HYPERSWITCH NETWORK FOR THE HYPERCUBE COMPUTER

E. Chow, H. Madan, J. Peterson
Jet Propulsion Laboratory,
California Institute of Technology
Pasadena, Calif. 91109

D. Grunwald, D. Reed
University of Illinois,
Urbana-Champaign

ABSTRACT

The performance of a parallel algorithm depends in a large part on the interconnection topology of the multicomputer system. The method presented in this paper realizes a kind of interconnection network, called a hyperswitch network, that is achieved using a mixture of static and dynamic topologies. Here, available or fault free paths need not be specified by a source because the routing header can be modified in response to congestion or faults encountered as a path is established. This method can be accomplished in a static topology such as the hypercube network if the nodes have switching elements which are capable of performing the necessary routing header revisions dynamically. Detailed simulation results show that the hyperswitch network is consistently more efficient than fixed path routing for large message traffic conditions. The simulation results also show that the hyperswitch network has equivalent latency overhead for messages with localized and antilocal destinations (i.e., less then a 25% difference between diameter 1 and 5).

INTRODUCTION

A multicomputer interconnection network may be judged by many different criteria. Network link reliability and availability are among those attributes that should be given the most emphasis. With the aim of designing a very richly interconnected multicomputer system, a hypercube topology with a switching element in each computer node is proposed and being implemented. (A three-dimensional hypercube computer is shown in Figure 1.) This interconnection network connects a great number of small computer nodes using the fixed hypercube topology, which is characterized by point-to-point links between computer nodes.

Tasks among various nodes (such as the eight nodes depicted in Figure 1) can be performed with either a minimal or a high degree of interaction. Systems which have this wide latitude of interaction are generally designated as loosely coupled systems. The performance of such systems depends largely on the efficiency of communication between computer nodes. The previously known hypercube communication limitations are high message latency overhead, low channel utilization, static routing (e-cube algorithm), and a lack of fault tolerance. Certain routing algorithms (e.g. virtual channels, e-cube routing[1,2]) avoid deadlocks by ordering the links to be allocated. Thus a lower order link resource cannot be committed if a required higher order link resource cannot be obtained. The disadvantage of this approach in an interconnection network is that it limits the number of paths connecting a source to a destination to exactly one, even though several alternate free paths may exist at that moment.

In general, a hypercube topology with $N=2^n$ nodes is arranged as an $n=\log N$ dimensional hypercube with two nodes in each dimension. The nodes are represented by binary equivalents of the decimal numbers between 0 and N-1, where adjacent nodes differ by one bit [3]. The diameter of this topology is n, and there exist n disjoint paths between a source and a destination. For example, with a three-dimensional hypercube computer a message from node 0 to node 7 can be routed through one of the following disjoint paths:

000 -> 001 -> 011 -> 111
000 -> 010 -> 110 -> 111
000 -> 100 -> 101 -> 111

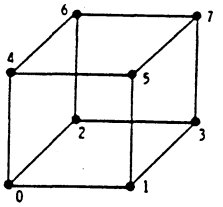

Fig 1. Hypercube network

Hence the structure is highly fault-tolerant. The number of I/O links per node is again $n=\log N$.

Steven's Post Office chip for the FAIM machine [10] uses a dynamic routing algorithm which does not wait for links that are congested. But, if all the desired outbound links are busy, then the routing chip randomly selects an available link for reroute. The hyperswitch routing algorithm uses an informed heurestic search algorithm, which can automatically avoid congested links based on it's previous experience congestion history information. Therefore, a message does not wait for a busy link because the hyperswitch network will try to route the message through non-congested links.

As can be seen in Figure 1, any two computer nodes which are not directly connected by a link need to have their messages relayed by intervening nodes.

However, in contrast with the fixed hypercube topology, the proposed hyperswitch network does not use fixed dedicated links between nodes, but rather establishes these links through explicit switching elements and message routing control logic (adaptive routing algorithm) located in each node. Therefore, the multicomputer system can dynamically route messages by this routing control logic, setting each switching element properly so that the links are shareable among a number of node pairs.

The adaptive routing algorithm with congested path and node/link failure learning and avoidance capability has been developed and implemented in the hardware for the hyperswitch network. As shown above, multiple paths exist between hypercube node pairs. The routing control logic enables messages to select the first available path which does not repeat the previously experienced congested traffic or node/link failures. Thus, known busy or bad links can be avoided by network pruning and the shortest path setup time can be achieved. The links within a hypercube topology are pruned by eliminating paths already attempted unsuccessfully. Then the untried, or virgin network territories are systematically explored to select one of the many possible paths. Once a complete path has been traced and reserved, the path sections are latched to form a complete path for message transmission from the originating node to the destination node. The complete path may span many nodes and, since it is latched by the routing control logic before any data is sent, transmitted data need not enter any intervening node for interaction with that node's computer or memory during its transmission. Data may be pipelined up to the hardware limits for the communication path.

This type of network has the advantages of both static and dynamic networks. The static network characteristics enable the hyperswitch network to maintain locality, increase the number of I/O links, and provide multiple paths between two communicating nodes. The dynamic network characteristics provide high permutability, short message latency, and the capability to explore other computational/problem mapping models.

ARCHITECTURE

The architecture of any one of the identical hypercube nodes is depicted in Figure 2. Each node includes a computation processor (node processor) and its associated node memory. Each node also has associated with it a switching element, containing a plurality of hyperswitch units, a channel interface and crossbar switch and a dispatch processor. The dispatch processor is a memory interface for the hyperswitch units and manages the arrival and departure of messages. Each dimension of the cube provides a separate communication path into and out of each node. Those paths are shown simply as paths 1, 2, through n, where n is the dimension of the cube.

Figure 3 depicts a simplified example of the steps involved in transmitting from a source node through an intermediate node's hyperswitch processor to a destination node. Each hyperswitch processor has two separate operating modes. These two modes are called a "path setup" mode and a "data transmission" mode. For purposes of describing this architecture, we shall focus primarily on the path setup mode, which is the routing mode for each hyperswitch processor.

