Evaluation of Multiprocessor Interconnection Networks

F.W. Howell & R.N. Ibbett

Technical Report CSG-38-98
Department of Computer Science
University of Edinburgh

July 27, 1998

Abstract

This report describes the work carried out under the EMIN project to set up a testbed for simulating multiprocessor networks. All levels from low level hardware to the software interfaces affect performance, and so the initial simulation testbed provided an MPI interface on top of a cycle level simulator. The networks modelled included a crossbar and the Cray T3D network. Meaningful simulations at this level of detail proved infeasible, however, and an alternative approach was to use microbenchmarking, of both shared memory and message passing network primitives, as a means of characterising network performance in a way which is meaningful to programmers. This led to a refined simulation testbed which cleanly separates workload models from network models, using an interface based on the microbenchmarking work. In a further development, a web version of the testbed was developed and the value of this approach to modelling is evaluated, in particular the accessibility of the simulation models and the importance of visualisation.

Contents

Chapter 1

Introduction

This report documents the work carried out as part of the EMIN (Evaluation of Multiprocessor Interconnection Networks) project under EPSRC Grant GR/K19716.

The aim of the project was to set up a simulation testbed for evaluating interconnection networks, stretching from low level hardware details up to the software which uses them.

The report is structured into four chapters which follow the development of the project through time.

The first approach was to include everything in the model, from a clock cycle level simulation of the network up to the interface which parallel software sees (Chapter 2). This was successful in showing that it was possible to interface real parallel software (in this case the MPI interface) to low level hardware models, but was expensive in terms of time to construct models and time to execute them.

The next stage was to develop an appropriate interface layer between hardware and software models to reduce the complexity of the combined models (Chapter 3). The idea was that hardware models would reach up to this interface, software models would reach down to it, and there would be scope for 'plug and play' between models. The interface layer built on earlier work in microbenchmarking and extended its scope.

Once the interface layer had been constructed, the next stage was to use it as the basis for building a simulation testbed. This decoupling of hardware models and software workloads is described in Chapter 4.

For a design tool to be effective, the turnaround time must be small and the results must be visible and understandable. Visualisation of simulation results is therefore crucial. Figure 1.1, for example, is a screen shot of a visualisation tool developed for viewing simulation results as a timing diagram.

Simjava [1, 2, 3] takes this one step further, and allows live simulations to be included in documentation. The simulation testbed developed originally using C++ was therefore ported to the simjava simulation language to allow wider access to models across the web (Chapter 5). The on-line HTML version of this report [4] include examples of the models developed in this project.

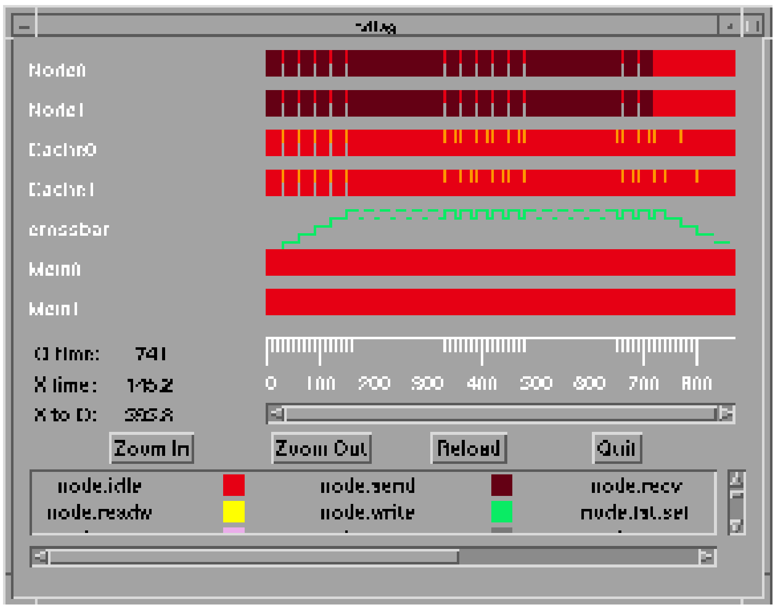

Node0
Node1
Cache0
Cache1
crossbar
Mem0
Mem1
O time: 741
X time: 145.2
X to O: 292.8
0 100 200 300 400 500 600 700 800
Zoom In
Zoom Out
Reload
Quit
node.idle
node.send
node.recv
node.readw
node.write
node.tst.set

Figure 1.1:

Chapter 2

The MPI Models

The initial aim of the EMIN project was to build realistic simulation models of actual networks, ranging from the clock cycle level up to high level software. This chapter describes the design of these models, and the lessons learnt while building them. The models built were a large packet switching crossbar, a generic multistage network and the Cray T3D torus interconnect.

2.1 The VLXbar

The VLXbar [5] was a proposal for a packet switched crossbar switch design based on an earlier design (Xbar [6]) which had been fabricated in silicon. The aim of the simulation study was to investigate some design alternatives and to visualise how the system behaved under a variety of workloads. Figure 2.1 shows the simulation model constructed using HASE. Down the left hand side are eight sources; these feed packets into the input queues. The outputs are also buffered; each output is shown in a dotted box and includes two output queues and arbiters to decide (a) when a flit (each packet transmission is composed of a number of flits) can go into an output queue and (b) which one. The output sinks are at the right hand side of the picture. The crossbar interconnect is an $N \times N$ connection between the input queues and the output queues (not shown in this visualisation).

During model animation the queues fill and empty, the arbiters show which way they are arbiting, and the input and output sinks and sources show when they are blocked and busy.

2.1.1 Traffic generation

A random workload was developed for the model's source entity. The parameters were **utilisation** and **hot spot probability**. Utilisation is defined as the proportion of time that the source is sending flits to the network. The easiest way of generating an *X% utilisation workload* is to use a random number generator to choose the

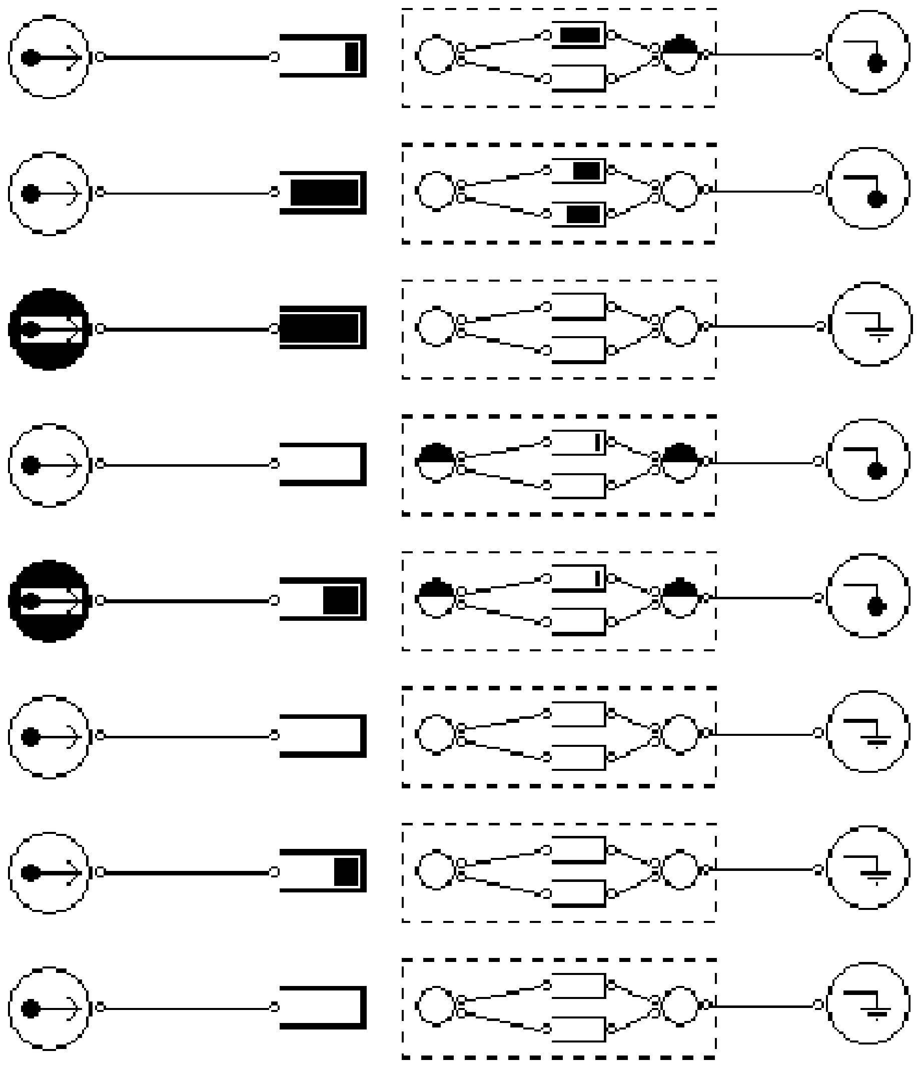


Figure 2.1: The HASE model of the VLXbar network

time delay between sucessive packets (e.g. for 50% utilisation, choose a random delay with an average of the packet transmission time). This tends to produce more evenly distributed traffic than is realistic. Each packet transmission is composed of a number of flits, so a more sophisticated traffic generator was used which samples the random number generator after every flit to decide whether to initiate a new packet. If the decision is positive, the new packet starts immediately after the current one. This approach to generation leads to more realistic 'bursty' traffic.

For this model a **hot spot probability** parameter was used to decide to send a packet to a fixed destination rather than to a random one.

From a single run of the simulation, the output is a timing diagram and statistics collected by each entity. As an example, the sink entity gives the number of packets and flits received at each output, and the minimum, average and maximum delays which packets suffered, as shown in the following table.

```
(90% util, pkt size = 8, 10% hotspotting, 8 inputs, 1000 clk cycles):-

u:sinks0 at 1000.050000: C Pkts:68, flits:545, delay:(9,24,47)
u:sinks1 at 1000.050000: C Pkts:114, flits:915, delay:(15,127,221)
u:sinks2 at 1000.050000: C Pkts:30, flits:241, delay:(15,19,35)
u:sinks3 at 1000.050000: C Pkts:40, flits:326, delay:(15,21,36)
u:sinks4 at 1000.050000: C Pkts:76, flits:608, delay:(15,51,130)
u:sinks5 at 1000.050000: C Pkts:51, flits:408, delay:(15,20,45)
u:sinks6 at 1000.050000: C Pkts:103, flits:824, delay:(9,119,240)
u:sinks7 at 1000.050000: C Pkts:115, flits:926, delay:(9,74,157)
```

For multiple runs of the simulation, the effect on output performance of varying input parameters was obtained. The following graphs show the effects on the system components of varying the utilisation.

Figure 2.2 shows how the *maximum* input queue length increases fairly linearly as utilisation increases.

Figure 2.3 shows how the average output queue lengths increase up to an average of 1 at 70% utilisation and then level out. This is because once the output queue length hits an average of one, the outputs are saturated.

The upper graph in Figure 2.4 shows how the *average* and *maximum* packet delay varies with utilisation. The lower graph shows the 10 percentile distribution. There is a big jump at 80% in the maximum packet delay (and the average packet delay then grows more slowly); some of the packets start to be very slow above 80% utilisation.

Figure 2.5 shows the percentages of time the output arbiters are in the states **NONE** (*i.e.* idle), **Q1** (arbitrating for output queue 1), **Q2** (arbitrating for output queue 2), while Figure 2.6 shows the percentages of time the output multiplexors are in the states **NONE**, **SEND1** (*i.e.* sending from output queue 1 to the sink), **SEND2** (*i.e.* sending from output queue 2 to the sink).

