Distributed Route Initialization Algorithms for the Monash Secure RISC Multiprocessor

V. J. Fazio
Dept. of Computer Science
Monash University
Clayton, Vic 3168
Australia
vjf@cs.monash.edu.au

R. D. Pose
Dept. of Computer Science
Monash University
Clayton, Vic 3168
Australia
rdp@cs.monash.edu.au

Abstract

The Monash Secure RISC Multiprocessor is a general purpose large scale multiprocessor with a capability-based distributed shared virtual memory. The physical design of the Monash SRMP (Secure RISC Multiprocessor), consists of passive backplane bus sections and dual-ported processor-memory modules connected in an unusual topology.

This paper details three algorithms used for route initialization. The first is a simplification of the flooding algorithms commonly used in distributed networks. The second algorithm uses a variation of the depth-first search algorithm, which has been used to route mesh and hypercube architectures. The final algorithm uses a theoretical solution to the gossiping (all-to-all broadcast) problem for mesh networks; this is the first known simulation of this algorithm. The algorithms are evaluated via simulations of the traffic flow over the multiprocessor network.

1 Introduction

The Monash SRMP (Secure RISC Multiprocessor) is an ongoing research project in the Department of Computer Science, Monash University. It is a general purpose, large scale multiprocessor with a secure capability-based distributed shared virtual memory.

The system architecture consists of modules connected by passive backplane buses. These modules contain a small internal bus (the Memory-Processor bus), which is used to connect components, such as 1 GHz serial links, 64-bit RISC processors, and large memory arrays. Each module has two 64-bit bus ports that are used to communicate with other modules via external passive buses. Modules can also communicate via the 1 GHz serial links. This architecture has the distinct advantage that all the switching and bus transmission componentry is internal to each module. This allows for easy incremental expansion without the need to purchase the additional switching hardware that many other multiprocessor systems require. Pose et al. [2] provide an introduction to the main architectural features.

Figure 1a. A configuration of buses and modules.

The Figure 1a depicts one possible configuration. Once a planar mesh has been set up using a particular configuration, the user can then either join a pair of edges to create a cylinder, or join all edges to create a sphere, or a torus connected bidirectional 4-ary n-cube. Each module has three separate lookup tables to coordinate the routing process.

The concept of independent modules has been extended to the initialization of routing tables which is the core topic of this paper. The routing initialization algorithms are also designed to initialize the routing lookup tables with no knowledge of the surrounding configuration. This concept has much in common with the initialization of distributed networks.

It is known that temporary faults and software faults form the bulk of faults in any computing system [18] and [19]. Thus some faulty requests are injected into the network. Permanent (fail-silent) faults in modules are not only rare, but when they do occur, they are usually easier to detect.

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

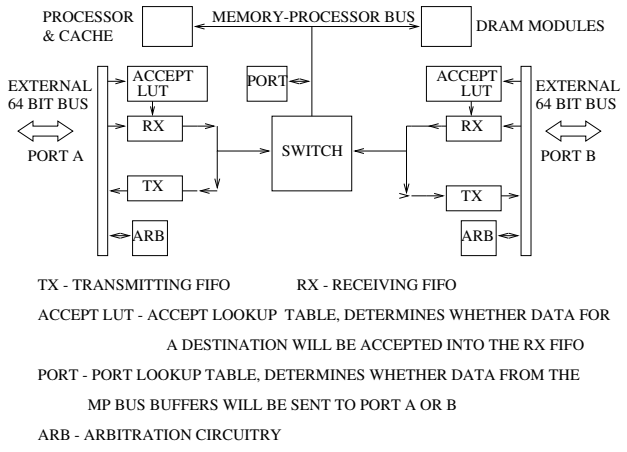


Figure 1b. A block diagram of a module's internal features.

2 Static Routing Hardw are

Travel across the bus is fully pipelined, so requests may quickly proceed across the bus if there is no contention. This has a similar effect to virtual cut-through, as used by many routing chips. A block diagram of the internals of a module is shown in Figure 1b. The interface between the bus and each module is bordered by FIFO queues, one for requests going to the bus (transmit FIFO), the other for requests coming from the bus (receive FIFO).

