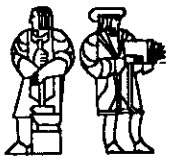
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Design of a Fault-tolerant Packet Communication Computer Architecture

Computation Structures Group Memo 196
July 1980

Clement Leung
Jack B. Dennis

This research was supported by the National Science Foundation under grant MCS75-04060 A01.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Design of a Fault-Tolerant Packet Communication Computer Architecture

Clement K. C. Leung
Jack B. Dennis

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Mass 02139
USA

Abstract

A dynamic redundancy scheme for masking hardware failures in a multiprocessor architecture designed to execute parallel programs organized by data flow principles is presented. Hardware in this architecture is organized as an interconnection of self-timed packet communication modules. Novel features include use of packet networks to support communication among processing elements and dynamic allocation of a homogeneous set of specialized functional units to service requests. Program organization and hardware module designs to support the dynamic redundancy scheme are described.

Keyword: Dynamic redundancy, fault detection, network repair, self-timed hardware modules, data flow computer architecture.

1. Introduction

Computer systems which are significantly more powerful than those presently available are needed in weather forecasting and aeronautical design for solving differential equations numerically. Data flow concepts provide an attractive approach to build high performance systems for these applications. Current projects on data-driven computations and highly concurrent systems based on these concepts are surveyed in [21], [22]. A graphical data flow language with procedures and data structure operations has been presented in [10].

Unlike process control and interplanetary spaceflight, there are no stringent reliability requirements inherent in physics simulation applications. It is nonetheless important to design for availability and maintainability. System throughput is improved if hardware failures can be masked, especially since it is not unusual for a numerical computation in physics simulation to execute for many hours. Methodologies for incorporating fault tolerance into high performance computing systems and the associated technical problems have been discussed in [2], [3], but most computer systems [1], [14], [24], [28] designed to tolerate hardware failures are intended for high reliability or long life applications with modest computational requirements. In this paper we study fault tolerance techniques to cope with hardware failures in a multiprocessor designed to execute programs expressed in a subset of the data flow language presented in [10]. We shall refer to this multiprocessor system as a data flow processor (DFP) for convenience. This DFP has several novel features:

1. High performance and fault tolerance are achieved by using pools of identical hardware units

Fault-tolerant multiprocessor systems [4], [14] have been designed using multiple, identical, hardware units. A program running on the DFP is partitioned and stored on a set of identical

processing elements. The DFP also has a homogeneous set of specialized functional units for performing complex operations. These functional units are allocated dynamically to service requests from the processing elements. The dynamic allocation scheme provides direct support for graceful degradation with respect to these functional units. Programs prepared for execution on the fault-free DFP can run without modification if only a subset of functional units has failed.

2. Communication between processing elements and functional units is supported by packet networks

In the DFP modules serve two distinct functions - processing and communication. Processing elements in a DFP execute subcomputations concurrently. Communication between the processing elements is supported by packet networks, to be constructed out of a few basic LSI cell types. This architecture is quite different from most fault-tolerant computer architectures reported in the literature, which are bus-oriented von Neumann architectures [1, 25, 24] or bus-oriented multiprocessors [14, 28]. Store-and-forward packet network designs which can handle a large number of packets concurrently have been analyzed in [5]. In some of these networks the number of basic modules and the length of connections between them both exhibit faster than linear growth as the number of processing elements being serviced increases. It thus appears that a substantial amount of hardware in a practical data flow processor will be used to implement packet networks. The reliability of these networks will be an important factor in assessing system reliability and availability and it is important to minimize the amount of redundant hardware invested in them to achieve a desired level of fault tolerance.

3. Hardware in the DFP is organized as a packet communication architecture

The DFP hardware is organized by a *packet communication discipline* to support concurrency and modularity. Computer architectures are commonly implemented as synchronous digital systems in which events in all modules are synchronized with reference to a common timing signal. Many

fundamental fault tolerance techniques have been developed in this context. In contrast, a packet communication computer architecture is implemented as an interconnection of self-timed modules whose activities are synchronized through localized signal exchange, in accordance with the adopted packet communication protocol. Fault tolerance techniques for asynchronous systems have previously been demonstrated in a fault-tolerant clock design [8] and reported in a paper on *synchronization voting* by Daniels and Wakerly [7]. We have generalized these techniques to show that modular redundancy and coding techniques can be used to mask and detect failures in self-timed systems under a random wave train fault model. The basic concepts and techniques of this work are reported in a doctoral thesis [17].

In this paper we present a dynamic redundancy scheme for masking hardware failures in a DFP. In this scheme hardware failures are detected and diagnosed, the DFP is repaired, and then afflicted subcomputations are reexecuted on the repaired, possibly degraded, DFP. We assume that programs for the DFP are prepared on a host machine, loaded into the DFP, and then executed. This host machine is also assigned the tasks of configuration control, and of coordinating diagnosis, repair and recovery activities with program execution. We will explain the strategy to be implemented on this host machine, and present hardware redundancy and packet encoding techniques for implementing hardware modules to support this strategy. A fault-tolerant implementation of this strategy on the host machine can be constructed using conventional techniques, and will not be considered in this paper.

The hardware organization of the DFP and its operation are explained in Section 2, where we will focus on those aspects of its operation relevant to fault tolerance considerations. The dynamic redundancy scheme is explained in Section 3. Hardware module designs to satisfy fault tolerance requirements imposed on them by the dynamic redundancy scheme are presented in Section 4. Strategies for incorporating additional hardware into a network to support rapid repair are discussed

in Section 5. Concluding remarks are given in Section 6.

2. A Packet Communication Computer Architecture

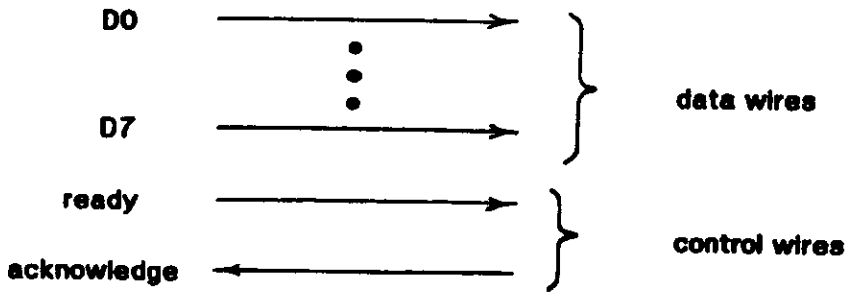
Hardware Organization

Hardware in a packet communication computer architecture is organized as an interconnection of self-timed modules which communicate by sending packets to each other. Each packet is transmitted byte-serially between modules. Packet bytes are delivered and received by hardware modules using an asynchronous protocol. Each *module port* consists of a bundle of data wires and a pair of control wires (Figure 1a). Packet communication is synchronized by sending control signals over the control wires. Availability of a new packet at a connection is signaled by sending a *ready* signal over the ready wire, its receipt by returning an *acknowledge* signal over the acknowledge wire. *Ready* and *acknowledge* signals are represented by signal transitions (Figure 1b) on the respective wires.

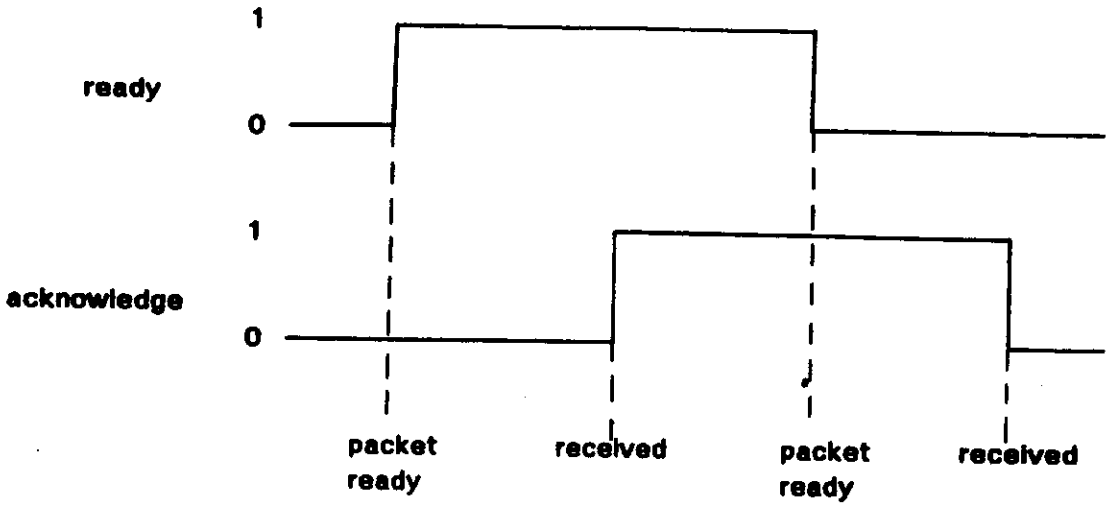
The major modules in the DFP (Fig. 2.) are processing elements (PEs), specialized functional units (SFUs), a routing network and an allocation network. Scalar operations are processed in the PEs and SFUs. The networks support packet traffic among the PEs and SFUs.

Operating Principles

A machine level program for the DFP is stored in the PEs as a set of *activity templates* [9]. An activity template A contains an *operation code* and the *addresses* of one or more activity templates, called A's *target templates*, which should receive the result generated by processing A. The address of an activity template uniquely identifies the activity template; it has two components:



(a) Hardware structure



(b) Transition signaling

Fig. 1. Packet transmission protocol

---

a destination tag which specifies the PE in which the template resides, and the location of the template within that PE.

Operations are divided into two classes, according to whether they are executed by a PE or by a SFU. When activity template A is enabled (See below for a discussion of the enabling conditions) and the operation it specifies is executable on a PE, the operation is applied to the operands A has

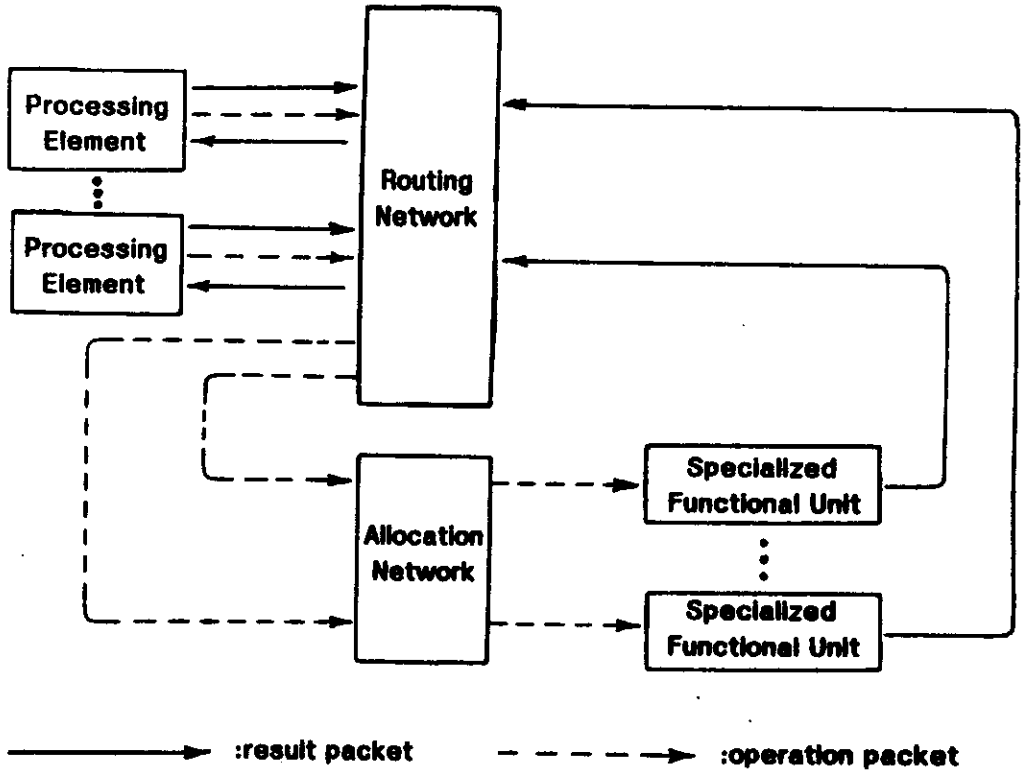


Fig. 2. Hardware architecture of the data flow processor

received and the result of the application is dispatched to A's target templates. If a target template B resides on another PE, a copy of the result is tagged with B's address and a byte count to form a *result packet*, and delivered to B via the routing network. For an operation executed on a SFU, an *operation packet* consisting of the operation code, operands and the target template addresses is formed and delivered to a SFU via the routing network and the allocation network. At a SFU the operation specified in an operation packet is applied to the operands and result packets are generated and dispatched to the target templates via the routing network.

An activity template A is enabled when two conditions are met. First of all the operands required for the operation must have arrived. The second condition is a consequence of organizing machine level programs in the DFP to support pipelining and iteration. Suppose that in such a program activity template A sends results to activity template B residing in another PE. To avoid deadlock [19], an operand sent from A to B must be processed before A can send B a second operand. The execution of A and B are synchronized by conditioning each activation of A on receiving an *acknowledge packet* from B. This acknowledge packet is transmitted when B is processed with the previous operand received from A. Thus before an activity template can be executed, it must have received the necessary acknowledgments from its target templates. A detailed explanation of this synchronization scheme is given in [13]. Under this scheme every result packet transmitted through the networks is acknowledged by an acknowledge packet returned by the target template. We will also make the assumption that *each operation packet contains exactly one target template address*. This assumption results in no loss of generality since the result obtained by processing an operation packet can be further distributed through its target template. In fault-free operation under the above synchronization scheme and this assumption on operation packets, a PE will receive exactly one acknowledge packet for every result packet or operation packet it delivers to the routing network. This property will be used to incorporate fault tolerance in the DFP in later sections.

