Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications
Monterey, California, pp. 103-108, March 1989.

APPLICATION-TRANSPARENT ERROR-RECOVERY TECHNIQUES FOR MULTICOMPUTERS †

Tiffany M. Frazier and Yuval Tamir

Computer Science Department
University of California
Los Angeles, California 90024
U.S.A.

Abstract

We describe and compare error recovery techniques for restoring a valid system state in large ''general-purpose'' multicomputers following component failures. The techniques discussed are application-transparent, i.e., they do not require application processes to be aware of the fault tolerance characteristics of the system. These techniques are based on checkpointing and rollback: process states are periodically saved to stable storage and are restored from stable storage in the event of a hardware failure. We divide existing recovery schemes into two classes: Message Logging and Coordinated Checkpointing, describe several techniques in each class, and present the advantages and limitations of the schemes when used to provide fault tolerance for large multicomputers.

I. Introduction

Multicomputer systems, with hundreds of nodes interconnected via point-to-point links, may consist of thousands of large VLSI chips and hundreds of printed circuit boards. If the entire system is likely to fail as a result of the failure of any of its components, the expected mean time between system failures is only a few hours [11, 18]. Since there are many important applications that require continuous correct operation for tens or hundreds of hours, such a high system failure rate is unacceptable. Hence, the system must be able to operate correctly despite the failure of some of its components, i.e., it must be *fault tolerant*. In order to ''tolerate'' hardware faults, the system must be able to *detect* errors, *locate* the faulty component, *restore* a valid system state, and (possibly) *reconfigure* the system so that it does not continue to use the faulty components [13].

In this paper we describe and compare *error recovery* techniques for restoring a valid system state in a multicomputer following an error. We focus on schemes for ''general-purpose'' applications without hard real-time constraints. In this environment, the performance overhead and redundant hardware dedicated to fault tolerance must be minimized, even at the expense of increased recovery time when an error does occur. Given the difficulty in writing large distributed applications, the *virtual machine* visible to the programmer must not be made more complex by the fault tolerance scheme. Hence, we focus on *application-transparent* techniques, where no restrictions are made on the behavior and structure of the application software.

The recovery techniques discussed are based on *checkpointing and rollback*. Process states are periodically saved (checkpointed) to stable storage. When an error is detected, a previously saved valid state is restored. The overhead of these schemes includes the time and storage overhead incurred during normal operation, and time lost during recovery. Practical existing application-transparent recovery schemes based on checkpointing and rollback can be divided into two classes: Message Logging and Coordinated Checkpointing. We describe the key features of the two classes, highlight the differences between the schemes within each class, and present the advantages and limitations of each scheme when used to provide fault tolerance for large multicomputers.

II. System Model and Assumptions

Our target system is a multicomputer which consists of hundreds or thousands of VLSI nodes which communicate via messages over point-to-point links. Each node includes a processor, local memory, and a communication coprocessor. The nodes operate asynchronously and messages may have to pass through several intermediate nodes to reach their destinations. The system is used for ''general purpose'' applications which have no hard real-time constraints. Errors can occur at any time as a result of *hardware* faults in the nodes or in the communication links.

A fault-tolerant multicomputer must be able to detect and recover from errors in interprocessor communication. Many error recovery schemes [15, 9] are based on the use of reliable FIFO communication channels between nodes. These channels are implemented by underlying communication protocols which typically involve sending redundant bits for error detection, appending sequence numbers to each message, and use of end-to-end acknowledgements [20]. Alternatively [18, 19], detection and recovery from errors in communication may be an integral part of the general error recovery mechanism. In either case the overhead for dealing with the possibility of lost or corrupted messages must be considered as part of the system overhead for fault tolerance.

Most schemes assume that nodes are *fail-stop* [14] so that they either generate the correct results or no results at all. Such nodes can be implemented using self-checking hardware which is guaranteed to signal an error to the neighbors immediately when incorrect output is generated [17]. Any node that generates an error signal is assumed to be permanently faulty and no attempt is made to continue to use it.