It is assumed that all full FIFOs will destroy any excess packets and that if any bit errors are detected in a packet, then the packet will also be destroyed, after it has reached its destination. The memory module also has a request queue which will destroy excess requests.

The Monash SRMP selects paths for its requests using the destination address (and by implication, the current position). As mentioned in the Introduction, each module contains three lookup tables. There are two types of lookup table, both indexed on the destination address. A port lookup table determines which port a request will go through after it is created at the source. An accept lookup table at each port determines whether that port will accept any request from the bus. The route taken by a request is a function of its destination, the first port lookup table entry for that destination and a number of accept lookup table entries. It is these accept lookup table entries that guide the request along its route to the destination, once it has left the source module. Broadcasting across the bus is achieved by setting most accept lookup tables to "accept" for a broadcast address.

Each module, at initialization, assumes no knowledge of the number of modules or their positions on the backplane. It does assume that each module has a unique identification number, and a minimal amount of hardware used to initialize lookup tables from scratch. All algorithms have similar hardware requirements.

In order for modules to gain knowledge of each others' existence there must be a mechanism for an exchange of module identifiers across each bus. These need only be unique to the bus itself. This can be implemented using two special types of module number. Each module connects to two slots on the backplane, one for each port. Each slot on a bus has a number, called the bus number. There is also a module number, which in unique to the whole system.

This paper is related to the work presented in [12] and [13], but there are a number of important architectural differences. First of all, square and hexagonal planar mesh architectures are used, which are very different to the mesh architecture used in the Monash SRMP. Second there is a degree of synchronisation between modules in the other meshes, so that data exchange can be carefully coordinated to avoid traffic hot-spots. The Monash SRMP has no suc h capability. Third, as mentioned previously, when a FIFO is full it is able to destroy any incoming requests; this is not a feature of the HARTS system, as outlined in [12]. Bagchi et al. [20] have done work similar to ours' in that each module has its own identity and has to discover the identity of its neighbours. This paper approaches the problem from a more theoretical viewpoint. It applies to general point-to-point networks, and uses a minimal number of transmissions. The algorithm presented does not allow for transmission errors or for messages culled due to full FIFOs. It also relies on a "leader" to commence the algorithm, which is a concession to centralisation. Large amounts of information are accumulated in a single "transmission", which in a real system may need to be broken into packets for transmission.

A series of different routing algorithms are outlined in the next few sections. Each algorithm only goes as far as allowing each module to decide a set of shortest paths to all other modules. Another algorithm has to ensure that consistent values are written into the lookup tables, but examination of this topic is beyond the scope of this paper.

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

3 Routing Lookup Table Initialization via Distance V ector Flooding

The routing algorithm proposed is a variation of the well known "flooding" algorithm [6] and used in ARPANET [7] and NELHNET [5] to broadcast changes to network topology. Each request contains a number which denotes the distance from the source that the request has travelled, and broadcasts it to all modules on the mesh. Eac h module can then determine the shortest distance to other modules and the direction of the shortest path, and initialize its lookup tables.

Figures 2a,2b,2c contain pseudo-code samples.

3.1 Initialization

- Processor reads three values from special fixed internal address(es): the module number and the bus number for Port A, and the bus number for port B.
- Processor initializes two arrays, one for distance values, another for sender values. The distance values are initialized to a very large number. Sender values can be initialized to zero.
- Processor sets all accept lookup table values to not_accept, except for the bus number, which is always set to accept. Port lookup tables can be left uninitialized.

```
int distance[max_modules];
int sender[max_modules];
```

Figure 2a. Arrays used for Flooding.