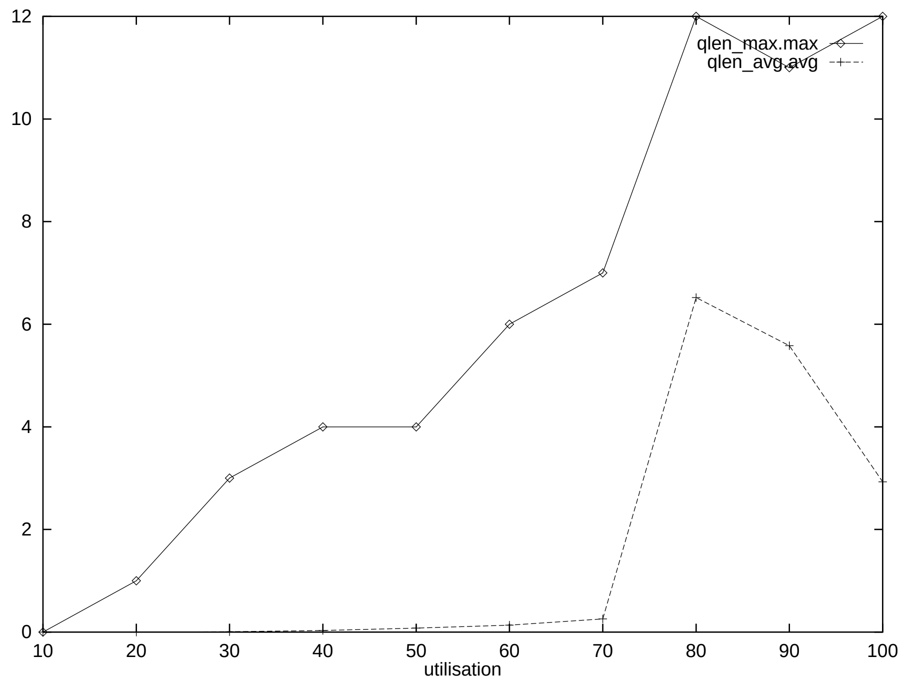
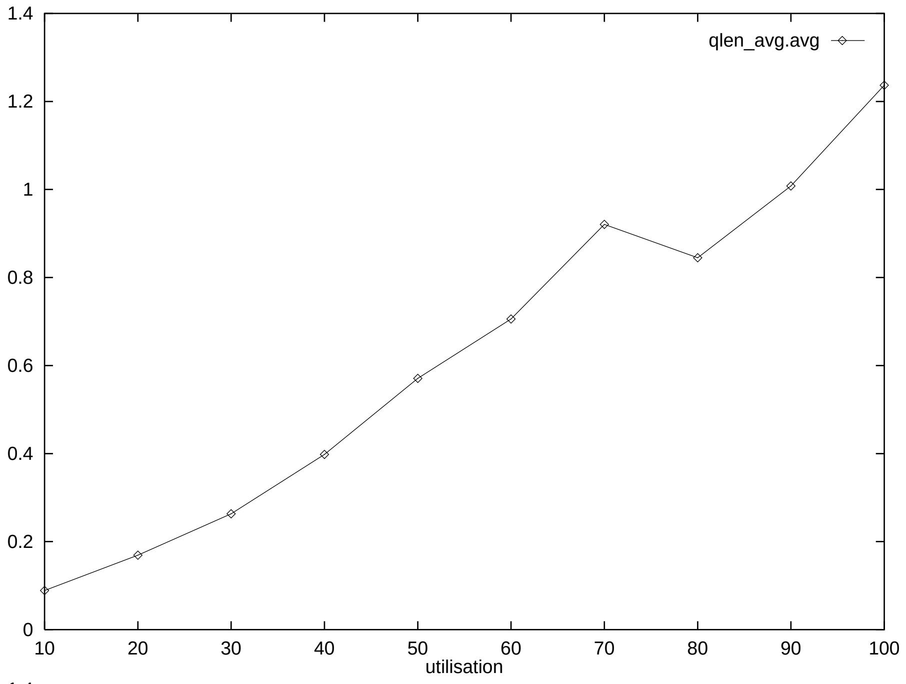


Figure 2.2: Maximum input queue length vs utilisation.

Figure 2.7 shows the influence of the network on the source (upper graph) and sink (lower graph); the source is never blocked until 80% utilisation, and the sink processes progressively more flits until the network starts to saturate.



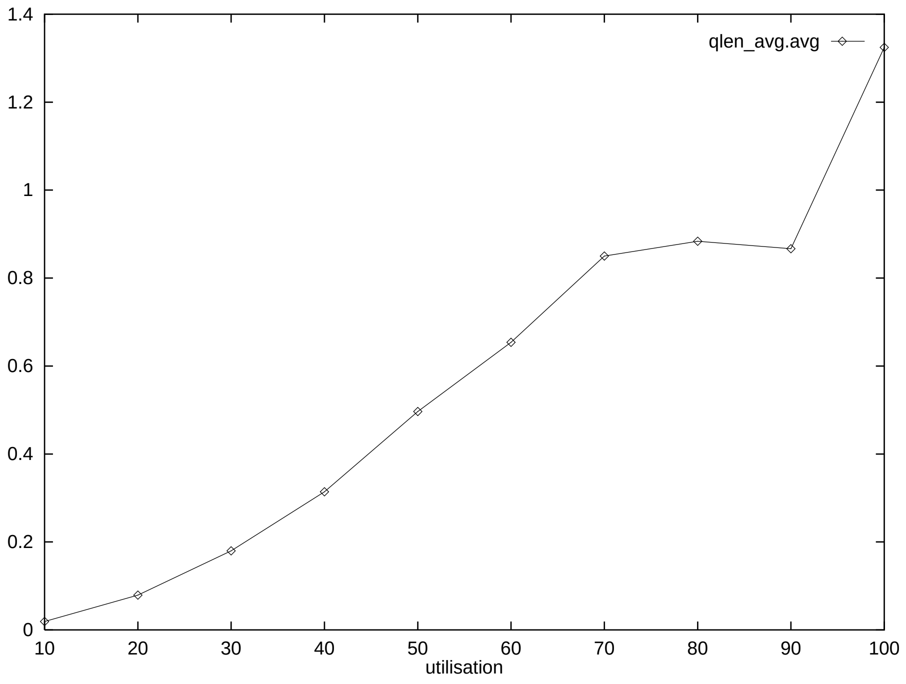
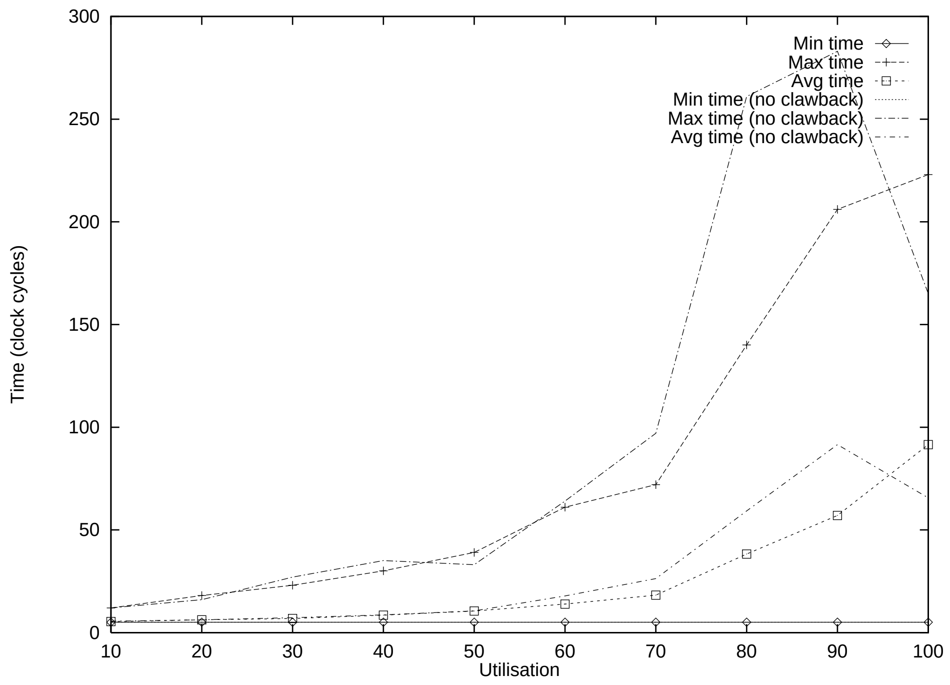


Figure 2.3: Output queue lengths (Q1 and Q2) vs utilisation.



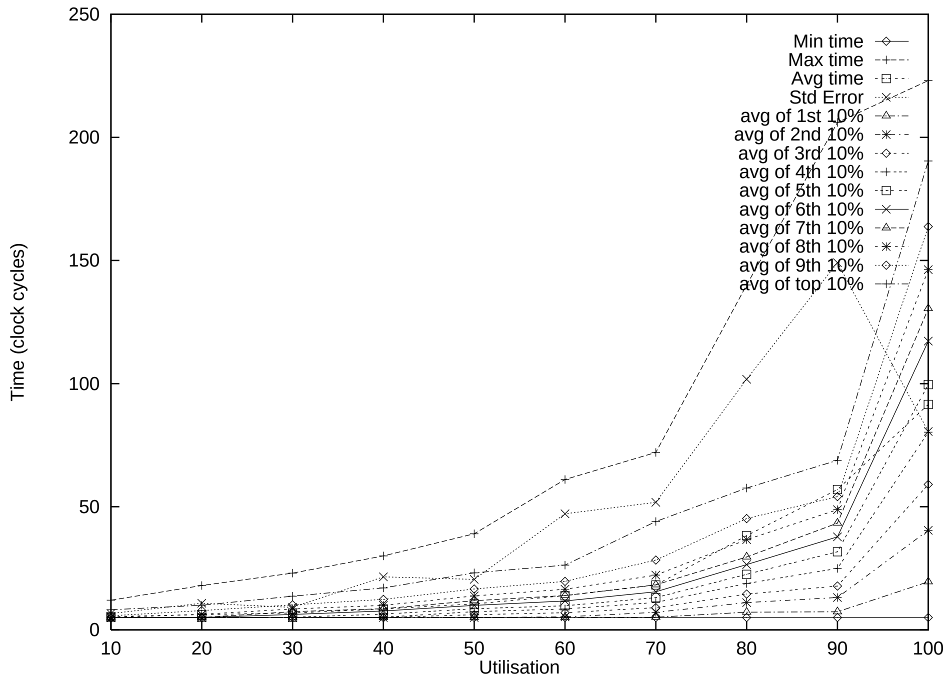


Figure 2.4: Packet delay distribution (with and without clawback) vs utilisation.

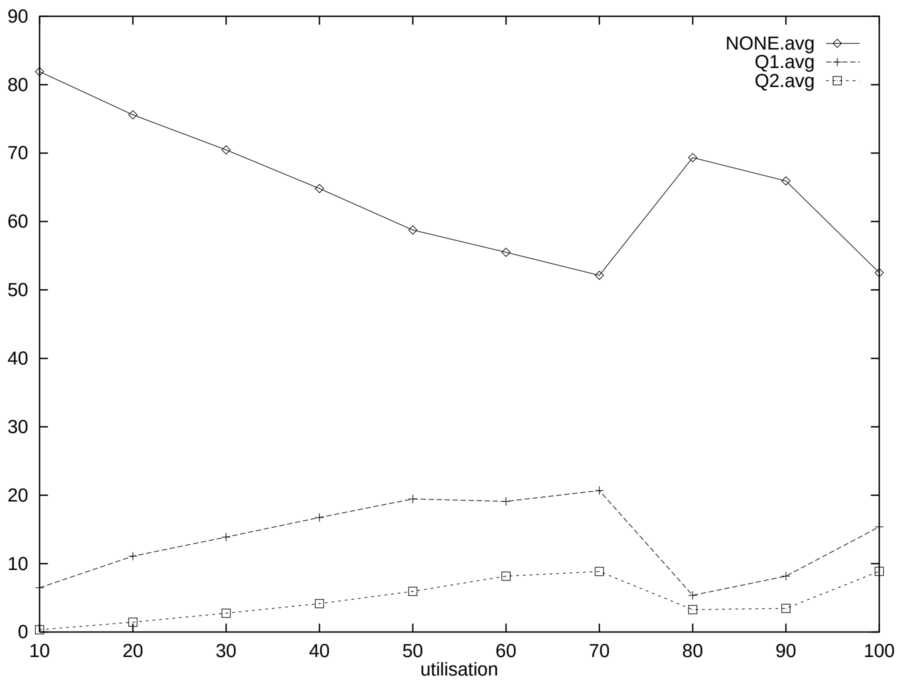


Figure 2.5: Output Arbitration Performance vs Utilisation

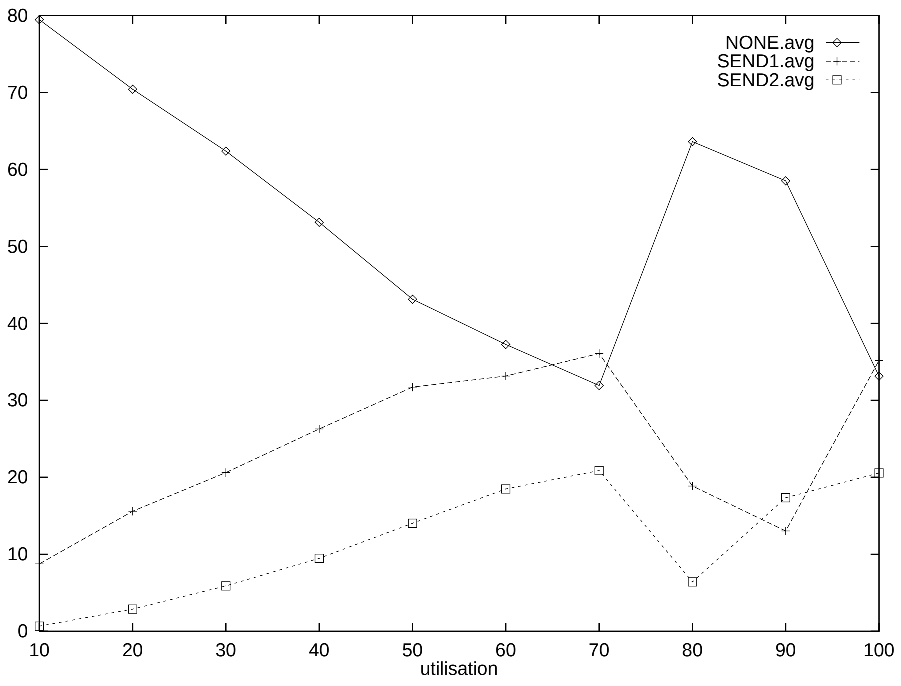
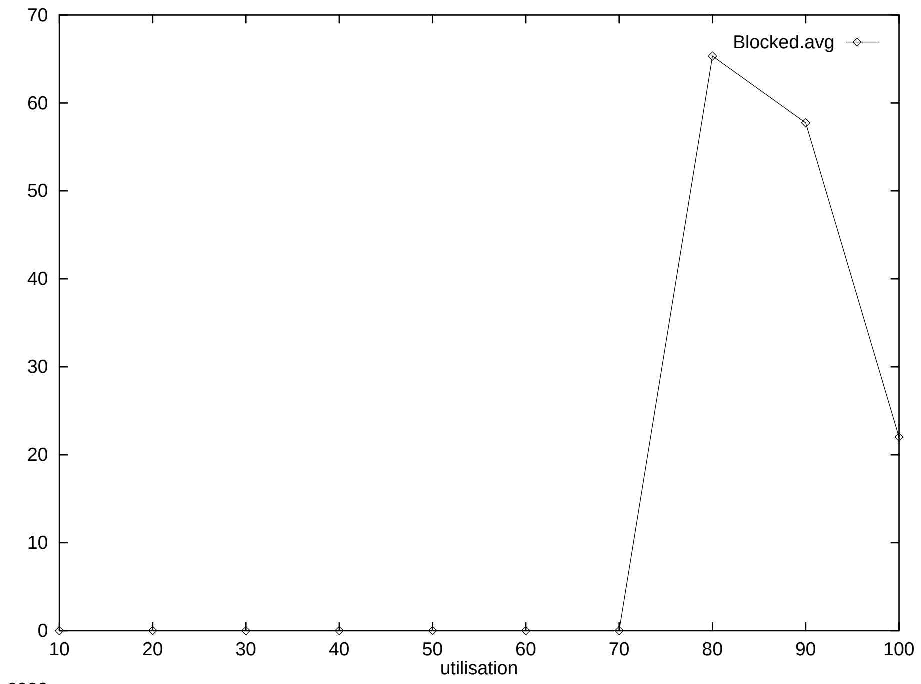


Figure 2.6: Output Multiplexor Performance vs Utilisation



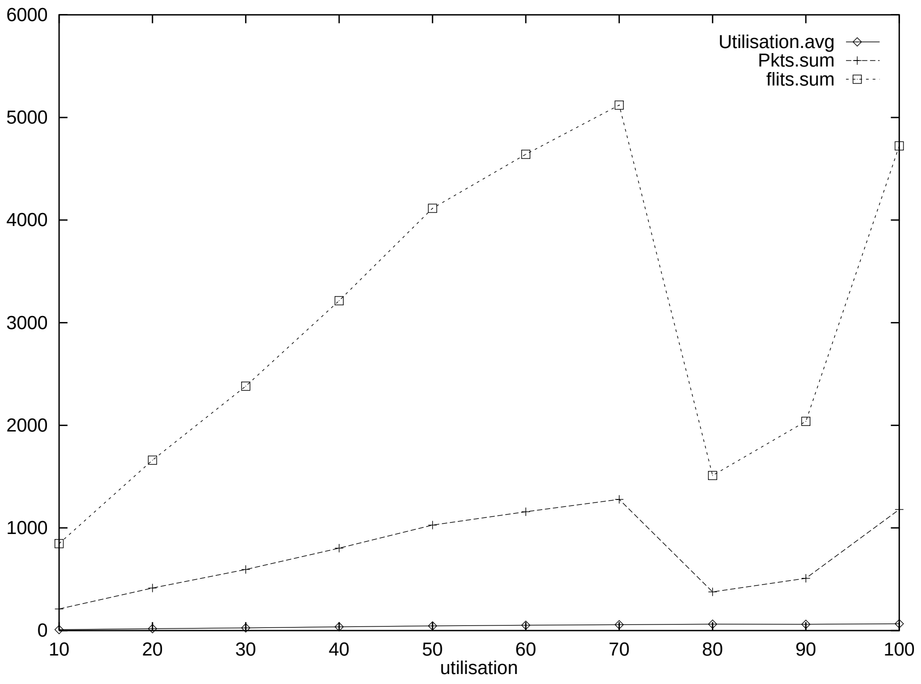


Figure 2.7: Source and sink performance vs utilisation

2.2 Multistage networks

The VLXbar was designed to be useful as a component of multistage switching networks, so the simulation model described in Section 2.1 was also used as a component to build such multistage networks. A framework was constructed which allowed any size of crossbar to be used as the switching element and the detailed multi-entity model described in the previous section was replaced by a single entity encapsulating all the behaviour.

2.2.1 Routing

The routing algorithm for the general N way omega network is given in C++ in figure 2.8. The basic idea is that the first flit arriving at the crossbar is the destination address which is stripped from the packet when it leaves. In a multistage network with three stages, the first flit of the packet determines the route through the first stage, the next determines the route through the second and the third flit determines the address through the third crossbar. When the packet emerges, it has lost its address flits and just contains data.

```
void mstage_src::construct_route(int dest, int *buf, int *i)
{
  /* Construct route from here to there */
  *i = 0;
  /* For each stage */
  /* dest = bit pattern */
  /* 00 11 01 10 */
  /* Shift dest by log2(swsize)
   * and with mask of swsize -1
   */

  int mask = swsize-1;
  int lgswsize = ilog2(swsize);
  for (int s=nstages-1; s>=0; s--) {
    buf[s] = dest & mask;
    dest >>= lgswsize;
  }
  (*i) = nstages;
}
```

Figure 2.8: Routing algorithm for N way omega network.

2.3 The MPI interface

As well as using random workloads, the low level simulation models were also interfaced to the MPI interface. The MPI interface is a standard for writing message passing programs on parallel machines. Providing an MPI interface on top of a simulation model of a parallel network effectively creates a working implementation of MPI on the (simulated) platform.

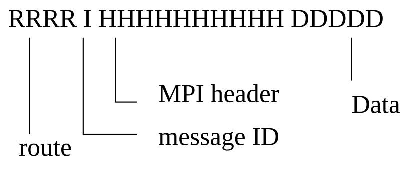
This is not a trivial task. The MPI interface has a large number of functions ranging from basic message passing to collective reductions, gathering and scattering of complex data sets, and a wide variety of synchronisation options. MPI source code for an implementation based on LANs was available, however, and this was used as the basis for the simulation interface.

The layers from the MPI program down to the low level network simulation model are as follows:-

1. Code in the workload generator "**src**" calls **MPI_Send()** (for example).
2. **MPI_Send()** (a method of the **src** class) checks the MPI level group information encoded in the parameters, then calls a lower level routine **send_mpi_packet**.
3. **send_mpi_packet** operates as in figure 2.9.

1. Break MPI messages into a number of fixed-size network packets

2. Insert route into the header of each packet:



3. Send all network packets, calling send_raw_pkt

Figure 2.9: Sending MPI messages through the network

4. **send_raw_packet** breaks the network packet into its flits and performs flow control to send it to the input of the network simulation component.

2.3.1 Validation

To validate the MPI interface implementation, a number of basic test programs were run (point-to-point, scatter, gather, barrier), and the MPI level output was checked for correctness.

2.3.2 Performance

A clock cycle simulation of a parallel machine was developed, running actual application code: each MPI call on each simulated processor caused many simulated packets to be generated; each simulated packet was made up of a number of flits; each flit caused a number of events to occur as it moved between the different input and output queues of each simulated switch it went through. As might therefore be expected, this resulted in a very slow implementation of MPI, and so in practice it was only realistic to run short test sequences. On a 64-processor model with crossbar, a simulation of MPI level all to all communication took approximately 20 minutes on a workstation.

2.4 A real system routing network: the T3D

A model of the T3D interconnection network model was constructed at a similar level of detail to that built for the VLXbar. This was to see how the torus interconnect would compare with a crossbar or multistage network. The T3D was an ideal choice as it was (at the time) a state of the art network and a T3D system was available in EPCC. Detailed hardware information was provided by Cray staff working at EPCC.

2.4.1 Low level behaviour of the T3D routing network

There are 6 different sizes of packet which can be sent through the Cray network, ranging from 3 flits up to 26. The types are given in Table 2.1. The *transactions* use the packet types for requests and responses as shown in Table 2.2.

Type	Size	Contents
0	3	route,dest,cmd
1	6	route,dest,cmd,ad1,ad2,src
2	8	route,dest,cmd,ad1,ad2,src,rad1,rad2
3	8	route,dest,cmd,d1,d2,d3,d4,E5
4	11	route,dest,cmd,ad1,ad2,src,d1,d2,d3,d4
5	23	route,dest,cmd,d1..16
6	26	route,dest,cmd,ad1,ad2,src,d1..16

Table 2.1: Packet types on the Cray T3D network

PE is the processing element, the BLT is a DMA device which has access to the network, pfetch stands for prefetch, f+inc and swap are used for synchronisation.

The 'node' of the torus contains two CPUs, a network interface and a router with 6 bidirectional connections (left, right, up, down, in, out) as well as connections to and from the network interface. The router is actually made up of three switches:

Transaction	Request	Response
PE noncache rd	1	3
PE wr 1word	4	0
PE cache rd	1	5
PE wr 4word	6	0
BLT rd 1word	2	4
BLT wr 1word	4	0
BLT blk rd	2	6
BLT blk wr	6	0
pfetch rd	1	3
f+inc rd	1	3
f+inc wr	4	0
swap	4	3

Table 2.2: Transaction types on the Cray T3D network

the X switch, the Y switch and the Z switch. Each switch routes both ways in its own dimension, and also has input and output connections for messages which are turning a corner.

The HASE++ simulation code for building a node is:-

```
proc *pe0 = new proc(pe0name, new sim_port(nifname),
                     x,y,z,0,SRC_OK);
proc *pe1 = new proc(pe1name, new sim_port(nifname),
                     x,y,z,1,SRC_OK);
nif *n = new nif(nifname,
                 new sim_port(pe0name), new sim_port(pe1name),
                 new sim_port(btename), new sim_port(netoname),
                 new sim_port(netiname),x,y,z);

router *r = new router(new sim_port(nifname), new sim_port(nifname),
                       xp_i, xm_i,
                       yp_i, ym_i,
                       zp_i, zm_i,
                       x,y,z);
```

The router is made up of three switches:-

```
cswitch *xsw = new cswitch(xname,pein_i,xp_i,xm_i,
                           new sim_port(yname),
                           x,y,z,0);
cswitch *ysw = new cswitch(yname,new sim_port(xname),yp_i,ym_i,
                           new sim_port(zname),
                           x,y,z,1);
```

```
cswitch *zsw = new cswitch(zname,new sim_port(yname),zp_i,zm_i,
                           peout_i,
                           x,y,z,2);
```

2.4.2 Deadlocks and virtual channels

Deadlocks can occur if there is a cyclical dependency (such as can happen in a torus). In order to break the cycle, *virtual channels* are implemented in the T3D routing strategy. Although there is only one physical connection between two switches, there are four virtual channels. Requests and responses use different virtual channels (so that they cannot block each other); there is also the idea of an "international date line"; messages must switch channels when they cross this point in the torus, breaking the cycle.

This was implemented in the model of the switch using an array of buffers:-

```
// Virtual channels - each is an array of 8 flits.
vc *vchan[3][4];
```

Arbitration must be done for the virtual channels as well as the physical connections. To give an idea of the complexity of the switch, the class definition file is given here:-

```
class cswitch : public hsim_entity {
protected:
  sim_port pdim, mdim, pein, swout;

  // Routing
  int x,y,z;
  int dim;              // 0=xsw, 1=ysw, 2=zsw
  virtual int route(int dest);  // Return op no.
  int route_dim(int i, int di, int dsz);  // Return M_OUT/P_POUT

  // Internal blocking
  int op_blocked[3][4];
  int ip_blocked[3][4];
  int ip_dest[3][4];    // Dest op for given iq
  int op_ipno[3];       // ip no for given op
  int op_ipvc[3];       // ip vc no for given op
  void ok(int i, int v); // Send OK to ip/vch

  // Virtual channels - each is an array of 8 flits.
  vc *vchan[3][4];
  void break_link(int input_no, int vc_no, int op);
  void make_link(int input_no, int vc_no, int op);
  // Conflict resolution for input
  int ip_pri;
  int vc_pri;
  int ip_dist( int ip );        // distance of ip from pri
  int vc_dist( int vc );
  void rotate_priorities();
  void resolve_conflict(int&i1,int&vc1,int i2, int vc2);

  sim_port* get_op_port(int i);
  sim_port* get_ip_port(int i);
  int get_op_no( sim_event &ev );
  int get_ip_no( sim_event &ev );

  void send_qe(queue_elem &qe, int dest);

  void wait_for_msg();
  void swallow_clocks();
  void make_clock();
  void recv_input_msgs();
  void recv_oks();
  void forward_flits();
  void arbitrate();

  void handle_msgs();

  void dump_state();

public:
  cswitch(char*n,
         sim_port *pein_i, sim_port *pdim_i, sim_port *mdim_i,
         sim_port *swout_i,
         int x_i, int y_i, int z_i, int dim_i );
  void body();
}
```

2.4.3 Results

With tracing switched off, the T3D model ran at 12500 events per second (using HASE++ on a Sun SPARC-20). For a 32 processor simulation running MPI code, this worked out at just under 1 second real time per simulated clock cycle. Thus it proved infeasible to run anything other than small test programs to verify the model, and so no meaningful performance figures could be obtained by this technique.

