Flexible oblivious router architecture

by J. Park
S. Vassiliadis
J. G. Delgado-Frias

In this paper we present a router architecture that accommodates a family of oblivious routing algorithms. The architecture is suitable for current technologies, and it is intended for multiprocessor and massively parallel systems. Via the proposed architecture, we suggest that general-purpose routers can be designed to accommodate a variety of multiprocessor interconnection networks. In particular, the routing algorithms of those interconnection structures that can be classified as trees, cubes, meshes, and multistage interconnection networks can be accommodated with a flexible, easily implemented architecture. Our investigation strongly suggests that a common design can satisfy at least forty network topologies with the introduction of a few instructions that are very simple to implement. The overall conclusion is that general-purpose cost-effective routers can potentially be designed that perform equally as well as customized routing logic, suggesting the possibility of a common router for multiple interconnection networks. Furthermore, the proposed architecture provides programming capabilities that allow other oblivious routing algorithms not considered in our investigation to be accommodated.

1. Introduction

Interconnection networks have an important role in the design of parallel systems; they influence the performance of such systems. A key component of an interconnection network is the router, which is responsible for handling different switching technologies, flow controls, and routing algorithms [1]. The router can be categorized into two types, depending on the network topology supported. The first type of router relates to the networks with a fixed topology. In this case, the router performs only one routing algorithm, which depends primarily on the chosen topology. Examples within this category include routers such as the Torus routing chip [2], the router supported in the iPSC/2 [3], and the Cosmic Cube router [4]. The second type of router relates to networks with physically reconfigurable topologies. In such systems, the router is able to execute multiple routing algorithms or an algorithm that can handle multiple topologies. The Intel®/CMU's iWARP [5] and the INMOS transputer [6] are examples of this type of router. In this scheme, there are two approaches in routing: source routing and lookup-table routing [7]. In source routing, the source node determines the routing paths on the underlying network topology. The packet used to communicate between processors must carry the complete routing information in the header. In lookup-table routing, the router has an entry in a table that indicates which output channel must be used to reach each destination node.

If the first approach is used, different routers must be developed for different network topologies. The implication here is that because the router has no flexibility, it cannot be used as an "off-the-shelf" component for the design of parallel systems. The second type of router resolves the previously mentioned problem, since it can be used as an off-the-shelf component to satisfy various network requirements; however, it introduces some new problems. In particular, in the source routing scheme, network

0018-8646/95/$3.00 © 1995 IBM

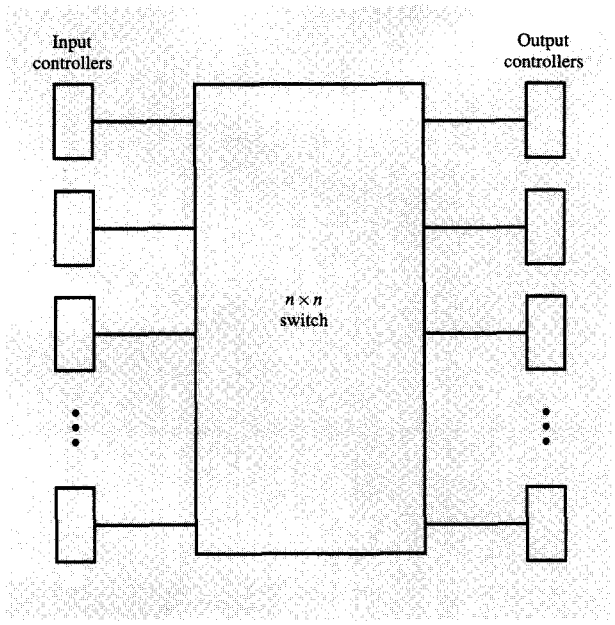


Figure 1

Logical structure of a router.

bandwidth can be wasted when the complete routing information is included in the header packet. The lookup table can be viewed as a flexible scheme. However, the size of the lookup table may grow in proportion to the size of the network, and it may be possible that, for a large network, the size of the lookup table may become prohibitively large, especially when the destination address spaces are not easily partitioned into contiguous ranges.

In this paper, based on the investigation of forty network topologies representing five families of networks, we propose a flexible router architecture that can support fixed and/or reconfigurable network topologies. Our proposal can be viewed as a RISC-like general-purpose router architecture which has a small set of instructions to perform various routing algorithms. Our work represents the first attempt at proposing a RISC-like general-purpose router architecture, and our goal is to build an off-the-shelf general-purpose router that can be used in a wide range of network topologies. In this paper, the term *architecture* denotes the attributes of a system as seen by the programmer (i.e., conceptual structure and functional behavior); it is distinct from the organization of the dataflow and the physical implementation of the machine.

The paper is organized as follows. In Section 2, the background and the direction of our investigation are briefly explained. In Sections 3 and 4, the flexible router architecture and its general operation are introduced. Examples of routing programs using the proposed router architecture are given in Section 5. Section 6 reports various program characteristics, and Section 7 contains concluding remarks.

2. Routing algorithms

In our investigation, we have considered oblivious routing algorithms for the determination of a general-purpose router architecture. An oblivious routing scheme always produces the same communication path, given the same source and destination address. The architecture is developed around topologies that have been used extensively in the design of parallel systems. The routing algorithms we have considered have been divided into five families: tree, cube, mesh, multistage interconnection networks, and others (i.e., networks that do not fall into any of the preceding four types of network topologies). We considered for direct implementation the functions required by the routing algorithms of twelve interconnection networks classified as trees, six classified as cube networks, five as mesh, ten as multistage interconnection networks, and eight that were not classified with the other network families. The major objective of the investigation was to make instructions simple to implement and keep the number of instruction sets as small as practicable. Furthermore, we were interested in providing an architecture that allows parallelism in its implementation. For the forty interconnection networks we considered, we used optimal routing algorithms when possible. Consequently, our first criterion was to define a set of instructions that would perform the algorithmic requirements. Since the entire study is rather lengthy to report here, the interested reader is referred to [8] for a detailed discussion.

Identifying algorithmic requirements, however, is only a part of the operations of an architecture. An architecture must also support the routines necessary for "handshaking" with other units, and must provide flexibility of operations. We support these capabilities with additional functions and protocols described in detail later.

3. Flexible router architecture

Generally speaking, the router consists of three major parts: n input controllers, an n-input, n-output switching mechanism, and n output controllers. The input controller receives packets from the paired output controller of a neighboring router, performs a routing algorithm based on the routing information in the packet, and determines the output controller through which packets are to be forwarded to another neighboring router. The n-input, n-output switch connects n input controllers and n output controllers. The output controller sends packets to the paired input controller of the next router. **Figure 1** shows the logical structure of a router.

The functions of the $n \times n$ switch and the output controllers are the same for all routing algorithms, so the flexibility of the router depends entirely on how many different algorithms the input controller can process.

The logical functions of the input controller can be broken into three major blocks: input port, routing algorithm handler, and packet flow controller, as shown in the block diagram of **Figure 2**. The input port (incorporated in the port controller) is responsible for carrying out the communication protocol for the reception of packets from a neighboring router. The input port extracts the header portion of the packet and transfers it to the routing algorithm handler, while forwarding data bytes from the packet to the packet flow controller. The routing algorithm handler executes the routing program on the header information and sends the result (output controller number) to the arbiter of the $n \times n$ switch and also to the packet flow controller. The packet flow controller stores the data bytes sent from the input port in its buffer and waits until the routing algorithm handler sends the address of the output controller to which the data bytes should be routed. Depending on the method of assigning buffers in the packet flow controller to the data bytes, there are three well-known flow control schemes: store-and-forward, virtual cut-through [9, 10] and wormhole [7]. Once the packet flow controller is notified of the output controller through which the data bytes should be forwarded, it waits for the connection to the switch. Once it receives the acknowledgment from the arbiter of the $n \times n$ switch, it transmits the data bytes to the output controller.