3.2 Startup

Each processor sends two write requests, first it alters the port lookup table to point to Port A, sends a request, then it alters the table to point to Port B and sends another one. The destination is set to the bus number of the port. The source is a unique module number. Each request contains a distance, which is initially set to zero.

```
/* request.dest = destination field of request */
/* request.source = source field of request */
/* request.data = distance field of request */
/* send() = sends request to address at "dest" */
/* port_lookup[] = local port lookup table */
/* mem[] = refers to memory at a location */

/* bnumA = address of bus number at port A */
bus_numA = mem[bnumA];
/* bnumB = address of bus number at port B */
bus_numB = mem[bnumB];
request.dest = bus_numA;
request.source = source_addr;
request.data = 0;
port_lookup[bus_numA] = 0;
send(request);
request.dest = bus_numB;
port_lookup[bus_numB] = 1;
send(request);
```

Figure 2b. Flooding Initialization.

3.3 Processing

After the request is taken from the bus by a module, it will be sent to the MP bus. The request is sent to the memory con troller, generates an interrupt, and is soon serviced by the processor. The processor extracts the contents of the request from the memory controller's buffer and proceeds to process it.

When the processor receiv es the request it increments the distance value, and compares it with that held in the distance array. If the request's distance is less than that in the array, the array's value gets replaced by the request's. The bus number that was part of the data field in the request gets replaced by the other port's bus number. The destination field also gets altered in the same manner. The source field and its component in the data field are both left unchanged. If the distances are the same, and the sending module numbers are the same, then the request is destroyed. If the distances are the same, but the sending numbers are different, then the sending number, and port direction are stored for future reference. If the distance is too large to replace the array value, then the request is also destroyed.

The new request is soon sent out to the other port, and the process is repeated until no more requests are received by a processor for a certain length of time.

Figure 2c contains the relevant pseudo-code.

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

```
 /* distance estimate interrupt routine */
request.data = request.data+1;
if (request.data<distance[request.source])
 { /* store request details in array */
 distance[request.source]=request.data;
 sender[request.source]=request.sender;
 direction[request.source]=request.dirn;
 send(request);
 }
else if ((request.data=distance[request.source])
 and (sender[request.source]<>request.sender))
 {/* stores extras in array of linked lists */
 append(extras[request.source],request);
 }
```

Figure 2c. Flooding Algorithm.

3.4 Completion

This part of the algorithm will complete when all the shortest paths have been traversed, and no more requests are propagated through the system. Some sort of timeout mechanism must be employed to enable this stage of the algorithm to successfully complete, depending on the perceived number of modules in the system.

3.5 Resistance to failures

The lack of any feedback paths from the receivers of distance data to the senders prevents any sort of informed retry, other than repeating the procedure whether repetition is required or not. Each request is sent twice in the simulation.

3.6 Efficiency Improvement

The algorithm outlined above describes each module as sending two requests each with a source, destination and distance from source. It would certainly be more efficient for each module to wait for a random (but fairly short) amount of time before sending the requests. If other requests are sent to a module before it has sent its own requests, then it can amalgamate its request with the one passing through. The bundled requests can significantly reduce the amount of overhead required to transmit data across the mesh. The simulation model uses this technique, and can bundle up to four requests into one.

4 Routing Lookup Table Initialization with Feedback using Read Requests or Breadth-First Searching

4.1 Introduction

The algorithm outlined in the previous section has the disadvantage that each module is unable to differentiate between a request that was lost because of an error in transmission or because a module does not exist. The requests were just transmitted and re-transmitted until it was highly probable that an accurate picture of the outside world had been developed. A technique that would avoid the repetition involves each module sending read requests to neighbouring modules, and waiting for a reply. If no reply is given then the request can be resent. After two attempts, it can be assumed that the module does not exist. This is similar to the depth-first searches used in hypercube architectures, [14], [15], [16].

The algorithm works by first allowing each processor to investigate the validity of the links to its nearest neighbours. Then each processor sends a series of enquiries (read requests) to determine the global number of each module and surrounding links' health. Since each individual module knows the state of all nearby links, and sends this information with its module number, there is no need to test the exact path taken by any request, since the status of the links in the path can be deduced from the information given. Therefore, requests are free to use paths which minimise the number of requests rather than paths that minimise the point-to-point distance between source and destination modules.