The error recovery schemes discussed in this paper are based on the existence of *stable storage* where checkpoints can be safely maintained [10]. Such stable storage may be

† This research is supported by TRW Corporation and the State of California MICRO program.

approximated by ''mirrored'' disks with multiple access ports [8] in conjunction with reconfiguration which takes place in response to node or disk failure [18]. In this paper we assume that some of the nodes in the system, called *disk nodes*, are connected to these ''reliable'' disks and the failure of a disk or a disk node causes a *crash* (i.e., an unrecoverable error) [18]. We further assume that the system connectivity is sufficiently high that network partitioning will never occur, i.e., there will always be a path between every node and every disk node.

The state of a process consists of the contents of all of the memory and registers used by the process. This can include some system tables, such as the list of all processes with whom communication is currently ''open.'' The ''unit'' of checkpointing and recovery may be a process [9, 19], a set of processes [15] (*recovery unit*), or a complete node [18].

Each node can be time-shared between multiple processes. If checkpointing and recovery are done at the level of processes, a node may thus have to participate in multiple simultaneous checkpointing and recovery sessions [19]. Whenever checkpointing or recovery of a particular process is initiated, the kernel spawns a *handler* process that performs the necessary operations. The handler can suspend the process, manipulate its state, or allow it to resume normal operation. In the rest of the paper we will often discuss the actions of participants in checkpointing and recovery sessions. However, the techniques described are all application-transparent so the ''participants'' are really the handlers corresponding to the processes being checkpointed or recovered.

III. Checkpointing and Recovery

A. Checkpointing/Recovery of Independent Processes

In a system which supports only non-interacting processes, checkpointing involves periodically suspending each process, copying the process' state to stable storage, and then allowing the process to resume normal computation. When a node fails, all information on the node is assumed to be lost. The checkpointed states of all the processes that were on the failed node are restored from stable storage to working nodes. The processes may then resume computation from these older states [13]. In order to be able to tolerate node failures that occur during checkpointing, the old checkpoint of a process is erased only after the new checkpoint has been completely written to stable storage.

The overhead of a fault tolerance scheme is determined by the frequency of checkpointing, the ''cost'' of saving each checkpoint, extra operations required during normal computation for error detection and maintenance of bookkeeping information, and the ''cost'' of recovery. The rest of this subsection is a detailed description of the different overhead components.

a) *computation time lost during checkpointing*: Without the use of redundant hardware, a fraction of the processing cycles on each node must be devoted to checkpointing. Specifically, each node that has processes that are participating in a checkpointing session, must suspend normal operation in order to run the handlers that perform the various functions necessary for checkpointing. If the communication coprocessor can operate independently of the application processor, normal execution of other processes on the node may resume before the checkpointing session is complete. The process being checkpointed may need to be suspended until the checkpointing session is complete and its state is stored in stable storage [19].

b) *load on the communication network caused by sending process checkpoints to disk*: In the ''worst case'' the entire process state has to be sent to disk. If virtual memory is employed, part of the state is already on disk and the size of the state in local memory, which has to be sent to disk, is related to the process' *working set*. With appropriate hardware support for identifying modified pages, only the process state which has been changed since the last checkpoint is sent to disk.

c) *disk bandwidth required for saving the checkpoints*: A significant fraction of the total disk bandwidth available on the system may be needed for saving checkpoints, thus limiting the performance of I/O-intensive applications.

d) *disk space for storing checkpoints*: The minimum storage requirement is one complete checkpoint state per process plus additional space for storing a second copy of the checkpoint of processes during checkpointing [9]. Many error recovery techniques require storage of multiple checkpoints per process as well as bookkeeping information and message logs [15].

e) *local memory usage*: Part of local memory is needed for storing program code for handlers, bookkeeping information (e.g. a list of processes that have communicated with local processes [19]), and, possibly, storage for volatile checkpoints [1] or message logs [16]. Such use of local memory can increase paging activity, load on communication links and the average memory access time.

f) *special processing during recovery*: Each node participating in a recovery session must suspend normal operation so that it can run the recovery handlers.

g) *computation time lost due to rollback*: When a process is rolled back, the computations it performed since its last checkpoint are lost and have to be repeated. This overhead is inversely proportional to the frequency of checkpointing.