A flexible router must have a routing algorithm handler that can handle multiple routing algorithms. In this section, we propose a novel routing algorithm handler architecture that provides support for all of the required instructions and manipulations of data to manage multiple routing algorithms. The architecture of the port/packet flow controller is reported elsewhere [11, 12]. When necessary, the functions and protocols of the port controller are discussed to some extent.

• *Routing algorithm handler architecture*

To support the algorithms directly, we have identified twelve general-purpose, easily implementable instructions (**Table 1**) that satisfy the requirements. The particular instructions required by the routing algorithm families of the interconnection network are presented in **Table 2**; OUT, CMP, and BC instructions are not listed in this table because they are used in all networks. A second instruction set, the control instruction set, is shown in **Table 3**. The four instructions in this set are not needed for the routine algorithms themselves, but are used by the communication controller to initialize the algorithm handler and load the appropriate routing program in cases where multiple programs are available. A detailed description of all of the algorithms, types of interconnection networks, and routing programs can be found in [8].

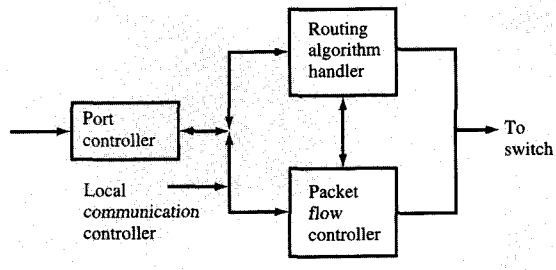


Figure 2

Logical structure of an input controller.

Table 1 General instruction set.

		Format	*Operations*
ALU instructions			
ADD	R1, R2, R3	RR	R3 = R1 + R2
SUB	R1, R2, R3	RR	R3 = R1 − R2
CMP	R1, R2	RR	Compare R2 to R1 (sets condition code)
AND	R1, R2, R3	RR	R3 = R1 AND R2
XOR	R1, R2, R3	RR	R3 = R1 XOR R2
PLO	R1, R2	RR	R2 = position of leading one bit in R1
Shift instructions			
SHR	R1, R2, R3	RR	R3 = Shift right R1 by (R2)
SHL	R1, R2, R3	RR	R3 = Shift left R1 by (R2)
Data transfer instructions			
MOV	R1, R2	RR	R1 = R2
Control instructions			
BC	Address	MI	Branch on condition
OUT	Channel no.	I/R	End of program
Communication instructions			
MSG	R1, R2, R3	RR	Send message to local processor

R# = a register and its number.
R2 = content of register R2.

General instruction set

As indicated earlier, the general instruction set is used to perform routing algorithms. All operands are either stored in registers or made available within the immediate field of the instruction. Most of the arithmetic and logical instructions have three operand fields, to reduce the demand for registers and help reduce the number of data

Table 2 Instructions required for interconnection networks.

Network name	*Instructions required*	*Reference*
Binary tree	AND PLO XOR	[13]
Fat tree	AND PLO XOR	[14]
Flip tree	AND PLO XOR MOV	[15]
Binary tree with a full ring	AND PLO XOR MOV SUB SHIFT ADD	[16]
Binary tree with a half ring	AND PLO XOR SUB SHIFT	[16]
Hierarchical mesh	AND SUB ADD	[17]
Hypertree	AND PLO XOR ADD SHIFT SUB	[16]
Diamond network	AND PLO XOR	[18]
KYKLOS structure	AND MOV PLO SUB SHIFT XOR	[19]
Tree of meshes	AND XOR PLO SUB	[20]
Quad tree	AND XOR PLO ADD SHIFT	[21]
Hypercube	AND PLO SHIFT	[22]
Folded hypercube	AND PLO SHIFT ADD	[23]
Banyan hypercube	AND XOR PLO	[24]
Spanning multiaccess channel (SMAC)	AND	[25]
Base-*m* *n*-cube	AND SHIFT	[26]
Cube-connected cycles (CCC)	PLO	[27]
Mesh array		[27]
Torus network	AND SUB	[2]
k-ary *n*-cube	AND SUB	[28]
Hexagonal mesh	AND SUB	[29]
GNNM hypercube	AND SUB	[30]
Omega network	AND SHIFT	[31]
Delta network	AND SHIFT	[31]
Baseline network	AND SHIFT	[32]
Benes network	AND SHIFT	[33]
Shuffle exchange network	AND SHIFT XOR	[34]
Augmented data manipulator net (ADMN)	AND SHIFT	[35]
Generalized cube network	AND SHIFT	[31]
Extra stage network	AND SHIFT	[36]
Rectangular SW banyan network	AND SHIFT	[37]
Gamma network	AND SHIFT	[38]
Ring network	AND SUB	[8]
Completely connected network		[39]
Pyramid network	AND ADD	[31]
Chordal ring network	SHIFT	[40]
Crossbar		[41]
Cube-connected cycles w/virtual channel	AND SHIFT XOR MOV ADD	[34]
k-ary *n*-cube w/virtual channel	AND SHIFT XOR MOV ADD	[34]
Shuffle exchange net w/virtual channel	AND SHIFT XOR MOV ADD	[34]

Table 3 Control instruction set.

Instructions		*Format*	*Operation*
LPG	R1, R2, R3	RR	Load program
LSR	R1, R2	RR	Load status register
LR	R1, R2	RR	Load general register
ECP	Address	I/R	End of control program

transfers between registers. All instructions have equal length, which is assumed to be 32 bits. Each instruction has one of three formats: register to register (RR); immediate or register (I/R); or immediate with mask (MI). In the RR format, all operands are in registers. In the MI format, I is the absolute address and M is the mask value. In the I/R format, there are two mode bits, m1 and m2. The m1 bit indicates whether R1 or I is to be used for the OUT instruction. The m2 bit is used for the control instruction ECP; it indicates whether it has the I as its operand, or no operand. **Figure 3** shows the three instruction formats with the assigned bit positions.

Additional information regarding the instructions can be found in Table 1. The specific definition of the instructions is reported in [8]. The mnemonics and the functions of most instructions are self-explanatory, except for the instruction PLO (find position of leading one bit). The PLO instruction is used frequently in the routing programs of the tree and cube networks. As the name implies, it finds the position of the leading one bit, starting from the most significant bit position down to the least significant position, and leaves the result of the positional value in the target register. An example is the instruction PLO R1, R2; R1 = 00001000, where R1 is an 8-bit register labeled from 1 to 8, and the result is (R2 = 00000101) because the leading one bit was the fifth bit in R1.

Addressing All instructions in a program (except BC) imply sequential execution of the program. The branching instruction is the only one that may use an address in memory to determine the instruction to be executed. For simplicity of the architecture, there is only one type of addressing, and that is absolute. The absolute address is the address assigned to a memory location. An absolute address does not require any transformation of address to access memory.

Instructions and condition code The condition code is set only by the CMP (compare) instruction and tested by the BC (branch on condition) instruction. Overflow conditions are ignored and are not recorded anywhere. No other flag bits, such as "result equal to zero," are set as a result of arithmetic/logical, shift, or PLO instructions.

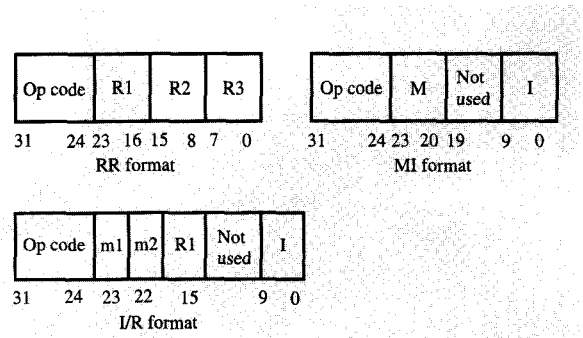


Figure 3

Three instruction formats.

Control instruction set

The instruction set we have described in the previous section can be used to determine the behavior of the routing algorithm handler. No additional instructions have to be developed for the design of a router. The router can be initialized with proper settings of its memory and states (to be discussed). If some of the functions necessary for the proper operation of the device are performed before the router is employed, the control instruction set need not be implemented. The control instruction set is introduced primarily to perform functions such as initialization, cooperative operations with a local processor, potential operation modes that allow adaptive routing, or execution of algorithms requiring complex functions, etc., which increase the flexibility of the router architecture. The control instruction set comprises four instructions, as shown in Table 3. The LPG (load program) instruction interrupts the routing algorithm handler, sets the state of the handler to privileged, and initiates the transfer of the routing program from an external source (usually the local communication controller, described later) to the memory of the handler. It has three operands. The first, R1, contains the address where the routing program is stored in the local communication controller. The second operand, R2, contains the address where the routing program should be loaded in the routing algorithm handler. The content of the third operand (R3) is a counter which specifies the size of the routing program to be loaded. The program-loading operation is performed until counter R3 is equal to zero. The LSR (load status register) instruction loads the new content into the status register. The LR (load register) instruction puts the new value into one of the general registers of the routing algorithm handler. The ECP (end control program) instruction terminates the control program and returns the state of the routing algorithm handler from privileged to normal; it may or may not set the instruction address depending on the value of the m2 field in the instruction. The execution of the control instructions is described further in the subsection on the local communication controller.

Status register

The status register in the routing algorithm handler keeps the information required for the execution of the active program; its implementation is always required. It includes the instruction address, the condition code, the interrupt code, the protects, the execute bit, and the state of the routing algorithm handler. **Figure 4** shows the bits assigned to each field. The content of the status register is set by the control instruction LSR.

States The routing algorithm handler has two states: normal and privileged. Each state has two modes: operating and idle. The control instructions are executed in the privileged state and the general instructions in the normal state. The interrupt is executed only when the routing algorithm handler is in the idle mode. After the completion of each routing program, the routing algorithm handler enters the idle mode. The mode of the routing algorithm handler changes from the idle mode to the operating mode when the input port transfers header information to the registers of the routing algorithm handler and causes an end-of-header packet interrupt. The state of the routing algorithm handler changes from normal to privileged when the local communication controller causes an execute control program interrupt. The MSG instruction can also change the state from normal to privileged.

Condition code The condition code is set as a result of the CMP (compare) instruction and recorded in the condition code field in the status register. The meanings of the bits in the field are as follows:

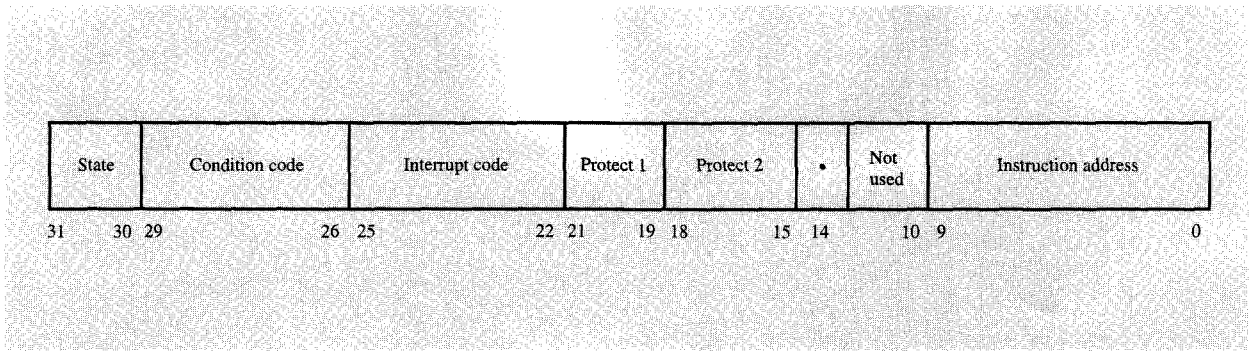


Figure 4

Status register format.

Condition code

0000 operands are equal;
0001 first operand is low;
0010 first operand is high;
0011 undefined.

Interrupt code There are five types of interrupt in the routing algorithm handler. The interrupt code field in the status register records types of interrupt as follows:

0000 hardware failure;
0001 input port;
0010 local communication controller;
0011 program check;
0100 instruction not implemented.

Protect fields As shown in Figure 4, the instruction format can accommodate up to 256 registers. However, the number of registers may be limited to fewer than 256 by the hardware technology in which the architecture is implemented. To avoid unnecessary architecture, once the number of registers that can be implemented has been decided, the protect1 field is used to indicate that number. The protect1 field contains three bits and can represent numbers ranging from 0 to 7. If a register number in the instruction is given as the binary number $r_7 r_6 \cdots r_0$, a value i in the protect1 field represents the value $j = i$ for which all r_j, $j \geq i$, must be zero—i.e., the maximum number of registers allowed by the current implementation. If the register number in the instruction exceeds this limit, the program check interrupt will occur. The protect2 field is used in a similar way. It determines the actual addressing space that a program can use and indicates whether the address exceeds the implemented address range.

Execute bit It indicates whether the execution occurs from the local memory of the routing algorithm handler or from an external device.

Instruction address The instruction address field contains the address of the next instruction for either control instructions or general instructions. When the state bit in the status register indicates that the routing algorithm handler is in the normal state, the instruction address represents the next instruction address for general instructions. Otherwise, it represents the next instruction address for the control program.

Address generation and data format Execution of instructions by the routing algorithm handler involves generating the addresses of instructions and operands. When an instruction is fetched from the location designated by the current status register, the instruction address is increased equally after the execution of each instruction. For the branching instruction, the address of the next instruction is either the address of the next instruction in the sequence or the address specified in the I field in the instruction, depending on the branching decision made in the branch instruction. All instructions treat data as only one type, two's-complement numbers. In a two's-complement number, the most significant bit is used as the sign bit indicator. The logical structure of the routing algorithm handler is shown in **Figure 5**.

- *Storage*

Registers There are up to 256 registers in total. For simplicity in program writing, we consider two types of registers: general registers, which store values or results of computations, and constant registers, which hold constant values used in the routing program.

Memory The routing program is loaded into memory and executed from there. The routing algorithm handler does not allow any operand to be stored in the memory. The word length of the memory is 32 bits.

Interrupts The interrupt facility allows the routing algorithm handler to react to hardware failures in the router, monitor the program execution status, initiate the routing program stored in the memory, and also communicate with the local processor through the local communication controller (described later). We discuss the following types of interrupts.

Interrupt from input port When the input port receives a packet, it stores the header part of the packet into the predetermined register(s) in the routing algorithm handler. Then, it causes an interrupt to the routing algorithm handler so that the routing algorithm handler may begin executing the routing program on the newly arrived header information. When the interrupt occurs, the routing algorithm handler executes the routing program if the routing algorithm handler is in the normal state. Otherwise, the interrupt remains pending until the state changes to normal.

Interrupt from local communication controller The local communication controller is an interim location for communication instructions between the local processor and the input controllers. The local processor sends data, the routing program, and the control program to the local communication controller and instructs the local communication controller to notify the input controller that the control instructions should be executed. The local communication controller performs this operation by causing an interrupt to the input controller. When the interrupt occurs from the local communication controller, the input controller changes its state to privileged and executes the control instructions stored in the local communication controller.

Interrupt from program A program check interrupt occurs when the register number exceeds the allowed number or when an unimplemented instruction is received. The routing algorithm handler sends the program check message to the local processor and halts the program.

Interrupt from hardware A bus error or hardware failure from any component can cause an interrupt. The routing algorithm handler reports the error to the local processor and halts the program.

- *Local communication controller mechanism*

In this section, we describe the concept and the conceptual structure of the local communication controller. The

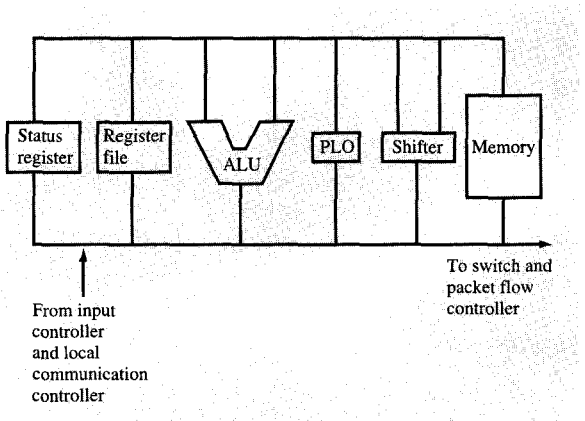


Figure 5

Conceptual structure of routing algorithm handler.

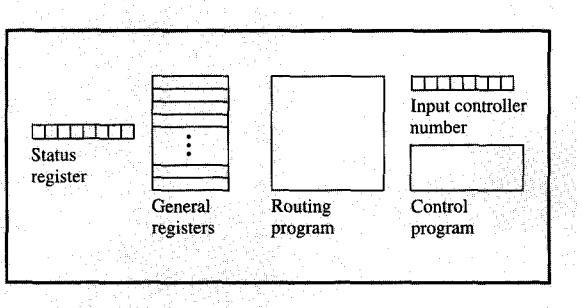


Figure 6

Reserved spaces in local communication controller.

facility need not be implemented if the control instruction set is not considered for implementation. This facility can be implemented either in hardware or in software. If the local communication controller is implemented, it is responsible for handling the communication between the local processor and the input controllers. The local processor sends programs and data to the input controllers via the local communication controller and vice versa. The communication between the local processor and the local communication controller is carried out through messages. The interrupt mechanism is used for communication between the local processor and the input controllers. The local communication controller reserves spaces for the status register, the general registers, the constant registers, routing programs, and control programs. Those spaces, except for the control instruction space, are replicas of storage in each of the input controllers. **Figure 6** shows

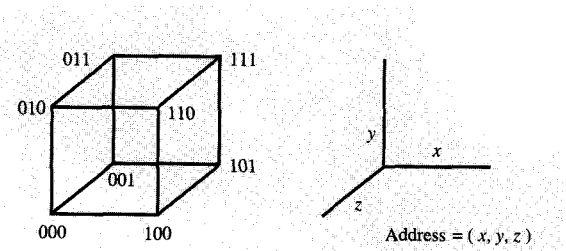


Figure 7

Hypercube network.

```
R1: Routing tag;
C1 = 0;
C2 = 1.
1.              CMP     R1, C1
2.              BC      B'1000', processor (branch if equal)
3.              PLO     R1, R2
4.              SHL     C2, R2, R3
5.              XOR     R3, R1, R1
6.              OUT     R2
7. processor:   OUT     4
```

Figure 8

Hypercube routing program.

reserved spaces in the local communication controller. Although the physical structure of the controller may not be necessary, its implementation may be highly desirable for performance reasons.

Communication between local processor and local communication controller

When the processor has to send data or routing programs to one or all of the input controllers and the structure is implemented in hardware, it first sends messages that contain such data to the local communication controller. The data and instructions are stored in the reserved spaces. The input controller number and the control instructions are also transferred to the communication controller. If it is desired that the routing algorithm handler execute the routing program from the communication controller, the status register must reflect this correspondence. Otherwise, the control program must be loaded into the routing algorithm handler. The control program is loaded at the beginning of the address space reserved for control instructions. The last address of the control program is used by the local communication controller to modify the addressing of the routing program, if necessary, by a constant offset so that no conflict of the addresses occurs.

Communication between local communication controller and input controllers

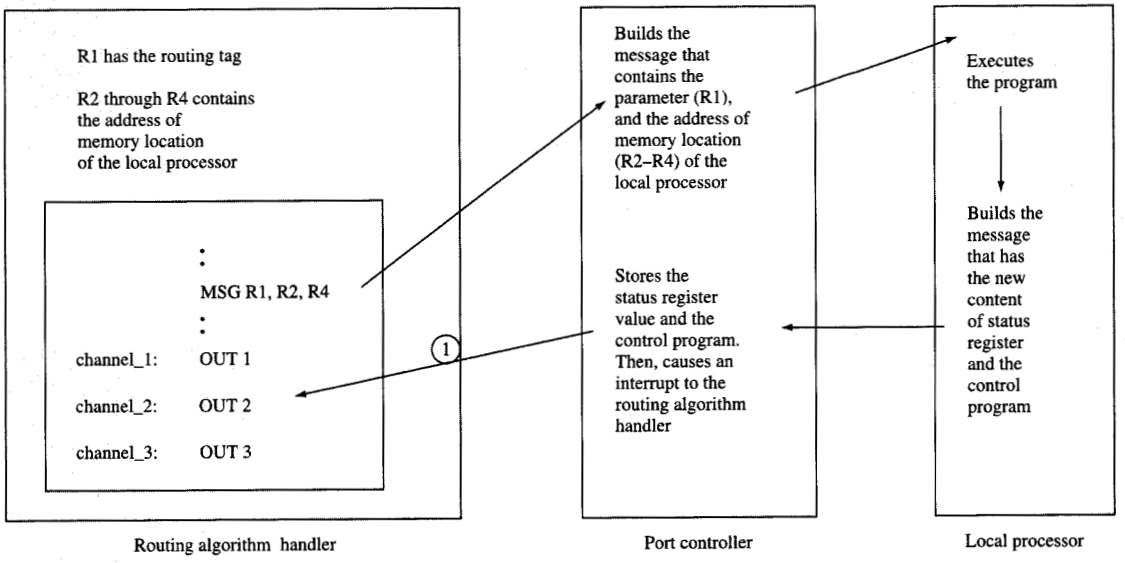
In all cases in which it is used, the local communication controller causes an interrupt to notify the input controller, which is supposed to receive data and/or the routing program, that the data/routing program has arrived. The interrupted input controller then changes its state to privileged and executes the control instructions stored in the local communication controller or the control instructions in its memory, depending on the status bit which reflects where the control instructions reside. As indicated earlier, the starting address of the control instructions is always the beginning of the address space for both techniques. By executing control instructions, the input controller may load data into the status register and/or general registers, or it may load the routing program into its memory.

4. Operating environment

Once the type of interconnection is decided for a parallel system, the routing program for the routing algorithm handler can be developed in either the host computer or the local processor. The executable program is downloaded to the routing algorithm handler in each input controller.

Routing algorithm handler initialization and downloading of data

The registers of the routing algorithm handler may hold information that is used repeatedly in the routing program. An example of such information is the address of the source node where the router is attached. This information is needed frequently in many of the routing programs; since it is known in advance, it does not have to be computed in the routing program. The other data that must be downloaded include the content of the initial status register for each routing algorithm handler. The value for the protect fields is assigned appropriately, depending on the number of registers to be supported. The initial instruction address for the routing program should be determined and set accordingly. The control program for initialization must also be present in the host computer or the local processor. The control program needed for the initialization is the sequence of LPG, LR, LSR, and ECP if it is assumed that the control instruction set is implemented. Once all the necessary data are ready, the processor transfers the data to the local communication



(1) The next instruction address in the new status register was channel_2.

Figure 9

Sequence of communication initiated by the MSG instruction.

controller, assumed for simplicity here to be implemented in hardware. After sending all the data, the processor sends another message that indicates the end_of_data. Upon receiving the end_of_data message, the local communication controller causes an interrupt to the routing algorithm handler in the input controller. Then the routing algorithm handler executes the control program loaded in the local communication controller and initializes the status register, the general registers, and its memory. The last instruction in the control program is the ECP instruction, which puts the routing algorithm handler into the normal state and stopped mode, waiting for header information to begin operating. Multiple input controllers using this example scheme can be initialized within the same environment by changing the input controller number and repeating the process. A parallel initialization is possible with proper hardware support and parallel loading of the program to all input controllers.

Execution of the routing program

Once the routing algorithm handler is initialized, it is in the normal state and stopped mode. The mode changes from stopped to operating when the input port causes an interrupt that notifies the end of header information transfer to the input controller. Then the routing algorithm handler executes the routing program on the new header information. The last instruction in the routing program is the OUT instruction, which sends the result of the routing program to the $n \times n$ switch as well as to the packet flow controller. Then it changes the operating mode to stopped.

5. Routing program examples

In this section, we show examples of routing programs written using the routing algorithm handler instructions. In particular, in the first example, we describe the operation of a routing algorithm entirely supported by the router. In the second example, we demonstrate the flexibility of the proposed architecture by showing an example which requires both the processor and the router operating in synergy. In the third example, we show how to support reconfigurable topologies using the Inmos [6] table-lookup scheme, for the interval labeling method with the proposed architecture.

• *Example 1 (routing in hypercube)*

Figure 7 shows a hypercube network. For the given pair of source $(s_{n-1}s_{n-2}\cdots s_0)$ and destination address

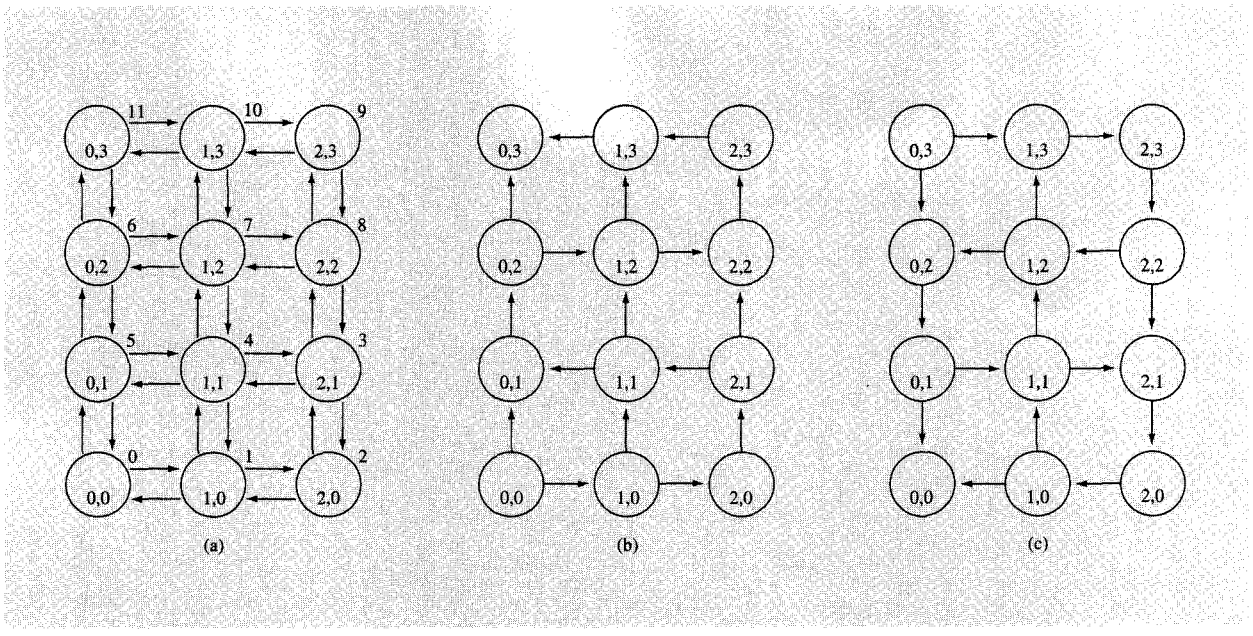


Figure 10

The labeling of a 4 × 3 mesh: (a) physical network; (b) high-channel network; (c) low-channel network [42].

$(d_{n-1}d_{n-2}\cdots d_0)$, the routing tag $(r_{n-1}r_{n-2}\cdots r_0)$ is computed as

$$r_{n-1}r_{n-2}\cdots r_0 = (s_{n-1}s_{n-2}\cdots s_0 \text{ XOR } d_{n-1}d_{n-2}\cdots d_0).$$

Then, using this routing tag, each router performs the following algorithm:

- If $(r_{n-1}r_{n-2}\cdots r_0$ = all zeros), then forward the message to the local processor;
- Else find the position, i, of the leading one bit in $(r_{n-1}r_{n-2}\cdots r_0)$;
- Set $r_i = 0$;
- Send message in the ith direction.

The routing program developed for the routing algorithm handler that performs this hypercube routing algorithm is shown in **Figure 8.** The R1, C1, and C2 are general registers containing the routing tag R1 and the constant values C1 and C2. The values of C1 and C2 are set when the routing algorithm handler is initialized by the local processor. The value of R1 is loaded by the input port each time the input port transfers new header information. The output channels are numbered as follows: output channel in x direction = 1; output channel in y direction = 2; output channel in z direction = 3; output channel to local processor = 4.

The R1 has the routing tag, and the CMP instruction at line 1 compares C1 to the routing tag. At line 2, the BC instruction tests whether the routing tag contains all zeros. If true, the program branches to the location labeled processor and executes the OUT instruction, where operand 4 indicates that the message should be forwarded to the local processor. If the test result at line 2 is not true, the program continues to search for the direction in which the message should be forwarded. The PLO instruction at line 3 does this job by finding the position of the leading one bit in R1 and storing the result in R2. Thus, the value i of R2 represents the direction of the next node. The SHL instruction at line 4 shifts register C2(=1) i bit positions to the left and stores the result in R3. The value in R3 is used to zero out the ith bit of the routing tag by XORing the R1 and R3. Finally, the program sends the result of the routing program by OUT R2.

- *Example 2 (synergetic operation)*

As mentioned before, the instructions provided for the routing algorithm handler were selected carefully after investigating numerous network topologies. Yet, there are special cases where the routing algorithm should be executed in the local processor. This case may occur, for example, if the routing algorithm requires instructions not provided within the routing algorithm handler, or if the routing program is too big to be stored in the memory of the routing algorithm handler. The routing algorithm handler uses the MSG instruction to delegate some or all

```
R1: Routing tag (d);
R2: Label of the current node;
C1 = 0;
C2 = 5;
C3 = 0.

1.              CMP     R1, R2
2.              BC      B 1000, processor   (branch if equal)
3.              CMP     R1, C3
4.              BC      B'1010', +Y         (branch if greater than or equal)
5.              CMP     R1, C2
6.              BC      B'1010', −X         (branch if greater than or equal)
7.              CMP     R1, C1
8.              BC      B'1100', −Y         (branch if less than or equal)
9.              OUT     2
10. +Y:         OUT     4
11. −X:         OUT     1
12. −Y:         OUT     3
13. processor:  OUT     5
```

Figure 11

Routing program for 4 × 3 mesh with interval labeling.

of the routing program to the local processor. The MSG instruction has three operands, which are all registers. The first operand indicates the parameters to be sent to the local processor. One example of such a parameter is the address of the destination node. The second and the third operands are pointers to the beginning and end of the sequence of registers that contain the address of the memory location in the local processor where the desired routing program is stored. When the MSG instruction is executed in the routing algorithm handler, it causes an interrupt to the port controller. The port controller then builds a message containing the parameter and the address of the memory location in the processor, along with the number of the input controller which has the routing algorithm handler executing the MSG instruction. After building the message, the port controller sends it to the local processor. The local processor executes the desired program for the routing algorithm handler; it then sends the result to the port controller. The result includes the input controller number, the new content of the status register, and the control program. Included in the status register is the new instruction address, which the routing algorithm handler will fetch to execute the next instruction when control is returned to it. The control program has two control instructions, LSR (load status register) and ECP (end control program). At this point, the port controller causes an interrupt to the routing algorithm handler which is in the input controller specified in the message from the processor. The routing algorithm handler then executes the control program stored in the port controller and obtains the new content of the status register. The next instruction address of the routing algorithm handler is the location specified in the instruction address. **Figure 9** shows the sequence of communication initiated by the MSG instruction.

• *Example 3 (routing of 4 × 3 mesh with interval labeling)* The routing algorithm with interval labeling uses the table-lookup routing scheme. It is an efficient routing algorithm that reduces the table size [6, 42]. The architecture we propose can also support this scheme. We show a routing program example for a 4 × 3 mesh interconnection network reported in [42] using the proposed routing algorithm handler. As shown in **Figure 10**, each node of the 4 × 3 mesh 2D is assigned a label $\phi(x, y)$. In particular, the example routing program shows how the routing is done for node (1, 1). Let d be the label of the destination address in a packet. Each routing table requires only four entries, one for each outgoing channel. For example, the routing table at node (1, 1) contains the following information: For $d \geq 7$, the packet is routed using the +Y channel. For $5 \leq d < 7$, $1 < d \leq 3$, and $d \leq 1$, the packet is routed through channels −X, +X, and −Y, respectively. Shown in **Figure 11** is the routing program using the routing algorithm handler. It is assumed that 1, 2, 3, 4, and 5 represent respectively the channel numbers of −X, +X, −Y, +Y, and the local processor.

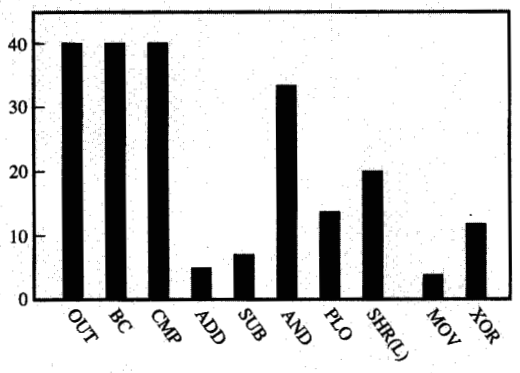


Figure 12

Frequency of instruction usage in the forty networks shown in Table 1.

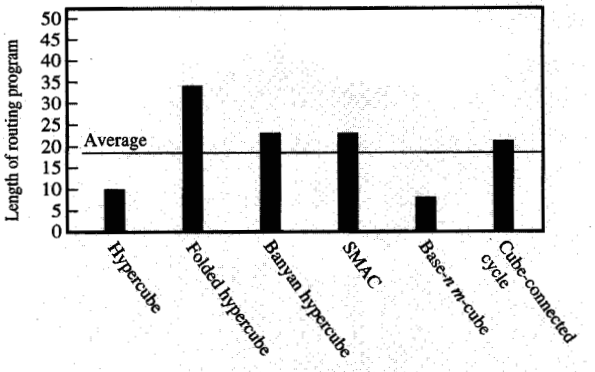


Figure 14

Routing program length for cube networks.

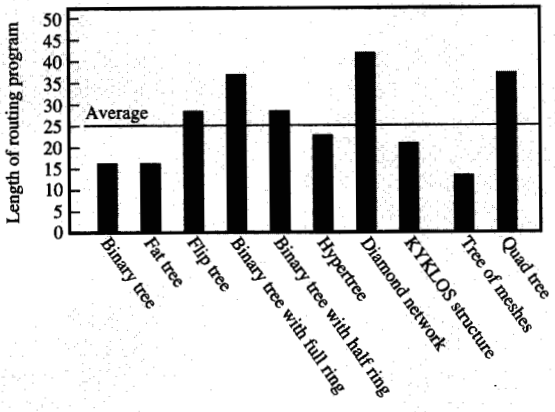


Figure 13

Routing program length for tree networks.

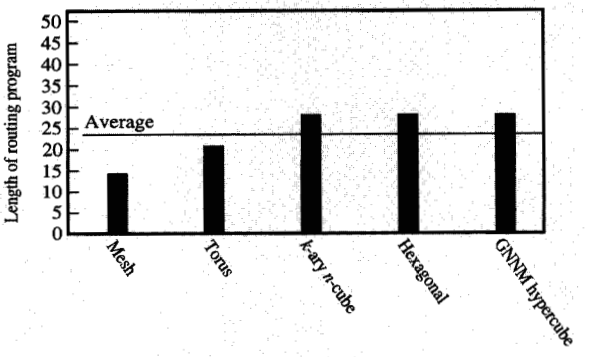


Figure 15

Routing program length for mesh networks.

6. Program characteristics

In this section, we discuss some of the program characteristics for the various routing programs we considered. (The programs have been reported elsewhere [8].) As shown in **Figure 12**, the CMP and BC instructions are used in all routing programs because every routing program must check at least one condition, i.e., whether or not the packets have reached their final destination node. The OUT instruction is also used in every routing program, since it sends the result of the routing decision made to the $n \times n$ switch as well as to the packet flow controller. Many routing algorithms require operations on the selected field of the given header information. The AND instruction is used to mask out the unnecessary field of data. The PLO instruction is used in the routing program of all tree networks and some cube networks. The

shift (SHR and SHL) instructions are used to align the data for comparisons and are also used in the routing programs of most multistage interconnection networks. The ADD and SUB instructions are used to increment or decrement values. The lengths of the routing programs for the tree networks, the cube networks, the mesh networks, the multistage interconnection network, and the networks not included in the preceding families are shown in **Figures 13–17**. The actual lengths of routing programs may vary depending on the size of the network.

7. Summary

In this paper, we have described a RISC-like general-purpose router architecture that accommodates a family of oblivious routing algorithms. The architecture is suitable for current technologies and is intended for multiprocessor and massively parallel systems. For this investigation, we studied the routing algorithms of more than forty interconnection networks. We have identified the common functions and instruction set that satisfy the requirements for handling all of the routing algorithms. The architecture has programming capabilities, and allows other oblivious routing algorithms not considered in our investigation to be accommodated as well. Furthermore, the architecture can handle the various types of header packet formats that are necessary to support different sizes of the interconnection networks. In addition, the fact that the architecture is programmable makes it easy to modify the routing algorithm if it is found to contain errors or if a better algorithm is subsequently developed. Because the architecture supports a wide range of interconnection networks, it can be mass-produced and has the potential to become an "off-the-shelf" product.

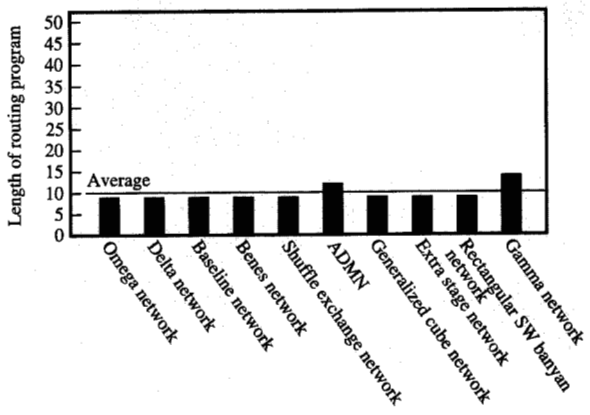


Figure 16

Routing program length for multistage interconnection networks.

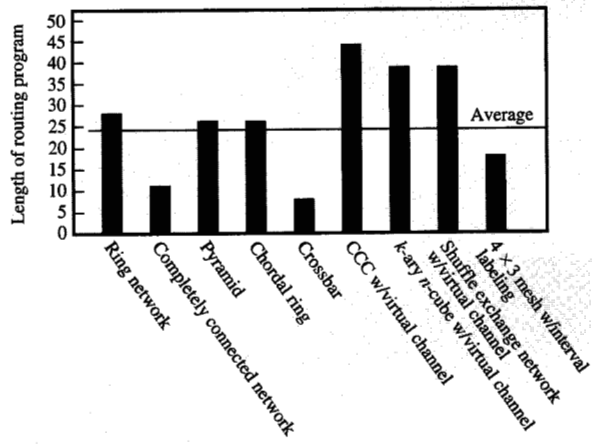


Figure 17

Routing program length for other networks.

References

1. B. W. O'Krafka, "Design and Evaluation of Directory-Based Cache Coherence Systems," Ph.D. Thesis, University of California, Berkeley, 1992.
2. W. J. Dally and C. L. Seitz, "The Torus Routing Chip," *J. Distr. Syst.* **1,** No. 3, 187–196 (1986).
3. "A Technical Summary of the iPSC/2 Concurrent Supercomputer," Order No. 280115-001, Intel Scientific Computers, Beaverton, OR 97006, 1988.
4. C. L. Seitz, "The Cosmic Cube," *Commun. ACM*, pp. 22–33 (1985).
5. S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moor, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb, "iWARP: An Integrated Solution to High-Speed Parallel Computing," *Proceedings of the IEEE Supercomputing Conference*, November 1988, pp. 330–338.
6. "The 9000 Transputer Products Overview Manual," Order Code: DB-TRANSPST/1, InMOS Ltd., 1000 Aztec West, Bristol BS12 4SQ England, 1991.
7. L. M. Li and P. K. McKinley, "A Survey of Routing Techniques in Wormhole Networks," *Technical Report MSU-CPS-ACS-46*, Department of Computer Science, Michigan State University, East Lansing, October 1991.
8. J. Park, S. Vassiliadis, and J. G. Delgado-Frias, "Flexible Router Architecture: Instruction Set and Organization," *Technical Report TR 01.C752*, IBM Microelectronics Division, Endicott, NY, September 1993.

9. P. Kermani and L. Kleinrock, "Virtual Cut Through: A New Computer Communication Switching Technique," *Computer Networks* **3,** 267–286 (1979).
10. W. J. Dally, "Virtual-Channel Flow Control," *IEEE Trans. Parallel & Distr. Syst.* **3,** 194–205 (1992).
11. J. Park, B. W. O'Krafka, S. Vassiliadis, and J. G. Delgado-Frias, "Design and Evaluation of a DAMQ Multiprocessor Network with Self-Compacting Buffers," *Proceedings of the IEEE Supercomputing Conference*, 1994, pp. 713–722.
12. J. Park, S. Vassiliadis, and J. G. Delgado-Frias, "Input and Output Port Controller Architecture and Organization," *Technical Report TR01.C767*, IBM System/390 Division Development Laboratory, Endicott, NY, 1994.
13. E. Horowitz and A. Zorat, "The Binary Tree as Interconnection Network: Applications to Multiprocessor Systems and VLSI," *IEEE Trans. Computers* **C-30,** No. 4, 247–253 (1981).
14. C. E. Leiserson, "Fat-Tree: Universal Networks for Hardware-Efficient Supercomputing," *Proceedings of the 1985 International Conference on Parallel Processing*, IEEE Computer Society Press, Silver Spring, MD, 1985, pp. 393–402.
15. F. J. Meyer and D. K. Pradhan, "Flip-Tree: Fault-Tolerant Graphs with Wide Containers," *IEEE Trans. Computers* **37,** No. 4, 472–478 (1988).
16. J. R. Goodman and C. H. Sequin, "Hypertree: A Multiprocessor Interconnection Topology," *IEEE Trans. Computers* **C-30,** No. 12, 923–933 (1981).
17. W. K. Tsai, Y. C. Kim, and N. Bagherzadeh, "A Hierarchical Mesh Architecture," *Proceedings of the 4th Annual Parallel Processing Symposium*, IEEE Orange County Computer Society, Fullerton, CA, 1990, pp. 923–933.
18. N. S. Woo and A. Agrawala, "A Symmetric Tree Structure Interconnection Network and Its Message Traffic," *IEEE Trans. Computers* **C-34,** No. 8, 765–768 (1985).
19. B. L. Menezes and R. Jenevein, "The KYKLOS Multicomputer Network and Its Message Traffic," *IEEE Trans. Computers* **C-34,** No. 8, 765–768 (1985).
20. F. T. Leighton, "New Lower Bound Techniques for VLSI," *Math. Syst. Theor.* **17,** No. 1, 47 (1984).
21. P. K. Bansal, K. Singh, and R. C. Joshi, "Quad Tree: A Cost Effective Fault Tolerant Multistage Interconnection," *Proceedings of IEEE INFOCOM '92: Conference on Computer Communications*, Vol. 2, 1992, pp. 860–866.
22. P. Banerjee, "Algorithm-Based Fault Tolerance on a Hypercube Multiprocessor," *IEEE Trans. Computers* **39,** No. 9, 1132–1144 (1990).
23. A. El-Amawy and S. Latifi, "Properties of and Performance of Folded Hypercubes," *IEEE Trans. Parallel & Distr. Syst.* **2,** No. 1, 31–42 (1991).
24. A. S. Youssef and B. Narahari, "The Banyan-Hypercube Networks," *IEEE Trans. Parallel & Distr. Syst.* **1,** No. 2, 160–169 (1990).
25. P. W. Dowd and K. Jabbour, "Spanning Multiaccess Channel Hypercube Computer Interconnection," *IEEE Trans. Computers* **37,** No. 9, 1137–1142 (1988).
26. N. Tanabe, T. Suzuoka, and S. Nakamura, "Base-*m* *n*-cube High Performance Interconnection Networks for Highly Parallel Computer Prodigy," *Proceedings of the 1991 International Conference on Parallel Processing*, Springer-Verlag, Berlin, 1991, pp. I509–I516.
27. L. D. Wittie, "Communication Structures for Large Networks of Microcomputers," *IEEE Trans. Computers* **C-30,** No. 4, 264–273 (1981).
28. W. J. Dally, "Performance Analysis of *k*-ary *n*-cube Interconnection Networks," *IEEE Trans. Computers* **39,** No. 6, 775–785 (1990).
29. M. S. Chen and K. G. Shin, "Addressing, Routing and Broadcasting in Hexagonal Mesh Multiprocessors," *IEEE Trans. Computers* **39,** No. 1, 10–18 (1990).
30. L. N. Bhuyan and D. P. Agarwal, "A General Class of Processor Interconnection Strategies," *Proceedings of the 9th Annual Symposium on Computer Architecture*, IEEE, New York, 1982, pp. 90–98.
31. A. DeCegama, *Parallel Processor Architectures and VLSI Hardware*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.
32. C. P. Kruskal and M. Snir, "A Unified Theory of Interconnection Network Structure," *Ultracomputer Note No. 106,* New York University, New York.
33. V. Benes, "Optimal Rearrangable Multistage Connection Networks," *Bell Syst. Tech. J.* **40,** No. 4, Pt. 2, 1641–1656 (1964).
34. W. J. Dally and C. L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. Computers* **C-36,** No. 5, 547–553 (1987).
35. R. J. McMillen and H. J. Siegel, "Routing Schemes for the Augmented Data Manipulate Network in an MIMD System," *IEEE Trans. Computers* **C-31,** No. 12, 1202–1214 (1982).
36. G. B. Adams and H. J. Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems," *IEEE Trans. Computers* **C-31,** No. 5, 247–253 (1982).
37. V. Cherkassky, E. Opper, and M. Malek, "Reliability and Fault Diagnosis of Fault Tolerant Multistage Interconnection Networks," *Proceedings of the 14th Symposium on Fault Tolerant Computers*, IEEE Computer Society Press, Silver Spring, MD, June 1984, pp. 246–251.
38. D. S. Parker and C. S. Raghavendra, "The Gamma Network," *IEEE Trans. Computers* **C-33,** No. 4, 367–373 (1984).
39. S. P. Dandamudi and D. L. Eager, "Hierarchical Interconnection Networks for Multicomputer Systems," *IEEE Trans. Computers* **39,** No. 6, 786–797 (1990).
40. B. W. Arden and H. Lee, "Analysis of Chordal Ring Network," *IEEE Trans. Computers* **C-30,** 291–296 (1981).
41. N. Pippenger, "On Crossbar Switching Networks," *IEEE Trans. Commun.* **COM-23,** 646–659 (1975).
42. X. Lin and L. M. Ni, "Deadlock-Free Multicast Wormhole Routing in Multicomputer Networks," *Proceedings of the 18th International Symposium on Computer Architecture*, IEEE CS Press, Los Alamitos, CA, Order No. 2146, 1991, pp. 116–125.

Received November 1, 1993; accepted for publication May 12, 1995

Joonho Park *Bell Communications Research (Bellcore), 331 Newman Springs Road, Red Bank, New Jersey 07701 (jpark@cc.bellcore.com).* Dr. Park received a B.S. degree in computer science and mathematics from the State University of New York (SUNY) at Binghamton and an M.S. in computer engineering from the Pennsylvania State University. In 1988 he joined IBM at the Endicott laboratory, where he worked on performance evaluation of I/O subsystems, visualization tool development, and LAN network management.
Dr. Park received his Ph.D. in computer engineering from SUNY–Binghamton in 1994 through the IBM Graduate Work Study Program. In 1995 he joined Bell Communications Research as a member of the technical staff to work on information modeling of communications networks. Dr. Park's areas of interest include parallel processing, network management, and network software development.

Stamatis Vassiliadis *Delft University of Technology, Faculty of Electrical Engineering, Mekelweg 4, 2628 CD Delft, The Netherlands (stamatis@duteca.et.tudelft.nl).*
Dr. Vassiliadis received the Dr. Eng. degree in electronic engineering from the Politecnico di Milano, Milan, Italy, in 1978. He is currently on the faculty of the Department of Electrical Engineering at the Delft University of Technology. He has been a faculty member at both Cornell University in Ithaca, New York, and the State University of New York (SUNY) at Binghamton. From 1992 to 1993 he was a Visiting Professor at the School of Electrical Engineering, College of Engineering, at Cornell. Dr. Vassiliadis worked for ten years at IBM, at the Advanced Workstations and Systems laboratory in Austin, Texas, the Mid-Hudson Valley laboratory in Poughkeepsie, New York, and the Glendale laboratory in Endicott, New York. His assignments included the development of new computer organizations and architectures, high-level design and technical leadership in the implementation of new computer systems, and advanced research in a variety of computer-related fields. Among other projects, he was involved in the design and implementation of the IBM 9370 Model 60 computer system. While at IBM he received numerous awards, including 21 levels of the IBM Publication Achievement Award, 14 levels of the IBM Invention Achievement Award, and an Outstanding Innovation Award for Engineering/Scientific Hardware Design in 1989. In 1990 he was awarded the highest number of patents in IBM. Dr. Vassiliadis is a member of the IEEE Computer Society. His research interests include computer architecture, hardware design and functional testing of computer systems, parallel processors, computer arithmetic, EDFI for hardware implementations, neural networks, fuzzy logic and systems, and software engineering.

Jose G. Delgado-Frias *State University of New York at Binghamton, Department of Electrical Engineering, Binghamton, New York 13901.* Dr. Delgado-Frias received a B.S. from the National Autonomous University of Mexico, an M.S. from the National Institute for Astrophysics, Optics and Electronics, Mexico, and a Ph.D. from Texas A & M University, all in electrical engineering. Since 1989 he has been with the Electrical Engineering Department at the State University of New York at Binghamton, where he is an associate professor. He has held academic positions at the University of Oxford, England, and the National Autonomous University of Mexico. His research interests include parallel computer architectures, VLSI/WSI design, computer hardware organization, and neural networks. Dr. Delgado-Frias has been the co-chairman of three international workshops on VLSI for artificial intelligence and neural networks which were held at the University of Oxford in 1988, 1990, and 1992. He is a co-editor of three books, has co-authored more than seventy technical papers, and holds four U.S. patents. In 1994, he received the State University of New York System Chancellor's Award for Excellence in Teaching. He is a senior member of the Institute of Electrical and Electronics Engineers (IEEE).