4.2 Initialization

- Processor reads three values from special fixed internal address(es): the module number, the bus position number for Port A, and the bus position number for port B.

- Processor writes to the accept lookup table at each port, setting all values to "not_accept" except for that corresponding to the bus position number for that port. The module will only accept requests which have a certain bus position number as the destination. Port lookup tables can be left uninitialized.

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

```
/* bus number at ports A and B */
int portA_num, portB_num;
int module_num; /* module number */
int memval,j,result;
/* =1 found, else 0 */
int module_found[max_modules]; struct packet
{
storage for the module number */
int modn;
int request_type;
/* maximum value of packet.pointer */
int arraylen;
/* points to current module in path */
int pointer;
}
```

Figure 3a. Some Bread-First Search data structures.

```
 /* PROCESS READ REQUEST */
if (packet.request_type=readreq)
 { /* is req at end of path? */
  if (packet.pointer=packet.arraylen+1)
   { /* set to local module number, send back*/
     packet.modn=board_number;
     packet.request_type=readdat;
     send(packet);
   }
  else /* if not at end..*/
   { /* point to next module on path */
     packet.pointer=packet.pointer+1;
     result=find_grid_elem(packet);
     if (result≥0) /*found local answer?*/
      { /* send result back */
        packet.modn=result;
        packet.request_type=readdat;
        send(packet);
      }
     else
      { /* enquire at next node */
        packet.pointer=packet.pointer+1;
        send(packet);
      }
   }
 }
```

Figure 3b. Breadth-First Search, read requests.

```
else /* PROCESS READ DATA */
if (packet.pointer=0) /* returned to start? */
 { /*insert result into data str.*/
   insert_grid_elem(packet.modn,packet);
   if (module_found[packet.modn]=0)
    {/* if not found, send new requests */
      module_found[packet.modn]=1;
      set_new_event(...);/* etc.*/
    }
 }
  else
 { /* send onwards */
  packet.pointer=packet.pointer-1;
  send(packet);
 }
```

Figure 3c. Breadth-First Search, read data.

4.3 Startup and Processing

Each processor sends a read request to each module on the neighbouring buses. Before sending read requests, the processor must write the correct value to the port lookup table to ensure that the request is sent to the correct port. The read request asks for the module number of the neighbouring modules. It uses bus numbers as the source and destination for the request. This enables the module that receives the request to identify which port the request came from by looking at the destination field, and to use the source field as the destination when returning the read data.

Upon receiving the read request, the switch inside the other module must accept the request into the MP bus, and not send it the the FIFO at the other port. Thus some hardware must be allocated to be hardwired to the module number and the bus position numbers, and allow the module to send the request to the MP bus if a comparison between the bus position numbers and the destination is successful. During normal operation, that is, after initialization, the hardware would be required to compare the module number with the incoming destination.

Figure 3 is a pseudo-code description of the breadth-first search algorithm, and below a short description of the algorithm.

Each module has an area of memory which holds a data structure representing the module numbers and relative positions of all the modules in the outside world. The values represented in the data structure can be either a module number, "unknown" (-1) or

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

"absent". "Unknown" is used to represent a part of the tree which has not been explored at present. "Absent" denotes a module that did not respond to requests for information, that is, when the local processor had to time out after sending multiple read requests. Each module, after finding the numbers of its immediate neighbours, can send more read requests for information about their more distant neighbours. If "unknown" is the result of a read request for a particular location, then the local processor must pass the read request on to the next module in the path. After the reply to the local request arrives, then the reply will be forwarded to the originator of the request.

If a request has had no response for a large amount of time, then another request is sent. If neither request brings a response, then the memory value is changed from "unknown" to "absent".

4.4 Completion

After the algorithm terminates, each node contains a map of the whole network. Each node must then execute an algorithm to determine the set of shortest paths to every possible destination, using an algorithm such as Dijkstra's [4] or Floyd's [8].