2.5 Summary

This phase of the project produced:

- A detailed simulation model of a packet switched crossbar design.
- A higher level model for use as a component in multistage switching networks.
- A detailed working model of the Cray T3D network capable of running MPI code.

MPI software was successfully interfaced with cycle level simulation models, but the simulation performance was such that it was unrealistic to simulate programs of more than a few MPI calls - and totally unrealistic to run large applications on the simulation models. Porting the simulation environment to the Cray T3D did not help particularly; the uniprocessors of the T3D are no faster than workstations, the T3D memory system is not optimised for DE simulation programs (it has a small, single level cache and no secondary cache; it was necessary to write a threading library as none was provided by Cray Research), and developing a parallel simulation engine was infeasible because of the amount of time this would have involved.

Building the models provided useful insight into the low level behaviours of these networks, and led to a new approach described in the next chapter of this report.

Chapter 3

Microbenchmarking

The idea of the 'benchmarking' approach is to take a standard application and measure its performance on a range of machines [7]. The value of this is that it provides a fixed point for comparing performance. The problem with it (from the point of view of system designers or people wanting to do performance prediction) is that the performance of a system running one application gives no clues as to the underlying *reasons* for the given performance; for this more detailed measurements are needed.

Microbenchmarking provides these measurements; rather than give one number for the performance, the time taken by each of the primitive operations making up the performance is measured. This is a complementary process to standard benchmarking, but provides more detailed performance information.

In this application 'microbenchmarks' can be seen as the performance characterisation of the interface layer between hardware and software of multiprocessor networks. Routines are described for *measuring* shared memory and message passing performance, as well as *calculating* 'microbenchmarks' analytically. The use of microbenchmarks in *simulation* is also described, both as the output of low level hardware models and as the input to high level software models.

The 'primitive operations' of a parallel machine depend on the programming model; the two most important models are message passing and shared memory.

3.1 Message passing

A message passing interface consists of a set of communications functions such as 'send', 'receive', 'broadcast'. The microbenchmark characterisation chosen consists of a table giving the performance of each communication function. The web page [8] contains code and examples. The table below shows the microbenchmark obtained by running a characterisation routine across a network of workstations.

These routines were originally developed for software performance prediction; more details are available in [8], [9] and [10].

MPI Function	Time (μs)			Goodness of fit (Q)
send	$5000 +$	$1 \times ndata$		0.95
ssend	$10000 +$	$2 \times ndata$		1.00
rsend	$5000 +$	$0.9 \times ndata$		0.94
recv	$8000 +$	$3 \times ndata$		1.00
recvmin	$3000 +$	$0.7 \times ndata$		0.23
irecv1	$70 +$	$0.001 \times ndata$		1.00
irecv2	$8000 +$	$2 \times ndata$		0.97
irecvoverlap	$5000 +$	$0.9 \times ndata$		0.98
sendrecv	$10000 +$	$3 \times ndata$		0.91
pingpong	$10000 +$	$4 \times ndata$		0.98
alltoall	$0 +$	$4000 \times p^2 +$	$2 \times p^2 \times ndata$	1.00
gather	$0 +$	$20000 \times log(p) +$	$2 \times p \times ndata$	1.00
allgather	$0 +$	$8000 \times p +$	$2 \times p^2 \times ndata$	1.00
reduce	$0 +$	$20000 \times log(p) +$	$2 \times p \times ndata$	1.00
allreduce	$0 +$	$8000 \times p +$	$2 \times p \times ndata$	1
bcast	$1000 +$	$600 \times p^2 +$	$0.9 \times p \times ndata$	1.00

Table 3.1: Microbenchmark characterisation of MPI performance on a network of workstations.

3.2 Shared memory

Microbenchmark measuring routines were also developed for shared memory architectures, including the Solaris and POSIX threads library (for multiprocessor workstations), the Cray SHMEM library, and the Sequent shared memory model. The characterisation takes the form of measuring the performance of synchronisation operations (such as barriers, semaphores, process creation and joining), as well as the thornier issue of measuring the performance of the shared memory system.

Characterising the performance of the memory system is difficult because of the number of levels in the hierarchy, and because of the possibility of both network and memory bank interference between different memory accesses. The following sections describe the techniques used for measurements on the Sequent Symmetry programming model, POSIX and Solaris threads on multiprocessor workstations, the Cray T3D shared memory operations and NT threads.

3.2.1 Routines common to all

To measure memory access times for local and remote data, it is essential to know whether the data is stored in the local cache or in main memory. This requires devious programming. The basis of the measurements is a data buffer, `int buf [csize];`: with size `csize` greater than the cache size.

A single pass through this array, reading every element, will flush out the old contents of the cache. If the array is at least twice the size of the cache, subsequent passes through the array will suffer continuous cache misses.

In addition to the cache size, the block size is also a parameter. The worst case involves stepping through `buf` with a step size equal to the block size.

It is possible to determine the cache size and the block size experimentally, but this is not easy, since the times involved are so short, and there is the danger of process switching interfering with the timings. So these figures are left as parameters.

It is most attractive to make the measurements using a compiled language like C++, but the measurements are at the mercy of the code generator, so it is essential to check that the generated assembler contains only the required memory references.

The loop overheads interfere with the timings. This may be measured using a "control" empty loop. The impact may also be reduced by "unrolling" the loop to contain a sequence of memory operations.

The actual function used to measure the time introduces another variable. The timer resolution is typically $1\mu s$ or the clock rate of the processor, and the overhead of calling the timer may be severe.

The loop to time reading `.nwords.` words from main memory (unrolled eight times) is:-

```
double e1 = MPI_Wtime();
for (int j=0; j<nwords; j+=8, index += 8*blk_size) {
  accum += buf[index];
  accum += buf[index+blk_size*2];
  accum += buf[index+blk_size*3];
  accum += buf[index+blk_size*4];
  accum += buf[index+blk_size*5];
  accum += buf[index+blk_size*6];
  accum += buf[index+blk_size*7];
}
time = MPI_Wtime() - e1;
```

The time returned will be composed of:

```
{time to read nwords} + {loop overhead} + {timer overhead}
+ {indexing and adding overhead} + e
```

where `.e.` is a random error caused by context switching/virtual memory paging.

In order to extract the required *time to read nwords*, it is either necessary to subtract the overheads, or to ensure that they form an insignificant portion of the time. The *indexing and adding overhead* is small compared to the memory access time, and is required in most programs.

The above loop must be repeated as shown below for long enough to minimise the one-off timer overhead (and to compensate for the timer's lack of resolution). This must be done with care to avoid spurious cache hits.

```
  double e1 = MPI_Wtime();
  for (int i=0; i<iters; i++) {
    for (int j=0; j<nwords; j+=8, index += 8*blk_size) {
      accum += buf[index];
      accum += buf[index+blk_size*2];
      accum += buf[index+blk_size*3];
      accum += buf[index+blk_size*4];
      accum += buf[index+blk_size*5];
      accum += buf[index+blk_size*6];
      accum += buf[index+blk_size*7];
    }
  }
  time = MPI_Wtime() - e1;
  time /= iters;
```

It would be possible to flush the cache after each loop (by cycling through another cache-sized array). However this would take a while (seconds), and it is more appealing to just use separate areas of the cache.

3.2.2 Posix threads

By default, global variables are shared between threads. Therefore, to arrange for threads to work independently on their own sets of data, it is necessary to allocate some data for each thread:-

```
struct local_data {
  int index;
  int buf[csize];
};
```

A thread may access another thread's local data using an array of pointers:-

```
local_data ** remote_data;
```

Each thread then runs its own independent function:-

```
void* thr_main(void *r)
{
  local_data *l = (local_data*)r;
  int rank   = (int)l->index;
  sem_wait(&set_going);
  barrier();
  time_all(rank);
  barrier();
  return NULL;
}
```

Synchronisation may be performed using POSIX and pthread semaphores and condition variables:-

```
pthread_mutex_init(&mtx, NULL);
pthread_mutex_lock(&mtx);
pthread_mutex_unlock(&mtx);
pthread_cond_init(&cond, NULL);
pthread_cond_wait(&cond, &mtx);
pthread_cond_broadcast(&cond);
sem_init(&set_going, 0, 0);
sem_post(&set_going);
sem_wait(&set_going);
pthread_create(&threads[i], NULL, thr_main, (void*)1);
pthread_join(threads[i], (void**)&status);
thr_setconcurrency(nprocs);
```

3.2.3 Solaris Threads

Solaris threads are largely identical to Posix threads. The main difference concerns the separation of lightweight processes (LWPs - operating system constructs, and the unit of concurrency) and threads (user level constructs). Many threads may share a LWP; or there may be one thread per LWP. The advantage of this two layer model is that user level context switching is faster than operating system level switching. The disadvantage is that it adds complexity.

3.2.4 Windows NT threads

Windows NT threads are (relatively) heavyweight operating system level objects. The startup costs are greater than Posix threads.

3.2.5 The Cray T3D

Shared memory on the Cray T3D differs from threads. Each processor runs a separate copy of the whole program, so N copies of main (), shown in the example below, are executed. Global variables are not shared; to access another processor's variables, it is necessary to use shmem_get and shmem_put. These can only be used on *symmetric* data objects, i.e. C and C++ data allocated by shmalloc. Mutual exclusion locks are simply symmetric variables of type `long` which are initialised to zero. A barrier operation is built in.

```
int buffer[csize];
int lbuf[csize];
```

```
main()
{
  nprocs = _num_pes();
  rank   = _my_pe();
  fill_buffer();
  if (rank==0) printf("Shared memory microbenchmarker (%d procs)\n",
                      nprocs);
  barrier();
  time_all();
  barrier();
  if (rank==0) printf("Completed\n");
}

void time_read_remote(int nwords, int iter, double &time)
{
  double e1 = MPI_Wtime();
  for (int i=0; i<iters; i++) {
    shmem_get(lbuf,buffer+i,nwords,1);
  }
  time = MPI_Wtime() - e1;
}

extern "C" void barrier();
nprocs = _num_pes();
rank   = _my_pe();
shmem_get(lbuf,buffer+i,nwords,1);
shmem_put(lbuf,buffer+i,nwords,1);
shmem_clear_lock(long *lock);
shmem_set_lock(long *lock);
shmem_test_lock(long *lock);
shmem_wait( long *ivar, long cmp_value );
```

3.2.6 Sequent Symmetry

The parallel programming model of the Sequent Symmetry contains elements of both the Cray model and threads. Only one instance of `main` is launched; an `m_fork(func);` is needed to start up the processes. By default, global data is *not* shared. Shared data must be enclosed within `pragma sequent_shared` and `pragma sequent_shared_end`, as shown in the example below. Synchronisation is performed using built in barriers and lock variables. Remote reads may be done by accessing the `shmalloced` data of another process.

```
#include <parallel/microtask.h>
#include <parallel/parallel.h>
```

```
#pragma sequent_shared
int nprocs;
#define CSIZE (1 << 16)
const int blk_size = 8;          /* Cache block size (# of ints) */
const int maxword = 1024;
int iters;
int smallsize,smalliters;
int medsize,mediters;
int largesize,largeiters;
#pragma sequent_shared_end

#pragma sequent_shared
local_data_s **remote_data;
#pragma sequent_shared_end

m_set_procs(nprocs);
remote_data = (local_data_s**)shmalloc(sizeof(plocal_data) *nprocs);
s_init_barrier(&bp, nprocs);
m_set_procs(nprocs);
m_fork(func);
m_kill_procs();
s_lock();
s_unlock();
```

3.3 Measurement examples

Examples of results obtained by running the shared memory microbenchmarker for two threads on a workstation are given in Figure 3.1.

The figures given are average times in nanoseconds to perform the given tasks. Barrier, timer overhead and loop overhead times are given first. The figures afterwards are for reads and writes to local main memory, local cache, and the neighbour process's memory. The `allread` and `allwrite` measurements are taken when all processes are performing the operation concurrently; with the other measurements, only one process is active.

This set of measurements is just one of many possible such sets of measurements which could have been carried out. On this machine (a 2-processor workstation with shared bus) it makes no difference whether a processor reads its own or its neighbour's section of shared memory and local cache reads are about 7 times slower than local cache writes. These factors may be interesting for the program designer, but extracting the influence that the *interconnect* (in this case the Sparc MBus) has on these figures is not easy. A clue may come in the difference between the `write` and the `allwrite` figures - if it takes longer for all processes to write at the

```
   Microbenchmark    :      Time (ns)
                     :   avg    ( max  )
----------------------------------------
         empty loop :        6 (     6)
            timer() :        4 (     4)
            barrier :      263 (   264)
    read_localcache :       69 (    69)
 allread_localcache :       69 (    74)
         read_local :      328 (   328)
      allread_local :      350 (   362)
     read_neighbour :      350 (   350)
  allread_neighbour :      359 (   362)
   write_localcache :       11 (    11)
allwrite_localcache :       11 (    11)
        write_local :      184 (   184)
     allwrite_local :      235 (   235)
    write_neighbour :      229 (   229)
 allwrite_neighbour :      218 (   218)
```

Figure 3.1: Shared memory microbenchmarks for measuring times to local and remote memory.

same time, one could conclude that contention was the cause. But this could as well be memory bank contention as network contention, and there is no way to find out which is which without opening the computer box and placing logic analyser probes on the DRAM chips (or similarly instrumenting simulation code). Of course, the application programmer doesn't particularly care where contention occurs, only about the possible degraded performance. Some of the measurement problems are described below.

3.3.1 Influence of the cache

Since the caches play such an important part in the memory hierarchy performance, ensuring consistency of their effects is vital. The secondary cache size and block size are inputs to the characterisation routine. To ensure a steady state before each measurement, the cache is flushed with a routine:-

```
void flush_cache(int rank) {
  int accum;
  for (int i=0; i<csize; i+=blk_size) {
    accum += remote_data[rank]->buf1[i];
  }
}
```

This steps through the entire cache reading one word per cache line, to ensure that the cache starts in a known state.

3.3.2 Inaccuracy of the timer and loop overheads

The time spent in the timer and on loop overheads is measured using an empty loop. Possible lack of resolution of the timer is dealt with by repeating each measurement a large number of times. The high resolution timers typically give time in nanoseconds to the nearest 500ns, so for nanosecond resolution measurements are repeated at least 1000 times (and typically 1,000,000 times).

3.3.3 Code generation

The measurement code is written in C++; this means that there is a possibility that the compiler will generate more memory references than would be expected for a given loop. Rather than code the timing loops in machine code (which would guarantee the number of memory references, but would require different code for each architecture), the compiler was set to generate assembler code and this was checked manually. It could be argued that people never write in machine code anyway, so this step was unnecessary - what matters is machine performance given normal code.

3.3.4 Memory bank contention

This is difficult to quantify, as it depends on the precise memory subsystem of the machine. Because of this, it is left unquantified and assumed to be one of the reasons different memory access patterns will perform differently.

3.4 Presentation of results

One useful way to characterise the performance of operations is in the form of a logarithmic table (Figure 3.2).

```
Ten to power minus:
         9
         8
         7         Cache read hit   Wr main (if buffered OK)
         6         Rd Main memory
         5         daxpy100, fprintf
         4         daxpy1000
         3         thr_create       daxpy10000
         2         fopen, fclose
         1
         0
```

Figure 3.2: Simplified representations of times on a logarithmic scale

Routines were written to carry out these measurements for some common operations (using files, reading and writing memory, creating threads). The example in figure 3.3 shows results obtained for a measurement on a Sun SPARCstation-20.

This format of results presentation is more useful for rough performance estimation than the ‘exact’ timing of operations in nanoseconds - but less useful for comparing detailed contention issues.

3.5 Summary

This chapter has presented methods for measuring and estimating the microbenchmark performance of parallel machines, for both shared memory and message passing programming models. The memory subsystem is the most difficult part to characterise meaningfully; some of the reasons for this were discussed, and some attempts at solution were presented. Detailed notes and download details for the software are available on the shared memory microbenchmarker home page [11].

```
     Operation       : Order of magnitude (log10(time in secs))
-----------------------------------------
              fopen : -1
    fprintf (1 char) : -4
     printf (string) : -3
             fclose : -2
    rd main mem 8000 : -6
   rd cache mem 8000 : -7
    wr main mem 8000 : -6
   wr cache mem 8000 : -7
         thr_create : -3
           daxpy100 : -4
          daxpy1000 : -3
         daxpy10000 : -2
```

Figure 3.3: Basic thread microbenchmarks for a single processor machine.

Microbenchmarks can be used in conjunction with simulation models in several ways; as measurements from a network simulation model, feeding into a simulation of parallel software and as a simplified workload. The next chapter considers the last of these approaches, using a microbenchmark style of interface between network models and workloads.

Chapter 4

The Multiprocessor Simulation Testbed

This chapter describes the simulation testbed developed to allow plug and play between different simulation models and workloads. The idea was to provide a clean interface between the two, and to allow development of models and workloads to proceed independently.

The model and workload are separate programs, compiled into separate libraries in separate directories, and only linked together at the end to produce an executable simulation program. This rigid division of the two allows the mix and match process. For example, a bus model and a random workload could be combined using:

```
CC bus.a random.a
```

and the bus model with a memory bank saturating workload using:

```
CC bus.a memsat.a
```

The important features of the testbed are:

- Interface specification (between models and workloads).
- Model and workload parameterisation.
- Experimentation and graphing.

4.1 Interface specification

Allowing plug and play between different simulation models is the subject of a number of initiatives in simulation research, notably with the HLA (High Level Architecture) US DoD initiative. Similar concerns abound in the computer industry, from the Virtual Socket Interface used for mixing and matching cores onto a piece of silicon, to 'software component' technology like CORBA/IIOP and JavaBeans.

The crucial aspect is to get the interface right. Here a testbed interface for interconnection network simulations was developed. The *model* must provide N processors and N memories. The *workload* runs on each processor, and may also supply its own memory behaviour model. The interface defines how memory accesses, message passing and synchronisation take place.

The workload is called from the model, the interface being in the functions:-

```
void init_workload(p_fn &mf);
void do_workload( int argc, char **argv, int index, int nprocs );
```

init_workload must be called first by the model; it reads in the global parameters from the **input.params** file.

The workload is chosen using the **workload** parameter set in **input.params**, with options including a random workload, a thread workload, a Cray workload, a memory bank saturation test and a cache test.

The processor calls **do_workload** to perform the work. A workload-specific function to perform memory operations may be provided in **mf**, for example the Cray model includes a memory unit which responds to Cray specific memory operations.

The workload drives the model using the function:-

```
void do_cmd(int index, int cmd, int pno, int addr, int size,
            int *data);
```

This performs one of the commands given in Table 4.1.

Command	Explanation
SEND_C, RECV_C	Send and receive messages between processors.
READ_C, READ_R	Read memory Command, and Response.
WRITE_C	Write Command.
TST_SET_C	Read-modify-write Command.
PROC_C	Local processing Command.
USER_C	User defined Command.
USER_R	User defined Response.

Table 4.1: Interface commands

Note that messages to memories are Commands (such as read requests); messages in response to commands are Responses.

Memory units may be driven using:-

```
int  read_mem(int index, int a);
void write_mem(int index, int a, int val);
void do_mem_cmd(int index, int cmd, int pno, int addr, int size,
                int *data);
```

Workload specific timings may be obtained using:

```
double get_time(int index);
```

And results may be placed on the trace file using:-

```
void put_result(int index, char *string );
```

4.2 Models

Several network models were constructed for the testbed. The network can be selected from a bus, a crossbar and a multistage network. Cache models and memories are also included in the system. The basic parameters are shown in Table 4.2.

Parameter name	Description
interconnect	1=Bus, 2=Crossbar, 3=Multistage
usecache	include cache (yes/no)
nprocs	Number of processors
cmd_startup, cmd_perword	Startup and per word delay (ns) at processor
bus_startup, bus_perword	Startup and per word delay (ns) at bus
mem_startup, mem_perword	Startup and per word delay (ns) at memory
cache_size	size (in bytes) of cache
cache_access_time	Cache hit time (ns)
cache_tag_access_time	Cache tag memory access time (ns)
cache_assoc	Cache associativity
cache_blk_size	Cache block size
cache_tagmems	Number of tag memories

The simulation code accesses these parameters using code like:-

```
double cache_size        = get_int_param(0,"cache_size");
double cache_access_time = get_int_param(0,"cache_access_time");
```

4.2.1 The Bus

The bus entity acts as a passive router of commands, holding for a message dependent time, then passing the command on. Only one command may occupy the bus at a time. Requests are dealt with on a First Come First Served basis, but no actual arbitration or priority is implemented apart from this. Its delay parameters are *Bus hold time* and *Bus time per word.*

4.2.2 The Crossbar

The crossbar entity has similar basic delay parameters to the bus. It acts as a passive router of commands, waiting for each command to arrive and then passing it on (a message-dependent time) later. It never actually blocks itself.

4.2.3 The Multistage Network

The multistage network entity acts as an array of crossbars. Events from processors or memories are passed into the internal network of switches. When they emerge, they are sent on to the appropriate processor or memory.

4.2.4 The Processor Node

The processor node parameters are shown in Table 4.2.

Command hold time	Command time per word
Send hold time (H_{send})	Send time per word
Recv hold time (H_{recv})	Recv time per word
Read hold time (H_{read})	Read time per word
Write hold time (H_{write})	Write time per word
Process cycle time (H_{proc})	
User hold time (H_{user})	

Table 4.2: Processor node parameters

It always holds for the basic Command hold time, then the delay is computed depending on the message (Table 4.3).

SEND.	Holds H_{send}, then sends to network instantly.
RECV.	Holds H_{recv}, waits for posted send, returns instantly.
READ.	Holds H_{read}. Sends request. Waits for reply. Returns instantly.
WRITE.	Holds H_{write}, posts write, returns.
TEST_SET.	Holds H_{read}. Sends request. Waits for reply.
PROCESS.	Holds $H_{proc}\times$ number of words.
USER.	Holds H_{user}. Sends request. Waits for reply. Returns instantly.

Table 4.3: Processor node behaviour

User code is in:

```
void do_workload( argc, argv, index, nprocs );
```

This calls:

```
void do_cmd(int cmd, int pno, int addr, int size, int *data);
```

do_cmd holds for the command hold time, then behaves as described in Table 4.3. All hold delays are modelled as startup plus time per word. There is no extra delay on a read request returning.

4.2.5 The Memory

The memory entity holds for the Memory hold time, performs the read or write operation and returns instantly. For user commands (needed to implement more sophisticated memory operations than the basic read/write ones provided), it provides read and write methods which operate instantaneously. User delays may be inserted by calling the hold method.

4.2.6 The Cache

The parameters for the cache object are given in Table 4.4.

Access time.	
Tag access time.	
Number of tag memories.	
Write policy.	
size	number of words in cache.
blk_size	number of words in block.
up_width, down_width	width of up/down buses in words.
up_cycle_time, down_cycle_time	speed of up/down buses.

Table 4.4: Cache parameters

The possible operations on a cache access are given in Table 4.5.

RD_HIT.	Passes back contents.
WR_HIT.	Sets contents. If shared, invalidate others.
RD_MISS.	Read block.
WR_MISS.	Read block. Write contents.
INV_ADDR.	Invalidate block.

Table 4.5: Cache operation

The cache waits for requests from the processor (the "up" connection). It performs a tag lookup, on a tag hit it returns the data. If it is a miss, it sends the request down the memory hierarchy. The cache also receives invalidate requests from below.

4.3 Workloads

Workloads were developed to drive the models including a random workload, a multi-threading workload, a Cray workload, memory bank saturation workload and a cache testing workload. Workload-specific memory operations were incorporated into the Cray workload to model the Cray memory system behaviour.

4.3.1 A Random workload

Each processor sends N fixed length messages to other processors, randomly selected. Parameters include utilisation, and the length of messages.

4.3.2 A Cray workload

This workload simulates a mix of the standard Cray network packets.

The standard Cray T3D network operation **PE_noncache_rd** is used. The memory is specialised to deal with the different remote memory operations supported by the Cray, i.e.:

```
enum {
  PE_noncache_rd,  PE_noncache_rd_res,
  PE_wr_1word,     PE_wr_1word_res,
  PE_cache_rd,     PE_cache_rd_res,
  PE_wr_4word,     PE_wr_4word_res,
  BLT_rd_1word,    BLT_rd_1word_res,
  BLT_wr_1word,    BLT_wr_1word_res,
  BLT_blk_rd,      BLT_blk_rd_res,
  BLT_blk_wr,      BLT_blk_wr_res,
  pfetch_rd,       pfetch_rd_res,
  f_inc_rd,        f_inc_rd_res,
  f_inc_wr,        f_inc_wr_res,
  swap,            swap_res
};
```

4.3.3 Memory bank saturation

This workload generates continuous requests to a given number of memory banks. The behaviour is:-

```
for (i=0; i<nmsgs; i++) {
  pno    = index % memmodules;
  do_cmd(index, READ_C, pno, addr, size, data);
  do_cmd(index, PROC_C, 0, 0, 1, NULL);
}
```

4.3.4 Threaded workload

This workload performs a set of standard thread synchronisation operations using mutual exclusion (mutex). The mutex tests are performed with the code:-

```
    // Get lock
```

```
do {
  val = 1;
  do_cmd(index, TST_SET_C, mp, ma, 1, &val);
  tries ++;
} while (val==0);

// process
do_cmd(index, PROC_C, 0, 0, 5, (int*)"got lock");

// Release lock
do_cmd(index, WRITE_C, mp, ma, 1, &zero_val);
```

4.3.5 Cache test workload

This workload tests cache hit and miss operations. Parameters include number of messages and cache size. The workload fills the cache, performs *nmsgs* hits then *nmsgs* misses.

4.4 Experiments

A number of experiments were run comparing the performance of the workloads on different networks, investigating the effects of including caches, and investigating the scalability of the networks.