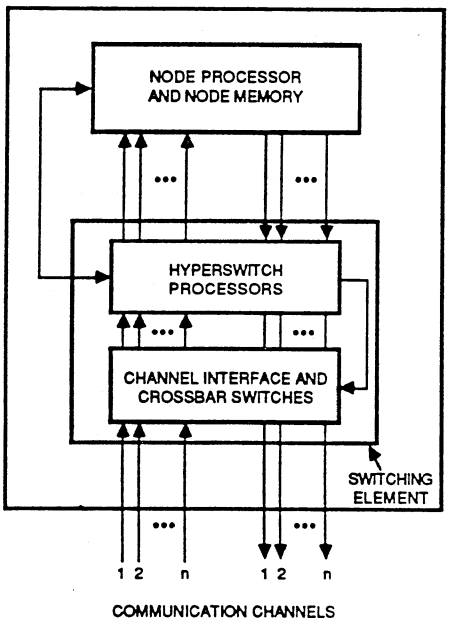

Figure 2. Hypercube Computer Node

The protocol for the hyperswitch network is as follows:

Step (a)- The source node hyperswitch processor sends a header word with the destination node's address embedded in the header to its selected next neighbor hyperswitch processor and reserves the path through which the header travels.
Step (b)- The intermediate hyperswitch processor evaluates the header, reroutes this header to its selected next neighbor hyperswitch processor which is not busy, and then switches the local crossbar switch to connect the header inbound channel to the outbound channel. If all the selected hyperswitch processors are busy, then this intermediate hyperswitch processor will send backtrack status (by status signals) to the previous hyperswitch processor.
Step (c)- Step (b) is repeated until the header reaches its destination node.
Step (d)- The destination hyperswitch processor sends an acknowledgment back (by status signals) to the source hyperswitch processor through the intermediate node's channel interface and crossbar switch circuit. Network connection (a pipeline communication path) is set and information (data messages) can start to be transmitted.

The status signals in the hyperswitch network are two bits of encoded information passed back

through the special status links. These two bits are interpreted differently in the path setup mode and the data transmission mode. In the path setup mode, the status bits are:

00 - Ready to receive header
01 - Ready to receive data
10 - Header backtracked to the previous node
11 - Header error, retransmit header

Initially, the hyperswitch processors are in a path setup mode. After the header is received by a destination hyperswitch processor, the link status is changed to 01. Then the reserved hyperswitch network path enters a data transmission mode. In the data transmission mode, the status bits are:

00 - Break pipeline to source
01 - Ready to receive data
10 - Wait for next data packet
11 - Data error, retransmit data packet

The routing protocol of this architecture involves a header word for each message. A header is shown in Figure 4 and includes a total of thirty-two serial bits. The header will be described in more detail in the next section. Suffice it to say at this point that the header includes a channel history tag portion, a node destination portion, and a distance portion. These portions are used by a hyperswitch processor along with its routing control logic to perform an informed best-first heuristic search algorithm. The destination portion of the header defines the ultimate node's identification for which the message is intended. The distance and history portions of the header allows the hyperswitch processors in the cube-connected network to keep track of the progress of the system as they establish a complete path from an originating to a destination node.

Because the hyperswitch processor is designed to support many different applications, seven communication modes are provided to the users. These modes are all software programmable by changing the bit fields in the message header. These modes are described as follows:

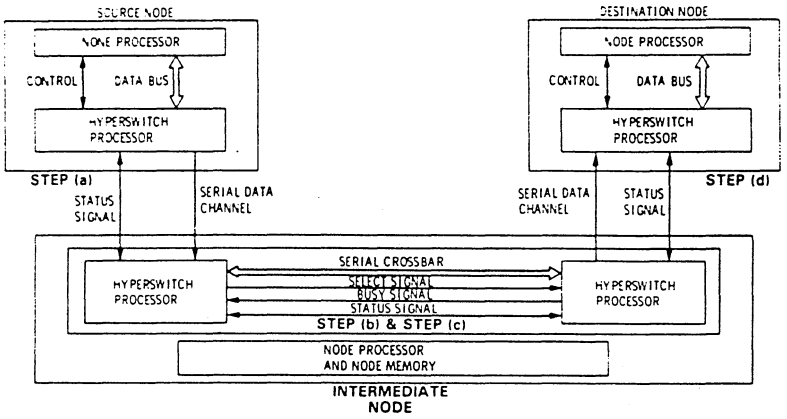

FIG 3. Steps involved in setting up a path.

(a) Circuit Switching: Apply the real-time routing algorithm, where the maximum message length is 3 Kbyte. This is the normal recommended routing mode and is good for unpredictable congestion patterns.
(b) Circuit Switching: Apply the real-time routing with 64-bit message header, with no message length limitation. All intermediate nodes taken have their node ID's recorded in the message header, which is good for system debugging and traffic analysis.
(c) Designated Route Circuit Switching: Routing paths to be used are controlled by the sender. This mode can be used for problems with known communication patterns.
(d) Packet Switching: Compatible with the current Mark III packet switching mode [4]. This mode interrupts all of the intermediate nodes so the packet can be forwarded.
(e) Stage Circuit Switching: Circuit switch as described in (a) is performed until an I/O congested node is reached. The hyperswitch processor then interrupts the node processor for continued message handling.