5 KCV algorithm: toroidal grids, minimal communications

In graph theory parlance "gossiping" is the process whereby each vertex in a graph must send a message to all other vertices. "broadcasting" is a subset of this process, whereby only one vertex must send a message to all other vertices. Variations of gossiping have been well studied over the years, as indicated by Hedetniemi's extensive survey [10]. An optimal time algorithm for grid graphs using a weak communication model is presented in a paper by Krumme, Cybenko and Venkataraman [11]. Its communication pattern is used as the basis for the asynchronous KCV routing algorithm. Both communication patterns are depicted in Figure 4. The weak communication model constrains a module to either send one message to another module or receive a message from another module at each time step, but it cannot do both. Nor can it send a message to multiple modules. Unlike the flooding algorithm, KCV's requests were not able to be bundled together because KCV requires significantly more data to be stored inside each request.

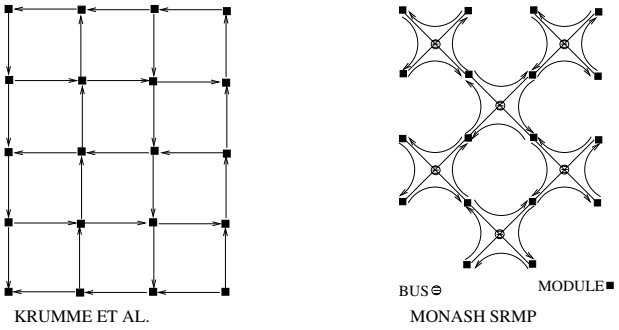


Figure 4. Synchronous communication scheme by Krumme et al. [11], together with the equivalent scheme for the Monash SRMP.

5.1 Startup

As in the previous section, the algorithm commences with each module examining the state of the surrounding links. A simple read request, similar to that initially used in the breadth-first search is used for this purpose. The algorithm proceeds in a similar way to flooding, whereby each module sends its module number and the state of the surrounding links (bit vector) along the paths shown in Figure 4.

5.2 Processing

When a request arrives, it causes an interrupt, and appropriate code is executed. The code reads the module number and the distance value inside the request, and if the distance value is smaller than one presented in the distance array, then it is stored in the array. This is similar to the flooding algorithm presented previously. Assuming that the multiprocessor is connected in a full wrap-around torus, then provided no mishaps occur, the request will eventually find its way back to its source module. This allows the module to estimate the shortest distance to all modules. By only allowing one way travel along loops in the torus, the distance to some modules along the loop will be inaccurate. By subtracting this distance from the total loop distance, the correct distance can be obtained. The major difference between this algorithm and the flooding one is that requests are not broadcast across the bus, but sent one-to-one like normal requests. Each request is sent twice, in a similar way to the flooding algorithm, but each time it is sent in a different direction, along the x or y-axis. Routing algorithm pseudo-code is shown in Figure 5.

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

```
/* stores the smallest distance to each module */
int distance_array[max_modules];
/* stores which direction request comes from */
int came_from[max_modules][1];
/* smallest distance in each dimension */
int distance_x[max_modules];
int distance_y[max_modules];
/* local board number */
const int bn;
/* used to store loop length */
int xlooplen,ylooplen;
/* source and destination */
int d,s;
/* direction within a dimension */
int pn;
struct packet_type
{
/* total distance travelled so far */
int net_dist;
/* distance travelled in x,y dimensions */
int x_dist,y_dist;
/* origin module number */
int modn;
/* dimension travelled in */
bit xy_dirn;
}
```

Figure 5a. Some KCV Data Structures.

```
/* Functions */
/* if (x<0) x= -x; return(x); */
int abs(x);
/* returns the slot number at port */
int get_slotnumber(port);
/* returns packet attributes */
int get_source(),get_dest(),get_dirn();
/* destroys packet, so it isn't sent */
void terminate(packet);
/* sets up packet on event queue */
void set_new_event(packet);
/* create a new packet and assign values */
packet_type assign_values();
```