4.4.1 Effect of cache on network performance

The following graphs show the effects of including a cache in the system for the different networks.

- Bus Figures 4.1 and 4.2.
- Crossbar Figures 4.3 and 4.4.
- Multistage network Figures 4.5 and 4.6.
- Summary graph Figures 4.7 and 4.8.

4.4.2 Scalability measures

Figures 4.9 and 4.10 show the effects of varying the number of processors on memory utilisation and time to complete the workload.

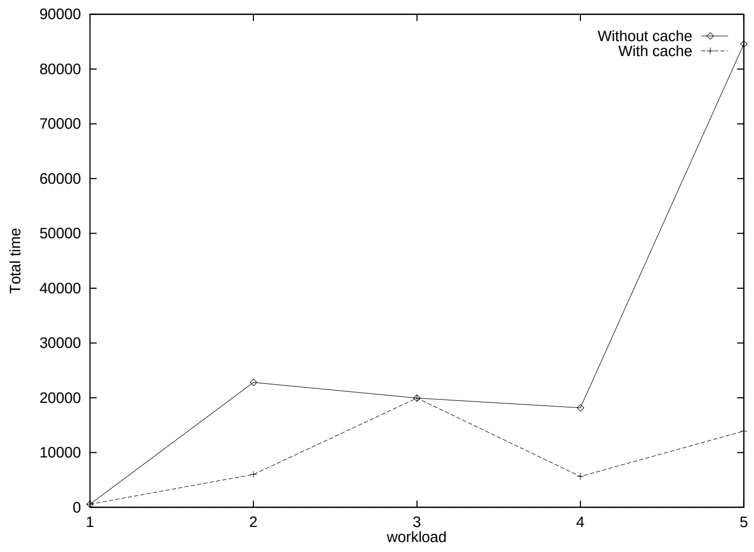


Figure 4.1: Total time for the workloads on a bus, with and without a cache.

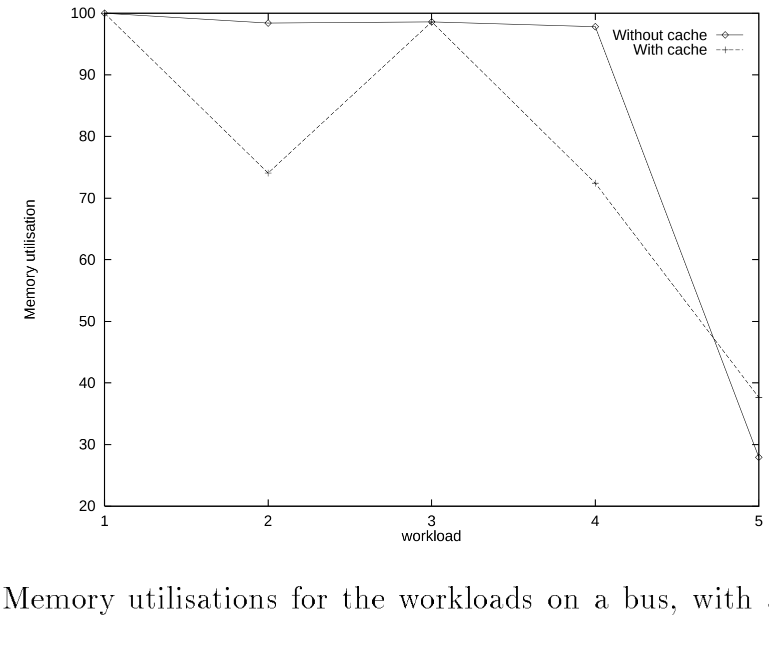


Figure 4.2: Memory utilisations for the workloads on a bus, with and without a cache.

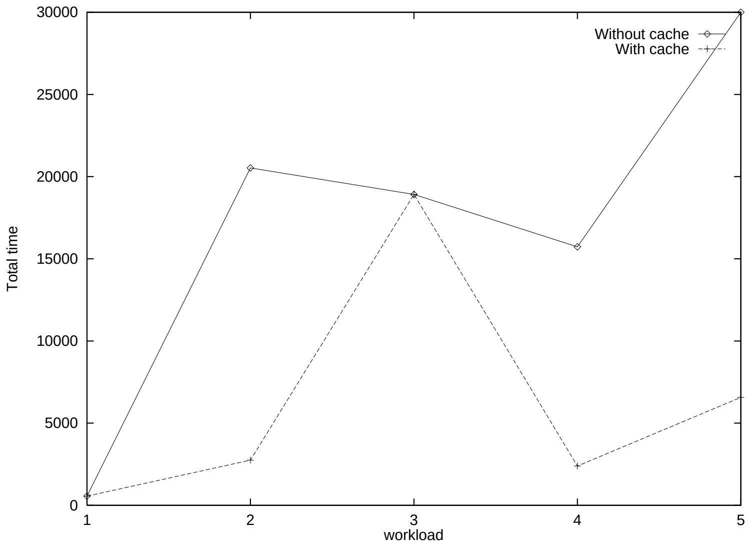


Figure 4.3: Total time for the workloads on a crossbar, with and without a cache.

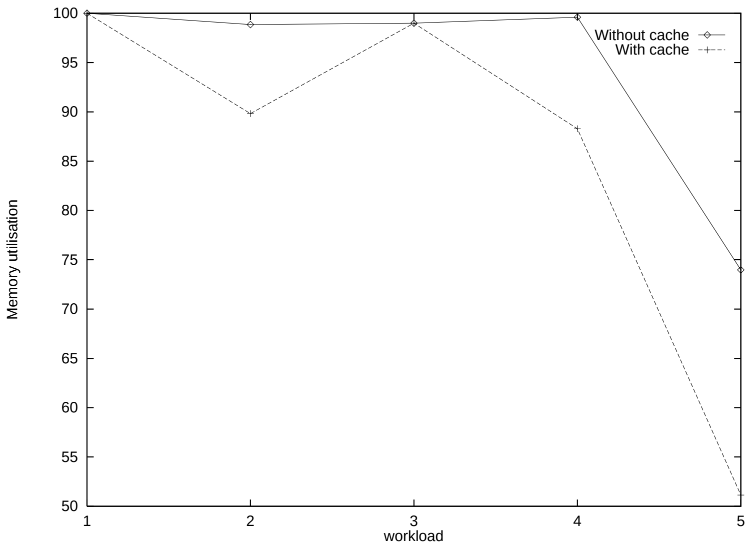


Figure 4.4: Memory utilisations for the workloads on a crossbar, with and without a cache.

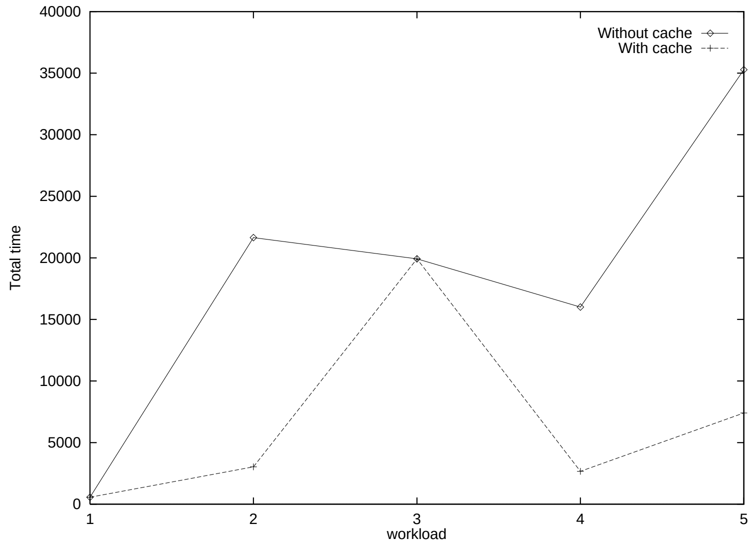


Figure 4.5: Total time for the workloads on a multistage network, with and without a cache.

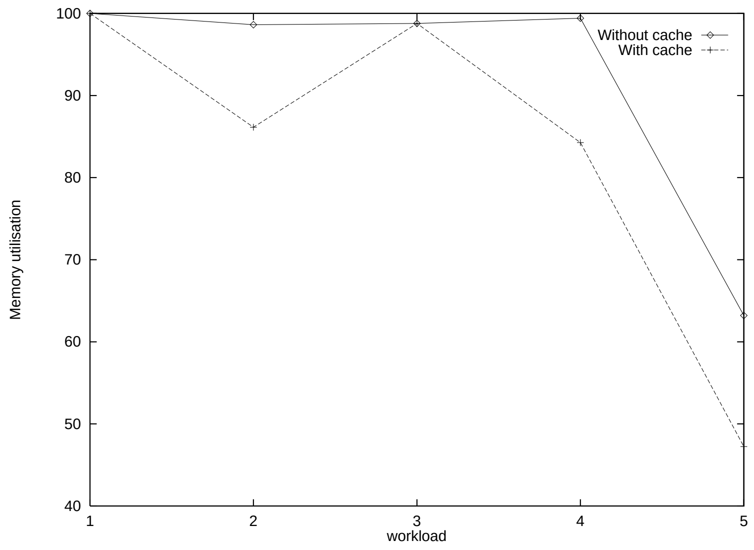


Figure 4.6: Memory utilisations for the workloads on a multistage network, with and without a cache.

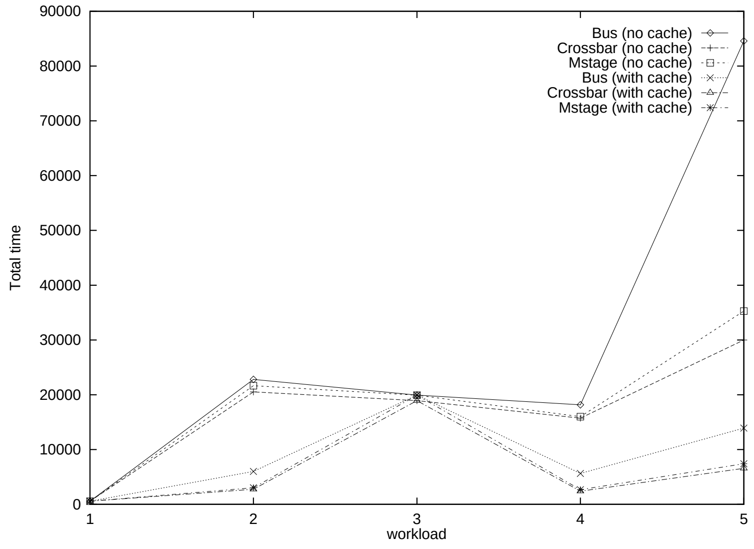


Figure 4.7: Total times for all workloads and networks, with and without cache.

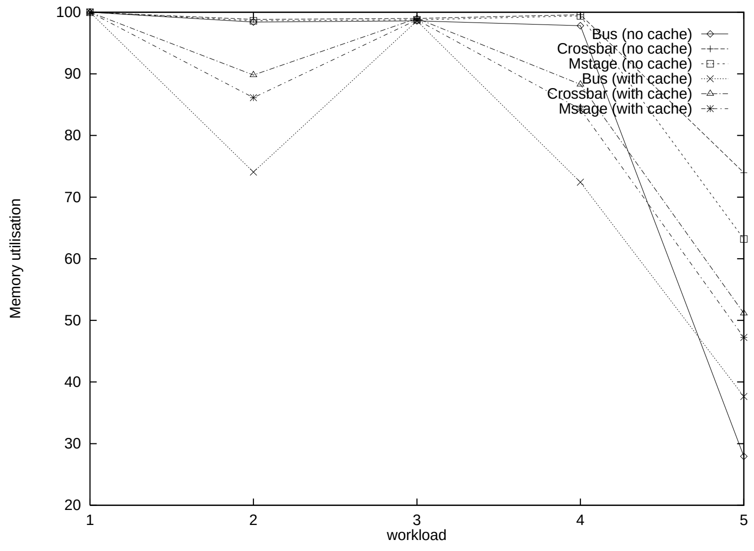


Figure 4.8: Memory utilisations for all workloads and networks, with and without cache.

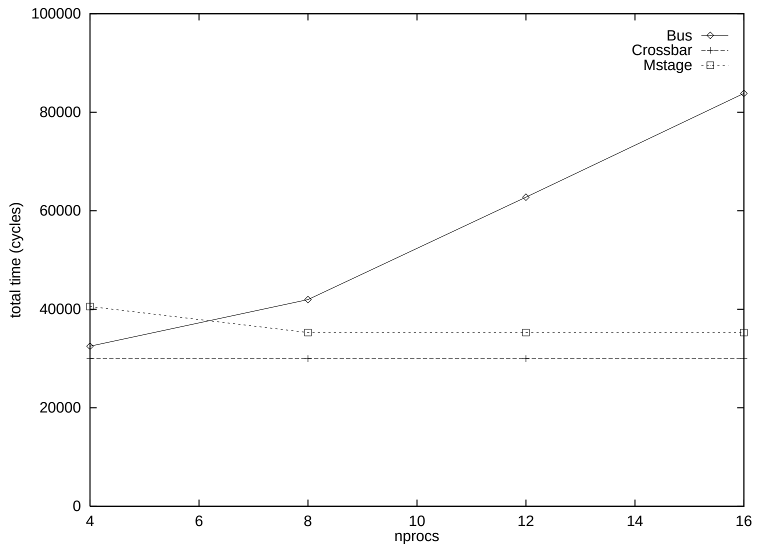


Figure 4.9: Total time for variable numbers of processors and the same workload. Note that the bus time grows linearly with number or processors, the crossbar time is constant, and the multistage network is slower for 4 processors than 8, 12 or 16. This apparently anomalous result is because of increased contention in the 4 processor case.