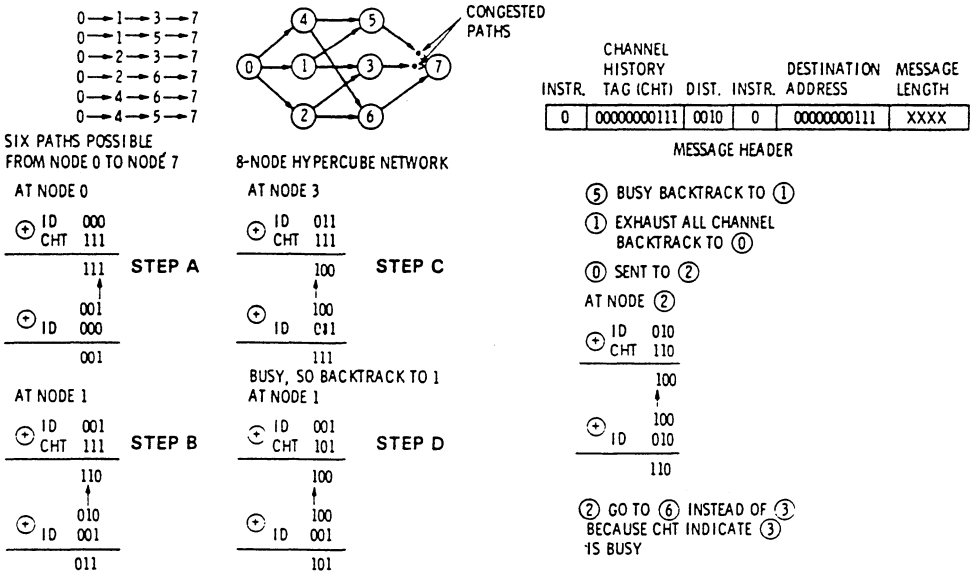

FIG 4. Adaptive routing control flow.

(f) Loop Back: Used for network testing
(g) Near Neighbor Circuit Switching: Used to process synchronize with near neighbor nodes and for message broadcasting.

ADAPTIVE ROUTING MODEL

There are three methods for sources to generate routing headers that specify a noncongested or fault free path. The first is nonadaptive routing, where a source learns of a fault only when the path it is attempting to establish reaches the congested or faulty network component. Notice of the fault is sent to the source, which tries the next alternative path. This approach requires little hardware, but will have poor performance. The second is notification-on-demand adaptive routing. With this method, a source maintains a table of congestion and faults it has encountered in attempting to establish paths and uses this information to guide further routing. Broadcast notification of congestion and faults are made to all sources as they are diagnosed. This method also has poor message-routing performance due to the global broadcast messages adding to the message traffic of the network. The method we use is a real-time adaptive routing network. Here, available or fault-free paths need not be specified by a source, because the routing header can be modified in response to congestion and faults encountered as a path is established. This method can be accomplished within hypercube networks if the nodes have switching elements which are capable of performing the necessary routing header revisions. To perform this header revision, we have developed an informed best-first heuristic search algorithm that is hardwired in the switching elements.

The basic routing model is predicated on the following goal: To reach the destination node using the shortest path in the shortest amount of time (best-first search).

To follow the heuristic search procedure:

(a) Only visit virgin or untried areas of the network following a backtracked routing header status reception. These areas are evaluated by using the channel history tag portion of the header.
(b) Perform a channel latch on the first available channel of all terminal channels that terminate in a destination node. This implies that all penultimate nodes in the network must be attempted for visits.
(c) If an outbound channel is found blocked, all the successor links are considered blocked for the purpose of the channel history tag.

A simplified explanation will be used to describe the optimality of the search algorithms of this routing model. Two algorithms will be explained and are designated as the k algorithm and the k*(k-1) algorithm. In both of the search algorithms, all nodes traversed along the entire message path are visited exactly once.

ALGORITHM k DEFINITION:

GOAL: To latch the first available terminal channel to the destination node using the first accessible penultimate node.

SEARCH CRITERIA: At most one attempt will be made to reach all given penultimate nodes.

TIME COMPLEXITY: Order of (k + C1) hops
where k = number of channels per node log (N)
C1= number of terminal channels busy
Therefore, algorithm k has a performance on the order of k.

ALGORITHM k*(k-1) DEFINITION:

GOAL: To latch first available terminal channel to the destination node using the first accessible penultimate node.

SEARCH CRITERIA: Every possible attempt will be made to reach all the given penultimate nodes.

TIME COMPLEXITY: Order of (k +C2) hops
where k = number of channels per node = log (N)
C2= (number of terminal channels busy)^2
Therefore, algorithm k*(k-1) has a performance on the order of k*k and hence is more exhaustive than algorithm k.

The heuristic search strategy outlined above is an informed search (i.e., it uses the current real-time network history tag to direct the search parameters). This type of search is particularly useful where the number of network links grows exponentially as the number of nodes grow, as seen in Figure 5.

The reliability and performance improvement obtained from a multipath network like the static hypercube interconnection depend upon how effectively the alternate available paths are used by the routing algorithm. The following is a routing complexity assessment where backtracking is part of the search strategy.

STATEMENT: Let H(n) be a hyperswitch network connected using a hypercube topology that utilizes real-time routing with backtracking capability for the message header during the path setup process. Let there exist a node x(i) and let c(i) be the associated collision factor (i.e. the number of colliding message headers at node x(i)) encountered by a message header such that x(i) is a distance of i from the destination node and that x(i) is forced to return the header to the last forwarding node.

TO PROVE: That there exists a limited number of virgin paths (as shown in Figure 5) that may be used by a backtracked message header (according to a search strategy specified by algorithms k and k*k-1) such that it does not encounter the already discovered congested nodes on a message track. This number of

untried paths is a function of i, and n where n is the number of hops between the node evaluating such virgin paths and the destination node.

PROOF: Node x(i) has i! paths to destination node, where i is the number of hops from x(i) to destination node. Node x(i+1) that precedes node x(i) has (i+1)! paths to destination, where (i+1) is the number of hops from node x(i+1) to destination.

When a header backtracks from node x(i) to node x(i+1), it gains (i+1)!-i! = (i)*(i)! paths to destination.

It is assumed that i is selected such that smallest i satisfies: (i) >= (c(i)+1), where c(i) is the collision factor at x(i).

Number of paths outside the i! paths that belong to node x(i)
$=\{(i!)*(i) - ((i-1)!)*(i-1)*(n-1)\}$
$=\{(i-1)!*[i*i - (i-1)*(n-1)]\}$

Number of virgin paths (i.e. VPATHS_K-1(i)) belonging to node x(i) as part of this subgraph as a fraction of (i)*(i!) using algorithm k*k-1:

$=\{(i-1)!*[i*i-(i-1)*(n-1)] * n(n-1)/(i)*(i!)\}$ for permuting factor of n*(n-1) at node x(i) for no header returned; and

$=\{(i-1)!*[i*i-(i-1)*(n-1)]*(n*n-i*i-n+i)/(i)*(i!)\}$ for permuting factor of (n*(n-1)-i*(i-1)) at node x(i) for one header returned.

Number of virgin paths (i.e.VPATHS_K-1(i+1)) belonging to node x(i+1) as part of this subgraph as a fraction of i*(i!) using algorithm k*k-1:

$=\{[(i+1)!*n*(n-1)/(i)*(i!)]$-VPATHS_K-1(i)$\}$ for a permuting factor of n*(n-1) at node x(i+1) for no header returned; and

$=\{[(i+1)!*(n*n-i*i-n+i)/(i)*(i!)]$-
VPATHS_K-1(i)$\}$
for a permuting factor of (n*(n-1)-i*(i-1)) at node x(i+1), for one header returned. Likewise, the permuting factor of 2 is used to compute VPATHS_K-1(i+1) at node x(i+1) when the header is returned twice.

Total number of virgin paths belonging to node x(i+1) that will be attempted when using algorithm k*k-1 are summarized below:

Header_returned at node x(i+1)	VPATHS_K-1(i+1)
0	= (i*n+1-n)*(n*n-n)/i*i = i*i+i when n=i+1;
1	= {[(i*n+1-n)*(n*n-n-i*i+i)]/i*i} = 2*i when n=i+1;
2	= [(2*n(i-1)+2)/i*i] = 2 when n=i+1

NUMBER OF NODES	NO. OF TOTAL PATHS (K!)	NO. OF PATHS TESTED BY ALGORITHM K*(K-1)
16	24	12
32	120	20
64	720	30
128	5040	42
256	40320	56

FIG 5. Paths tested by the K*(K-1) search algorithm as the number of nodes increase.

Total number of virgin paths belonging to node x(i+1) that will be attempted when using algorithm k are summarized below:

Header_returned at node x(i+1)	VPATHS_K(i+1)
0	= (i*n*n+n-n*n)/i*i = i when n=i+1;
1	= [(i*n*n - i*n*i+n-i-n*n+i)/i*i] = 1 when n=i+1

Q.E.D

For example, at node 0, when n=4, i=3,

VPATHS_K-1(i+1)=12 for 0 headers returned
VPATHS_K-1(i+1)=6 for 1 header returned
VPATHS_K-1(i+1)=2 for 2 headers returned

The probability that a path will be successfully formed is based on computing the probability of successfully latching an outbound channel in a series of steps at each intermediate node. Let the process of latching a channel in hop n be denoted by event E(n).

We are interested in the probability that a series of events E(1), E(2), ... E(n) will occur exactly in that order, thus forming a given path between the source and the destination node. Probability events E(1), E(2), ... E(n) will occur in that order in n trials at n intermediate nodes = p(n) = p(E(1) * p(E(2) | E(1)) * ... * p(E(n) | E(1) E(2) ... E(n-1)) where probability p(E(2) | E(1)) is a conditional probability that event E(2) occurs, given that event E(1) has occurred (see Figure 6). This is essentially true since a message header can only try to latch a channel in hop 2 if and only if it has already latched a channel in hop 1. Hence the sequence of occurrence is important in computing the overall probability of success (i.e., E(1) cannot occur before E(2)). Nodes closer to the destination node contribute a progressively decreasing probability of success and hence need to be pruned more exhaustively than earlier nodes (i.e., nodes closer to the source node). In addition, the effective number of retried paths for the header at each node x(i) is given by:

$$1/\Sigma(\text{VPATHS\_K(i)}*p(i)).$$

This is the expected number of retries that can be anticipated at node x(i) with a collision factor of c(i).

ADAPTIVE ROUTING CONTROL EXAMPLE

Having explained in simple terms the operations of the k and k*(k-1) algorithms, a representative example will be used in conjunction with the information supplied above.

Figure 4 depicts a simplified 8-node hypercube network. Also shown are the six paths that are possible from an originating node, node 0, to a destination node, node 7. An originating node formats a message header, which includes an instruction bit at bit position number 0, a channel history tag at bit positions 1-11, a distance field at bit positions 12-15, another instruction bit at position 16, a destination field at bit positions 17-27, and a message length field at the remainder of the header's bit positions. Obviously, timing, framing parity, and other information may be formatted into the serial bit stream of the header. Along with the message header fields, a series of operations are shown in Figure 4. Each node, whether it is a source, intermediate, or destination node, follows the same routing protocol and the same algorithm, whether k or k*(k-1). The hyperswitch hardware routing logic can be adapted to the k or k*(k-1) algorithms by simply changing the "distance" field in the message header. For defining algorithm k, the distance field is set as the number of hops minus 1. For defining algorithm k*(k-1), the distance field is set as the number of hops minus 2.