Figure 5b. KCV Functions.

```
packet.net_dist=packet.net_dist+1;
/* increment net (total) distance */
/* if moving in x-direction ... */
if (packet.xy_dirn=1)
packet.x_dist=packet.x_dist+1;
else packet.y_dist=packet.y_dist+1;
if (packet.modn=bn) /* traversed the loop?*/
 { /* store loop length */
  if (packet.y_dist=0) xlooplen=abs(packet.x_dist);
  if (packet.x_dist=0) ylooplen=abs(packet.y_dist);
 }
/* ensure no packets sent in this direction */
came_from[packet.modn][packet.xy_dirn]=1;
if (packet.net_dist<distance_array[packet.modn])
 {
  distance_array[packet.modn]=packet.net_dist;
  distance_x[..]=... distance_y[..]=... /*etc.*/
if (came_from[packet.modn][1-packet.xy_dirn]==0)
   { /* request has not been sent?*/
   s=get_source(...); d=... pn=... /* etc.*/
   /* create new packet, set attributes */
   new_pack=assign_values(1-packet.xy_dirn..);
    set_new_event(new_pack);
   }
  /* set source and dest fields for next dest */
  packet.dest=...
  packet.sendr=...
 }
else
 terminate(packet);/* do not send onwards */
```

Figure 5c. KCV Algorithm.

6 Comparison by Simulation

As with any simulation of such a complicated system, there are a number of variables which can be varied in order to compare the performance of the algorithms. Here is a point by point discussion of some of the more important variables used in the simulations:

- The simulation model contains a number of queues inside each module. Each queue can have a maximum length, which if exceeded results in all subsequent arrivals being destroyed.
- Requests can be made to contain some sort of error, such that, on completion of their journey through the system, they must be destroyed, without using the information contained within.

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

- A very important variable is the rate at which requests are released. If there are too many requests released, then the performance may suffer. For example, if the maximum queue length is fixed, then this will be because many requests will be culled once queue lengths reach the maximum. If there are too few requests released, then the system may suffer from underutilisation.

It must be noted that the simulation models the communication process between the modules, and not the software execution in each module as requests arrive. Simulation models are available for modelling the cache, processor and main memory interaction, given a stream of machine code. MINT is one such example [17]. There is a large variety of processor types, cache sizes, MMU policies that could be implemented within the modules. This area of research has been adequately covered. Another source of variation is the data structures used by the software part of the algorithms. It was decided to avoid the duplication of research and combinatorial explosion of simulation parameters making the request processing time an input variable, instead of simulating the CPU, cache etc. This has advantages in the form of: a strong degree of memory-CPU system independence, an ability to form an accurate comparison of the algorithms' communication requirements and an approximate estimate of total execution time.

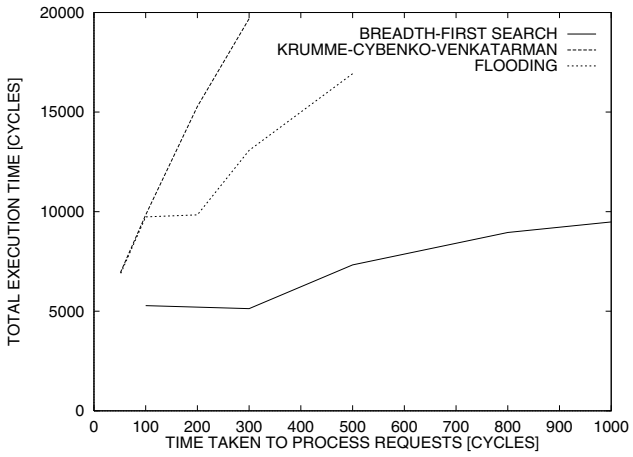
The average amount of time spent executing sections of code is called the request generation period. It is generated from a uniform distribution in order to properly model the even spread of delays that would occur when processing different path lengths, degrees of request bundling etc. All results shown are averages of 20 simulation runs, using different random number seeds. A 16 module toroidal configuration was used for all simulation runs.

6.1 Comparison without transmission errors

In this section, simulations are used to compare the error-free performance of all algorithms. Parameter settings are as follows:

- Maximum FIFO queue sizes are fixed to the same value, 6, for all simulation runs. Maximum memory queue size is 25.
- No requests contain errors.
- The request generation period is varied, in order to measure the effect on run time.

