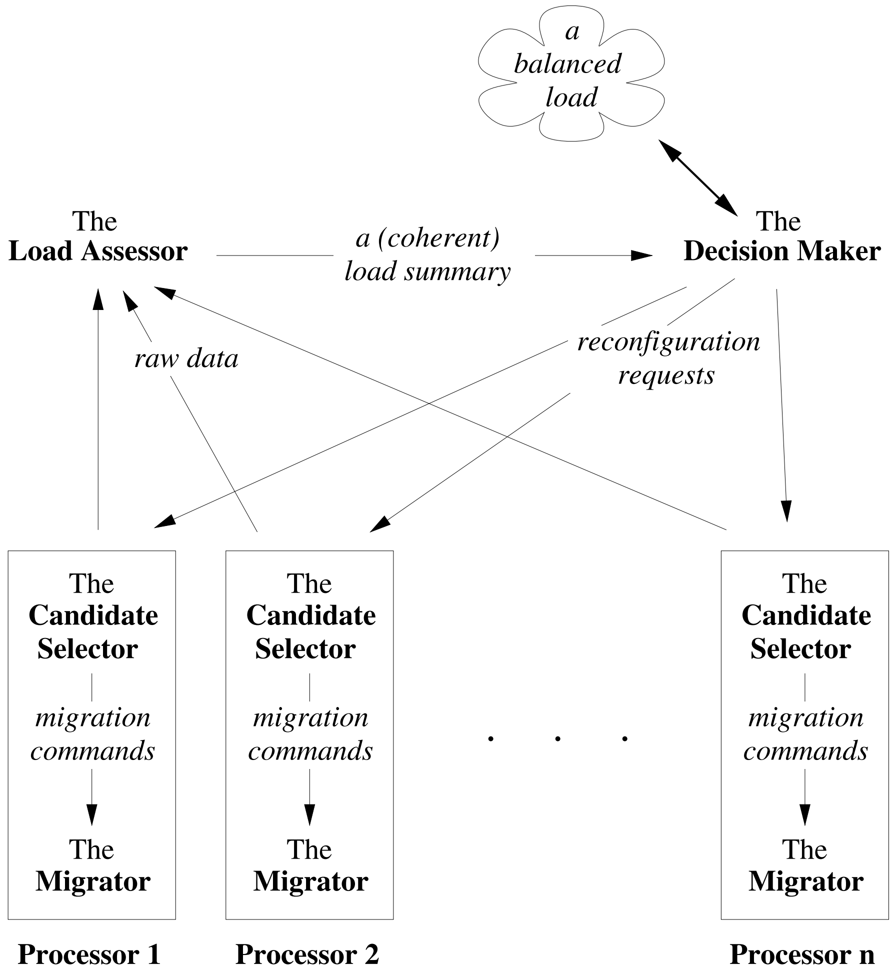

Figure 39: *Components of the Testbed's load balancing system.*

CPU load *and* communications load since, as Table 6 (page 39) 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 selector. The components in the Testbed's load balancing system are depicted by Figure 39.

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 Section 4.1. 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.

4.2.1 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 40. Threads with children are rejected because the formal specification of the Testbed requires that parent threads do not migrate. Threads

```
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 optimising a link..*/
      if (last_comm(t)==l) /*..and thread used link..*/
        return t;          /*..then return thread*/
      else if (!ct ||      /*Else if no candidate yet or..*/
          !last_comm(ct))  /*..cand's last comm not remote..*/
        ct=t;              /*..then remember new candidate*/
    }
    else return t;         /*If not optimising any link..*/
  }                        /*..then return thread*/
  return ct;               /*Return best candidate*/
}
```

Figure 40: *The candidate selection algorithm, in a C-like pseudocode.*

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.

4.2.2 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 processors. The Testbed is additionally restricted since the proofs of correctness of the migration protocols in Martin [26] 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 Section 3 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 13). In the worst-case scenario (as described in Section 3.3.5) 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 6). 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 $41.5 + 28.4 + n * 20$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 4.3. While these values obey the constraints described above, 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.

4.3 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 [33]). The first approach takes a simple program whose behaviour and optimal assignment to processors is obvious: Boillat and Kropf [5], 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 [23] 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 [30], for example, reports on the execution times of common Unix utilities such as LaTeX and `ls` and Ogle *et al* [29] simulate a war game. This report uses both approaches.

4.3.1 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 `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

```
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 41: *Extract from the Testbed load balancer's log file.*

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 `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 41: the `TIM` field gives the time in ms at which the load summary was obtained; the `LNK` field gives the number of communications on each of the ten links; the `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 `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 42 to 44. 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

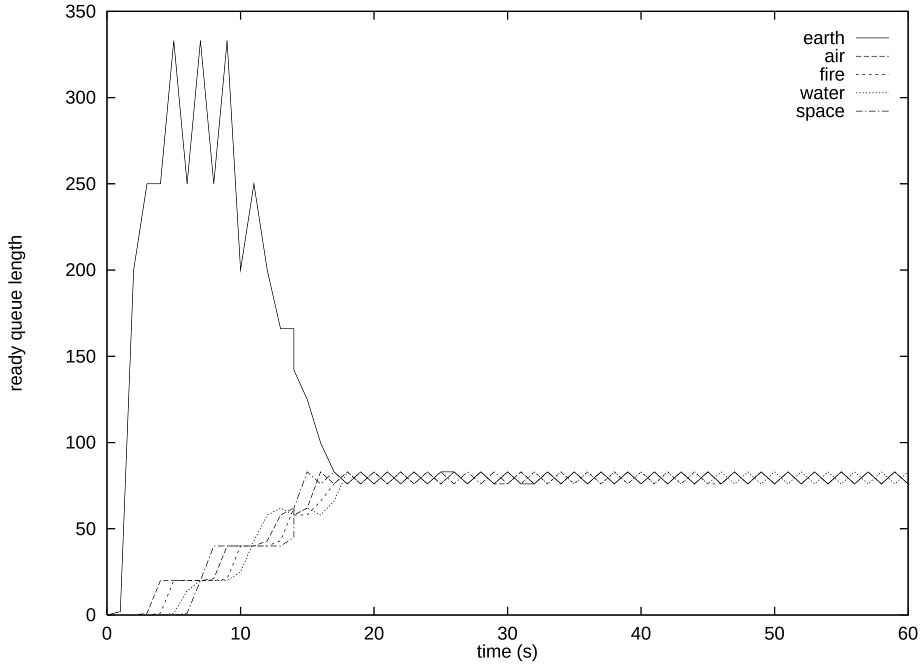

Figure 42: *Length of the ready queue on each of the five slave processors while executing 20 compute-bound threads.*

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 45 to 47). The computational 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 47 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 48).

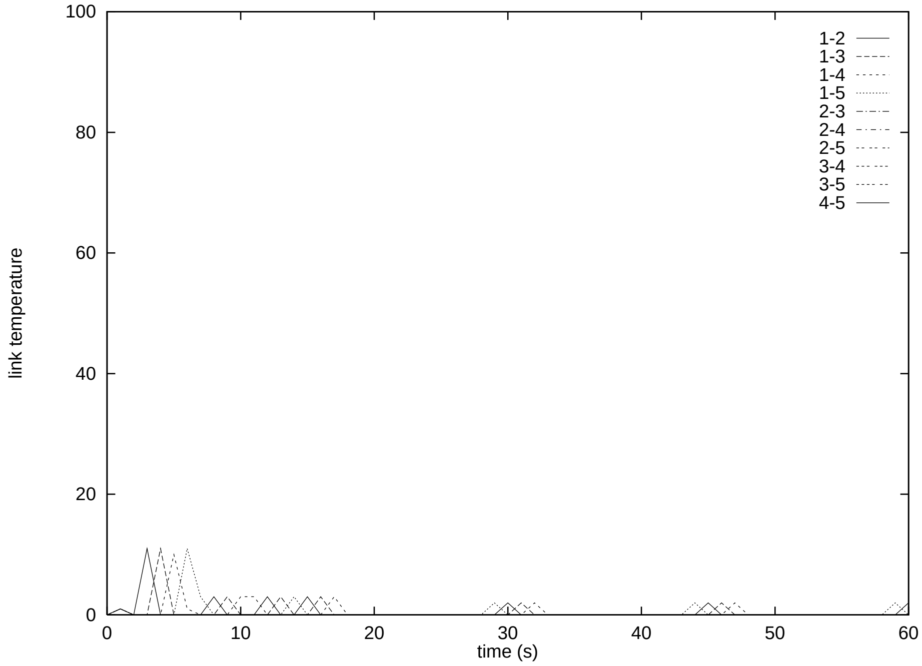

Figure 43: *Number of communications on each link while executing 20 compute-bound threads.*

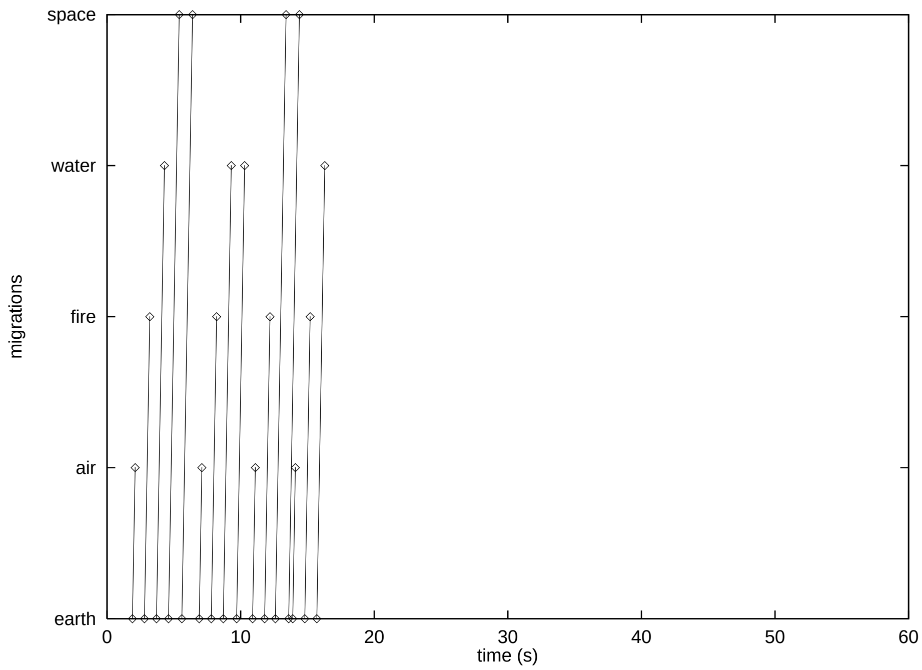

Figure 44: *Thread migrations while executing 20 compute-bound threads.*

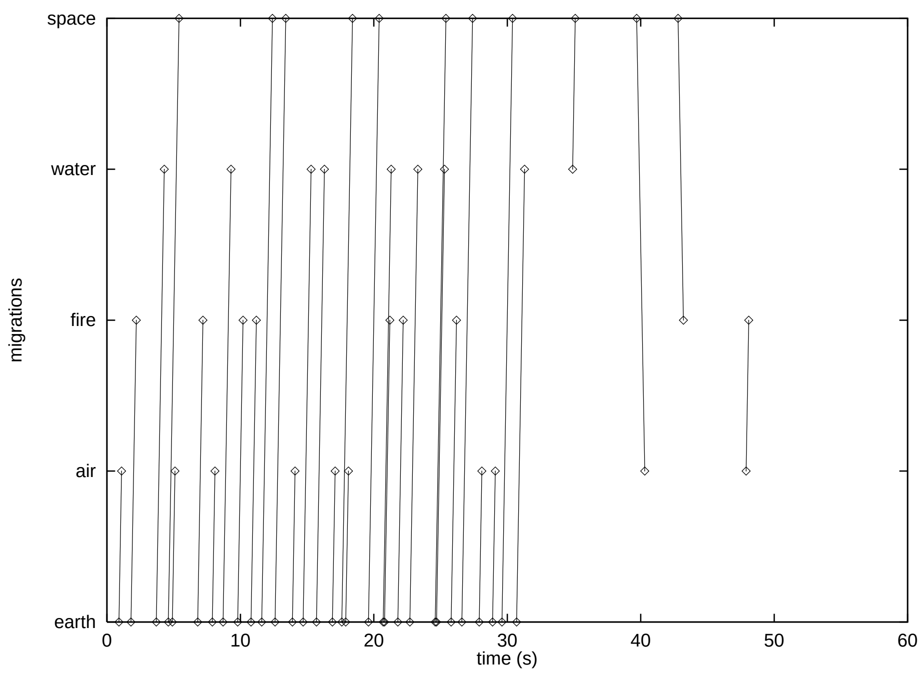

Figure 45: *Length of the ready queue on each of the five slave processors while executing 20 pairs of communication-bound threads.*

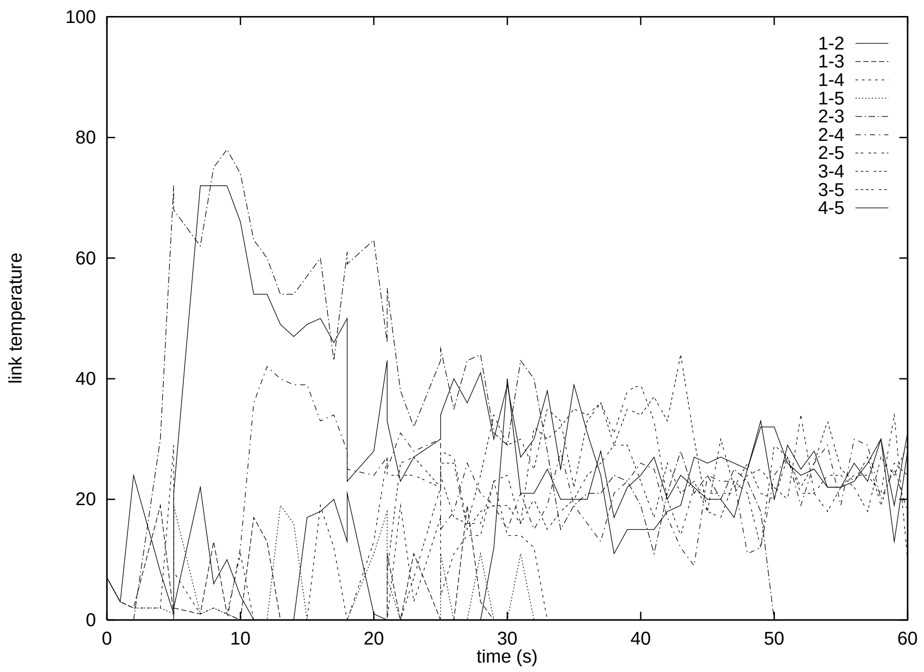

Figure 46: *Number of communications on each link while executing 20 pairs of communication-bound threads.*

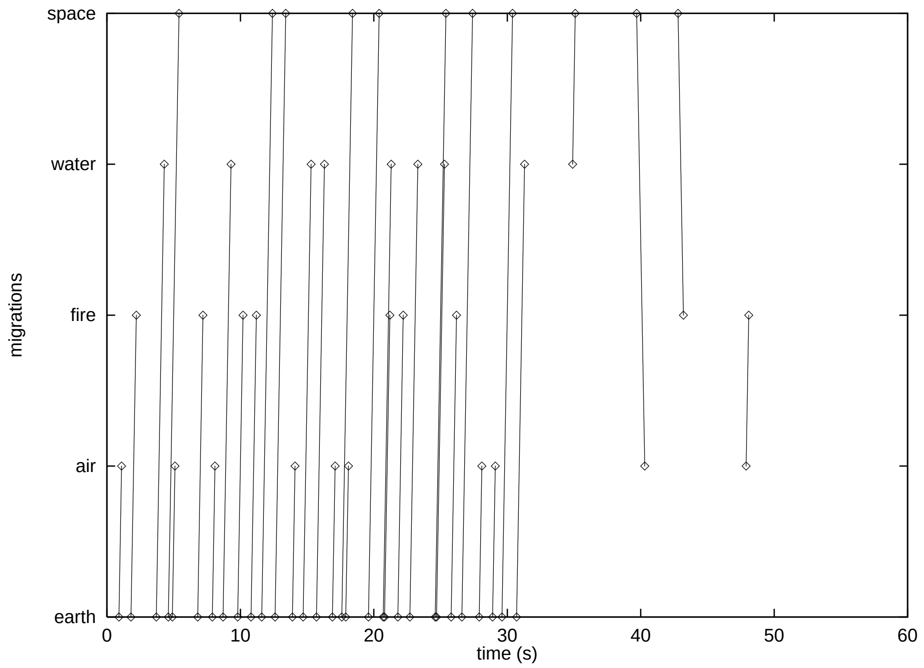

Figure 47: *Thread migrations while executing 20 pairs of communication-bound threads.*

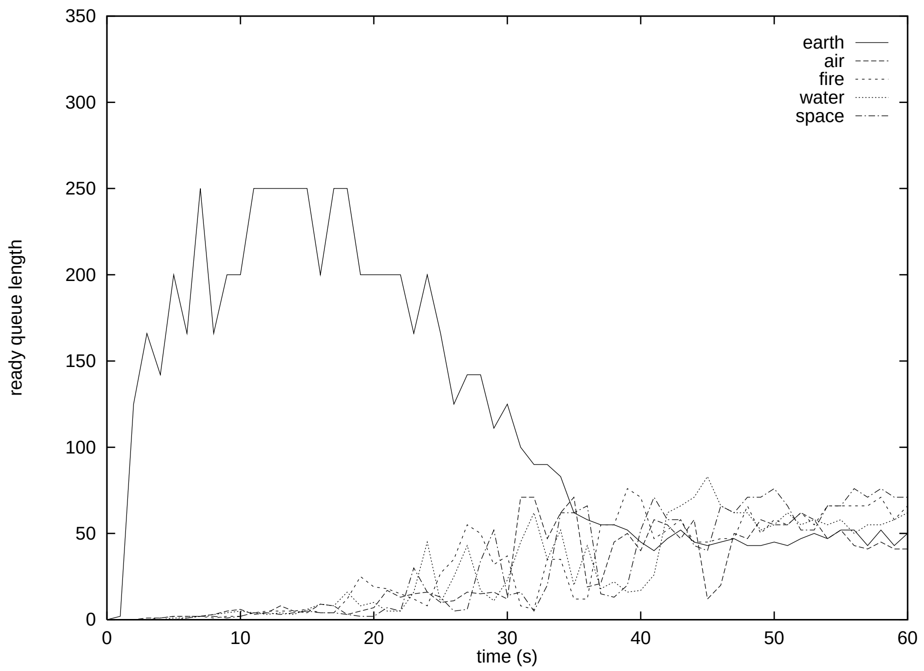

Figure 48: *Length of the ready queue on each of the five slave processors while executing 20 mixed threads.*

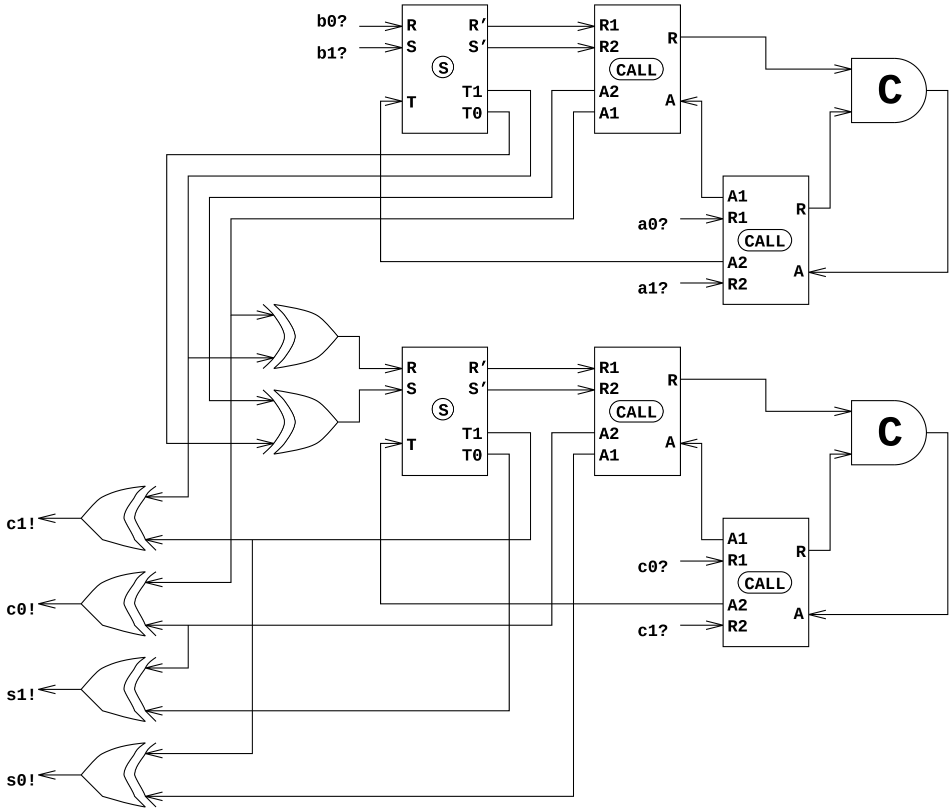

Figure 49: *An asynchronous adder circuit.*

4.3.2 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 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 [34].

Simulation of asynchronous circuits can be easily achieved on the Testbed. The `adder` program, which simulates the 8-bit adder circuit shown in Figure 49, 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

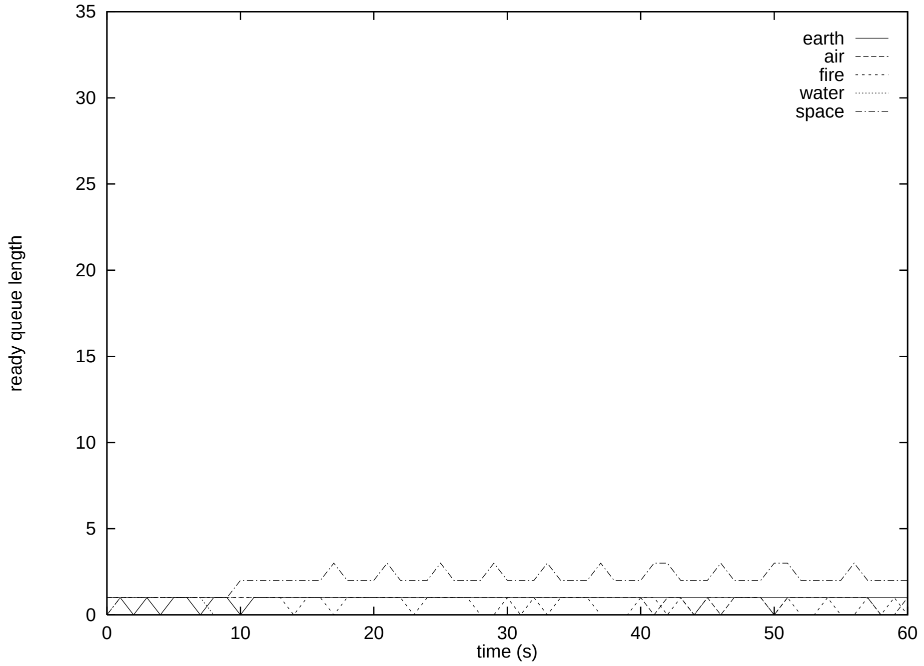

Figure 50: *Length of the ready queue on each of the five slave processors while executing* `adder`*.*

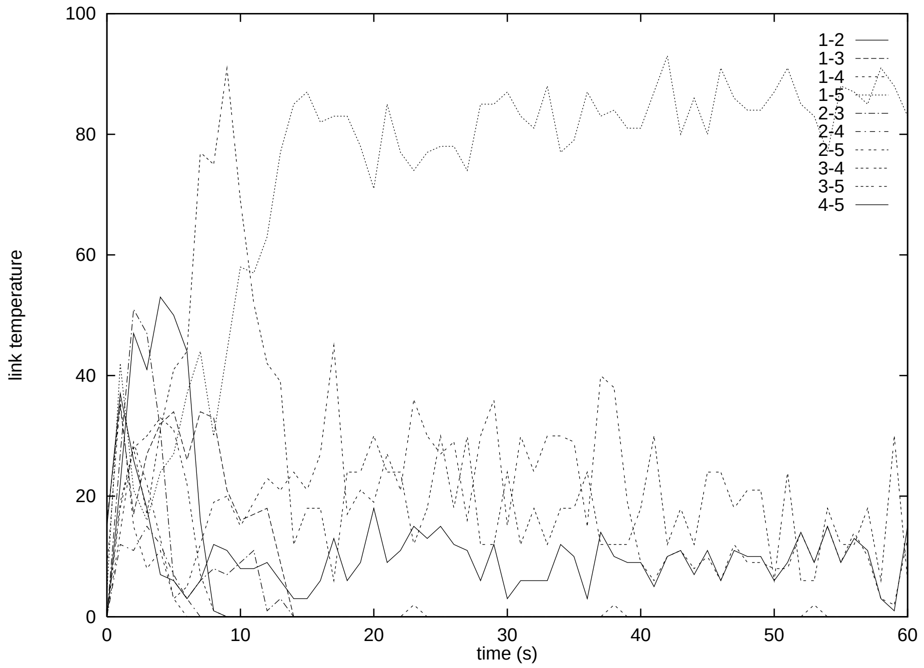

Figure 51: *Number of communications on each link while executing* `adder`*.*

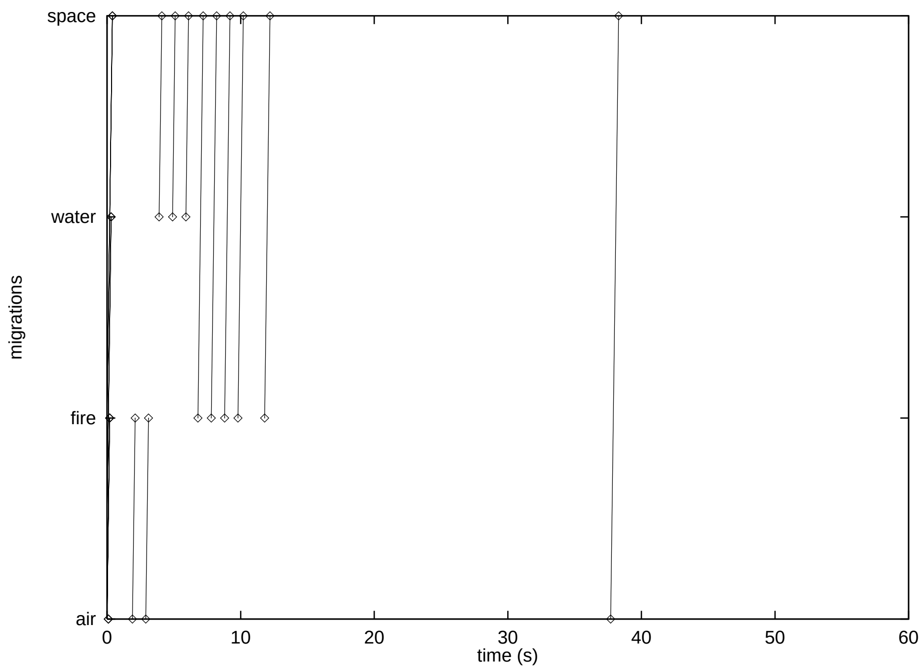

Figure 52: *Thread migrations while executing* `adder`.

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 50 to 52. 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 50 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 shows that reconfiguration requests were issued to correct the problem but that the slave processors could not find appropriate threads for migration.

4.3.3 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* [11, Section 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 53 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 (Figures 54 and 55) and then by showing the same loads *with* load balancing in action (Figures 56 to 58).

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 53 can be approximately matched against the graph in Figure 54 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 53 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 53 as it does not have any region

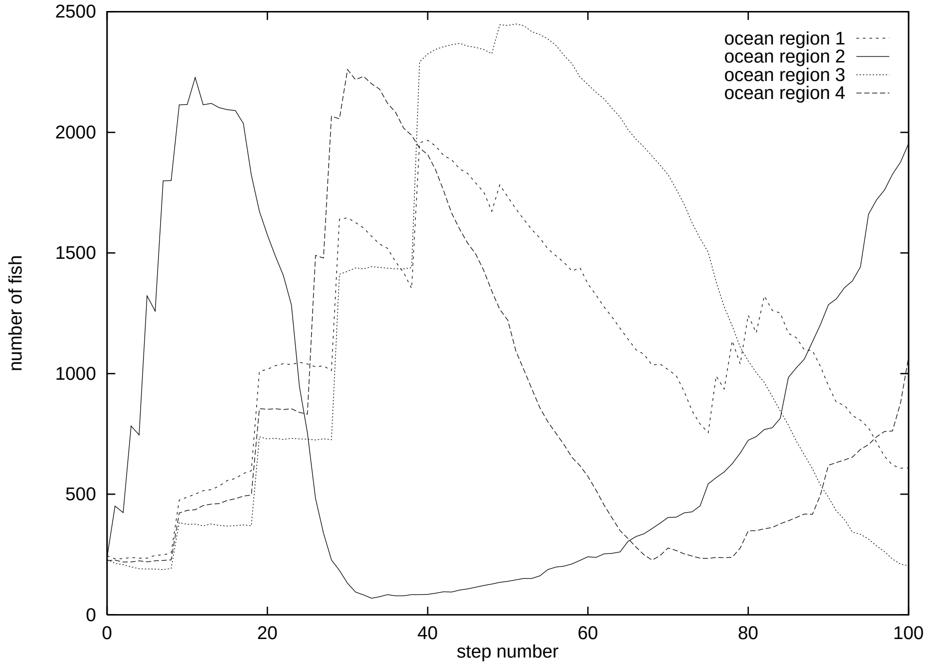

Figure 53: *Variation in number of fish during 100 steps of the WaTor simulation.*

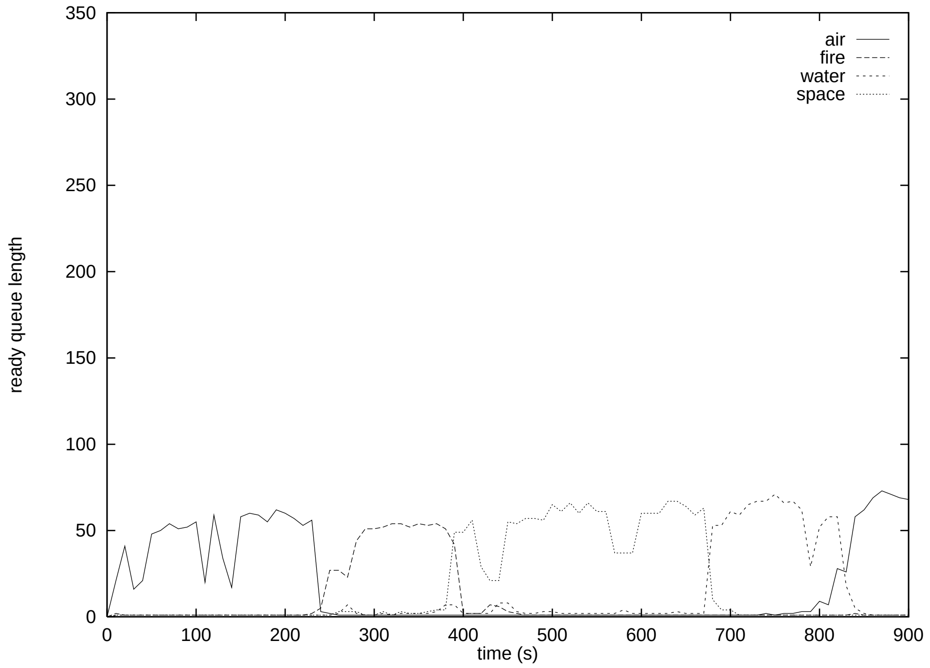

Figure 54: *Ready queue variation during 100 steps of the WaTor simulation* ***without*** *load balancing.*

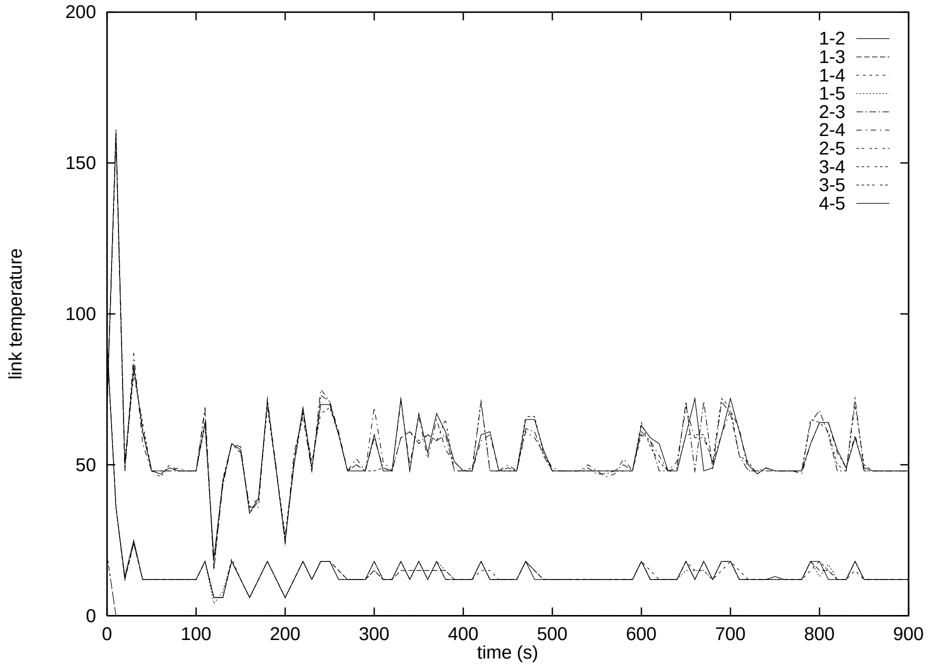

Figure 55: *Variation in link load during 100 steps of the WaTor simulation* ***without*** *load balancing.*

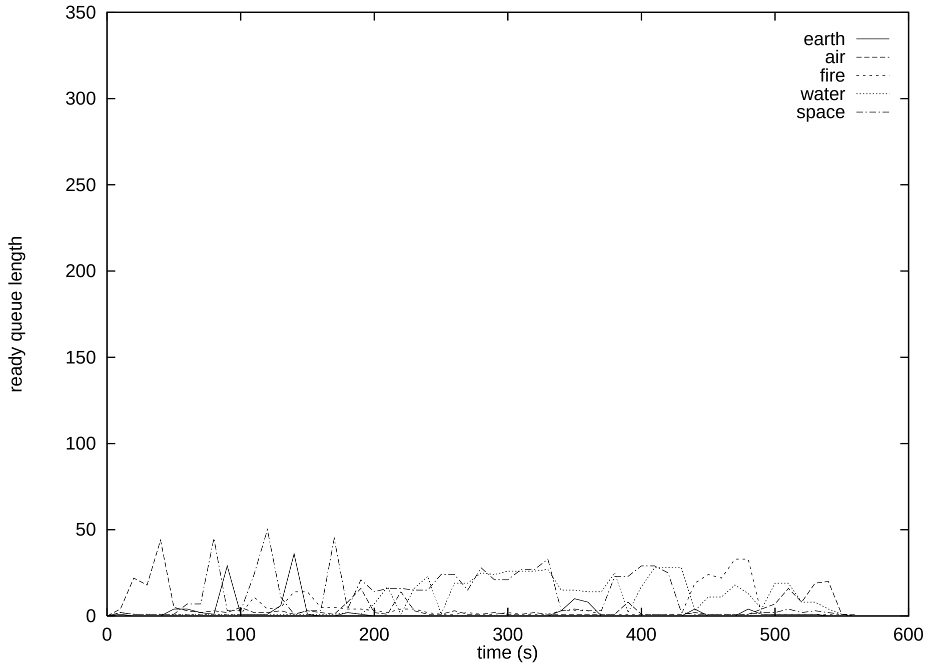

Figure 56: *Ready queue variation during 100 steps of the WaTor simulation* ***with*** *load balancing.*

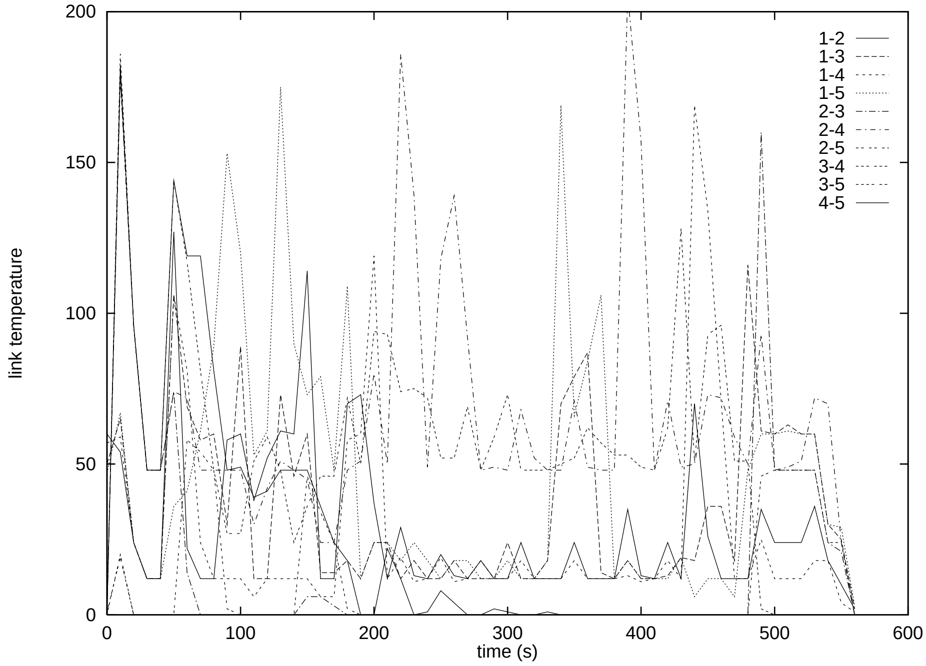

Figure 57: *Variation in link load during 100 steps of the WaTor simulation* ***with*** *load balancing.*

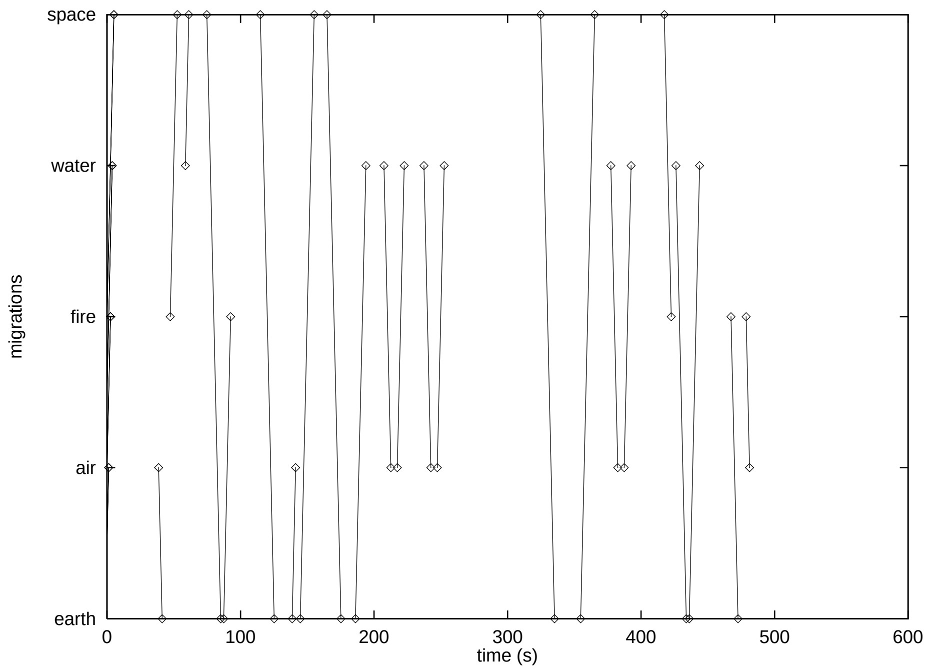

Figure 58: *Thread migrations while executing 100 steps of the WaTor simulation.*

Application	*Sequential*	*Random*	*Balancing*
20 compute-bound threads	54 (100%)	15 (28%)	21 (39%)
20 communication-bound threads	72 (100%)	41 (57%)	35 (49%)
`adder` simulation	93 (100%)	183 (197%)	126 (135%)
`wator` simulation		900 (100%)	550 (61%)

Table 9: *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.*

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 55.

The effect of load balancing the `wator` simulation is seen in Figure 56 where the average ready queue length is reduced (cf. Figure 54). This strategy leads to an increase in remote communication—compare Figures 55 and 57—but 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 54 and 56. Figure 58 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.

4.3.4 Summary

The effectiveness of the Testbed's load balancer is summarised in Table 9. 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.

5 Conclusions

In Section 3 a detailed and accurate profile was made of the performance of the Testbed operations involved in task migration. 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.

In Section 4 I 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 presented selection criteria for load metrics and illustrated their use in several case studies. I 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 Douglis and Ousterhout [8], for example, argue that the additional overheads of load collection make dynamic migration prohibitively expensive and Ieumwananonthachai *et al* [15] say the information collected will usually be incomplete and out of date, the work presented in this section (and that of, amongst others, Livny and Melman [24]) 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.

It is a conclusion of this report 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 4.3 showed that significant speed-ups could be achieved with the RQ-LENGTH and REM-COMM metrics.

- The case studies on load metrics in Section 4.1.3 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 3.3.7 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 [31]—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.

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

References

[1] Yeshayahu Artsy and Raphael Finkel. Designing a Process Migration Facility, The Charlotte Experience. *IEEE Computer Magazine*, 22(9):47–56, September 1989.

[2] Thomas Bemmerl and Arndt Bode. An Integrated Environment for Programming Distributed Memory Multiprocessors. In A. Bode, editor, *Distributed Memory Computing, LNCS 487.* Springer, 1991.

[3] Thomas Bemmerl, Arndt Bode, Olav Hansen, and Thomas Ludwig. A Testbed for Dynamic Loadbalancing Distributed Memory Multiprocessors. 1990.

[4] Thomas Bemmerl, Robert Lindhof, and Thomas Treml. The Distributed Monitor System of TOPSYS. In H. Burkhart, editor, *Proceedings CONPAR*, pages 756–765, September 1990.

[5] Jacques E. Boillat and Peter G. Kropf. A Fast Distributed Mapping Algorithm. In *Proceedings CONPAR*, pages 405–416, 1990.

[6] Raymond M. Bryant and Raphael A. Finkel. A Stable Distributed Scheduling Algorithm. In *Second International Conference on Distributed Computing Systems*, pages 314–323, April 1981.

[7] Thomas L. Casavant and Jon G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[8] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Concurrency—Practice and Experience*, 21(8):757–785, August 1991.

[9] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[10] Raphael A. Finkel, Michael L. Scott, Yeshayahu Artsy, and Hung-Yang Chang. Experience with Charlotte: Simplicity and Function in a Distributed Operating System. *IEEE Transactions on Software Engineering*, 15(6):676–685, June 1989.

[11] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, 1988.

[12] Dieter Haban and Kang G. Shin. Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times. *IEEE Transactions on Software Engineering*, 16(12):1374–1398, December 1990.

[13] Dieter Haban and Dieter Wybranietz. A Hybrid Monitor for Behaviour and Performance Analysis of Distributed Systems. *IEEE Transactions on Software Engineering*, 16(2):197–211, February 1990.

[14] Roland N. Ibbett, D. A. Edwards, T. P. Hopkins, C.K Cadogan, and D. A. Train. Centrenet—A High Performance Local Area Network. *The Computer Journal*, 28(3):231–242, July 1985.

[15] Arthur Ieumwananonthachai, Akiko N. Aizawa, Steven R. Schwartz, Benjamin W. Wah, and Jerry C. Yan. Intelligent Process Mapping through Systematic Improvement of Heuristics. *Journal of Parallel and Distributed Computing*, 15(2):188–142, June 1992.

[16] Kayhan Imre. *A Performance Monitoring and Analysis Environment for Distributed Memory MIMD Programs.* PhD thesis, University of Edinburgh, 1993.

[17] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[18] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[19] Orly Kremien and Jeff Kramer. Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):747–760, November 1992.

[20] Thomas Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[21] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler. Analyzing Parallel Program Executions Using Multiple Views. *Journal of Parallel and Distributed Computing*, 9:203–217, 1990.

[22] INMOS Limited. *occam2 Reference Manual*. Prentice Hall International (UK) Ltd, 1988.

[23] Frank C. H. Lin and Robert M. Keller. The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engineering*, 13(1):32–38, January 1987.

[24] Miron Livny and Myron Melman. Load Balancing in Homogeneous Broadcast Distributed Systems. In *Proceedings of the ACM Computer Network Performance Symposium*, pages 47–55, April 1982.

[25] Masoud Mansouri-Samani and Morris Sloman. Monitoring Distributed Systems. Technical Report DOC92/23, Imperial College, April 1993.

[26] Paul Martin. The Formal Specification in **Z** of Task Migration on the Testbed Multicomputer. Technical Report ECS-CSG-2-94, University of Edinburgh, June 1994.

[27] Paul C. Messina. Parallel Computing in the 1980s—One Person's View. *Concurrency—Practice and Experience*, 3(6):501–524, December 1991.

[28] Lionel M. Ni, Chong-Wei Xu, and Thomas B. Gendreau. A Distributed Drafting Algorithm for Load Balancing. *IEEE Transactions on Software Engineering*, SE-11(10):1153–1161, October 1985.

[29] David Ogle, Karsten Schwan, and Richard Snodgrass. Real-Time Collection and Analysis of Dynamic Information in a Distributed System. Technical Report OSU-CISRC-TR-85-12, Computer and Information Science Research Center, Ohio State University, Columbus, September 1985.

[30] William Osser. Automatic Process Selection for Load Balancing. Master's thesis, University of California, Santa Cruz, June 1992.

[31] Joseph Phillips. *A Statistical Investigation of the Factors Influencing the Performance of Parallel Programs, with Application to a Study of Process Migration Strategies*. PhD thesis, University of Edinburgh, 1994.

[32] Dick Pountain. *A Tutorial Introduction to Occam Programming*. INMOS Limited, March 1987.

[33] C. Gary Rommel. The Probability of Load Balancing Success in a Homogeneous Network. *IEEE Transactions on Software Engineering*, 17(9):922–933, September 1991.

[34] Roger Sayle. *Automated Synthesis of Delay Insensitive Circuits*. PhD thesis, University of Edinburgh, 1994.

[35] Y. T. Wang and R. H. T. Morris. Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, C-34(3):204–217, March 1985.

[36] Peter H. Welch. The Role and Future of Occam. Available by ftp from unix.hensa.ac.uk (129.12.21.7), 1993.

[37] Simon Woods. Error Diagnosis in a Distributed Environment. Master's thesis, University of Edinburgh, September 1990.

[38] Martina Zitterbart. Monitoring and Debugging Transputer-Networks with NETMON-II. In H. Burkhart, editor, *Proceedings CONPAR*, pages 200–209, September 1990.

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 (op1=0; op1<256; op1++)
    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');
    }
  exit();
}

keller()
{
  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);
    }
  }
}
```