(FOR N = 16)

PROBABILITY A RESULTING PATH IS FORMED

P = P (1 OF 4 IS AVAILABLE IN HOP 1)

$\times P_{cond}$ (1 OF 3 IS AVAILABLE IN HOP 2 | 1 IS AVAILABLE IN HOP 1)

$\times P_{cond}$ (1 OF 2 IS AVAILABLE IN HOP 3 | 1 IS AVAILABLE IN HOP 2)

$\times P_{cond}$ (1 OF 1 IS AVAILABLE IN HOP 4 | 1 IS AVAILABLE IN HOP 3)

LET PROBABILITY OF LATCHING A CHANNEL = P

THEN

$P(E(1) \cdot E(2) \cdot \cdots E(n)) = [1 - (1 - P)^n] \times [1 - (1 - P)^{n-1}] \cdots \times [1 - (1 - P)]$

FOR EXAMPLE:

FOR P = 1/2, n = 3, N = 8

P (E (1) · E (2) · E (3)) = 7/8 x 3/4 x 1/2

SINCE PROBABILITY OF EVENTS E (1), E (2) . . E (n) PROGRESSIVELY DECREASES WITH INCREASING n

i.e. P (E (1) = 7/8
P (E (2) | E (1)) = 3/4
P (E (3) | E (1) E (2)) = 1/2
} FOR P = 1/2

IT IMPLIES THAT THE FAILURE PROBABILITY RISES IN A SERIES PROGRESSION, ALWAYS BEING

(1 - P) FOR HOP n
$(1 - P)^2$ FOR HOP n - 1
$(1 - P)^n$ FOR HOP 1

HENCE NODES IN THE LAST FEW HOPS CONTRIBUTE VERY HEAVILY TOWARDS PROBABILITY OF SUCCESSFULLY FORMING A PATH. AS A RESULT THEY SHOULD BE PRUNED MORE EXTENSIVELY

FIG 6. Probability that a resulting path is formed.

The source node processor establishes that a message is to go to a particular destination node and selects an appropriate hyperswitch processor. Thereafter, the following nodes' hyperswitch processors in the network perform the routing algorithm operations as shown below.

As an example, assume that node 0 originated a message header and transmitted it to node 1. Node 0's identity is "000" and step A of figure 4 is applicable. The destination node is "111" or node 7. The distance is a total of three hops, so the distance field is set to one less than three or two (k algorithm), which is "010". The channel history tag is initialized to an all 1 condition and its output is "111". The channel history tag and node 0 identity signal are EXclusive ORed and the result is "111". The first one from the right, as shown by step A in Figure 4, is reflected as a term "001". The logic EXclusive ORed operation of the reflected "001" and the node's identity, "000", and designates the resultant "001", or node 1, as the next channel to select. The hyperswitch processor at node 0 responds by moving the header to node 1 and reserving the circuit section between node 0 and node 1.

At node 1 and node 3 the procedure repeats and the header would next go to node 7, its destination, as shown by steps B and C in Figure 4. But, a busy channel from node 3 to node 7 occurs and the header does not wait at node 3. Instead a status signal from node 3 back to node 1 sets the channel history flag in the message header at node 1 to reflect this busy condition and to inform node 1 that the message header must either be forward on another outgoing channel from node 1, or it must be back tracked to node 0. The channel busy status signal from node 3 to node 1 alters the channel history tag portion of the message header at node 1. The channel history tag is now changed to read "101". In short, while the channel history portion previously contained all 1's, the middle 1 was responsible for getting the header to "busy" node 3, so that middle 1 is changed to a zero. At node 1, the operation reflected at step D in Figure 4 occurs and the header is now routed to node 5.

At node 5 the header is again supposed to be routed to node 7 but again encounters a busy channel. All of node 1's outgoing channels are now exhausted, so the message header's channel history portion at node 0 must be updated to reflect that nodes 1, 3, and 5 have been pruned or eliminated from any further consideration for this one link selection attempt. The channel history portion is thus set to "110" and passed to node 2 where it is again EXclusive OR by that node's routing control circuit in the hyperswitch processor. At node 2 the pruned, or eliminated, status of node 3 causes the header to be routed to node 6 rather than to node 3. As shown in Figure 4, a path from node 6 to node 7 is available and thus the message header reaches its destination at node 7.

At node 7, that node's hyperswitch processor will recognize that the message header has reached a destination node. An output from the destination node is sent back through the status signal from node 7 to node 6. A complete path is then latched, because the sections from node 0 to 2 and from 2 to 6 were held as the message header progressed through those nodes' hyperswitch processors. Data transmission, as described from node 0 to node 7, can now occur over a high-speed data line. The data does not

interact with the node processors or memories at nodes 2 and 6 nor does it require any further interaction with the hyperswitch processors at those nodes.

ORGANIZATION OF THE ROUTING CONTROL LOGIC

The hardware organization of the routing control logic, which is part of the hyperswitch unit, is shown in Fig. 7. Outputs from this logic determines whether or not a routing header received in the header storage register, is to be moved forward, is to be backtracked, or is at its destination. A move forward command for the routing header will appear as an output on the channel req. signal from the Exlcusive OR gate. A backtrack signal appears as an output on the set backtrack signal, and a destination signal output is presented at the output of the zero detect circuit.

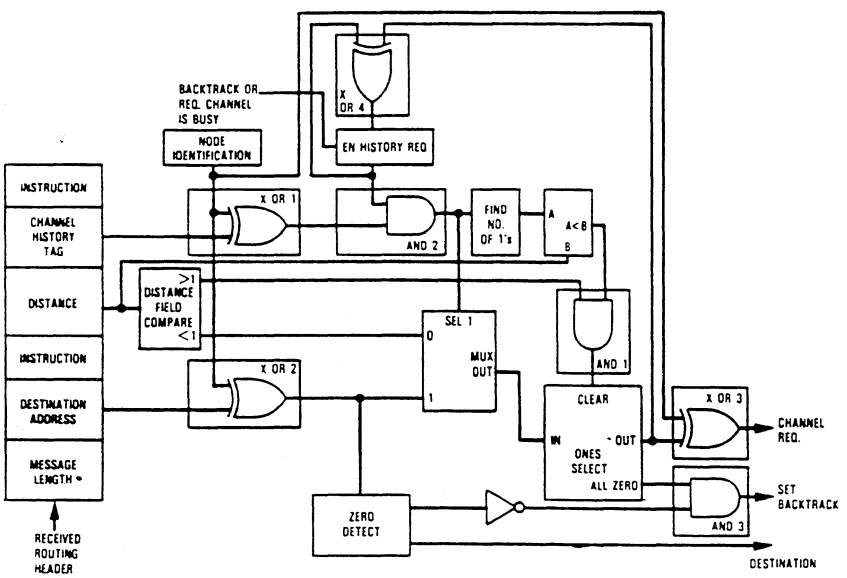

Figure 7. Routing Control Logic

Each node knows its own unique identity and that node's identity is stored in a node identification register, Fig. 7. The node's identity is applied as one term to an EXclusive OR_1 and the other input term is the channel history tag from a received routing header. The node's identity is also applied as an input term to a second EXclusive-OR_2 and the other input is the destination address field also from the received routing header.

As an example, assume that node 0 originated a routing header and transmitted it to node 1 (see step A of Fig. 4). Node 1's identity is 001 and step B of Fig. 4 is applicable. At the node identification register, in node 1, is an identity signal "001" for node 1. The destination node is "111" or node 7. EXclusive OR_1 receives these two inputs terms and the summation is "110" which appears on the input lead to AND_2.

The distance as shown by the message header, Fig. 4, is a total of three hops so the distance field is set to one less than three or two, which is "010". The distance field compare circuit enables the greater-than-one lead which applies a true condition to AND_1.

The history register is initialized to an all "1" condition and its output is "111". AND_2 applies its output "110" to a circuit which will count the number of ones supplied to it. The answer of two is applied to the A term of a difference circuit and the B term is the distance field, or 2. Since 2 is not less than 2, this circuit applies another true condition to AND_1, and in response an output lead from a ones-select circuit is cleared to reflect the condition of an input signal selected by a multiplexer (MUX).

The multiplexer applies the output from AND_2 as an input term to the ones-select circuit. The ones-select circuit selects the first "1" from the right. The first one from the right, as shown by step B in Fig. 4, is reflected as a term "010". EXclusive OR_3 has two inputs terms, the node's identity "001" and the putput from the ones-select circuit. The logic operation of EXclusive OR_3 designates "011" or node 3, as the next channel select. The hyperswitch processor at node 1 responds by moving the routing header to node 3 and reserving the path section between node 1 and node 3. The status lines are used to make that selection as was described earlier.

At node 3 the procedure is repeated by an identical routing control logic. The routing header next tries to go to node 7, its destination, as shown by step C in Fig. 4. A busy channel from node 3 to node 7 is present. Therefore, the status line from node 3 back to node 1 reflects this busy condition and to inform that the routing header must either be forwarded on another outgoing channel from node 1, or it must be back-tracked to node 0. The channel busy status signal from node 3 to node 1 appears as an "EN" signal at the history register. That input signal alters the history register and the channel history tag field of the routing header at node 1. The history register is now changed to read "101". The channel history portion previously contained all 1's and the middle 1 was responsible for getting the routing header to "busy" node 3, so that middle 1 is changed to a zero. EXclusive OR_4 has two inputs terms, its current history register "111" and the output from the ones-select circuit "010". . The logic operation of EXclusive OR_4 designates "101". The logical operations reflected in Fig. 4 occurs and the routing header reaches its destination at node 7.

At node 7, that node's routing control logic, via the zero detect circuit, recognizes that the header has reached a destination node. An output from the destination lead is sent back through the status lines from node 7, to node 6. A complete pipeline data path has now been latched and data transmission can now occur over the path.

EVALUATION OF NETWORK PERFORMANCE

To quantify the performance of the hyperswitch network and the k-family adaptive routing method, we simulated the hyperswitch processor architecture to measure the system performance, including the overhead of message management and memory contention. The simulation was written using the process oriented CSIM[8] simulation environment and the C++ programming language[9].

At the time the simulations were conducted, the architecture of the hyperswitch units was well understood, but the design of the dispatch processor that manages the hyperswitch units was incomplete. One goal of the simulations was to aid the design of the dispatch processor. The performance of the simulated dispatch processor was based on conservative estimates of a VLSI design.

The dispatch processor is a memory interface for the hyperswitch units and manages the arrival and departure of messages. Each node of the simulated hypercube contains a computation processor, a single dispatch processor and sufficient hyperswitch units to form a hypercube interconnection. Table 1 shows salient charactertics of the simulated system, including the overhead of the dispatch processor for sending or receiving a message.

With these parameters, each dispatch processor can support approximately six concurrent transactions (i.e., either six message receptions, six transmissions or some combination of receptions and transmissions). All active transactions are serviced in round-robbin order, and additional simultaneous transactions extend the duty-cycle of the dispatch processor. This lengthens the time of each transaction and the time that each transmission path is reserved.

Message length was fixed at 512 bytes, and unless otherwise noted, message destinations are equiprobable across all hypercube nodes, (i.e., each node is as likely as any other to be the destination for a message). Each simulation was run for 11 batches of 1000 message receptions per node, with the first batch discarded to remove initial transients. Batched statistics were maintained for message latency, transmission queue time, routing overhead, message transmission time, and transmission retries. The results shown are batch means of the gathered statistics. All errors bars indicate 99% confidence intervals.

Several switching algorithms were compared. The simulated switching algorithms are stored and forward message switching with fixed path routing (the method used in existing hypercubes), and circuit switching with both fixed path routing and the k(k-1) member of the adaptive k-family routing methods. To aid comparison, each method used the same simulated hardware while varying either the switching algorithm (i.e. message switched versus circuit switched) or the routing method (i.e. fixed path routing versus k-family routing).

Figure 8 shows the relative performance of the simulated systems and to previously measured[6,7] performance. The graph shows the time for message delivery as a function of the distance that a message travels, measured in hops. Message arrivals for the simulated system are modeled as a Poisson process with an interarrival time of 250 microseconds. The poor performance of the store and forward message switching networks can be traced to the dispatch processor overhead at each hop of the transmission path.

Table 1. Parameters of the Simulated Hypercube Computer

CHARACTERISTIC	VALUE
MEMORY◄► DISPATCH PROCESSOR BANDWIDTH	100 MBYTES/SEC
HYPERSWITCH◄► HYPERSWITCH BANDWIDTH	128 MBITS/SEC
PACKET SIZE	192 BYTES
HYPERSWITCH HEADER SIZE	4 OR 8 BYTES
DISPATCH PROCESSOR HEADER SIZE	12 BYTES
TRANSMISSION OVERHEAD	35 MICROSECONDS
RECEPTION OVERHEAD	60 MICROSECONDS
HYPERSWITCH SWITCHING TIME	0.78 MICROSECONDS/HOP

Although Figure 8 illustrates the performance of the hyperswitch network in absolute terms, the range of data obscures the differences among the circuit switching methods. Figure 9 shows the message latency as the size of the simulated system increases. Expanding the network can be expected to increase message latency by increasing the average distance of each message transmission, since an equiprobable destination distribution is used. Moreover, the increased number of nodes leads to more messages in transit, increasing the contention for the dispatch processor. As the number of nodes increases, the probability of successfully constructing a circuit decreases drammatically for fixed path routing, and the 64 node system is completely staurated with this arrival rate. The k(k-1) circuit network is more resistant to saturation because the routing algorithm search for alternate paths, leading to a higher probability of circuit establishment. Although not shown, the k(k-1) network remained stable for networks up to 256 nodes, the largest network simulated.

Figure 10 depicts the effect of increasing the message arrival rate. The fixed path circuit switching network is saturated for a message interarrival time of 200 microseconds and less, due to an increased probability of each circuit connection failing.

The preceding simulations drew message destination using a equiprobable distribution. Existing applications attempt to exploit spatial communication locality, i.e., they tend to communicate with near nodes rather than distant nodes. Figure 11 shows the result of simulations using a decay distribution

$$\Phi(\iota) = \text{Decay}(d, \iota\max) \bullet d^{\iota}, \quad 0<d<1$$

where $\iota\max$ is the diameter of the network, d is a selected decay factor, and Decay(d, $\iota\max$) is a normalizing constant chosen such that

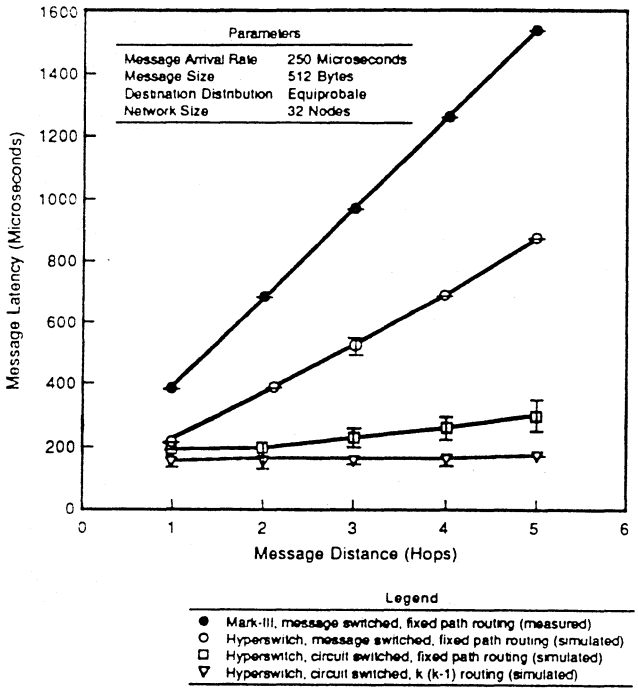

Figure 8. Comparison of Network Performance

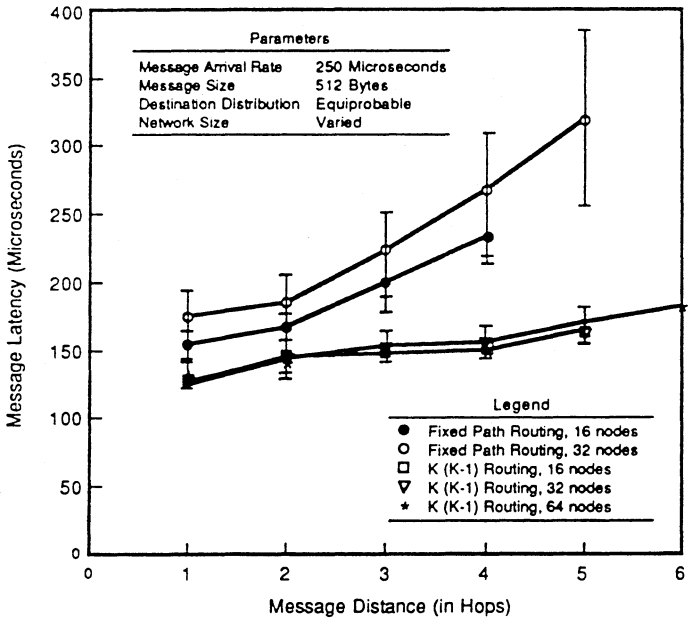

Figure 9. Circuit Switch Performance With Increased Network Size

$$\text{Decay}(d, \iota\max) \bullet \sum_{\iota=1}^{\iota\max} d^{\iota} = 1$$

This model reflects the diffusion of work from areas of high utilization to areas of lower utilization. Small values of d yield localized destinations, while large values give antilocal destinations. The simulations in Figure 11 have a diameter, ιmax, of five and decay

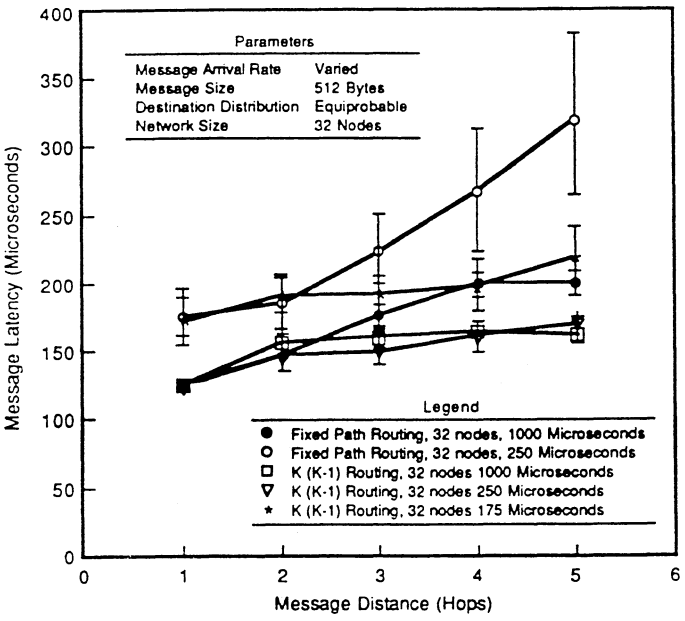

Figure 10. Effects of Varied Message Intervals

rate, d, of 0.2. Using this distribution, 96% of all messages travel two hops or less, and both fixed path and k(k-1) routing show statistically equivalent latency, although the k(k-1) routing algorithm delivers the remaining messages in less time. The wide confidence intervals for messages traveling three or more hops are due to the limited number of samples at those distances.

CONCLUSION

The reliability and performance improvement obtained from a multipath network depend upon how effective the alternate available paths are used by the routing algorithm. The hyperswitch network significantly improves the communication performance of the hypercube computer. It fully utilizes all the hardware capability of the hypercube multipath links and has the ability, with low overhead, to reroute with backtracking. Special features of this network are: low connection complexity, high fault-tolerance, ease of message routing, and very low message latency, as well as suitability for distributed computer architecture. All of these are general requirements which should be satisfied by an interconnection network.

Detailed simulation results show that the k-family routing method is more stable than fixed path routing, and that circuit switched networks have lower latency than message switched networks if there is a large overhead for each message transmission.

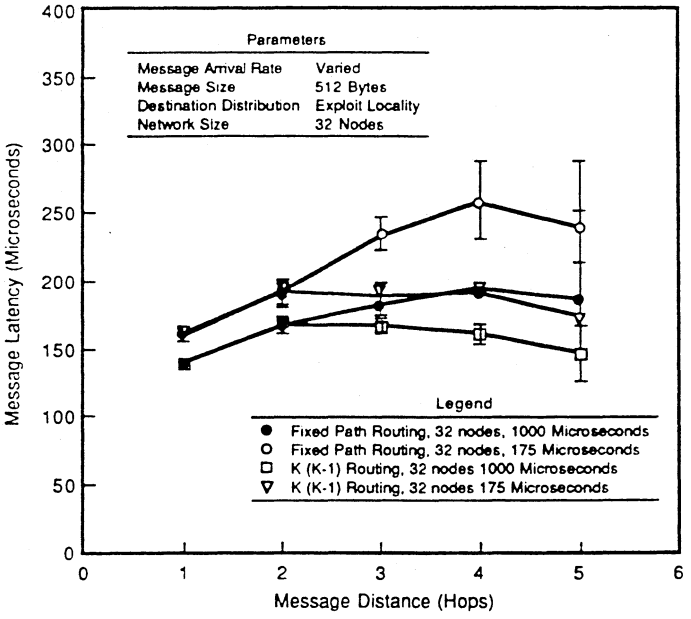

Figure 11. Effect of Message Locality

Photograph of the Hyperswitch Network Testbed

Analysis of the simulation results show that the k-family routing method is consistently able to send messages with a lower message latency than fixed path circuit switching methods.

The hyperswitch network has been designed by the Caltech/JPL Concurrent Processor Group for a next-generation concurrent computer. A 48,000 gate VLSI hyperswitch chip has been designed and implemented and can support, with a companion interface chip, up to 400Mbits/s throughput. The adaptive routing algorithm is implemented in the chip as hardwired logic to minimize the routing decision latency. In the initial test bed, we intend to use the hyperswitch chips in an existing hypercube computer.

ACKNOWLEDGMENTS

This research was supported by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and the United States Air Force, Electronic Systems Division.

REFERENCES

[1] W. Dally, C. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks" IEEE Trans. on Computer, May 1987.

[2] W. Dally, C. Seitz, "The Torus Routing Chip," Department of Computer Science, California Institute of Technology, Technology Report, 5208:TR:86, January 1986.

[3] M. C. Pease, "The Indirect Binary n-cube Microprocessor Array," IEEE Trans. on Computers, May 1977.

[4] J. Peterson, J. Tuazon, M. Pniel, and D. Lieberman, "The Mark III Hypercube Ensemble Concurrent Computer," Proc. of the 1985 International Conf. on Parallel Processing

[5] C. Wu, and T. Feng, "On a Class of Multistage Interconnection Network," IEEE Trans. on Computer, August 1980.

[6] D. Grunwald and D. Reed, "Benchmarking hypercube hardware and software". In M. Heath, editor, Hypercube Multiprocessor, pages 169-177, Society for Industrial and applied Mathematics, 1987.

[7] D. Reed and D. Grunwald, "The performance of multicomputer interconnection network". IEEE Computer, 20(6):63-73, June 1987.

[8] H. Schwetman, CSIM Reference Manual (Revision 9). Microelectronics and Computer Technology Corperation, 9430 Research Blvd, Auston TX, 1986.

[9] B. Stroustrup, The C++ Programming Language, Addison- Wesley, 1986.

[10] K. Stevens, S. Robison, A. Davis, "The Post Office Communication Support for Distributed Ensemble Architectures", Ai Technical Report, Feb. 1986.