A graph of request generation period versus request total execution time is plotted in Graph 1, for the three algorithms.



Graph 1. Time taken to process requests (cycles), against total algorithm execution time (cycles).

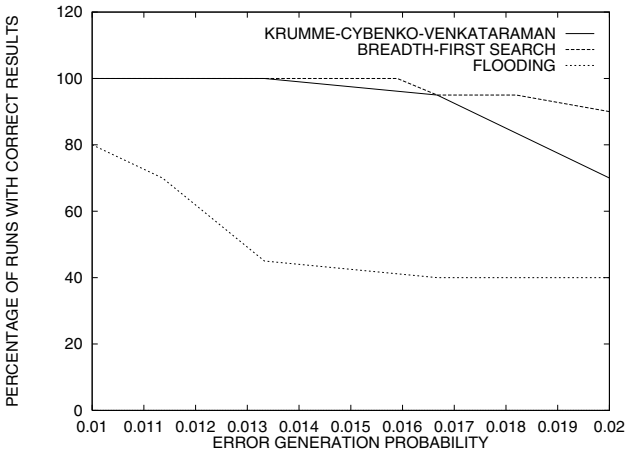
Graph 1 shows that the breadth-first search has a superior performance to the other algorithms, given the same request generation period. The flooding and KCV algorithms performed similarly, with flooding proving to be slightly faster than KCV. Ultimately, the comparative performance of the algorithms is determined by the speed of their software components. Flooding requires the smallest amount of code, and breadth-first search requires the largest. For each simulation run, the breadth-first search references its data structures two to three times more often than the other algorithms. This is indicative of a much longer average request generation period.

6.2 Comparison with transmission errors

For these simulations, the probability of a transmission containing an error each time it is sent from the source and destination modules is varied, and the percentage of correct runs is measured. Maximum FIFO queue sizes are fixed to the same value, 6, for all simulation runs. Maximum memory queue size is 25. A graph of probability versus percentage of successful runs is shown in Graph 2.

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE



Graph 2. Error generation probability, against percentage of runs generating correct results.

In Graph 2 the flooding algorithm exhibits a poor performance under transmission errors. This is probably due to errors limiting the propagation of bundled requests. The non-bundled requests in KCV are much more resistant to these errors.

7 Conclusion

The aim of this study was to compare the performance of three different types of algorithms, in order to determine the relative advantages and disadvantages of each, in terms of their communication performance.

It would be easy to conclude from the results that the breadth-first search is the ideal algorithm for initialising lookup tables, but there are two considerations: the first is that the average request generation period for breadth-first search is very likely to be several times higher than the other algorithms. This means that its performance in terms of total execution time is far less favourable than previously assumed. The second consideration is more practical, that a request can only hold a small search path. Thus breadth first search is only the most useful algorithm if there are a small number of modules and request processing can be done efficiently.

Of the other two algorithms, flooding is more sensitive to transmission errors, but slightly faster than KCV. The practice of bundling requests together in order to improve speed and processing efficiency appears to inhibit performance under transmission errors.

In the final analysis, algorithm performance depends more on the ability to implement the software components efficiently, than the time taken to communicate over the network. This takes the form of a tradeoff between software complexity and communications efficiency. Algorithms that use less sophisticated software, such as flooding, often require excessive communications bandwidth. A breadth-first search algorithm has the opposite requirements.

Acknowledgemen ts

Vincent Fazio gratefully acknowledges the financial assistance provided by a Monash Graduate Scholarship. The Secure RISC Architecture project was supported by an Australian Research Council grant.

References

[1] M. Castro and R.D. Pose: "The Monash Secure RISC Multiprocessor: Multiple Processors Without a Global Clock", in G. Gupta (ed) *Australian Computer Science Communications*, Seventeenth Annual Computer Science Conference (ACSC-17) Christchurch, New Zealand, No. 16, Vol. 1, pp. 453-59, 1994.