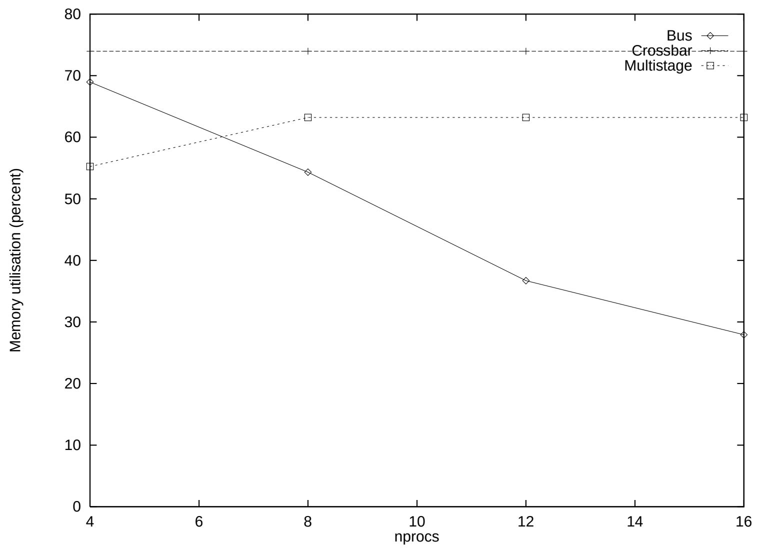


Figure 4.10: Average memory utilisation vs number of processors. Note the falloff for the bus, as the contention means that the memory units are not kept busy. Memory utilisation remains constant for the crossbar and multistage network (apart from the special case at 4 processors).

4.5 Timing diagrams.

For detailed analysis of the underlying behaviour of each simulation run, timing diagrams can be produced and inspected (Figure 4.11).

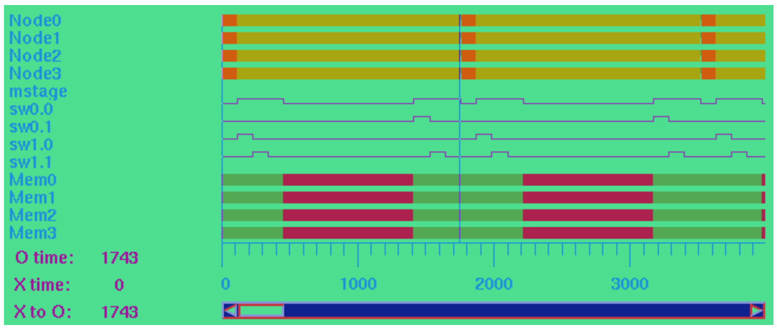
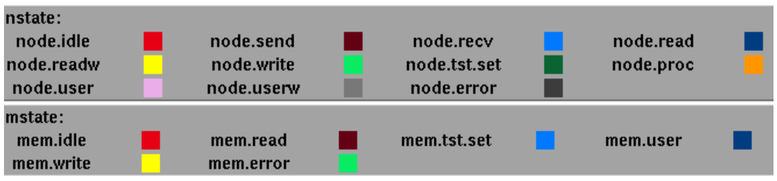


Figure 4.11: A timing diagram showing the detailed behaviour of a single simulation run. This workload is a sequence of accesses to memory in parallel; at the top are four processor nodes; they issue memory read requests through the multistage network (the timelines for the four switches sw0.0, sw0.1, sw1.0, sw1.1 which make up mstage.) to the memory modules at the bottom.



nstate:							
node.idle	🟥	node.send	🟥	node.recv	🟦	node.read	🟦
node.readw	🟨	node.write	🟩	node.tst.set	🟩	node.proc	🟧
node.user	🟪	node.userw	⬜	node.error	⬛		
mstate:							
mem.idle	🟥	mem.read	🟥	mem.tst.set	🟦	mem.user	🟦
mem.write	🟨	mem.error	🟩				

Figure 4.12: Key for timing diagrams.

The timing diagram (Figure 4.13) shows the top level behaviour of a 4 processor multistage network system running workload 5. Figure 4.14 shows the equivalent diagram for 8 processors.

The next two diagrams (figures 4.15, 4.16) show the detail of the first few transactions. Note that the transaction time for the 8 processor case is shorter than for the four processor case as the traffic is split between two switches.

4.6 Summary

This chapter has described the interconnection network testbed in detail. The crucial aspect was the interface specification between workload and model. This interface was designed to be flexible, and a Cray T3D memory model was layered on top of it to demonstrate this flexibility. Results are available from the testbench in the form of graphs and timing diagrams; the scalability and effects of caches on

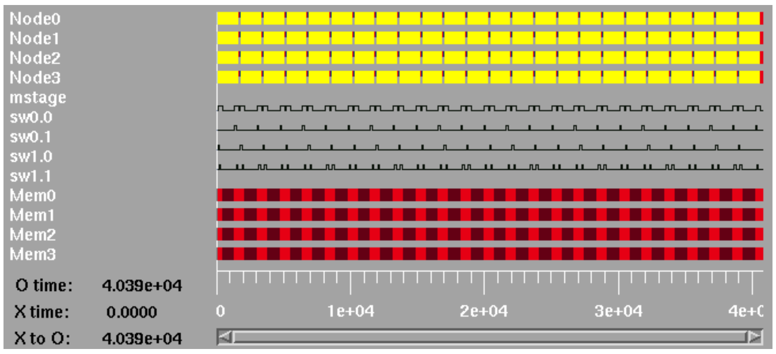


Figure 4.13: Top level timing diagram for 4 processors on a multistage network.

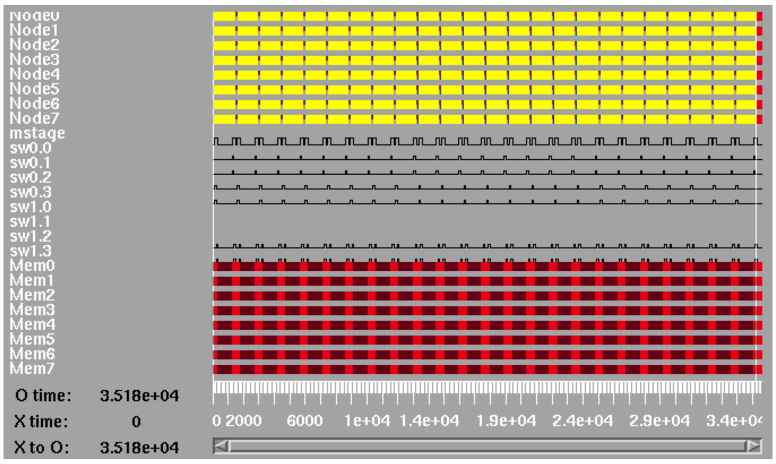


Figure 4.14: Top level timing diagram for 8 processors on a multistage network.

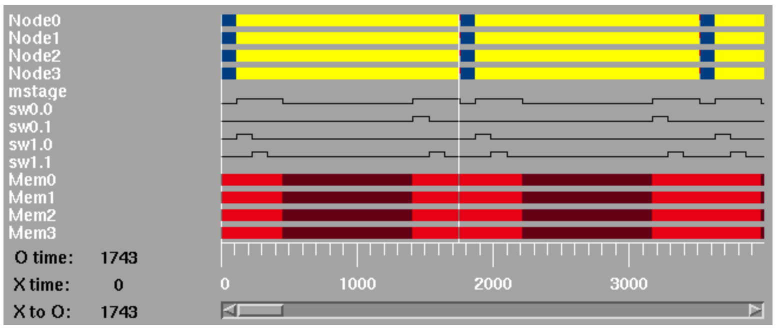


Figure 4.15: Zoomed timing diagram for 4 processors on a multistage network.

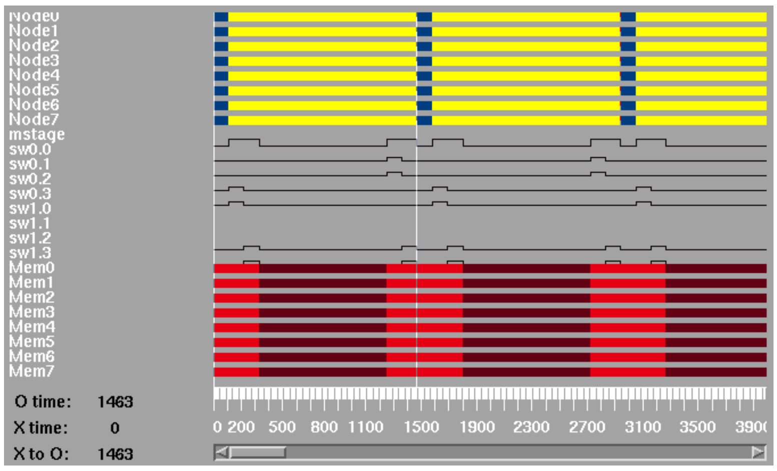


Figure 4.16: Zoomed timing diagram for 8 processors on a multistage network.

three networks were measured for the different workloads. Future development of the model could add more sophisticated shared memory models and workloads; this extension could be incorporated into the existing structure.

Chapter 5

The simjava Models

This chapter describes the simjava version of the multiprocessor testbed, including zoomable timing diagrams for displaying results obtained on the fly, 3D visualisation and graphs. The HTML version of this report, available on the WWW [4], includes the applets for some of the models.

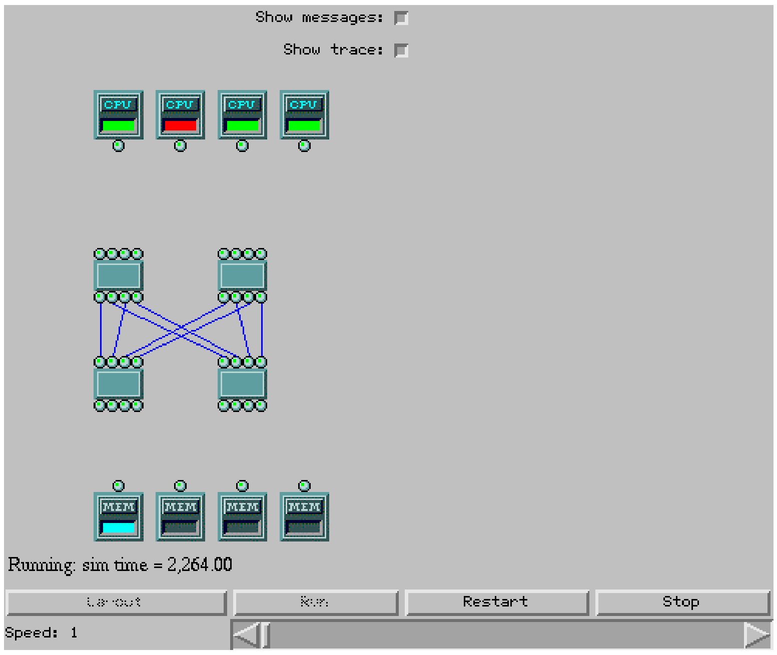
5.1 The models

Figure 5.1 shows four CPUs connected to four memory modules via a multistage switching network. The inputs to the network are at the top: the first four ports are inputs from the CPUs and the next four ports are inputs from the memory modules. Along the bottom (output) of the multistage network, again the first four ports go back to the CPUs and the next four ports go to the memories.

Running the simulation (by pressing the **Run** button) causes the CPUs to generate their workload. For this simulation model, each CPU does a number of memory accesses to the first memory module. The activity of the network switches can be seen when their state changes from idle to busy. Similarly a CPU can be idle, reading or waiting for a read request to return, whilst a memory can be either idle or reading.

Toggling the **show trace** button switches the animation on and off (with the animation switched off, the simulation runs faster, but since this model has no results display, there is nothing to see on screen). The **show messages** button usually selects whether or not to animate individual messages down links, but this feature is not enabled in this model. The **speed** slider selects the time in milliseconds between animation updates. **layout** resets the display, **run** starts the simulation, **pause** pauses it and **stop** stops it.

The purpose of this model is to demonstrate the basic model animation facilities provided by the testbed. In this case the visual feedback shows that only the first MEM is accessed, and also shows the dynamic pattern of accesses through the network.



Running: sim time = 2,264.00

Figure 5.1: The basic testbed. The working model can be included into a web page.