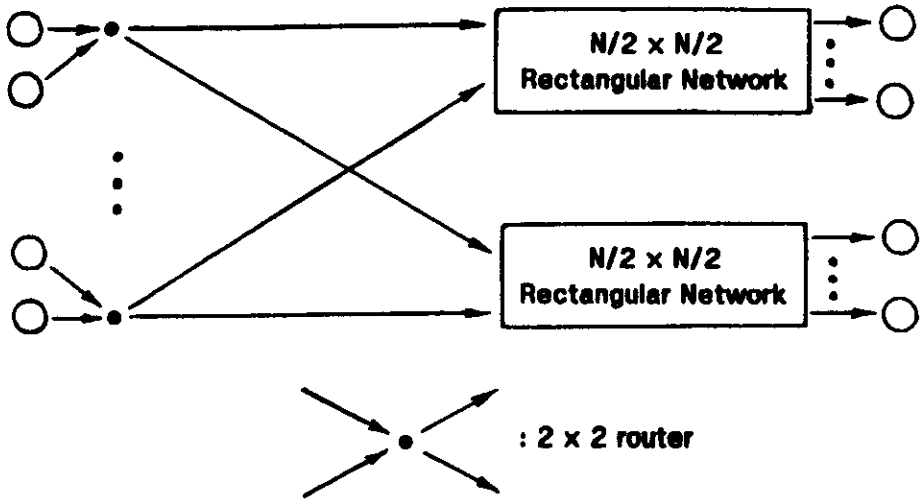
A PE provides storage for activity templates, executes simple operations, sends packets to and receives packets from the routing network. A SFU is designed to execute complex operations, such as floating point arithmetic, efficiently, with additional capabilities to receive operation packets from the allocation network and send result packets to the routing network. Implementation of these hardware modules will be examined after their fault tolerance requirements have been determined. In the remainder of this section we take a closer look at the structure of the routing network and the

allocation network.

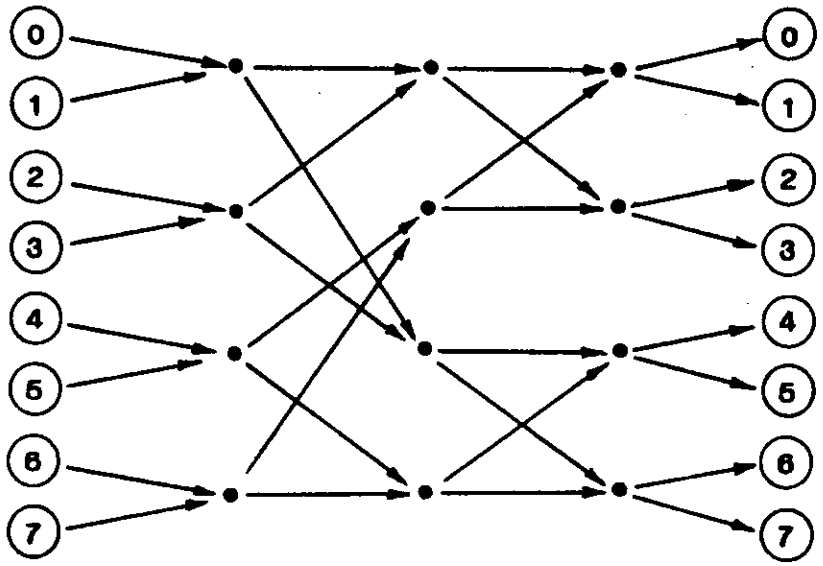
Packet Networks

An N x N routing network, with N input ports and N output ports, supports packet communication among N PEs. It accepts packets at its input ports and transmits each packet at the output port specified by a bit field comprising a header or destination tag of the packet. The destination tag and a byte count specifying the length of the packet are contained in the first few bytes of a packet. Routing networks can be constructed using 2 x 2 routers. A 2 x 2 router receives packets at its two input ports and delivers each received packet at one of two output ports according to the destination tag carried by the packet. The 2 x 2 router is designed so that packets to be forwarded at different output ports can be processed concurrently.

Methods and techniques for fault tolerance will be illustrated using *rectangular* routing networks. An N x N *rectangular* network is built from 2 x 2 *routers* by the recursive construction illustrated in Fig. 3. An N x N network so constructed has $\log_2 N$ stages each of which contains N/2 routers. All packets sent to an output port of the routing network, regardless of their source, have identical destination tags. Routers in succeeding stages in the network examine successive bits in a destination tag to forward the packet along the proper path. Path control in a routing network is distributed among the routers. There is no centralized control mechanism whose complexity must grow with network size and which may become a performance bottleneck. Many packets can be forwarded concurrently to provide a high throughput rate. One of the design objectives for a fault-tolerant routing network is to retain all these characteristics during fault-free operation: decentralized path control, parallel processing, and asynchronous byte-serial communication.



(a) Recursive construction



(b) An 8 x 8 rectangular network

Fig. 3. Rectangualr routing networks

In the DFP the allocation network receives operation packets from the routing network and distributes them among the SFUs. Each of its input ports has its own routing network address. In

this paper we use a rectangular allocation network which also has the topology shown in Fig. 3, constructed out of 2 x 2 allocators. An allocator receives operation packets from its two input ports and forwards them at its output ports as these output ports become free. Arbitration is performed whenever it receives two operation packets simultaneously (within a short time interval), and whenever both output ports become free simultaneously with operation packets pending for output. It is possible for operation packets to be temporarily "trapped" in a section of the Allocation Network waiting for service even though SFUs not reachable from this subnetwork are free. Such trapping has the pleasing property of automatically diverting other operation packets from the congested subnetwork.

3. A Fault Tolerance Strategy Based on Dynamic Redundancy

It is possible to apply static redundancy techniques uniformly in the DFP to implement a fault masking capability in hardware. The main disadvantages are the prices to be paid in additional power consumption and in redundant hardware packages and connections, especially in the packet networks. We have examined two fault masking network designs and a fault detecting network design [17]. The latter scheme has the lowest hardware redundancy requirement. It is thus attractive to derive a fault masking strategy based on this fault-detecting network design. Such a strategy leads to a *dynamic redundancy* scheme. In a dynamic redundancy scheme, failures are masked through hardware-implemented fault detection, diagnosis, repair, followed by reexecution of the afflicted subcomputations. To support the dynamic redundancy scheme, each PE must store additional information that is not used in a non-redundant data flow processor. A copy of every result and operation packet delivered to the *packet transport and processing* (PTP) subsystem consisting of the packet networks and the SFUs must be kept until the packet is acknowledged. At each PE the sender of every received packet must be known. This information can be maintained

by associating each operand position in an activity template with a template address for returning acknowledgments. The sender of each acknowledgment packet and retransmission request (See below) must also be identifiable.