[2] R.D. Pose, V.J. Fazio, J.R. Wells: "An Incrementally Scalable Multiprocessor Interconnection Network with Flexible Topology and Low-Cost Distributed Switching", *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, Summer-Fall 1994, pp. 31-36.

[3] V.J. Fazio: "Four Arbitration Circuits: Analysis and Comparison", *Technical report 94/209*, Department of Computer Science, Monash University, Melbourne, 18pp, 1994.

[4] E.W. Dijkstra: "A Note on Two Problems in Connection with Graphs", *Numerische Mathematik*, 1, pp269-271, 1959.

[5] D.J. Nelson, K. Sayood, H. Chang: "An Extended Least-Hop Distributed Routing Algorithm", *IEEE Transactions on Communications*, Vol. 38, No. 4, pp. 520-528, April 1990.

[6] K.M. Chandy and J. Misra: "Distributed Computation Graphs: Shortest Path Algorithm", *Communications of the ACM*, Vol. 25, No. 11, pp. 833-837, November 1982.

[7] D. Bertsekas and R. Gallager: *Data Networks*, Prentice-Hall, Eaglewood Cliffs, NJ, 2nd Edition, 1992.

[8] R.W. Floyd: "Algorithm 97: Shortest Path", *Communications of ACM*, 5, 1962, p345.

[9] V.J. Fazio, R.D. Pose and J.R. Wells: "Monash Secure RISC Multiprocessor: Performance Simulation", in R. Kotagiri (ed) *Australian Computer Science Communications*, Eighteenth annual computer science conference (ACSC-18) Glenelg, Australia, Vol. 17, No. 1, pp. 161-165, 1995.

[10] S.M. Hedetniemi, S.T. Hedetniemi and A.L. Liestman: "A Survey of Gossiping and Broadcasting in Communication Networks", *Networks*, Vol. 18, 1988, pp. 319-349.

[11] D.W. Krumme, G. Cybenko, and K.N. Venkataraman: "Gossiping in Minimal Time", *SIAM Journal of Computing*, Vol. 21, No. 4, February 1992, pp. 111-139.

[12] D.D. Kandlur and K.G. Shin: "Reliable Broadcasting Algorithms for HARTS", *ACM Transactions on Computer Systems*, Vol. 9, No. 4, November 1991, pp. 374-398.

[13] S. Lee and K.G. Shin: "Interleaved All-to-all Reliable Broadcast on Meshes and Hypercubes", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 5, May 1994, pp. 449-458.

[14] M-S. Chen and K.G. Shin: "Depth-First Search Approach for Fault-Tolerant Routing in Hypercube Multicomputers", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp. 152-159.

[15] J.M. Gordon and Q.F. Stout: "Hypercube message routing in the presence of faults", *Proc. of 3rd Conf. Hypercube Comput. Appl.*, Jan 19-20, 1988, pp. 318-327.

[16] E. Chow et al.: "Hyperswitch network for the Hypercube computer", *Proc. 15th Annu. Int. Symp. Comput. Arch.*, May 30-June 2, 1988, pp. 90-99.

[17] J.E. Veenstra and R.J. Fowler: "MINT Tutorial and User Manual", *Technical Report 452*, Computer Science Department, The University of Rochester, New York, 52pp, April 1994.

[18] I. Eslick, A. DeHon and T. Knight: "Guaranteeing Idempotence for Tightly-Coupled Fault-Tolerant Networks", *Proceedings First International Workshop Parallel Computer Routing and Communication Workshop* (PCRCW'94), pp. 219-225, Springer-Verlag, 1994.

[19] D.P.Sieworek: "Faults and Their Manifestation", *Fault-tolerant Distributed Computing*, A. Spector and B. Simons (eds.), Springer-Verlag LNCS 448, pp. 244-261, 1987.

[20] A. Bagchi, S.L. Hakimi and E.F. Schmeichel: "Gossiping in a Distributed Network", *IEEE Transactions on Computers*, Vol. 42, No. 2, February 1993, pp. 253-256.

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE