Proceedings of the
13th International Conference on Parallel Processing,
Bellaire, Michigan, pp. 32-41, August 1984.

# ERROR RECOVERY IN MULTICOMPUTERS USING GLOBAL CHECKPOINTS

*Yuval Tamir and Carlo H. Séquin*
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

**Abstract** — Periodic checkpointing of the entire system state and rolling back to the last checkpoint when an error is detected is proposed as a basis for error recovery on a VLSI multicomputer executing non-interactive applications. Detailed algorithms for saving the checkpoints, distributing diagnostic information, and restoring a valid system state are presented. This approach places no restrictions on the actions of the application tasks, and, during normal computation, does not require the complex communication protocols that are part of most other schemes. Estimates of the overhead of the proposed scheme are presented and extensions for efficient handling of transient faults, input/output operations, and disk failures are discussed.

## I. Introduction

Some applications, such as large circuit simulation, weather forecasting, and aeronautical design, require very high performance computer systems that are also highly reliable. The performance requirements imposed by these applications originate from their complexity (size) rather than from real time constraints. The ''size'' of these problems also determines the reliability requirements for the system — ''correct results'' must be produced after many hours of continuous operation.

In order to achieve maximum performance, the parallelism inherent in the tasks to be performed must be exploited. One possible system architecture that can exploit parallelism and is compatible with the constraints of VLSI uses a large number of computation nodes interconnected by high-speed dedicated links [9, 19]. The feasibility of such a system (henceforth called a *multicomputer*) is demonstrated by the recently announced Inmos *Transputer* [3].

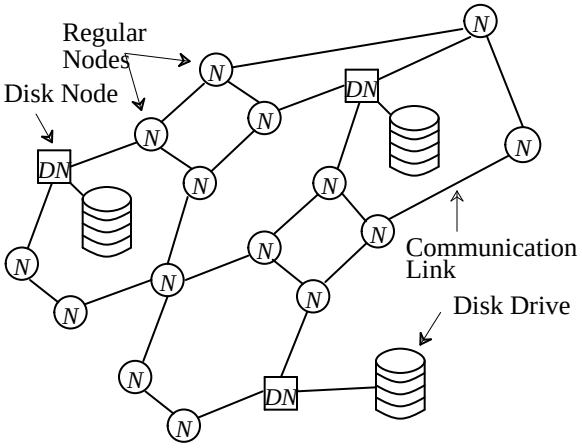


**Fig. 1:** *A Multicomputer*

In order to achieve system reliability that exceeds the combined reliability of all of its components, fault tolerance techniques must be used. These techniques require the ability to detect errors (*error detection*), to identify the components responsible for the corruption of the system state (*fault location*), and to reestablish a valid, consistent system state that allows continued operation of the system (*error recovery*) [1]. A multicomputer system is especially well suited for reliability enhancement using fault tolerance techniques since it contains no single component, such as a shared memory or bus, whose operation is critical to the continued operation of the system.

In a distributed system, reestablishing a valid system state, such that the state of each component is consistent with the state of all other components, is much more difficult than in a uniprocessor [16]. Some recovery techniques suggested in the literature are suitable for systems where interprocessor communication is over a common bus or Ethernet but are unsuitable for a multicomputer [4]. Other proposed techniques restrict communication between the processors and/or require devoting a significant fraction of the processing power and communication bandwidth to maintaining the information necessary for recovery [11]. While this high overhead may be necessary for interactive applications with real-time constraints, alternative approaches are needed for batch applications, such as those mentioned above, where very high throughput is essential but an occasional delay due to error recovery is tolerable.

This paper proposes and discusses a scheme for error recovery in a multicomputer based on the use of *global checkpoints*. The entire system state is periodically saved on disks and that information is used to restore a valid system state after an error has been detected. This approach is suitable for non-interactive, computation-intensive applications such as those discussed above. No restrictions are imposed on the ''behavior'' of application tasks, and message transmission is not slowed down by the complex acknowledgement and timeout mechanisms required by most schemes.

We begin by discussing the requirements for error detection and error recovery in a multicomputer. Following a brief review of alternative error recovery schemes, we present detailed algorithms for checkpointing the entire system state, distributing diagnostic information, and using the checkpointed state for error recovery. An informal ''proof'' that the algorithms are correct is presented, and an estimate of the overhead required by this scheme is given. Finally, we discuss how the scheme can be expanded to allow interactions with the ''outside world,'' deal more effectively with transient faults, reduce the latency in detecting errors in communication links, and handle the failure of disks and the nodes that control them.

## II. Fault Tolerance Requirements

We now discuss the requirements for error detection and error recovery in a multicomputer that consists of thousands of VLSI chips and is used for large non-interactive applications. We assume that chips fail independently of each other, and that failure rates are constant, resulting in exponentially distributed interarrival times (Poisson arrivals) of failures.

One of today's high-performance systems, the CRAY-1, contains more than 100,000 integrated circuits. Assuming that similar effort and resources will be spent on future high-end computer systems, any such system will contain thousands of chips. The rate of permanent faults in VLSI chips is currently on the order of one per million part hours [15, 5] while the rate of transient faults is at least an order of magnitude higher [7].

This work was supported by the Defense Advance Research Projects Agency (DoD), ARPA Order No. 3803, monitored by Naval Electronic System Command under Contract No. N00039-81-K-0251.

Hence, in a system that consists of 2,000 VLSI ICs, the rate of hardware faults resulting from chip failure alone is on the order of one transient fault every fifty hours and one permanent fault every five hundred hours. Thus, the probability of a hardware fault during the execution of a large task cannot be ignored.

A very high probability of detecting errors caused by hardware faults is required in order to have confidence that the results produced by the system are correct. Self-checking nodes implemented by duplication and matching [13, 21] with a self-testing comparator [6, 22] can ensure the detection of errors caused by faults in the ICs. Errors in message transmission can be detected with error-detecting codes such as checksums or cyclic codes [8].

After an error is detected, the system must be reset to some consistent state so that it can resume useful work. A possible approach to this error recovery process is to abort the task immediately when an error is detected, repair and reset the system, and then restart the task from scratch. Unfortunately, based on the above discussion, it is clear that the MTBF of a large multicomputer system is only on the order of tens of hours. For a large simulation task requiring, say, 100 hours of continuous execution, the expected number of times that the job would have to be restarted from scratch is on the order of several thousand. Hence, there is a need for an error recovery technique that allows the system to recover most of the work completed prior to the occurrence of a fault.

## III. Error Recovery Using Global Checkpoints

Much of the work on error recovery assumes a system where all communication is over a common bus or Ethernet [4, 16]. This allows the implementation of a ''recording node'' that keeps a record of all inter-node messages transmitted in the system [16] and facilitates the implementation of an efficient *atomic* operation that transmits a message to a ''primary'' process and to its ''backup'' that resides on another node [4].

On a *multicomputer*, communication is point-to-point between nodes. In order to keep track of messages that are transmitted throughout the system, they must be explicitly forwarded to the ''recording node'' [16] or to the ''backup node'' [4]. This requires extra delays in processing since before taking any action that assumes that a message has been reliably transmitted, an acknowledgement from the destination and the backup node must be received.

Barigazzi and Strigini propose an error recovery procedure for multicomputers that involves periodic saving of the state of each process by storing it both on the node where it is executing and on another backup node [2]. The critical feature of this procedure is that all interacting processes are checkpointed together, so that their checkpointed states can be guaranteed to be consistent with each other. Therefore, the *domino effect* that may require backing up to successively older states [18] cannot occur. As a result, it is sufficient to store only one ''generation'' of checkpoints.

With the recovery scheme described in [2] a large percentage of the memory is used for backups rather than for active processes. The resulting increased paging activity leads to increases in the average memory access time and the load on the communication links. This load is also increased by the required acknowledgement of each message and transmission of redundant bits for error detection. The communication protocols, which are used to assure that the message ''send'' and ''receive'' operations are atomic, require additional memory and processing resources for the kernel. Thus, performance is significantly reduced relative to an identical system where no error recovery is implemented.

The idea of simultaneously checkpointing the state of all processes belonging to the same ''task'' can be taken a step further: simultaneous checkpointing of the complete state of all the user and system processes on the system. A new global checkpoint is periodically stored on disks. When an error is detected, diagnostic information is distributed throughout the system. Normal operation is resumed after all the operational nodes are set to a consistent system state using the last checkpoint.

Although creating and saving a global checkpoint is expensive, if the time between checkpoints is sufficiently large compared with the time it takes to establish a new checkpoint, the net system overhead for error recovery is relatively small. In a large multicomputer the expected time to establish a new checkpoint is on the order of one minute (see Section V). Thus, keeping the overhead low requires that a new checkpoint be established only once or twice an hour. It is clear that the loss of as much as an hour of processing when an error is detected is tolerable only for non-interactive applications.

In the following six subsections we describe in detail the creation and storing of a global checkpoint and its use for recovery. In subsection A we present some basic assumptions that are made about the system. In subsection B we differentiate between *normal* packets that are used for the application tasks and *fail-safe* packets that coordinate the creation of checkpoints and recovery from errors. Subsection C describes the eight types of fail-safe packets used by the system. At each point in time, a node may be engaged in normal computation, creation of a checkpoint, or recovery from an error. Subsection D describes the possible modes or *logical states* of a node. Subsection E describes how a consistent global checkpoint is established and stored on disk. Finally, in subsection F, we show how the global checkpoints can be used to recover from errors.

### A. Assumptions

The algorithms for establishing global checkpoints and for using those checkpoints for error recovery are based on several simplifying assumptions. We will discuss later how some of these assumptions can be relaxed.

We assume a closed system that consists of nodes, links, and disks. Input is stored on disk before operation begins. Output is stored on disk when the job ends.

The nodes are self-checking and are guaranteed to signal an error to their neighbors immediately when they send incorrect output [21]. Any node that generates an error signal is assumed to be permanently faulty and no attempt is made to continue to use it.

Hardware faults either cause a node to generate an error signal or cause an error in transmission. A fault can occur at any time, including during the creation of a checkpoint. However, if a second fault occurs during recovery from a previous fault, the system stops execution and does not attempt recovery. This and other situations where the system must stop due to an unrecoverable state will henceforth be called a *crash*. It should be noted that, even if a crash occurs, the system still does not generate incorrect results. Furthermore, since recovery takes only a few minutes and the system has an MTBF of tens of hours, the probability of a fault occurring during recovery is very small.

Since the disks are extensively used for paging, checkpointing, and I/O, the average number of ''hops'' from

each node to the nearest disk should be small. Hence, disks are connected to several nodes throughout the system. As a first step, we allow an error in disk I/O or a fault in a node that controls a disk (henceforth called a *disk node*) to cause a crash. Each disk node uses an error-detecting code for all data written on the disk. When the node reads from the disk, any error caused by a faulty disk or a fault on the path between the node and the disk is detected by using the code. If an error is detected, the disk node signals a crash.

The structure of the system is relatively stable — it changes only due to hardware faults. Since all the nodes in the system are informed of each fault, every node is able to maintain tables that reflect the structure of the operational part of the system. This includes information about which nodes and links are operational and which nodes are disk nodes.

Each node has a unique identifier and there is a total ordering of these identifiers. All the nodes in the system know the identifiers of all the other nodes. For simplicity, we assume that the identifiers of an $n$ node system are the integers 1 through $n$.

To send a message, a process assembles it in memory and executes a system call. The kernel may divide the message into *packets* which are the unit of information actually transmitted. Packets may arrive at their destination out of order (if they arrive at all). It is the responsibility of the receiving node's kernel to order the packets before delivering them to the receiving process.

The interconnection topology of the system is of crucial importance for achieving fault tolerance. A large body of research on the tradeoffs between various topologies is available [24, 17] and will not be discussed in this paper. One parameter that is especially important for fault tolerance is the *connectivity* of the network. The node/edge connectivity is the minimum number of nodes/edges whose failure partitions the network so that there is at least one pair of working nodes that can no longer communicate. We assume that the connectivity of our system is sufficiently large that there is a very low probability of partitioning. Hence, it is acceptable if partitioning causes a crash.

If a node or a link fails, routing of packets through the network has to be modified to use alternate paths. This process of *reconfiguration* requires updating routing tables throughout the network. We assume that one of the well-known reconfiguration procedures [20] is used in conjunction with our recovery scheme but do not discuss this problem any further in this paper.

### B. Normal Packets and Fail-Safe Packets

One of the main advantages of our error detection and error recovery schemes is that it does not require the substantial delays in normal inter-processor communication that are a necessary part of most other such schemes. In particular, during normal operation no redundant bits are transmitted with the messages or packets for error detection, no acknowledgement of messages or packets are transmitted by their recipients, and neither processes nor processors have to wait for acknowledgement of messages or packets.

Since the probability of an error in transmission is low, it is wasteful to check the validity of each message or packet independently. Instead, in each node each port has two special purpose registers for error detection. One of these registers contains the CRC (Cyclic Redundancy Check) check bits for all the packets that have been sent from the port. The other register contains the CRC check bits of all the packets received. These special purpose registers are linear feedback shift registers (LFSRs) and their contents are updated in parallel with the transmission of each packet [8].

In order to check the validity of all the packets transmitted through a particular link, each node sends to its neighbor the contents of the LFSR used for outgoing packets. The neighbor can then compare the value it receives with the value in its LFSR for *incoming* packets and signal an error if it finds a mismatch. All the links in the system must be checked in this way before committing to a new checkpoint. Otherwise, the state of a node corrupted by an erroneous message may be checkpointed and later used for recovery.

The information in packets used to coordinate the creation of checkpoints and for error recovery must be verified before it is used. Hence, for these packets, an error detecting code is used and redundant bits are transmitted with the packet. Thus, there are two types of packets in the system: the normal packets that do not include any information for error detection, and special control packets that are used only for transmitting information between kernels and include a sufficient number of redundant bits so that any likely error in transmission will be detected. These special packets are called *fail-safe* packets since they are either error-free or the error is easily detectable by the receiving node.

As discussed in Section VI-C, it is possible to speed up the detection of errors caused by faulty links if some redundant bits are transmitted with each normal packet. However, even if this is done, the normal packets can still be handled more efficiently than the fail-safe packets. In particular, the latency associated with forwarding a normal packet through a node can be significantly reduced if the node can begin forwarding the packet before all of it has arrived [19]. This is not possible for a fail-safe packet since a node receiving such a packet must verify that it is correct before forwarding it. A node receiving a normal packet may begin forwarding it immediately and initiate error recovery if, after the complete packet is received, it is found to be invalid.

The first bit of each packet is used to distinguish between a normal packet and a fail-safe packet. If the bit is 0, the packet is usually accepted and processed or forwarded regardless of whether it is correct or not. The LFSR for incoming packets is updated as the packet is received. If the first bit is 1, the packet is not accepted until it is checked. The two LFSRs in each port are not modified by incoming or outgoing fail-safe packets.

The above scheme works even if a fault on the link modifies the first bit of the packet. If the original packet is a normal packet, the fault causes it to become a fail-safe packet. The receiving node checks the packet, assuming it is coded using some error-detecting code, and finds an error. If the original packet is a fail-safe packet, the fault causes it to become a normal packet. The LFSR for incoming packets in the receiver node is modified. The error is detected when the two nodes compare the value of the sender's LFSR for outgoing packets with the value of the receiver's LFSR for incoming packets.

### C. Types of Fail-Safe Packets

Any two nodes $i$ and $j$ are *neighbors* if, and only if, there is a link between them.

For every pair of neighbor nodes, $i$ and $j$, $CKV(i,j)$ is the correct CRC check vector of all the normal packets sent by $i$ to $j$ since the last checkpoint. At any point in time $CKV_i(i,j)$ is the value of $CKV(i,j)$ generated and stored in the LFSR in

node $i$. $CKV_j(i,j)$ is the value of $CKV(i,j)$ generated and stored in the LFSR in node $j$.

There are eight types of fail-safe packets:

*checkp(CKV)*: Used to initiate the creation of a new checkpoint. When sent by some node $i$ to its neighbor node $j$ it contains $CKV_i(i,j)$.

*state(dest,node,seq,size)*: Used to transmit the state of node *node* to node *dest* using fixed length packets. The *size* field contains the number of these packets required to transmit the entire state. Each packet includes a sequence number *seq*. These packets are used to transmit the state of a node to a disk node during checkpointing and to transmit the state of a node from a disk node during recovery.

*saved(coord,node)*: Used by a disk node to inform the checkpointing coordinator *coord* that the disk node is prepared to commit to a new state for node *node*.

*resume*: Used to signal the end of a checkpointing ''session'' or the end of a recovery session.

*fault(type,location,source)*: Used to broadcast the fact that a fault has occurred and to initiate recovery. *Type* is the type of fault detected: *node*, *link*, or *unknown*. The faulty node or link is indicated by *location*. The node that detected the error and initiated the distribution of diagnostic information is indicated by *source*.

*recover(version)*: Used to let the disk nodes know which version of the node states stored on their disks they should recover. *Version* may be 0 or 1.

*restored(coord,node)*: Used by the node *node* to inform the current checkpointing coordinator that *node* has received its complete state (as part of the recovery process) and is ready to resume normal operation.

*crash(type,location,source)*: Used to broadcast the fact that an unrecoverable situation has been encountered. The arguments are the same as those for the *fault* packet.

**D. The Logical States of a Node**

At any point in time, a node in the system may be engaged in normal operation, checkpointing, distribution of diagnostic information, or error recovery. The node's response to various packet types depends on its current activity. Hence, we can define several *logical-states* (henceforth *l-states*)[a] that are simply labels for the current activity of the node:

*normal*: The l-state of the node during normal operation. Normal packets are accepted and processed. A *checkp* packet causes an l-state transition to *checkp-begin*. A *resume* packet is ignored.

*checkp-begin*: The l-state of the node after it has received the first *checkp* packet from one of its neighbors but before it receives a *checkp* packet from all of its other neighbors. The checkpointing coordinator enters this l-state when it initiates checkpointing. Normal packets may be received only from neighbors that have not yet sent a *checkp* packet. Normal packets from other neighbors cause a transition to the *error* l-state. The arrival of valid *checkp* packets from all the neighbors causes an l-state transition to *checkpointing*.

*checkpointing*: The l-state of the node after it has received *checkp* packets from all its neighbors but before it completes sending its state to a disk node. Receipt of a normal packet causes a transition to the *error* l-state. The fail-safe packets *state* and *saved* may be received. Once the node completes sending its entire state to a disk node, it changes its l-state to *checkpointed*.

*checkpointed*: The node's l-state after it has completed sending its state to a disk node but before it receives the *resume* packet. Receipt of a normal packet causes a transition to the *error* l-state. The fail-safe packets *resume*, *state*, and *saved* may be received. A *resume* packet causes an l-state transition to *normal*.

*error*: The l-state of the node after it has detected (or has been informed of) an error but before it is ready to accept its recovered state. The node enters this l-state upon receiving a mismatch signal from a neighbor, an invalid fail-safe packet, a normal packet when only fail-safe packets are expected, or a *fault* packet. A transition to the *error* l-state may also be caused by a valid fail-safe packet whose contents indicate some error condition (see next subsection). Normal packets are ignored if sent by neighbors that have not yet sent a *fault* packet. Other normal packets cause a transition to the *crashed* l-state. The fail-safe packets *checkp*, *saved*, *resume*, *state*, and *restored* are ignored if sent by neighbors that have not yet sent a *fault* packet. The *fault* packet is ignored if the *location* it refers to is the same as the location of the fault that caused the transition to the *error* l-state. Any other *fault* packet causes a transition to the *crashed* l-state. The *recover* packet causes a transition the *recovering* l-state.

*recovering*: The node's l-state after receiving the *recover* packet but before it is ready to resume normal operation with its recovered state. Receipt of a normal packet causes a transition to the *crashed* l-state. The *recover* packet is ignored. The packets *state* and *restored* are processed (see Subsection F). The arrival of the node's complete state via *state* packets causes a transition to the *recovered* l-state.

*recovered*: The node's l-state after receiving its complete recovered state but before resuming normal operation. Receipt of a normal packet causes a transition to the *crashed* l-state. The packets *state* and *restored* are processed (see subsection F). All *recover* packets are ignored. The *resume* packet causes a transition to the *normal* l-state.

*crashed*: The l-state of the node after an unrecoverable error has been detected.

Each node includes the ''state variable'' *version* that determines what is the most recent *valid* version of the node's state that is stored on disk. This variable may have the values 0, 1, or *unknown*. When the system is initialized, the value of *version* in all the nodes is set to 0.

**E. Creating a Global Checkpoint**

The scheme for saving a consistent global checkpoint is an adaptation of the standard two-phase commit protocol used for preserving consistency in distributed data base systems [10].

Initially, a designated node, say node 1, is assigned to serve as the coordinator for establishing global checkpoints. If the coordinator fails, all the other nodes are notified and the next node, according to the total ordering between the nodes, takes over the task of being checkpointing coordinator.

Every node includes a ''timer'' that can interrupt the node periodically. Checkpointing is initiated by the checkpointing coordinator when it is interrupted by its timer [2] and it is in the *normal* l-state. Checkpointing is also initiated when a task is complete so that the system can commit to the output stored on disk.

Faulty operation of the timer is detected just like faulty operation of any other part of a node. The self-checking node is implemented using duplication and matching [21]. Each

[a] The l-state of a node is not to be confused with the node's ''state'' that is the contents of the node's memory that defines the state of all the processes and packets currently on the node.

duplicate module includes its own independent timer. Even if a fault disables one of the timers, the other timer still operates and causes the module it is part of to ''behave'' differently from the module with the faulty timer.

*The Actions of the Checkpointing Coordinator*

When the checkpointing coordinator, say node $i$, initiates checkpointing, it does the following:

[1] Node $i$ stops all work on application processes and stops transmitting normal packets. The node's l-state is changed to *checkp-begin*.

[2] Node $i$ sends to every neighbor node $j$ the *fail-safe* packet $checkp(CKV_i(i,j))$.

[3] Node $i$ waits for $checkp$ packets from all its neighbors.
If a normal packet arrives, it is included with the rest of the node state that must be checkpointed.
If a $checkp(CKV_j(j,i)$ packet arrives from neighbor $j$ then: If $CKV_j(j,i) \neq CKV_i(j,i)$, $i$ changes its l-state to *error* and sends the packet $fault(link,(i,j),i)$ to all its neighbors.
If no $checkp$ packet arrives from a neighbor $j$ within some fixed time limit, $i$ changes its l-state to *error* and sends the packet $fault(unknown,j,i)$ to all its neighbors.

[4] Node $i$ changes l-state to $checkpointing$ and sends its complete state to the disk node assigned to it.

[5] Node $i$ changes l-state to $checkpointed$ and waits for fail-safe packets of the form $saved(i,j)$ for all nodes $j$ in the system that are known to be working.
If for one of the nodes, say node $j$, no such packet arrives within some fixed period of time, $i$ changes its l-state to *error* and sends the packet $fault(unknown,0,i)$ to all its neighbors.

[6] After all the expected $saved$ packets arrive, node $i$ complements its $version$ variable, changes l-state to *normal*, and sends the packet $resume$ to all its neighbors.

[7] Node $i$ resumes normal operation.

*The Actions of a Checkpointing Participant*

Every node $j$, that receives the packet $checkp(CKV_i(i,j))$ from its neighbor $i$ while in its *normal* l-state, does the following:

[1] Node $j$ stops all work on application processes and stops transmitting normal packets. The node's l-state is changed to *checkp-begin*.

[2] If $CKV_i(i,j) \neq CKV_j(i,j)$, node $j$ changes its l-state to *error* and sends the packet $fault(link,(i,j),j)$ to all its neighbors.

[3] For every neighbor node $k$ (including $k = i$), $j$ sends $k$ the *fail-safe* packet $checkp(CKV_j(j,k))$.

[4] Node $j$ waits for $checkp$ packets from all its neighbors except $i$.
If a normal packet arrives, it is included with the rest of the node state that must be checkpointed.
If a $checkp(CKV_k(k,j)$ packet arrives from neighbor $k$ then: If $CKV_k(k,j) \neq CKV_j(k,j)$, $j$ changes its l-state to *error* and sends the packet $fault(link,(k,j),j)$ to all its neighbors.
If no $checkp$ packet arrives from some neighbor $k$ $(k \neq i)$ within some fixed period of time, $j$ changes its l-state to *error* and sends to all its neighbors the packet $fault(unknown,k,j)$.

[5] Node $j$ changes its l-state to $checkpointing$, and begins to send its state to the disk node assigned to it using *state* packets. The complete node state except for one last *state* packet is sent.

[6] Node $j$ changes its l-state to $checkpointed$, sets its *old-version* variable to $version$, sets its $version$ variable to *unknown*, and sends the last packet containing its state to the disk node assigned to it.

[7] Node $j$ waits for a $resume$ packet from one of its neighbors.

[8] When node $j$ receives a $resume$ packet from its neighbor $i$, it sets its $version$ variable to $1 - old\text{-}version$, changes its l-state to *normal*, and sends a $resume$ packet to all of its *other* neighbors.

[9] Node $j$ resumes normal operation.

*The Actions of a Disk Node*

A disk node may be a checkpointing coordinator or a checkpointing participant. In either case, it executes most of protocols described above as a regular node. However, a disk node also performs two additional tasks: (1) It stores node states so that they can be recovered in case of an error. (2) It handles input/output.

During a checkpointing session, a disk node accepts $state$ packets and stores them on its disk. Once the complete state of some node $j$ is received and stored on disk, the disk node sends a $saved(coord,j)$ packet to the checkpointing coordinator. Upon receiving a $resume$ packet, the disk node commits to the most recently saved versions of the node states, which correspond to the current value of the disk node's $version$ variable.

In order to roll back the entire system to a previous state, the state of the disks must be rolled back together with the state of the nodes. All files that are opened with write or read/write permission are duplicated by the disk node and I/O operations are performed on the duplicates until the disk node commits to the next checkpoint. Newly created files exist only as ''duplicates'' until the disk node commits to the next checkpoint. At that point, all the new and duplicate files are incorporated with the committed state. Files that remain open with write or read/write permission must, once again, be duplicated since all operations until the *next* checkpoint must not affect the files that are part of the current checkpoint.

**F. Fault Handling**

When a node detects an error, it informs its neighbors of that fact. The diagnostic information is then distributed throughout the system. Since an error may be detected in the middle of creating a new checkpoint, some of the disk nodes may have access to subsets of two different system states: the state currently being checkpointed and the previous state. The first node that can determine which version of the state should be used, distributes this information throughout the system. Once the disk nodes find out which version of the system state to use, they send the state to all the other nodes. When all the nodes receive their state, they inform the checkpointing coordinator, which subsequently initiates the resumption of normal operation. Since all the working nodes are informed of the cause of the error, they can avoid using the failed node or link so that it cannot be the source of any additional errors.

The rest of this subsection is a detailed description of the actions of the nodes when an error is detected.

*The Actions of a Regular Node*

When a node is in any l-state except *error* or *crashed*, it may change to the *error* l-state at any time, as described in Subsection D. When a node $j$ enters the *error* l-state, it does

the following:

[1] Node $j$ stops all work on application processes and deletes all normal *and* fail-safe packets that are waiting for transmission. If the fault is in one of the node's neighbors or in a link to a neighbor, all communication with that neighbor is terminated.

[2] Packets that arrive at the node are handled as follows: All normal packets and all fail-safe packets, except *fault* and *crash*, are ignored. *Fault* packets are ignored if their *location* field indicates that they were generated as a result of the same fault that caused node $j$ to enter the *error* l-state.

[3] Node $j$ sends *fault* packets to all its neighbors.

[4] If the *version* variable in node $j$ is not set to *unknown*, node $j$ sends a *recover*(*version*) packet to all its neighbors.
If *version* is set to *unknown*, node $j$ waits for a *recover* packet from one of its neighbors. When the *recover* packet arrives, node $j$ sets its *version* to the value in that packet and sends a *recover*(*version*) packet to all its neighbors.
If the error is a result of a fault in the checkpointing coordinator, all the working nodes in the system may have their *version* variable set to *unknown*. Hence, if node $j$ is a neighbor of the checkpointing coordinator whose *version* is set to *unknown*, and if the fault packets indicate that the checkpointing coordinator has failed, node $j$ waits for a *recover* packet only up to some preset time limit and then sets its *version* to *old-version* and sends a *recover*(*version*) packet to all its neighbors.

[5] Node $j$ changes l-state to *recovering* and waits for its complete state to arrive from a disk node.

[6] Node $j$ sends a *restored* packet to the (possibly new) checkpointing coordinator, changes l-state to *recovered*, and waits for a *resume* packet from one of its neighbors.

[7] If node $j$ is the checkpointing coordinator, when it receives *restored* packets from all the nodes which are known to be working (including itself), it sends *resume* packets to all its neighbors and changes l-state to *normal*.
If node $j$ is not the checkpointing coordinator, when a *resume* packet arrives from one of its neighbors, it sends a *resume* packet to all its neighbors and changes l-state to *normal*.

[8] Node $j$ resumes normal operation.

*The Actions of a Disk Node*

As previously mentioned, if an error is detected in a disk node, a *crash* is initiated (in Subsection VI-D we discuss modifications to the system that enable it to recover from failed disk nodes). However, a disk node can participate, or even initiate, the recovery process when the source of the error is some other node.

During recovery, every disk node $j$ uses *state* packets to send the checkpointed state to those nodes whose state is stored on the disk controlled by $j$. The disk node begins sending the checkpointed state to the nodes after it has gone through step [4] of the recovery protocol described above, in which it determines which version of the state should be used. The disk node's value for the *version* variable following step [4] corresponds to the state that it sends out.

Disk ''files'' must be restored to a state that is consistent with the state of the nodes. If the *version* variable in some disk node is not *unknown* (*i.e.*, it is set to 0 or 1) when it first enters the *error* l-state, the system is rolled back to the checkpoint to which the disk node has already committed. Updates to the disk that were performed after the last checkpoint are undone by the disk node that removes all the duplicate files and creates new duplicates from the ''master copy'' for all the files that are marked in the master copy as open with write or read/write permission.

If the disk node is in the middle of a checkpointing session, its *version* variable may be set to *unknown*. If recovery requires rolling back to the previous checkpoint rather than to the one being established, the value of *version* changes to *old-version* when the l-state changes to *recovering*. In this case the disk node performs the same actions as when *version* is initially set to a value other than *unknown*.

If recovery requires ''rolling back'' to the checkpoint that is currently being established, the value of *version* is *unknown* when the disk node first enters the *error* l-state but changes to $1-$*old-version* when the l-state changes to *recovering*. The disk node commits to the node states that have just been received as well as to all updates that were done to the disk since the last checkpoint. All files for which there are temporary duplicates are updated with the contents of the duplicates. For files that have been closed since the last checkpoint, the duplicates are removed. For files that are still active, the duplicates remain and continue to be used by the disk node.

If the disk node receives an *error* packet in the middle of committing to a new checkpoint, it forwards the packet to all its neighbors but does not proceed with any recovery actions until it completes the process of committing to the new checkpoint.

## IV. Correctness Arguments

Since a complete proof of the correctness of the protocols presented in this paper requires a lengthy case analysis, we limit our discussion here to showing that, despite hardware faults, the system will never produce (''commit to'') incorrect results. The last action of the system, before committing to the output of a task, is to establish a new checkpoint. Hence, it is sufficient to show that a checkpointing session can complete successfully only if the states of all the nodes are correct and there are no errors in transmitting output from the various nodes to the disks. In subsection A it is shown that the states of the individual nodes are correct. In subsection B it is shown that the states of all the nodes that are saved as part of a single system checkpoint are consistent with each other.

### A. The Correctness of Individual Node States

The correct operation of two neighbor nodes and the link connecting them is verified when both enter the *checkpointing* l-state. Errors that are a result of faults in nodes are detected immediately when they occur. Two neighbor nodes can enter the *checkpointing* l-state only after they have exchanged *checkp* packets. These *checkp* packets follow any normal packets transmitted between the two nodes. If the *checkp* packets do not cause one or both of the nodes to enter the *error* l-state (*i.e.*, the CRC check bits match), then there have been no errors in the transmission of normal packets between the nodes since the last checkpoint. Thus, if both nodes enter the *checkpointing* l-state the only way for one of the nodes to have received an incorrect normal packet from the neighbor is if the neighbor was correctly forwarding an incorrect packet from some other node.

Before checkpointing is complete, all the nodes go through the *checkpointing* l-state. Thus, all the normal packets received by each one of the nodes since the last checkpoint must

be correct. The internal state of a node may be erroneous as a result of incorrect processing of *correct* packets. However, since the nodes are self-testing, an erroneous internal state is detected when the state is sent to the disk node. Thus, if *all* the nodes complete sending their state (*i.e.*, enter the *checkpointed* l-state), all those states must be valid. It remains to be shown that these node states are consistent with each other and are stored and retrieved correctly.

The state of each node is sent to a disk node using fail-safe *state* packets. Errors in the transmission of the state are detected since the packets are fail-safe. Since the nodes are self-checking, any error in forwarding the *state* packets is detected immediately. If a *state* packet from some node is lost, no *saved* packet for the node is sent to the checkpointing coordinator and the checkpointing session is never completed. Hence, a checkpointing session can be completed only if the states of all the nodes arrive at the disk nodes intact.

Since the disk nodes are self-checking, errors in the operation of these nodes is detected by neighbors and causes a crash. An error-detecting code is used for transmitting and storing information on the disk itself. During recovery, the disk nodes either retrieve the node states correctly or detect an error and initiate a crash.

During recovery, the node states are sent from the disk nodes using *state* packets. Any errors in the transmission of these fail-safe packets are detected immediately and cause a crash. Each node can determine when it receives its entire state using the *seq* and *size* fields in the *state* packets. Normal operation is resumed only after the checkpointing coordinator receives *restored* packets from all the nodes, indicating that the entire checkpointed state has been retrieved correctly.

### B. The Consistency of Node States in a Checkpoint

The state of a node changes as a result of local computation or the transmission or receipt of a normal packet. The saved states of two neighbor nodes are consistent if no normal packets are transmitted between them after the state of one of the nodes has been saved but before the state of the other node is saved [2]. When a node enters the *checkp-begin* l-state, it stops local computation and transmission of normal packets. A node enters the *checkpointing* l-state only after all its neighbors have entered the *checkp-begin* l-state. Hence, no more normal packets are exchanged with *any* of the neighbors until normal operation is resumed. During a checkpointing session, each node enters the *checkpointing* l-state and sends its state to a disk node. Therefore, the saved state of the node is consistent with the saved states of all its neighbors. Thus, if a checkpointing session is not interrupted by an error, all the node states that are part of that checkpoint are consistent with each other.

The *version* variable stored with each node is used to ensure that all the nodes will ''recover'' with consistent states, even if an error is detected in the middle of a checkpointing session. During a checkpointing session, before a node completes sending its state to a disk node, it can determine independently of any other node that recovery requires rolling back to the *previous* checkpoint rather than the one being saved. Once a node completes sending its state, it can no longer make this determination and is forced to set its *version* variable to *unknown*. The checkpointing coordinator can always determine which checkpoint should be used. It receives *saved* packets for all the nodes in the system only after all the disk nodes together have a complete consistent new checkpoint. The *resume* packet sent by the checkpointing coordinator informs all the nodes that the new checkpoint is valid. If the checkpointing coordinator is in the middle of a checkpointing session and has already received *saved* packets for all the nodes in the system, the checkpoint being established is used. Otherwise, the previous checkpoint is used and the checkpoint being established is discarded. Every node either ''knows'' to what checkpoint the system should be rolled back, or ''knows'' that it is not able to make that determination. It is not possible for two nodes to have complementary values in their *version* variables. Since the disk nodes begin sending the checkpointed state only when their *version* variable is not *unknown*, the system is rolled back to a consistent state.

When the system commits to a new checkpoint, it also commits to the output generated since the last checkpoint. Disk output in the system is sent from the various nodes to the disk nodes during normal operation using normal packets. A checkpointing session can complete successfully only if no errors occurred as a result of faults in nodes or links. Thus, if the checkpointing session completes, all the output received by disk nodes since the last checkpoint must be correct. The disk nodes append redundant bits for error detection to all information stored disk. When the system commits to output on its disks, that output is either correct or, if it is incorrect, the error can be detected when the information is retrieved based on the error-detecting code used when the information is stored.

## V. Estimate of the Overhead for Fault Tolerance

An accurate estimate of the overhead of making the multicomputer fault tolerant requires detailed simulation based on information that is only available when system design is complete. In order to provide a rough estimate of the overhead associated with the proposed error recovery scheme, we make several specific assumptions about the system based on the intended application environment and on current and near-future technology:

(1) The system includes 1,000 nodes, each consisting of a high-performance (say, 5 MIPS [3]) processor. The processor state, which does not include code-space, is, on the average, 256,000 bytes.

(2) The topology of the system is dense, *i.e.*, the *diameter* is proportional to the logarithm of the number of nodes. Specifically, we assume a diameter of 15.

(3) Ten of the nodes in the system are disk nodes, each handling checkpointing and recovery of the state of 100 nodes.

(4) The communication links are assumed to have a bandwidth of $1.5\times10^6$ *bytes/second* [3, 12].

(5) Every fail-safe packet that is not a *state* packet is 16 bytes long, including redundant bytes for error detection.

(6) Each *state* packet is 1,000 bytes long and 260 such packets are required to transmit the entire state of a node.

(7) Each node can be simultaneously receiving a packet, processing a previously received packet, and sending a previously processed packet [3, 12].

(8) The bandwidth of the interface between the disk node and the disk drives it controls is much higher than the bandwidth of the communication links, and the node can transfer data to the disk drive simultaneously with all its other activities.

In order to initiate a checkpointing session, the *checkp* packets propagate from the checkpointing coordinator to all the other nodes. The 16 byte *checkp* packet passes through a link in $11\ \mu secs$. The processing required at each node to forward the packet is relatively simple. For a 5 MIPS processor, $50\ \mu secs$

is a pessimistic estimate of the delay introduced by this processing. Since the diameter of the systems is 15, all the nodes in the system can enter the *checkpointing* l-state within one milliseconds after the checkpointing session is initiated.

The state of each node is transmitted to a disk node using 260 *state* packets. Each 1,000-byte *state* packet can be transmitted through a single link in $670\ \mu secs$. Hence, every disk node begins receiving *state* packets within one or two milliseconds after all the nodes enter the *checkpointing* l-state. A regular node receiving a *state* packet can verify it and forward it to the appropriate output port within the $670\ \mu secs$ it takes to transmit the packet over a link. Hence, even if a disk node can only receive *state* packets through one of its ports, these packets can utilize the full bandwidth of the link. Since the bandwidth of the interface between the disk node and the disk drive has a much higher bandwidth than the communication link, the disk node can store the state packets as fast as they arrive. Thus, all 26,000 *state* packets containing the state of 100 nodes can be received by a disk node in approximately 18 seconds.

Sending *saved* packets from the disk nodes to the checkpointing coordinator is a similar process to distributing *checkp* packets from the checkpointing coordinator to the rest of the system. Hence, this process is expected to take approximately one millisecond (see above). Similarly, the resume packet from the checkpointing coordinator can be distributed to the rest of the system in approximately one millisecond. Thus, the entire checkpointing session can be completed in less than 19 seconds.

If a checkpoint is created twice an hour, the overhead involved in maintaining the checkpoint is, approximately 1.1 percent.

The process of recovery is very similar to the process of creating a checkpoint — 26,000 *state* packets are transmitted from each disk node. Hence, recovery is also expected to take approximately 20 seconds. When an error is detected, the system is rolled back and any computation done since the last checkpoint is lost. Since a new checkpoint is created every 30 minutes, on the average, 15 minutes of computation are lost every time the system is rolled back. If the MTBF of the system is 10 hours, the total overhead for fault tolerance during those 10 hours includes 6.6 minutes for creating checkpoints, 0.4 minutes for error recovery, and 15 minutes of lost computation. The total of 22 minutes amounts to an overhead of 3.7 percent.

## VI. Relaxing Some of the Assumptions

In this section we outline how some of the restrictive assumptions made in Subsection III-A can be relaxed. In particular, it is no longer assumed that the system must be ''closed.'' Some communication with the ''outside world'' is allowed. Since transient hardware faults are at least an order of magnitude more likely to occur than permanent faults [7], it is wasteful to logically remove a node or a link after it suffers from a fault. A more efficient way of dealing with transient faults is proposed. Finally, modifications to the system that allow recovery from faults in disk nodes are discussed.

### A. Input/Output from the System

The basic problem in allowing communication with the outside world is that it may be impossible to roll back the effects of such communication: if a page is printed, it cannot be erased; if a file is read and then deleted, it may not be possible to restore its contents; if input is obtained from a terminal, it is not acceptable to ask the user to retype the entire session when the system recovers. The problem is especially difficult with our scheme that allows the system to continue operating incorrectly for a relatively long time (up to the time between successive checkpoints) after an error occurs.

The multicomputer consists of nodes, links, and disk drives. We call any other system (computer) or device that interacts with the multicomputer, a ''peripheral.'' Some peripherals, such as another computer system, may be able to commit to checkpoints and roll back to those checkpoints upon demand. We call such ''devices'' *intelligent peripherals*. Most peripherals, such as printers or tape drives, cannot set checkpoints and roll back to them. We call these latter devices *simple peripherals*.

We call nodes that interact directly with the outside world *peripheral nodes* or *p-nodes*. Typically, a p-node is also a disk node. Information transfer from a p-node to a peripheral is *output* while information transfer in the opposite direction is *input*.

**Intelligent Peripherals.** With intelligent peripherals, input/output may occur virtually at any time. When a p-node commits to a new checkpoint during a checkpointing session, it directs its peripherals to commit to any data they exchanged with the multicomputer since the last checkpoint.

During recovery, the peripherals are instructed to roll back to a previous checkpoint or possibly, if the error is detected in the middle of a checkpointing session, to commit to a new checkpoint. As soon as the p-node determines the correct value for its *version* variable (see Subsection III-F), it informs the peripheral which of these actions to take.

**Simple Peripherals.** Regular nodes interact with peripherals indirectly by sending input/output requests to the appropriate p-nodes. These requests are sent using normal packets, some of which may be erroneous due to faults in links that are only detected during checkpointing. Since operations on ''simple peripherals'' cannot be undone, the p-node does not execute any of the I/O requests until they are verified to be correct by a checkpointing session. Instead, I/O requests are accumulated in temporary files on disk drives controlled by the p-node. If the system is rolled back to a previous checkpoint, all these I/O requests are discarded.

When the p-node commits to the new checkpoint, it also places the accumulated I/O requests in a queue of verified peripheral operations to be executed. Since data transfers to/from the peripheral cannot be repeated, this queue is *not* part of the node state that may be rolled back during recovery. After the checkpointing session is completed, the p-node performs the peripheral operations in the queue and keeps track of all data transfers to/from the peripheral so that they are not repeated if the system is rolled back.

For input operations, the p-node first stores the data received from the peripherals in temporary files on disk and later forwards the data to the nodes that had initiated the input requests. If the system is rolled back, any packets transferring data from the p-node to other nodes are lost. These packets can be considered ''safely on their way'' only after the next checkpointing session. In order to be able to resend packets containing data from the peripherals, the p-node keeps the temporary files until the next checkpointing session. *Immediately* after they are created, these temporary files are treated as though they are part of the *previous* checkpoint. The temporary files are logically ''removed'' when the data is sent from the p-node. However, since they are part of the previous checkpoint, they are only physically removed during the *next*

checkpointing session.

The p-node must keep track of input requests that have been verified as correct (by a previous checkpointing session) but are not yet completed since the data has not yet been sent to the node that requested it. At the same time (during checkpointing) that all I/O requests are placed in the queue of peripheral operations to be executed, the input requests alone are also placed on another queue of ''input requests that are not yet completed.'' This queue is handled in the same way as the temporary files mentioned above — it becomes part of the *previous* checkpoint and all modifications to it are committed only during the next checkpointing session.

The above scheme does not allow interactive access to the multicomputer. However, new tasks may be accepted during every checkpointing session and partial outputs may be produced as the task progresses. Hence, the multicomputer is no longer a ''closed system.'' There may be regular data transfers with a host system which interacts with the users, ''prepares'' jobs for the multicomputer, and handles the output from those jobs.

### B. Handling Errors Caused by Transient Faults

Most of the errors in computer systems are a result of transient faults [7]. Such errors can corrupt the state of the system so that rolling back to the last checkpoint is necessary. However, the hardware itself is not permanently affected and should be used again once a valid system state is established.

A fault may corrupt the state of a node and prevent it from cooperating with the rest of the system in establishing a ''sane state.'' Thus, a node that fails due to a transient fault should be *reset* to some valid initial state that allows it to communicate with other nodes. Once reset, the node can obtain information about the condition of the system (e.g., which nodes or links are faulty), accept its checkpointed state, and resume normal operation. Since a link does not contain any state, no special actions are required in order to ''reset'' it.

A node's neighbors must not be given the authority to reset it since that would allow a failed node to reset its fault-free neighbors. However, if the self-checking nodes are implemented using duplication and matching [21], the ''no-match'' signal from the comparator can also be used for self-reset. This reset causes the node to begin executing resident code that is stored in ROM.

Given the ability to reset a failed node, recovery is simpler if the error is caused by a transient fault than if it is caused by a permanent fault. There is no need to reroute packets around failed nodes or links, and it is not necessary to migrate processes that were assigned to the failed node to other nodes. When an error is detected, the node or link that caused the error is identified by the *location* field in the *fault* packets. In order to distinguish transient faults from permanent faults, each node in the system keeps a record, in its own memory, of the causes of the last few errors. A node or link that is the source of several consecutive errors is considered permanently faulty by all the other nodes in the system, which make no further attempts to use it.

### C. Faster Detection of Errors Caused by Faulty Links

With the scheme described so far, errors caused by faulty links are only detected during the next checkpointing session. Even after a faulty link causes an error, the system continues processing the erroneous information until the next checkpointing session.

Errors caused by faulty links may be detected quickly if every packet includes redundant check bits that are checked after every transfer over a link. Additional overhead is required for the communication bandwidth used to transmit the redundant bits, additional delay in relaying the packet at each intermediate node on its path, and additional hardware and/or software at each node for performing the validity checks. Since this scheme does not detect lost packets, it does *not* eliminate the need for checking the links during the checkpointing session, as described earlier.

The overhead of fast detection of errors caused by faulty links may be reduced by transmitting only one set of check bits for each message and examining those bits only at the final destination of the message. In this case, the delay in detecting errors is greater and there are more possible errors that can only be detected during the checkpointing session.

### D. Faults in Disks and Disk Nodes

With the scheme described so far, the system cannot recover from errors caused by faults in disks or disk nodes. The basic problem in recovering from such errors is that critical data, such as parts of the checkpointed state, may be corrupted or no longer accessible. Solving this problem requires a system that provides multiple paths to multiple copies of the critical data.

In several commercial systems [4, 14] multiple paths to data are provided using dual-ported disk drives and disk controllers. Each disk drive port is connected to a different disk controller and the two ports of each disk controller are connected to two independent nodes. Critical files are stored on two disk drives.

The above hardware can be used in the multicomputer. It requires some coordination between the two disk nodes connected to each dual-ported controller. As long as it is fault-free, one of these nodes is *active* and performs all the I/O operations. All output operations are checked by the active disk node by reading the data immediately after it is written and verifying that it is correct based on the error detecting code being used. Once the failure of an active disk node is detected, all the other nodes are informed of the failure in the same way they are informed of the failure of any other node. The *passive* disk node becomes active and all the other nodes begin sending it all I/O requests.

Special care must be taken to ensure that the passive disk node accesses the correct version of the checkpointed state when it becomes active. This requires the active disk node to maintain ''pointers'' to the last committed checkpoint in a location on the disk known to the passive disk node. The details of the exact nature of the information kept with these pointers and the way in which they are used are presented in [23] but are omitted here due to lack of space.

## VII. Summary and Conclusions

We have shown that fault tolerance in a VLSI *multicomputer* may be achieved based on the use of global checkpoints for error recovery. The proposed scheme requires relatively low overhead which is independent of the application tasks and is not affected by changes in the number of processes executing on each node or by variations in message traffic. The main disadvantage of the scheme is that it involves periodically ''freezing'' normal execution for tens of seconds in order to save a checkpoint and the system may lose minutes of processing if error recovery is necessary. These characteristics of the proposed error recovery scheme make it appropriate for non-interactive applications which require high throughput, but unsuitable for applications with strict real-time constraints.

We have presented detailed algorithms for saving the global checkpoints, distributing diagnostic information, and restoring a valid system state following an error. Rather than being applicable to a wide range of systems and environments, these algorithms exploit the characteristics of the multicomputer and of the intended applications (a batch environment) in order to gain efficiency. As a result, the algorithms are surprisingly simple and efficient relative to previously published schemes. In particular, the following properties of the system and applications were exploited:

(1) Each node in the system is implemented using a small number of VLSI chips.

As a result, it is economically feasible to implement self-checking nodes that are virtually guaranteed to flag their own failure [21, 22]. This allows nodes to ''trust'' each other without complex diagnostic procedures, thereby greatly simplifying all of the algorithms presented.

(2) The system is *not* geographically distributed.

Short communication links allow us to assume a very low probability of a fault in a communication link. This is necessary for our scheme since every error results in the time-consuming process of rolling back the entire system. The short distance between neighbors also allows simple, reliable transmission of the error signal from each node to its neighbors. Finally, the short distance between nodes allows high-bandwidth low-latency communication. This makes the system more amenable to algorithms where the critical ''decisions'' are made in one node, thereby avoiding complex distributed synchronization and voting protocols.

(3) The application tasks are non-interactive.

With interactive applications errors must be detected as soon as they occur. This requirement results in reduced throughput due to the need for complex communication protocols involving, for example, acknowledgement of each message. By allowing some latency in error detection, the complexity and overhead of the protocols is greatly reduced.

The system must interact with the ''outside world'' and deal with the failure of mechanical devices such as disk drives. We have discussed how these problems can be handled in the multicomputer and presented a way of dealing with transient faults so that working hardware is not unnecessarily disconnected from the system.

In addition to low overhead, an important property of the proposed error recovery scheme is that it does not impose any restrictions on the ''behavior'' of application tasks. A multicomputer which utilizes this scheme can thus provide a general-purpose, high-performance batch environment in which the fault tolerance features are completely transparent to the user.

**References**


1. T. Anderson and P. A. Lee, ''Fault Tolerance Terminology Proposals,'' *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 29-33 (June 1982).
2. G. Barigazzi and L. Strigini, ''Application-Transparent Setting of Recovery Points,'' *13th Fault-Tolerant Computing Symposium*, Milano, Italy, pp. 48-55 (June 1983).
3. I. Barron, P. Cavill, D. May, and P. Wilson, ''Transputer Does 5 or More MIPS Even When Not Used in Parallel,'' *Electronics*, pp. 109-115 (November 1983).
4. A. Borg, J. Baumbach, and S. Glazer, ''A Message System Supporting Fault Tolerance,'' *Proc. 9th Symp. on Operating Systems Principles*, Bretton Woods, NH, pp. 90-99 (October 1983).
5. R. L. Budziniski, J. Linn, and S. M. Thatte, ''A Restructurable Integrated Circuit for Implementing Programmable Digital Systems,'' *Computer* **15**(3) pp. 43-54 (March 1982).
6. W. C. Carter and P. R. Schneider, ''Design of Dynamically Checked Computers,'' *IFIPS Proceedings*, Edinburgh, Scotland, pp. 878-883 (August 1968).
7. X. Castillo, S. R. McConnel, and D. P. Siewiorek, ''Derivation and Calibration of a Transient Error Reliability Model,'' *IEEE Transactions on Computers* **C-31**(7) pp. 658-671 (July 1982).
8. S. A. Elkind, ''Reliability and Availability Techniques,'' pp. 63-181 in *The Theory and Practice of Reliable System Design*, ed. D. P. Siewiorek and R. S. Swarz, Digital Press (1982).
9. R. M. Fujimoto and C. H. Séquin, ''The Impact of VLSI on Communications in Closely Coupled Multiprocessor Networks,'' *Proceedings of COMPSAC 82*, Chicago, IL, pp. 231-238 (November 1982).
10. J. N. Gray, ''Notes on Data Base Operating Systems,'' pp. 393-481 in *Operating Systems: An Advanced Course*, ed. G. Goos and J. Hartmanis, Springer-Verlag, Berlin (1978). Lecture Notes in Computer Science 60.
11. S. H. Hosseini, J. G. Kuhl, and S. M. Reddy, ''An Integrated Approach to Error Recovery in Distributed Computing Systems,'' *13th Fault-Tolerant Computing Symposium*, Milano, Italy, pp. 56-63 (June 1983).
12. INMOS, *IMS T424 Transputer*, Advance Information. November 1983.
13. P. S. Kastner, ''A Fault-Tolerant Transaction Processing Environment,'' *Database Engineering* **6**(2) pp. 20-28 (June 1983).
14. J. A. Katzman, ''The Tandem 16: A Fault-Tolerant Computing System,'' pp. 470-480 in *Computer Structures: Principles and Examples*, ed. D. P. Siewiorek, C. G. Bell, and A. Newell, McGraw-Hill (1982).
15. G. Peattie, ''Quality Control for ICs,'' *IEEE Spectrum* **18**(10) pp. 93-97 (October 1981).
16. M. L. Powell and D. L. Presotto, ''Publishing: A Reliable Broadcast Communication Mechanism,'' *Proc. 9th Symp. on Operating Systems Principles*, Bretton Woods, NH, pp. 100-109 (October 1983).
17. D. K. Pradhan and S. M. Reddy, ''A Fault-Tolerant Communication Architecture for Distributed Systems,'' *IEEE Transactions on Computers* **C-31**(9) pp. 863-870 (September 1982).
18. B. Randell, P. A. Lee, and P. C. Treleaven, ''Reliability Issues in Computing System Design,'' *Computing Surveys* **10**(2) pp. 123-165 (June 1978).
19. C. H. Séquin and R. M. Fujimoto, ''X-Tree and Y-Components,'' pp. 299-326 in *VLSI Architecture*, ed. B. Randell and P.C. Treleaven, Prentice Hall, Englewood Cliffs, NJ (1983).
20. W. D. Tajibnapis, ''A Correctness Proof of a Topology Information Maintenance Protocol for a Distributed Computer Network,'' *Communications of the ACM* **20**(7) pp. 477-485 (July 1977).
21. Yuval Tamir and Carlo H. Séquin, ''Self-Checking VLSI Building Blocks for Fault-Tolerant Multicomputers,'' *International Conference on Computer Design*, Port Chester, NY, pp. 561-564 (November 1983).
22. Yuval Tamir and Carlo H. Séquin, ''Design and Application of Self-Testing Comparators Implemented with MOS PLAs,'' *IEEE Transactions on Computers* **C-33**(6) pp. 493-506 (June 1984).
23. Yuval Tamir, ''Fault Tolerance for VLSI Multicomputers,'' Ph.D. Dissertation, CS Division Report No. UCB/CSD 86/256, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA (August 1985).
24. L. D. Wittie, ''Communication Structures for Large Networks of Microcomputers,'' *IEEE Transactions on Computers* **C-30**(4) pp. 264-273 (April 1981).