The dynamic redundancy scheme assumes that the hardware implementation of the DFP has the following properties, even in the presence of hardware failures.

(1) Each packet received by the routing network is delivered at the output port specified by its destination tag, either to a PE or the allocation network. Each packet received by the allocation network is delivered to a SFU. Specifically, neither packets nor packet bytes will be lost in the networks.

(2) Each packet delivered to a network output port is either error-free or flagged as erroneous.

(3) Target template addresses carried in an operation packet are always delivered free of error.

(4) For every operation packet received, a SFU will deliver either an error-free result packet or one tagged as erroneous to the target template specified.

(5) Every acknowledge packet and retransmission request (See below) is delivered free of error to its destination.

Techniques to design redundant routers, allocators, PEs and SFUs that can be used to implement a DFP with these properties, even when up to one hardware package in each module has failed, will be presented in Section 4. The fault tolerance strategy is as follows.

Any packet which has encountered a faulty router or allocator in its journey through the networks will be marked as such upon delivery to the destination PE or SFU. When a SFU

receives an operation packet tagged as erroneous, it will generate a result packet, tag it as erroneous, and forward it to the target template specified in the operation packet. If a PE receives a result packet tagged as erroneous, it signals the host machine, which in turn signals the other PEs to stop sending packets to the routing network. All packets in transit will arrive at their destinations after a finite time period, which can be determined from hardware parameters specified for the PTP subsystem. After this time period the PTP subsystem can be repaired under the direction of error signals generated by fault detectors in this subsystem. After repair the processing elements are restarted. A PE which has received a contaminated packet will issue a *retransmission request* instead of an acknowledgment to the sender. Program execution can otherwise proceed normally.

Activity templates representing a machine level program, and a complete intermediate state of the computation in progress, are stored in the PEs. If only routers, allocators or SFUs have failed, the computation can always be restarted from the intermediate state stored in the PEs and run to completion after the networks have been repaired. If failures occur in the storage components of a PE, it may be necessary to abort the computation in progress, since the intermediate state may become inconsistent. Storage failures thus must be masked to achieve complete fault masking. Failure in other components of a PE need not be masked in hardware, but the activity templates and partial intermediate state stored in it must be relocated before processing can resume. If an activity template A is relocated, the entire activity template set must be relinked so that other templates having A as a target will contain the new address of A. It thus seems desirable to mask all failures in PEs locally in hardware. Communication between PEs and the host machine to coordinate repair can be implemented with an interprocessor bus. Fault-tolerant bussing structures have been presented in [14] and [28].

The dynamic redundancy scheme we have described is built directly on the execution control mechanism in a DFP. It has the merit that extensions to the execution control mechanism are

incorporated in low level hardware functions and require no extra programming effort to achieve fault tolerance.

The PTP subsystem must be repaired after failures are detected. A failed module can be replaced or repaired in place manually. Availability is improved by reconfiguring around the failed module automatically and then repairing the failed module off-line. For a computation in progress to proceed successfully, the full functionality of the routing network must be retained, i.e., packet communication between any input port and any output port must be maintained. Two strategies for incorporating spare routers and data paths into a routing network are discussed in Section 5. Under these strategies the full capability of a routing network is retained so long as spares are not exhausted. These strategies are also applicable to allocation networks. A SFU failure can be repaired by simply taking it off-line. An allocator will not forward operation packets to any SFU which has stopped acknowledging inputs. The machine architecture is gracefully degradable with respect to SFU failures in this sense. SFU failures have no effect other than degraded performance across the PTP subsystem boundary.

4. Module Design

Our basic unit for fault tolerance considerations is a *package*, which receives input signals and delivers output signals through its *terminals*. Hardware modules are constructed using packages. We also assume that the mean time to repair is much shorter than the mean time to failure. Modules described in this section are hence designed to tolerate up to one package failure per module. Logic design for error control often assumes a stuck-at fault model for hardware failures. We are interested in a fault model that reflects the sensitivity of self-timed modules to runt pulses and output hazards, as well as being applicable to most common hardware failure modes. We have adopted a *random wave train* fault model for hardware package failures: if a package has failed,

its output signals can wander arbitrarily in the region bounded by the signal values 0 and 1. When a signal takes on an intermediate value between 0 and 1, it may be interpreted by different receiving modules differently. A random pulse train fault model has been used in a fault-tolerant clock design in [8]. Daniels and Wakerly [7] have also presented a *synchronization voting* technique for implementing synchronization reliably without using a common timing signal for reference. We have generalized these results and shown that modular redundancy and coding can be used to mask and detect hardware failures in self-timed modules under the random wave train fault model. This work is reported in a doctoral thesis and will be assumed in this paper. The interested reader is referred to [17] for details. Failure mechanisms in MOS LSI technology that cannot be modeled by classical stuck-at fault models have been reported in [16], [27].

In this section we present hardware redundancy and packet encoding techniques to support the five fault tolerance properties stated in Section 3. We first describe the encoding techniques:

-- Control signals are generated in quadruplicate, from four failure-independent packages.[1]

-- Each data byte is protected by a parity bit. The nine bits of a parity-encoded byte are generated from nine failure-independent packages.

-- Each packet byte whose error-free transmission must be guaranteed is expanded into three bytes. The second and third bytes in each triplet are obtained by rotating bits in the given byte one and two positions to the right, respectively:

given byte: $b_0b_1b_2b_3b_4b_5b_6b_7b_8$,

1. Quadruplication is necessary for masking control signal faults under the random wave train model using the techniques given in [17]. Triplication is sufficient if the classical stuck-at fault model is assumed.

2^{nd} byte: $b_8b_0b_1b_2b_3b_4b_5b_6b_7$,

3^{rd} byte: $b_7b_8b_0b_1b_2b_3b_4b_5b_6$

This encoding scheme can be regarded as an implementation of triple modular redundancy in time instead of in space. Bytes in *byte count* fields and *template address* fields, and identification tags in acknowledgment packets and retransmission requests are protected using this technique.

-- An all 0's data byte is appended to the tail of each packet. This byte is used to flag packets which contain erroneous bytes. It is set to all 1's by the first module which detects the parity violation, and is otherwise retransmitted as received. The packet is accepted as error-free only if its flag consists of all 0's.

Using the hardware redundancy techniques explained in [17] and the above encoding techniques routers, allocators and SFUs can be implemented with the following fault tolerance properties:

-- If at most one of the four control signals delivered on each quadruple is faulty, its pathological effects can be masked.

-- If at most one data wire in each parity-checked data link carries faulty signals, the byte counts and addressing information in each packet can be retrieved. The last byte in the corresponding output packet will be flagged, i.e., will not be all 0's, if any packet byte has violated the parity check.