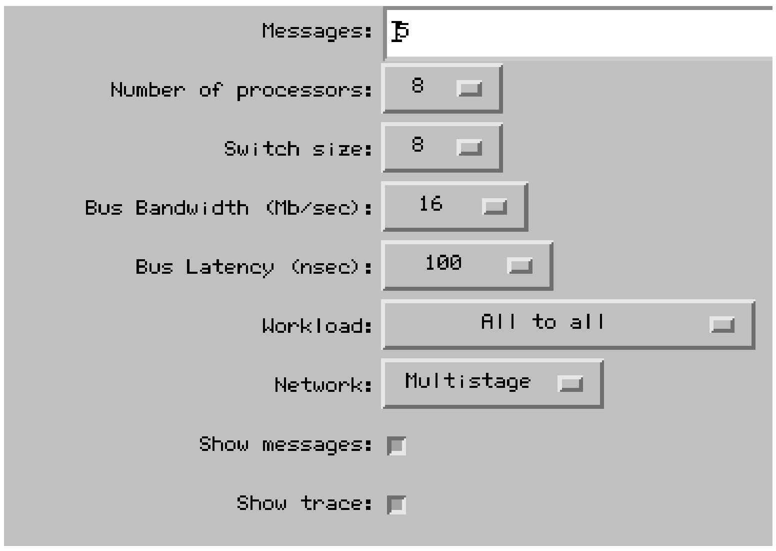
Figure 5.2 adds input parameters to the testbed. The effects of varying **number of processors**, individual **switch size**, **bandwith**, **latency**, **workload** and **network** can all be tried. Figure 5.3 shows a larger network.

Figure 5.4 shows how timing diagrams can be generated on the fly as simulations run. The timing diagram can be scrolled and zoomed once the simulation has finished; it is implemented as a *JavaBean* software component. Initially the timing diagram display is empty; as the simulation proceeds, the state changes are displayed on the diagram as different coloured bars (with the meaning provided by the key below).

Figure 5.5 shows the results of a multiple run experiment. In this case, the total time is displayed for different numbers of jobs on the network. Additional parameter boxes (*c.f.* figure 5.2) allow minimum and maximum range values to be entered. The results can be produced more swiftly by turning the **show trace** toggle off to disable animation.

5.2 3D network models

simjava may also be interfaced to 3D models constructed using VRML2. Figure 5.6 shows a 3D visualisation of the multistage switching network model. VRML allows more sophisticated visualisations than using 2D pixmaps, and provides good support for zooming in and out to obtain a visual feel for the behaviour of particular entities, or the whole system. VRML is a very modular language, and the effort required to



Messages:	5
Number of processors:	8
Switch size:	8
Bus Bandwidth (Mb/sec):	16
Bus Latency (nsec):	100
Workload:	All to all
Network:	Multistage
Show messages:	☐
Show trace:	☐

Figure 5.2: The testbed parameters allow interactive experimentation.

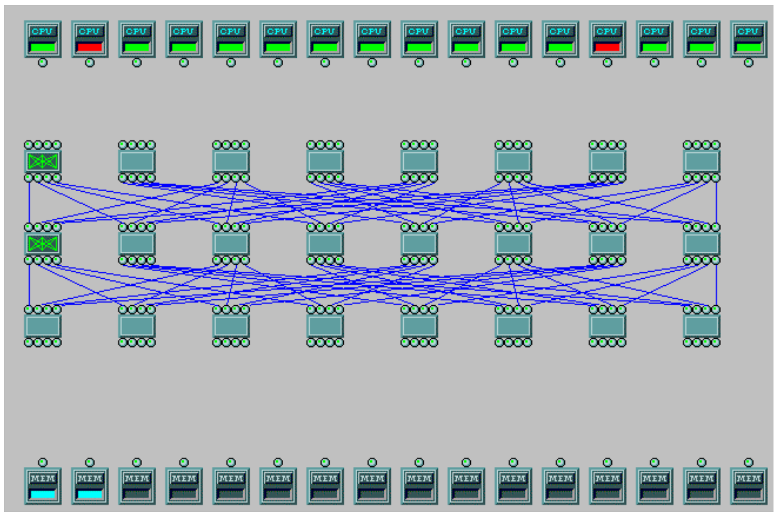


Figure 5.3: The testbed, with a larger number of processors.

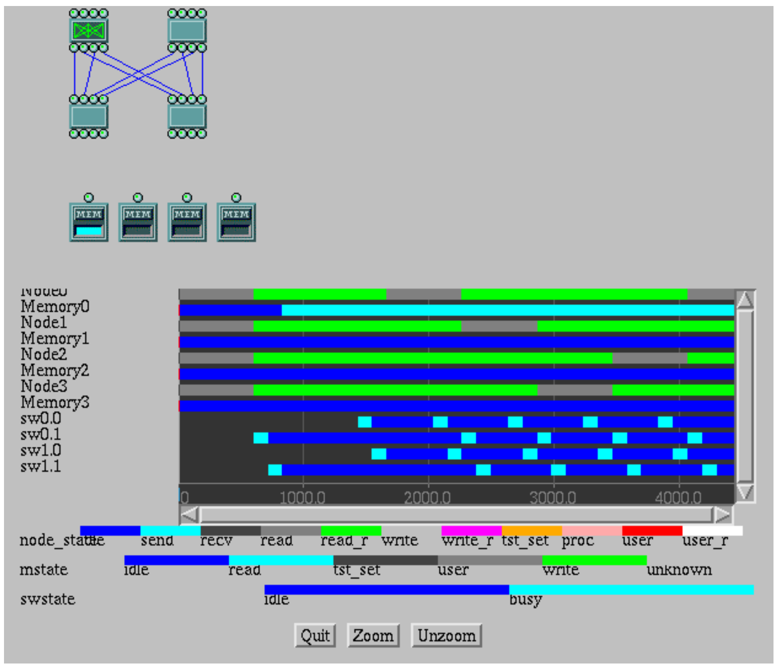


Figure 5.4: The testbed, with a dynamic timing diagram built in.

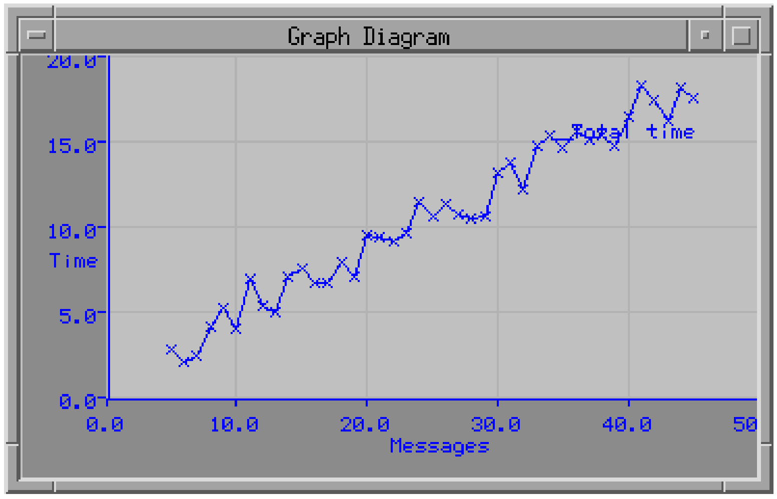


Figure 5.5: Dynamic graphs may also be included.

build a 3D visualisation is comparable to that required for the equivalent 2D model.

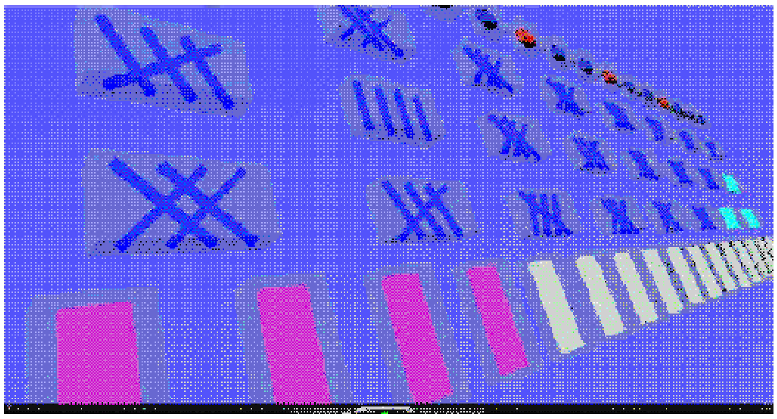


Figure 5.6: 3D model of a multistage network.

5.3 Summary

This chapter has presented a version of the multiprocessor testbed implemented using the simjava simulation language. The main advantage of using a web medium is accessibility of models. The obvious disadvantage of using Java for simulation (compared to C++) is that simulation run times are approximately 8 times slower than C++ [1]. However, this is offset by the ease of access to simulation models (i.e. by clicking on a web link), and the support for visualisation which is better in Java than C++.

Chapter 6

Conclusions

Simulating a complete system incorporating software and hardware requires decisions on the appropriate level at which to simulate. The initial aim of the EMIN project, of building realistic simulation models of actual networks and running real software on top of them, proved to be technically feasible but unrealistic in terms of performance measurements due to the very long simulation times required. The project therefore concentrated on using microbenchmark performance information as a practical interface between hardware and software, primarily looking at how the hardware (and low level software) affects the microbenchmarks.

Microbenchmarks provide a software view of the hardware performance, and are far more relevant than the optimistic figures of maximum bandwidth which are usually quoted (and which result in the oft reported 'disappointing' performance of parallel systems in practice). The project looked at measuring these microbenchmarks on real machines, as well as extracting them from simulation models which provide low level details of the *causes* of poor performance. This has meant concentrating on extracting as much detailed and visual information as possible from individual simulation runs.

The complexity of the hardware models and the interactions between software and hardware have made a visual approach essential; it is not sufficient to develop a 'black box' simulation model and hope that it will produce valid results. The simulation testbed developed using HASE provides visual feedback to the user and also allows performance graphs to be produced from the simulation traces. These graphs show that, other than in the case of a bus, system performance is influenced much more by the nature of the workload and by cache performance in the processors, for example, than it is by the type of interconnection network (*c.f.* Figure 4.7).

In terms of accessiblity of the simulations, it is very appealing to have visual models available on the World Wide Web; this extends the traditional idea of diagrams in printed documents to include 'active diagrams' in Web-based documents. The use of simjava allows models to be distributed and run extremely easily using the Web [12, 13], and also permits more advanced visualisations of behaviour using graphical modelling languages such as VRML2. This combination of visualisation and simulation is an effective tool for communicating the dynamic behaviour of complex systems, and has been applied to distributed software systems (such as transaction protocol analysis) as well as to network modelling.

Acknowledgements

The EMIN project was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) through Grant GR/K19716. The Hierarchical computer Architecture design and Simulation Environment (HASE [14]) was developed as part of the ALAMO project [15] supported by EPSRC through Grant GR/J43295.

Bibliography

[1] * F.W. Howell and R. McNab, "SimJava: a discrete event simulation package for Java". Home page with software, documentation and examples.
http://www.dcs.ed.ac.uk/home/hase/simjava/simjava-1.0 [1]

[2] * R. McNab and F.W. Howell *"Using Java for Discrete Event Simulation"* in proc. Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW), Univ. of Edinburgh, 219-228
http://www.dcs.ed.ac.uk/home/hase/simjava/UKPEWpaper.ps

[3] * F.W. Howell and R. McNab, *"SimJava: a discrete event simulation library"* in proc. SCS 1998 INTERNATIONAL CONFERENCE ON WEB-BASED MODELING AND SIMULATION, San Diego, January 1998.

[4] * Evaluation of Multiprocessor Interconnection Networks
http://www.dcs.ed.ac.uk/home/hase/projects/emin/csg38.html

[5] Claudia Wanke, "Very Large Crossbar Switches in Multistage Interconnection Networks", MSc dissertation, University of Edinburgh, 1993.

[6] D.J. Rogers and R.N. Ibbett. *"Xbar: a VLSI Circuit for Bit-sliced Packet Switching Networks."* In *Algorithms, Software, Architecture, J. van Leeuwen (Editor), Information Processing 92, Vol I*, 562-570, North Holland, 1992.

[7] R.H. Saavedra, R.S. Gaines, and M.J. Carlton. "Micro benchmark analysis of the KSR1." In *Supercomputing '93*, Portland, Oregon, 1993.

[8] * F.W. Howell, "MPI microbenchmarking". Software, documentation and results for message passing microbenchmarking.
http://www.dcs.ed.ac.uk/home/fwh/timing

[9] * Fred Howell, "Approaches to parallel performance prediction" PhD Thesis, 1996, University of Edinburgh.
http://www.dcs.ed.ac.uk/home/fwh/thesis

[1] *References marked with a * were produced as part of the EMIN project.*

[10] * F.W. Howell, *"Reverse Profiling"*, Software Engineering for Parallel and Distributed Systems : Proceedings of the First IFIP TC10 International Workshop on Parallel and Distributed Software Engineering, Chapman and Hall, 1996.

[11] * F.W. Howell, "SHMEM microbenchmarking". Software, documentation and results for shared memory microbenchmarking.
http://www.dcs.ed.ac.uk/home/fwh/emin/shmem

[12] * F.W. Howell, "EMIN: the simjava models",
http://www.dcs.ed.ac.uk/home/hase/projects/emin/simjavamodels.html

[13] E.H. Page. "Web Based Simulation at MITRE". A survey of simulation packages, with the distributed SimJava homepage
http://ms.ie.org/websim/

[14] * HASE home page
http://www.dcs.ed.ac.uk/home/hase

[15] * ALAMO home page
http://www.dcs.ed.ac.uk/home/hase/projects/alamo.html