Proceedings of the
International Conference on Parallel Processing,
St. Charles, Illinois, pp. 117-120, August 1987.

A SOFTWARE-BASED HARDWARE FAULT TOLERANCE SCHEME FOR MULTICOMPUTERS

Yuval Tamir and Eli Gafni
Computer Science Department
University of California
Los Angeles, California 90024

Abstract — A hardware fault tolerance scheme for large multicomputers executing time-consuming non-interactive applications is described. Error detection and recovery are done mostly by software with little hardware support. The scheme is based on simultaneous execution of identical copies of the application on two subnetworks of the system. Normal system operation is periodically suspended and the logical states of the two subnetworks are synchronized. Errors are detected by comparing the ‘‘frozen’’ synchronized states of the two subnetworks while they are being saved as ‘‘checkpoints’’ for possible subsequent use for error recovery. Algorithms for error detection and recovery using this scheme are discussed.

I. Introduction

Due to advances in VLSI technology it is now feasible to implement computer systems consisting of thousands of processors. Such systems can achieve high performance by exploiting parallelism. They also have the potential of achieving higher reliability than large monolithic systems since the individual processors are sufficiently powerful and independent so that they may cross-check each other and, if some system components fail, the others can modify their operation to adapt to this change and maintain correct system functionality [8].

One possible organization of systems consisting of a large number of processors is a network of computation nodes interconnected by high-speed dedicated communication links [5, 6]. Each node is a complete ‘‘computer’’ consisting of a processor, memory, and several communication ports. This type of system (henceforth called a *multicomputer*) has the advantage that there is no single component, such as a common bus or shared memory, whose correct operation is critical for the entire system.

Multicomputers can be used for large noninteractive applications, such as circuit simulation, weather forecasting, etc. A system composed of tens of thousands of VLSI chips has a mean time between component failure of, at most, a few hours [9]. Thus, one or more component failures are likely to occur during a computation that takes several hours to complete. Unless that system is able to detect errors and prevent failed components from participating in the computation, the results are likely to be incorrect. Hence, multicomputers used for noninteractive applications must be fault tolerant.

In systems used for non-interactive applications the major requirements are a high probability that the results produced are correct and high throughput. Unlike real-time systems, there are no strict constraints on the delays introduced when error recovery is necessary. This ‘‘flexibility’’ can be exploited by fault tolerance schemes which involve lower average overhead but are subject to more severe temporary service interruptions when errors occur.

Fault tolerance requires the ability to detect errors and identify the faulty components. Ideally errors should be detected as soon as they occur and before erroneous information spreads throughout the system [8, 9]. This can be achieved if error detection is performed by hardware. Specifically, each node should be *self-checking*, i.e., its implementation should guarantee that it will produce an error indication to the rest of the system if the results it produces are incorrect due to hardware faults [8, 10].

This work is supported by TRW Corporation and the State of California MICRO program. The second author is supported by NSF grant DCR-84-51396 and IBM grant D840622.

Based on the use of self-checking nodes, Tamir and Séquin [9] developed a low overhead fault tolerance scheme that takes advantage of the ‘‘flexibility’’ of non-interactive applications. The basic idea is to periodically checkpoint the state of the entire system and roll back to a previous checkpoint if an error is detected. The frequency of checkpointing is low (e.g. twice per hour) so the ‘‘cost’’ of recovering from errors is relatively high. It is estimated that the total overheard for fault tolerance and for dealing with faults in a multicomputer with one thousand nodes will be only a few percent [9].

The main disadvantage of the scheme proposed in [9] is that it relies on self-checking nodes. In this paper we examine the possibility of implementing fault tolerance in a multicomputer without the use of self-checking hardware in all the nodes. We present a software-based error-detection scheme which is coupled with an error recovery mechanism similar to the one proposed in [9]. The proposed scheme is based on partitioning the system into two identical subsystems. Identical copies of each task are executed on the two subsystems. Errors are detected by comparing the states of the entire subsystems during checkpointing.

The rest of the paper includes a discussion of the basic ideas in the proposed scheme, including identification of some key problems and their solutions. Some of the major advantages and disadvantages of the scheme are presented. An outline of the protocols used for the error detection and recovery phases of the scheme are shown. Due to lack of space, many issues cannot be fully discussed and explained. Since some of the ideas are closely related to those presented in [9], familiarity with that paper is assumed.

II. Software-Based Error Detection in Multicomputers

As mentioned earlier, one of the primary goals of this work was to find a way to perform error detection with very high coverage without relying on all the hardware to be self-checking. Given the complexity of the hardware and the difficulties in predicting all possible system behavior under faults, duplication and comparison is needed for error detection. In the hardware-based approach [8, 9] the duplication and comparison is accomplished by constructing each node using two modules that perform the same operations synchronously and a comparator for comparing their results.

With software, duplication and comparison implies that each process is executed on two identical nodes and the results are somehow compared. One possible approach is to allocate each process to two nodes and, when a message is received by a node in the system, the receiving node waits for the corresponding message from the sender’s ‘‘twin’’ and compares the two messages to verify their correctness. This scheme involves significant overhead during *normal* system operation and major complication in task allocation and message routing [11].

An alternative software-based error detection scheme involves (logically) partitioning the multicomputer into two identical multicomputers and executing identical copies of all system tasks on the two subsystems. Instead of each node performing a comparison each time a message and its duplicate are received, the states of the entire subnetworks are periodically compared. Fault tolerance requires the ability to recover from errors as well as detect them. Since the error recovery scheme proposed in [9] involves periodic saving of the entire system state, it is beneficial to combine the periodic state comparison with the state saving.

In order to detect errors by comparing system states, we must ensure that when the system is operating correctly the states of the two subsystems will, in fact, be identical. We are interested in multicomputers where the nodes are asynchronous. Thus, even though we have two identical systems executing identical tasks, there is no instant in time in which we can stop both systems and be guaranteed that their states will be identical. In order to synchronize system states, we need some measure of the *logical* progress made by the node in executing its processes. One such measure is a counter of machine instructions that are executed for each process on the node. Thus, our scheme requires that part of the process state will be a register with such a count. We call this counter the *process clock*. For each node the vector consisting of the process clocks for all processes on the node defines the logical ‘‘progress’’ made by the node. The hardware and the operating system must allow us to specify that a particular process should execute until its clock reaches a specified value.

With the proposed scheme checkpointing is periodically initiated in the two subsystems. As part of the checkpointing process, the states of the two subsystems are ‘‘frozen.’’ The entire states of the two subnetworks are then ‘‘synchronized’’ so that they can be compared. This is done by each node exchanging its vector of process clocks with that of its ‘‘twin’’ in the other subsystem. Each node then executes those processes that are behind until the process clock vectors match.

Once the subsystem states are synchronized (see below), the states of the two subsystems must be compared. The system state includes the entire contents of the memories of all the nodes. Thus, a simple comparison of subsystem states would involve too much overhead (the time to transmit the states and then to perform the actual comparison). However, as part of the checkpointing process, the entire system state is saved on disk [9]. Thus the entire state must pass through a small number of *disk nodes* [9]. Instead of comparing the subsystem states, *signatures* of these states will be compared. The signature of the state of each node can be calculated by adding a linear feedback shift register (LFSR) to each disk node [4]. The LFSR accumulates the signature of the state while it is being transmitted to the disk controller with negligible overhead. After the node states are saved on disk, a cumulative signature for all the node states saved by that disk node can be calculated. Comparing the subsystem states is now reduced to comparing the signatures calculated in the disk nodes of the two subsystem. Since signatures of, say, 128 bits, can provide extremely low probability that different states will lead to identical signatures [1], the cost of comparison is now small.

Matching the process clock vectors is not sufficient to ensure that the subsystem states are identical. Two other factors must be dealt with: (1) The order in which messages are received and processed by a node can affect its state. (2) Messages in transit between nodes when the subsystem states are frozen (and synchronized) can be ‘‘stuck’’ in different nodes in the two subsystems, leading to different subsystem states. The first problem can be solved by forbidding applications from using non-blocking receive. The non-blocking receive is inherently non-deterministic and is thus incompatible with our scheme. While we have not analyzed the cost of preventing non-blocking receive in realistic benchmarks, analysis of some simple situation indicate that in an extreme worst case, the cost may be as high as a factor of two slow down. We expect the cost in real programs to be *much* lower.

The solution to the second problem discussed above, is to ‘‘flush’’ messages in transit to their final destination before saving and comparing the subsystem states. This is a classic distributed termination problem and the solution we use in the next section is derived from [7].

In the error detection scheme discussed above we do not deal with the issue of locating the faulty component(s). In fact, for this entire discussion we will assume that all faults are transient. This is a reasonable simplifying assumption since transient faults are at least an order of magnitude more likely to occur than permanent faults [2]. Extension to this scheme that include fault location appear to be quite possible and will be the subject of future work.

Even if we restrict ourselves to dealing with transient faults, we must still be able to handle the problem that can occur when a transient fault causes a node to change its internal state so (e.g. corrupt part of the operating system) that it will no longer participate correctly in system operation. When self-checking hardware is used, implementing nodes which are *self-resetting* and can reestablish a ‘‘sane state’’ following a transient fault, is relatively easy [8]. While we are not able to discuss details of the solution here, a watchdog timer [3] can be used to periodically ensure that the node is still in a ‘‘sane state.’’

Another potential problem with this scheme is that nodes can send ‘‘misleading’’ information that will impede checkpointing or recovery. For example, during the process of synchronizing the two subsystems, incorrect process clock values may be exchanged. This can result in deadlock. Our solution to this problem is for the checkpointing coordinator to impose a loose time limit on the checkpointing session. If checkpointing is not complete by a certain time limit, a fault is declared and recovery is initiated.

Finally, as in [9], the problem of failure of disk nodes is difficult to solve. Specifically, since a disk node ‘‘handles’’ the state of other nodes, it has an opportunity to corrupt them. For this paper we will assume that disk nodes do not fail. In a realy system, we would have to implement the disk nodes as self-checking nodes and use duplicate (mirrored) disks, as discussed in [9].

III. Details of the Proposed Fault Tolerance Scheme

In this section we present some details of the proposed fault tolerance scheme. Due to lack of space, not all issue are dealt with and the algorithm specifications are incomplete. The purpose of this section is to present some *examples* of the type of protocols that need to be used rather than attempt to present complete solutions. Thus, only parts of the checkpointing process is described and the recovery process, which is very similar to the one describe in [9], is not presented.

As in [9] we assume that the system uses two basic types of packets: normal packets and fail-safe packets. Normal packets are used for normal computation and carry no redundant information for error detection. During normal processing there is no acknowledgement of normal packets. Fail-safe packets include error detection bit using a code such CRC. The coordination of checkpointing and recovery is all done using fail-safe packets. Fail-safe packets are always acknowledged.

The ‘‘checkpointing-coordinator’’ is selected as in [9] except that there is one checkpointing coordinator for each subsystem and one of these is the ‘‘master checkpointing coordinator’’ that initiates the checkpointing process for the entire system. We also distinguish between a *regular node* and a *disk node*. A disk node is connected to a disk drive and is used to save the states of several regular nodes on disk.

A. Types of Fail-Safe Packets

Any two nodes i and j are *neighbors* if, and only if, there is a link between them. Two nodes i and j are *twins* if, and only if, they are in different subsystems and are both assigned identical tasks. In the entire system each node has exactly one twin.

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 fifteen types of fail-safe packets:

freeze: Suspend normal processes.
synch(src,dest): Transmit the process clock vector from one node to its twin.
synch-done(node,coord): Inform the subsystem's checkpointing coordinator that process clock synchronization is complete.
flush: Signal the neighbors to begin ''flushing'' any packets in transit by forwarding them to the appropriate destination.
flush-ack(child-flag): Node i sends this packet to neighbor j to acknowledge the *flush* packet previously sent from j to i and to inform j whether i is now a child of j in the flushing spanning tree.
normal-ack: Acknowledge normal packets while in the *flushing* l-state.
flushed: Inform the parent in the flushing spanning tree that the sender and the subtree below it are flushed.
checkp(CKV): Initiate the saving 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.
check-sig(src,dest,sig): Used by a disk node *src* to inform its twin *dest* of the signature *sig* for all the node states that it has saved.
match-ok(src,coord): Disk node *src* informs the subsystem's checkpointing coordinator *coord* that the state that it saved matched the state saved by its twin.
resume: Signals the end of a checkpointing ''session'' or the end of a recovery session.
fault: Broadcasts the fact that a fault has occurred and initiates recovery. In most cases the originator of this packet is a disk node that has discovered a mismatch in signature with its twin.
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.

B. The Logical States of a Node

At any point in time, a node in the system may be engaged in normal operation, freezing processing of application processes, synchronizing it process clocks with those of its twin, flushing messages in transit, checkpointing, 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*) that are simply labels for the current activity of the node:

normal: Normal operation. Normal packets are accepted and processed. A *freeze* packet causes an l-state transition to *frozen*. A *resume* packet is ignored. Other fail-safe packets cause transition to the *error* l-state.
frozen: The l-state of the node after it has received the first *freeze* packet. Processing of application processes is suspended. Normal packets continue to be received and forwarded.
synching: The l-state of the node while it is catching up its process clocks with those of its twin. There is selective processing of application processes. Normal packets are received and forwarded. The l-state changes to *synched* when the catching up is complete.
synched: The node is waiting for all the other nodes in the subsystem to catch up with their twins. Normal packets are received and forwarded. Fail-safe packets of type *synch* and $synch-done$ are forwarded.
flushing: The node is receiving and forwarding normal packets so that all messages in transit will be delivered to their final destination. One of the neighbors is known to be the *parent* in the flushing spanning tree. Normal packets from all neighbors are acknowledged. Neighbors must acknowledge all normal packets. Processing of application processes is suspended. When *flushed* packets are received from all the neighbors except the parent, the l-state is changed to *flushed*.
flushed: Node i is in this l-state when i, and all the nodes in the subtree rooted in i are free of messages in transit. If a normal packet is received, the l-state changes back to *flushing* and the sender of the packet becomes the parent. A *checkp* packet causes transition to the *checkpointing* l-state. Other fail-safe packets cause transition to the *error* l-state.
checkpointing: The node is sending its state to a disk node. Normal packets and any fail-safe packet other then *state*, *check-sig*, and *match-ok*, cause transition to the *error* l-state. Once the node sends its entire state to a disk node, it changes its l-state to *checkpointed*.
checkpointed: The node has completed sending its state to a disk node but has not received the *resume* packet. A *resume* packet causes an l-state transition to *normal*.
error: The node has detected (or has been informed of) an error but it is not ready to accept its recovered state. A disk node enters this l-state if it is in the *checkpointed* l-state and it discovers that the signature for the node states that it just saved on disk does not match the signature computed by its twin. Other nodes can enter this l-state upon receiving a *fault* packet, an unexpected normal packet, an invalid or unexpected fail-safe packet, or if a neighbor fails to acknowledge a packet when it should. The *recover* packet causes transition to the *recovering* l-state.
recovering: The node has received *recover* packet but it is not ready to resume normal operation with its recovered state. The arrival of the node's complete state via *state* packets causes a transition to the *recovered* l-state.
recovered: The node has received its complete recovered state but has not resumed normal operation. The *resume* packet causes a transition to the *normal* l-state.

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[9].

C. Saving the Global Checkpoints

As in [9], every node has a ''timer'' that can interrupt the node periodically. Checkpointing is initiated by the coordinator when it is interrupted by its timer (while in the *normal* l-state). Checkpointing is also initiated when a task is complete since before the system can commit to the result of a task, it should attempt to detect errors in the results and error detection is performed as part of the checkpointing process. It is, of course, possible for a fault in the designated coordinator to prevent it from initiating checkpointing. The solution to this problem is based on the fact that all the other nodes also have timers[9].

Node Actions During a Checkpointing Session

The master checkpointing coordinator, node m, initiates checkpointing. The checkpointing coordinator for a particular subsystem (one of the two) is node c.

The checkpointing session begins when node m is interrupted by its timer. Node m stops all work on application processes. It stops transmitting normal packets whose origin is node m but continues receiving normal packets and forwarding normal packets originating elsewhere. The node's l-state is changed to *frozen*. Node m then sends to every neighbor node the *freeze* packet.

The actions of each node j that receives the *freeze* fail-safe packet are described below.

[1] Node j stops all work on application processes. It stops transmitting ''original'' normal packets but continues receiving and forwarding normal packets. The node's l-state is changed to *frozen*.
[2] Node j sends *freeze* packets to all its neighbors. Node j sends a *synch* packet to its twin and then waits for a *synch* packets from its twin. The node's l-state is changed to *synching*.
[3] Node j begins ''catching up'' with its twin. When all the processes are caught up, the node's l-state is changed to *synched*.
[4] Node j sends a *synch-done* packet to the checkpointing coordinator and waits for a *flush* packet.

[5] When a *flush* packet arrives from neighbor node i, node j changes l-state to *flushing* and sends a *flush-ack(true)* packet to node i. From that point on, node j acknowledges every normal packet sent by node i and expects node i to acknowledge every normal packet sent from j to i. Node j sends *flush* packets to all its other neighbors and, after getting *flush-ack* back begins to acknowledge every normal packet sent by a neighbor and expects those neighbors to acknowledge every normal packet sent from j to them. If all the *flush-ack* packets the node receive are *flush-ack(false)*, the node considers itself a *leaf node*.

[6] If node j is a leaf node and contains no messages in transit and has no outstanding unacknowledged messages, it changes its l-state to *flushed* and sends a *flushed* packet to its parent. When a node receives a *flushed* packet from all its children, it becomes a leaf node. When node j is in l-state *flushed* and receives a normal packet from its neighbor node i, it changes its l-state back to *flushing* and becomes the child of node i.

[7] When the checkpointing coordinator changes l-state to *flushed*, it immediately changes l-state to *checkpointing* and send $checkp(CKV)$ packets to all its neighbors. As in [9], when the *checkp* packets are distributed every pair of nodes exchanges CRC check vectors in order to determine if there were any transmission errors during normal computation. If a mismatch is found, recovery is initiated.

[8] After verifying the CRC check vectors, the node begins sending its state to its assigned disk node.

[9] The disk node accumulates the signatures for the states of all the nodes it saves. After receiving the states of all the nodes assigned to it, the disk node calculates an overall signature and exchanges signatures with its twin. If the signature matches, the checkpointing coordinator is informed using the *match-ok* packet. If they do not match, recovery is initiated.

IV. Summary and Conclusions

We have presented a new software-based fault tolerance scheme for multicomputers. The scheme provides a very high probability of detecting errors caused by hardware faults without requiring all the hardware to be self-checking. It should be noticed that there are situations where recovery is possible but this scheme will not be able to recover. However, this is a ‘‘fail-safe’’ mode. Our main goal is to prevent a situation where incorrect results will be accepted as correct. Such a situation is very unlikely to occur with this scheme. While we have not presented estimates of overhead and performance, such rough estimates, based on the current state of technology, indicate the the total overhead for fault tolerance with this scheme can be expected to be on the order of a few percent.

One of the main advantages of this scheme is that it presents an opportunity for a system to operate in both *safe* and *unsafe* modes. In the unsafe mode, all the nodes are used for achieving higher performance but there is not error detection. In the safe mode, the scheme described in this paper is used so that half the hardware resources are ‘‘wasted’’ for error detection. The sensitivity of the results to transient faults is different from one application to another (some applications will converge to the correct results despite an incorrect intermediate result at some point in the computation). Furthermore, depending on how the results of a computation is used, in some cases there is no significant damage if the results are incorrect. Given the fact that different applications have different fault tolerance requirements, the flexibility of scheme presented here is a very attractive and makes this approach worthy of further study and eventual implementation.

Acknowledgements

This work was motivated by discussions with A. Avizienis, M. Ercegovac, T. Lang, D. Rennels, and members of the Suprenum project at GMD-FIRST Berlin.

References

1. D. K. Bhavsar and R. W. Heckelman, ‘‘Self-Testing by Polynomial Division,’’ *1981 International Test Conference Proceedings*, Philadelphia, PA, pp. 208-216 (October 1981).
2. 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).
3. J. R. Connet, E. J. Pasternak, and B. D. Wagner, ‘‘Software Defenses in Real-Time Control Systems,’’ *2nd Fault-Tolerant Computing Symposium*, Newton, MA, pp. 94-99 (June 1972).
4. 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).
5. C. L. Seitz, ‘‘The Cosmic Cube,’’ *Communications of the ACM* **28**(1), pp. 22-33 (January 1985).
6. 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).
7. N. Shavit and N. Francez, ‘‘A New Approach to Selection of Locally Indicative Stability,’’ *13th ICALP (Lecture Notes in Computer Science 226)*, pp. 344-358, Springer-Verlag (1986).
8. 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).
9. Yuval Tamir and Carlo H. Séquin, ‘‘Error Recovery in Multicomputers Using Global Checkpoints,’’ *13th International Conference on Parallel Processing*, Bellaire, MI, pp. 32-41 (August 1984).
10. Yuval Tamir and Carlo H. Séquin, ‘‘Reducing Common Mode Failures in Duplicate Modules,’’ *International Conference on Computer Design*, Port Chester, NY, pp. 302-307 (October 1984).
11. 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).