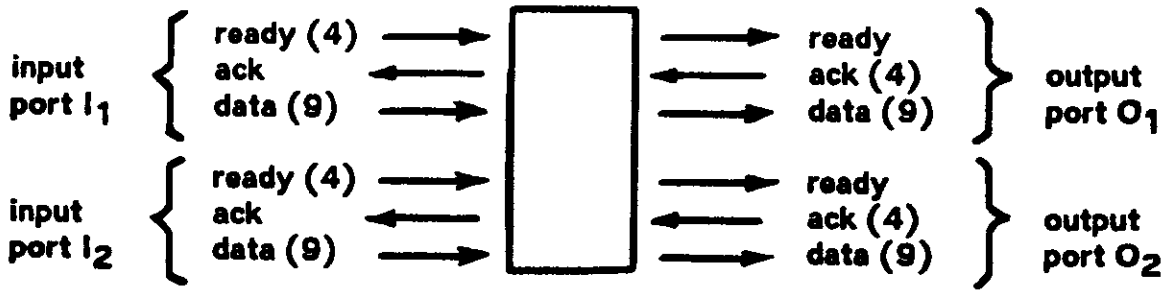
-- If at most one package in the module has failed, the above capabilities are not impaired and random wave trains are delivered on at most one output control wire and one output data wire at any module port.

We note that it is in fact possible to mask all single package failures in routers, allocators and SFUs with these fault tolerance capabilities if every packet byte is triplicated using the rotate-and-repeat encoding scheme given above. This approach to fault masking leads to lower performance during fault-free operation, as compared with the dynamic redundancy scheme explained in the last section, and hence is preferable only when the network hardware is quite unreliable.

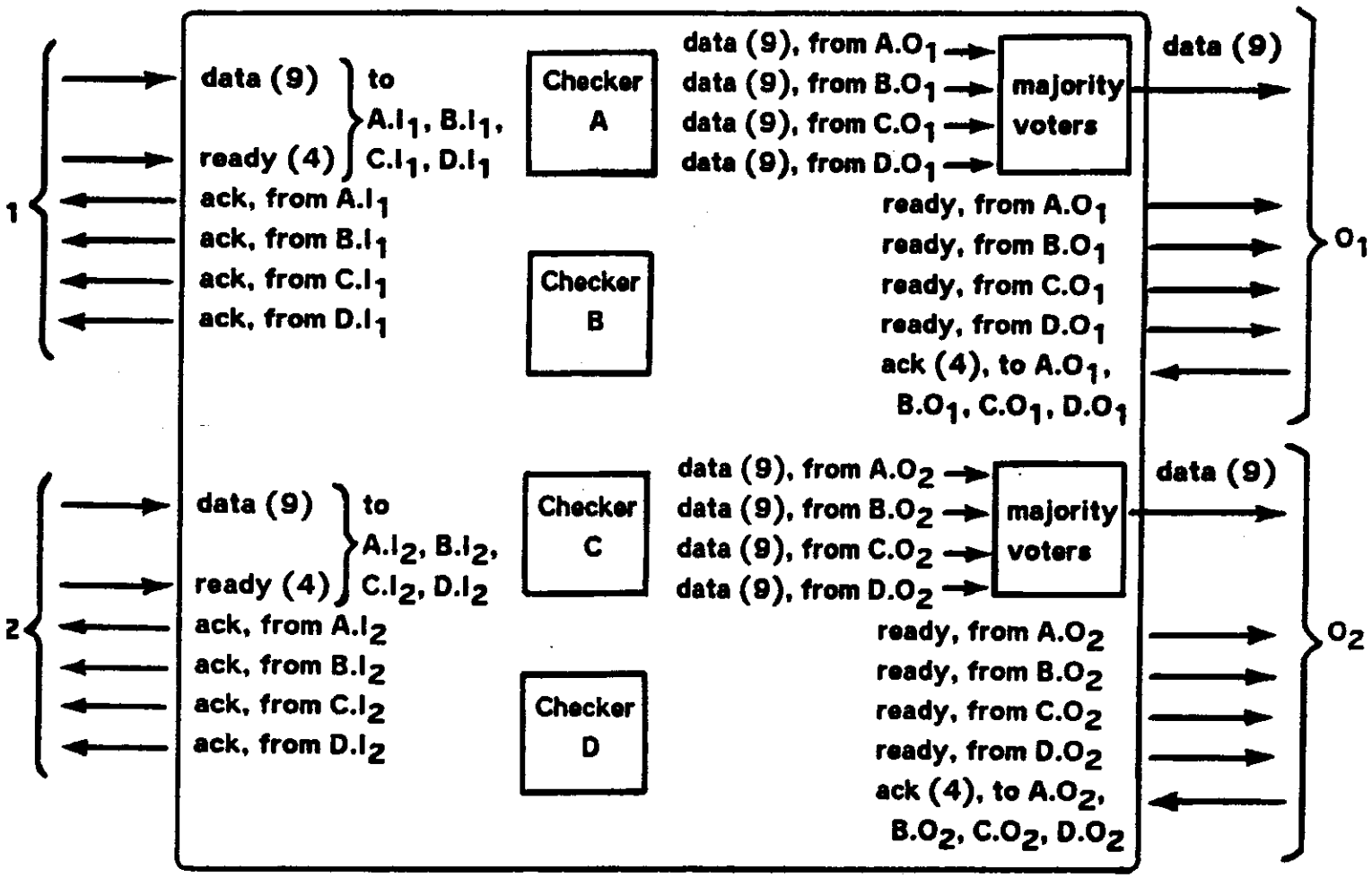
Router Module Design

We illustrate the design techniques using the router module. A *redundant* router module (Fig. 4) receives packets at its two input ports, and delivers each received packet at the output port specified by a destination tag carried in the packet. Packet receipt and forwarding are synchronized by control signals delivered in quadruplicate. Packet bytes are parity encoded. The redundant router is implemented using four *checker* packages and nine *voter* packages (Fig. 4). Each checker (Fig 4.a) has two input ports and two output ports. Control signals generated at the corresponding ports of the four checkers are grouped together at each port of the redundant router module. Thus, for example, the acknowledge signals generated by the four checkers at their I_1 input ports (Fig. 4b) are grouped together at input port I_1 of the redundant router. Data byte outputs from the corresponding output ports of the four checkers are collected together and voted upon at the voters to derive outputs for an output port of the redundant router. In Fig. 4, the number of wires represented by each arrow is given in its label in parentheses. This number is omitted if the arrow represents a single wire.

A checker receives all input control and data signals delivered to the router, and implements several fault tolerance capabilities in addition to packet routing:



(a) A Checker Unit



(b) The Redundant Router

Fig. 4. Hardware structure of a redundant 2 × 2 router.

-- mask control signal failures when at most one control signal in each quadruple is faulty.

-- deduce the error-free byte for every data byte encoded in triplicate using the rotate-and-repeat scheme given above.

-- set the last packet byte to all 1's if the last packet byte it has received is not all 0's or if parity violation has been detected.

In the redundant router shown in Fig 4. data faults due to single checker failures will be masked at the voter packages. Control faults due to single checker failures and data faults due to single voter failures are detected and dealt with in hardware modules receiving packets from this redundant router module.

Conflicting requests must be resolved at a router module. In an asynchronous hardware system there is no guarantee that the same packet will arrive at all three checkers in a redundant router simultaneously. Conflicting requests may arrive in different order at two checkers. If the conflicts are resolved differently, bytes belonging to different packets may be paired up with each other at the voters. To handle this problem the checkers must make unanimous decisions in conflict resolutions. Algorithms and circuit design techniques for reaching agreement in the presence of hardware failures are detailed in [17], [23], but will be omitted in this paper.

Processing Element Design

A PE provides storage for activity templates and their operands, as well as functional capabilities for activity template processing, and input/output capabilities for packet communication and error report. A detailed logic design for a non-redundant PE using commercially available components is given in [26], [11]. To support the dynamic redundancy scheme, a PE in a fault-tolerant DFP can be constructed using a fault masking bit-sliced memory [6], a redundant control unit, and a redundant functional unit (Fig. 5).

A partial state of the computation in execution is stored in the bit-sliced memory. For the computation to be recoverable after single package failures, all such failures must be masked along the data path used to retrieve this information from the memory to the host machine. This is achieved by using a bit-sliced memory protected by an error-correcting code to store this information, and a control unit with the same structure and operating principle, and hence the same

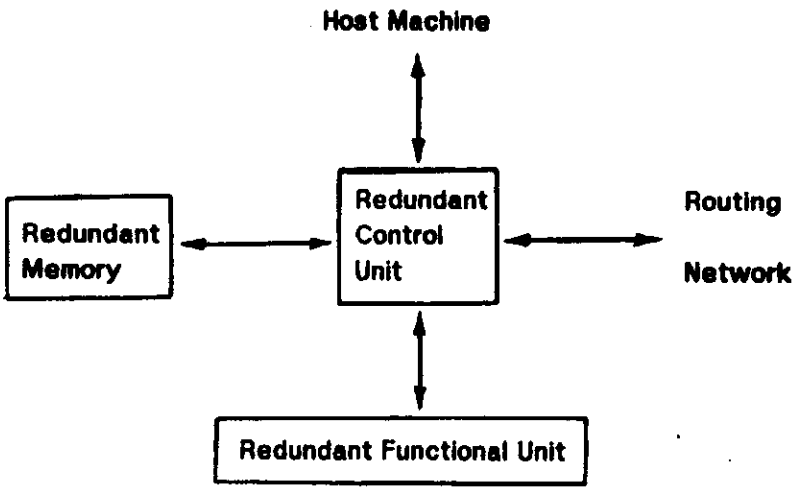


Fig 5. Fault-tolerant processing element design.

fault tolerance capabilities, as the redundant router module shown in Fig. 4. The control unit consists of four failure-independent packages each of which receives all input signals delivered to the control unit. The outputs of these four packages are grouped together or voted upon to form outputs of the redundant control unit.

Addresses and data transmitted between the redundant control unit and the bit-sliced memory system, and those transmitted between the control unit and the host machine, are encoded using an error-correcting code. Each bit-slice in the memory system stores one bit of a data word and has its own address decoder. Any hardware failure confined to within one bit-slice thus affects at most one bit of a data word and consequent errors can be corrected. Packet bytes transmitted between the redundant control unit and the routing network are parity-encoded. Since the redundant control unit has the same fault-tolerance capabilities as the redundant router, single package failures in the control unit cannot cause undetected erroneous packets to be delivered to another PE.

The functional unit is the only subunit in a PE that need not be completely fault-tolerant. Package failures in it must nonetheless be detectable. Many redundancy techniques are available for detecting failures in functional units. For a commercially available LSI functional unit chip, it is cost effective to detect single chip failures through duplication and mask these failures through triple modular redundancy, as desired. Communication between the redundant control unit and the redundant functional unit can also be protected using either an error-correcting code or an error-detecting code, depending on whether failures in the functional unit are to be masked or detected.

5. Network Repair Strategies

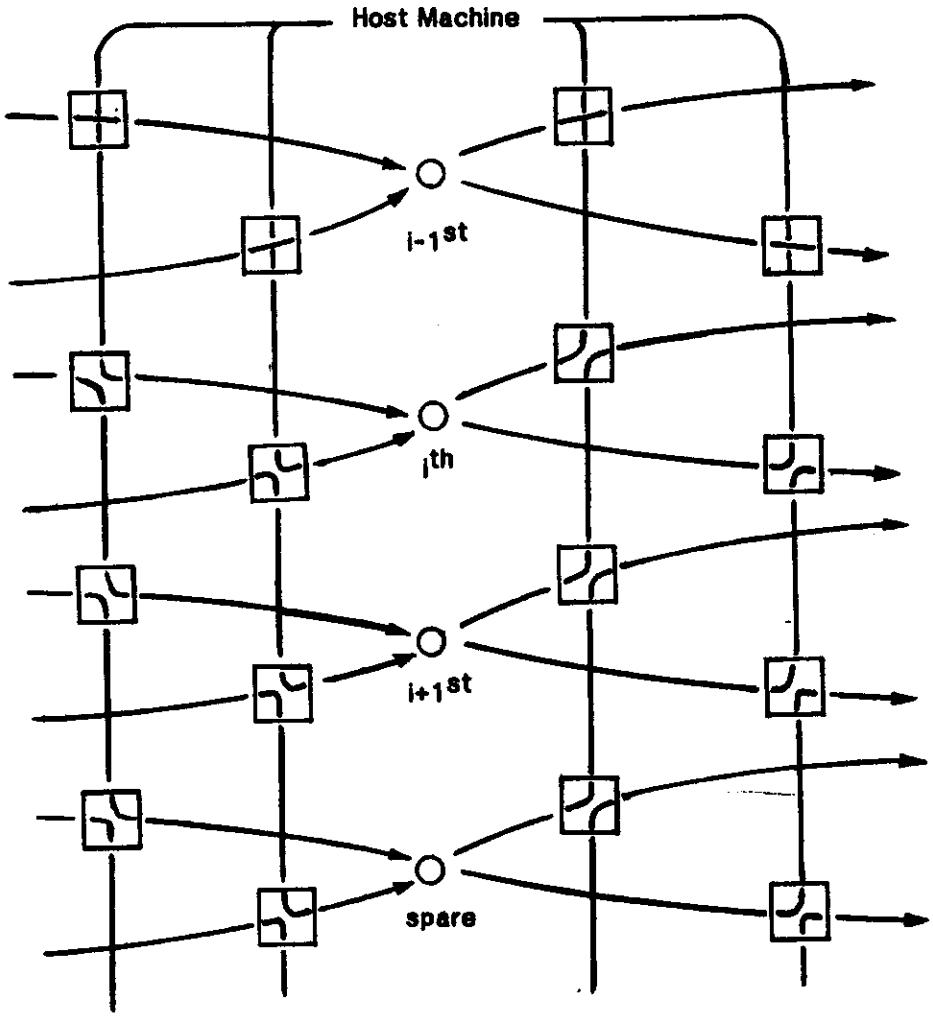
When a network hardware failure is detected in the dynamic redundancy scheme, all PEs will stop sending packets to each other and after a predetermined time period the networks will be dormant. Before normal processing can resume the failed unit must be located and the networks must be repaired. In this section we will illustrate two repair strategies using routing networks.

The first step in any repair procedure is to locate the failed router. Each checker package can generate an error signal upon detecting a parity violation. Failures in the last network stage are detected directly by parity checkers in the PEs and SFUs. Since checkers are quadruplicated in each router, PE and SFU, two or more error signals will be generated for each legitimate complaint under the single package failure assumption. These error signals can be used to locate the failed router. Further diagnosis will be necessary to locate the failed package(s).

The most straightforward repair procedure is to make use of error signals generated by checker packages to locate the failed unit and then replace it *manually* with a spare. This procedure requires no additional hardware, but system availability is directly related to the availability of maintainence personnel. The personnel requirement can be reduced by incorporating self-repair features into a network. The error signals will be monitored by the host machine which will direct repair activities. Additional modules or data paths must be incorporated directly into the routing network to support self-repair.

In the self-repair scheme illustrated in Fig. 6, a number of spare modules are appended to each routing network stage, switched in electrically to replace failed modules. Switching arrangements are incorporated systematically using *switch* packages, which have been introduced in [18], to support system reconfiguration. A switch can be set in one of two modes, either "crossing" or "bending" (Fig.

(a) The *switch* module



(b) Reconfigurable network stage

Fig. 6. A reconfiguration scheme for self-repair

6a) the pair of input leads to the pair of output leads. Spare routers are interspersed with active routers. The reconfiguration capability of this switching arrangement is illustrated in Fig. 6b where the i^{th} router is assumed faulty. Note that in this scheme a spare router cannot replace any faulty router below it in the column. Control signals for setting the switches can also be carried in the interswitch connections. This repair scheme requires many additional data paths and packages, and must be further enhanced to tolerate switch failures. It is thus practical only when the additional hardware costs are acceptable and the switches are much more reliable than the router modules. One technique to tolerate switch failures is to connect each switch to more than one neighboring switch so that an immediate neighbor which has failed can be bypassed. These switches are called *ripplers* in [24].

The additional data paths introduced can also be used for off-line diagnosis, testing out the routers systematically with pregenerated test patterns. In the configuration shown in Fig. 6, the i^{th} router can be tested by the host machine while the remaining routers carry the packet traffic. The fault detection mechanism in the dynamic redundancy scheme assumed single package failures in each router. Multiple package failures or lurking failures which have not yet manifested themselves are not detected. Network reliability can be further improved by testing the routers for these failures periodically or after detecting a fault in software.

In the above strategy the topological and operation characteristics of a rectangular routing network are retained after reconfiguration. In a rectangular network any router, except for those in the last stage, can be paired together with a neighbor in the same stage such that the two can be used interchangeably in packet routing. If one router in a pair fails, its duty can be taken over by its partner. The network can continue to operate, possibly under degraded performance, if at most one router in each pair has failed. This scheme can be implemented by adding two input ports and two output ports to each redundant router. If there is a path in the nonredundant network from router A

to router B, a new path between A and B's partner is added. The redundant paths incorporated into an 8 x 8 network (Fig. 3b) are shown in Fig. 7. The last stage can be repaired by using the previous scheme. A packet can be forwarded to its destination along two different paths at each enhanced router. Both of these paths can be used during normal operation when all routers are fault-free, or one of them may be designated a spare to be used only when the other path is blocked by a failed router. Information on the location of failed routers can be distributed by the host machine during repair, to disable connections to failed routers.

The host machine can keep a count of the number of failures reported for each router and take it off-line only when a predetermined maximum failure rate is exceeded. Spare modules can then be better utilized when transient failures dominate. We also note that neither of these repair schemes require recomputing destination addresses in a partially executed data flow program to complete its

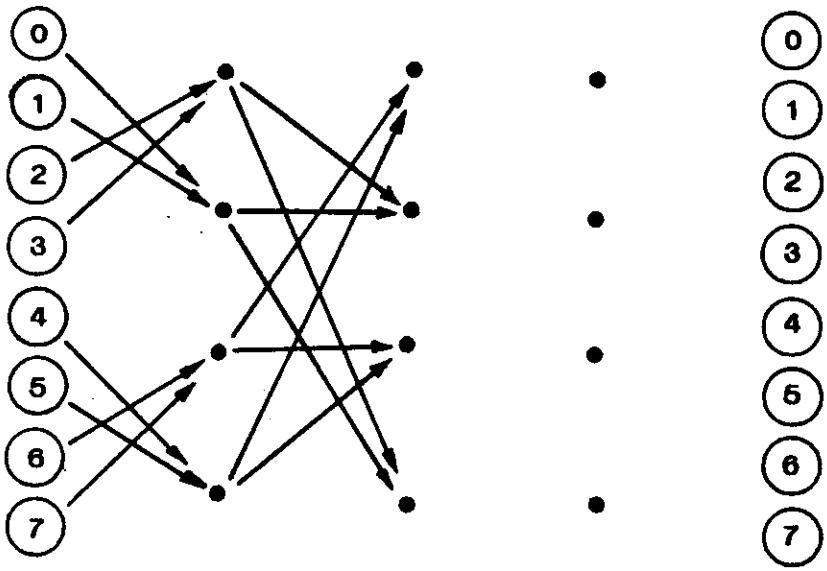


Fig. 7. Redundant data paths in a fault-tolerant 8 x 8 network.

execution after being interrupted by a network failure.

6. Concluding Remarks

The STAR computer [1] and the FTSC computer [15] are two examples of fault-tolerant computing systems based on dynamic redundancy. Both of them have a bus-oriented architecture designed for executing sequential programs. In this paper we have presented a dynamic redundancy scheme for masking hardware failures in a multiprocessor system designed to execute parallel programs organized by data flow concepts. These programming concepts and the quest for high performance also distinguishes our work from other fault-tolerant multiprocessor projects such as the FTMP system [14] and the SIFT system [28].

The dynamic redundancy scheme is explained by giving strategies for program organization and execution, and for co-ordinating fault-related activities such as fault detection, diagnosis and repair with normal execution. We have also described hardware redundancy and packet encoding techniques for implementing hardware modules and subsystems to support these strategies.

We have assumed that hardware packages fail under normal use, and that failures are readily repaired. The redundancy schemes have thus been presented assuming at most one package failure in each hardware module. As long as there is at most one failed package in any module, the computation in progress can always be completed. If the PEs are not designed to mask all single package failures, it may be necessary to relocate the activity templates and the partial state of the computation stored in a failed PE, and relink the activity templates, before program execution can be resumed. The redundancy techniques can be extended to accommodate multiple package failures by using more packages in each router and more elaborate coding techniques. In a physical realization several packages can share a physical unit as long as the physical system is partitioned so

that under the most common failure modes at most one package in each module can fail.

We have demonstrated how to methodically deal with hardware failures in a practical implementation of a highly parallel data flow processor, with no impact on its programmability. We have explained how hardware failures can be masked when the architecture is programmed in a restricted data flow language. Another operational restriction is that every packet transmitted over the packet transport and processing subsystem is acknowledged by another packet. The fault tolerance techniques proposed in this paper are directly applicable whenever the hardware architecture is programmed under these restrictions. It is expected that more sophisticated system strategies must be developed to incorporate fault tolerance cost-effectively into more advanced data flow architectures.

When detailed logic designs and hardware failure rates are available for a hardware implementation, alternative schemes should be carefully evaluated with reliability models [20] to determine their cost-effectiveness.

References

[1] A. Avizienis, et al., "The STAR (Self-Testing-And-Repairing) computer: An investigation of the theory and practice of fault-tolerant computer design," *IEEE-TC*, vol. C-20, no. 11, pp. 1312-1321, Nov. 1971.

[2] A. Avizienis, "Fault-tolerance and longevity: goals for high-speed computers of the future." *Proceedings of the Symposium on High Speed Computer and Algorithm Organization*. Academic Press, 1977, pp. 173-178

[3] A. Avizienis, M. Ercegovac, T. Lang, P. Sylvain & A. Thomasian "An investigation of fault-tolerant architectures for large-scale numerical computing." *Proceedings of the Symposium on High Speed Computer and Algorithm Organization*. Academic Press, 1977, pp. 159-171

[4] B. R. Borgerson, "Spontaneous reconfiguration in a fail-softly computer utility," *Proc. Datafair 73*, British Computer Society, London, pp 326-333, 1973.

[5] G. A. Boughton, *Routing Networks in Packet Communication Architectures*, Dept. of Electrical Engineering and Computer Science, M.I.T., S.M. Thesis, June 1978.

[6] W. C. Carter and C. E. McCarthy, "Implementation of an experimental fault-tolerant memory system," *IEEE-TC*, vol. C-25, pp. 557-568, June 1976.

[7] D. Daniels and J. F. Wakerly. "Synchronization and matching in redundant systems" *IEEE-TC* vol. C-27, no. 6, pp 531 - 539, June 1978

[8] W. M. Daly, A. L. Hopkins and J. F. McKenna. "A fault-tolerant digital clocking system" *Dig. 3rd Int. Symp. Fault-Tolerant Comp.* Palo Alto, CA, pp 17 - 22, June 1973

[9] J. B. Dennis, "The varieties of data flow computers" *Proceedings of the 1st International Conference on Distributed Systems*, IEEE, October 1979.

[10] J. B. Dennis, "First Version of a data flow procedural language," *Lecture Notes in Computer Science*, vol. 19, New York: Springer-Verlag, pp. 362-376, 1974.

[11] J. B. Dennis, G.A. Boughton & C.K.C. Leung "Building blocks for data flow prototypes," *Proceedings of the 7th Annual Symposium on Computer Architecture*, IEEE, pp. 1-8, May 1980.

[12] J. B. Dennis, & D.P. Misunas "A preliminary architecture for a basic data-flow processor." *Proceedings of the 2nd Annual Symposium on Computer Architecture*, IEEE, New York, 1975, pp. 126-132.

[13] J. B. Dennis, C.K.C. Leung & D.P. Misunas "A highly parallel processor using a data flow machine language" M.I.T. Laboratory for Computer Science, Computation Structures Group, Memo 134-1, Cambridge, Mass, also to appear in IEEE Transaction on Computers.

[14] A. L. Hopkins, Jr., T. B. Smith, III, and J. H. Lala "FTMP - A highly reliable fault-tolerant multiprocessor for aircraft" *Proc. IEEE*, vol 66, 10, pp 1221 - 1239, October 1978.

[15] Session on the fault-tolerant spaceborne computer, *Proceedings of the 1976 International Symposium on Fault-Tolerant Computing*, IEEE, June, 1976.

[16] J. Galiay, Y. Crouzet and M. Vergniault, "Physical versus logical fault models in MOS LSI circuits, impact on their testability," *Proc. of the 1979 Symp. on Fault-Tolerant Computing*, IEEE, pp 195-202, June, 1979.

[17] C. K. C. Leung, "*Fault tolerance in a packet commmunication computer architecture,*" doctoral thesis in preparation, Dept. of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass.

[18] K. N. Levitt, M.W. Green, & J. Goldberg, "A study of data commutation problems in a self-repairable multiprocessor." *AFIPS Conference Proceedings, vol. 32,* (1968 SJCC), Thompson Book Company, Washington, D.C, 1968, pp. 515-527.

[19] D. P. Misunas, "Deadlock avoidance in a data flow architecture." *Proceedings of the Milwaukee Symposium on Automatic Computation and Control,* IEEE, April 1975.

[20] Y. W. Ng and A. Avizienis, "A reliability model for gracefully degradable and repairable fault-tolerant systems," *Proc. 1977 Int. Symp. Fault-Tolerant Computing,* IEEE, pp. 22-28, June 1977.

[21] Sessions on Data Flow Computer Architectures. Proc. of AFIPS Conference, 1979.

[22] Sessions on Data Flow Computer Architectures. Proc. of Compcon 80, IEEE, Feb. 1980.

[23] M. Pease, R. Shostak and L. Lamport "Reaching agreement in the presence of faults," *JACM,* vol. 27, no. 2, pp. 228-234, April 1980.

[24] J. J. Stiffler, N. G. Parke IV, and P. C. Barr, "The SERF fault-tolerant computer," Parts I and II, *Dig. 1973 Int. Symp. fault-tolerant computing,* Palo Alto, CA, June, 1973.

[25] W. N. Toy, "Fault-tolerant design of local ESS Processors," Proc. IEEE, pp 1126-1145, Oct. 1978.

[26] E. Vishniac, "*A processor module for data flow computer development,*" Laboratory for Computer Science, M.I.T., CSG Memo 176, May 1979.

[27] R. L. Wadsack, "Fault modeling and logic simulation of CMOS and MOS integrated circuits," *Bell System Technical Journal,* vol. 57, no. 5, pp 1449-1473, May-June 1978.

[28] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT - The design and analysis of a fault-tolerant computer for aircraft control" *Proc IEEE,* vol 66, no. 10, Oct 1978, pp 1240 - 1255.