B. Checkpointing/Recovery of Interacting Processes

Checkpointing and recovery are more complex when processes are allowed to interact via messages. When one or more processes are rolled back care must be taken to ensure that each pair of processes are *consistent* with each other and hence that the entire system state is *valid* [13]. Specifically, every pair of processes must agree which messages have been sent and which have not [3] — such that no messages are lost or duplicated (see Figure 1).

A set of checkpoints, one per process being rolled back, which are consistent with each other is called a *recovery line* [13]. Graphically, a recovery line can be represented as a line which connects a set of checkpoints and intersects no communication lines (see Figure 1b). If processes are checkpointed independently, ignoring interactions with other processes, the recovery algorithm

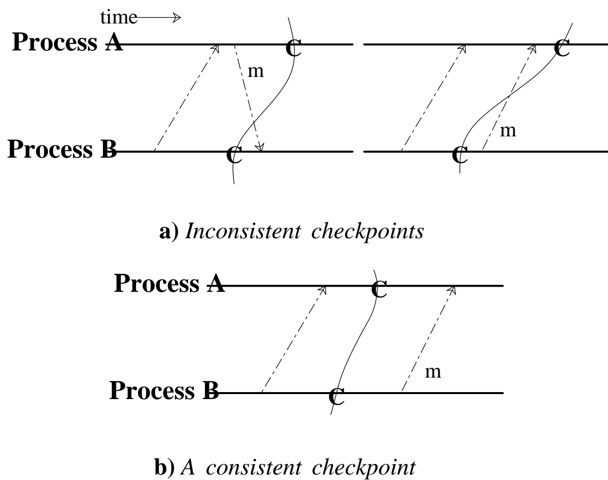

Figure 1: In a), if A and B roll back to their latest checkpoints their states will be inconsistent. In the first case B expects a message from A which A's rolled back state has already sent (lost message). In the second case B will resend a message which A has already received (duplicate message). In b), the latest checkpoints of A and B are consistent with each other.

must find a recovery line amongst the available checkpoints. The major problem with this method is the fact that there is no guarantee that a recovery line exists and *domino effect* [13] can occur, where, in the worst case, the entire system must be rolled back to its initial state (see Figure 2). Furthermore, many generations of checkpoints for each process have to be maintained on disk. This is necessary since, in the best case, it is desirable to roll back to a recent checkpoint (to minimize lost computations) while in the worst case it may be necessary to roll back to the initial process state.

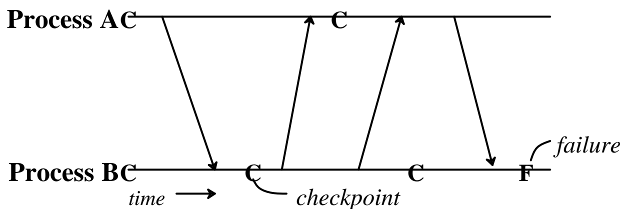

Figure 2: Process B fails and is rolled back to its latest checkpoint. Since B will later expect a message from A, A must also be rolled back, requiring B to roll back to a previous checkpoint. In this case, the entire system must roll back to its initial state.

One way to avoid the worst-case domino effect is for the application programmer to specify sequences of sets of recovery points that form recovery lines. If an error is detected, the system rolls back to the last programmer-specified recovery line. This approach is not an option if the fault tolerant characteristics of the systems are to be kept hidden from the programmer. In a large general-purpose system, keeping many generations of checkpoints for each process is not a viable option due to required disk space. Hence, recovery schemes that may require indefinite storage of all checkpoints (until the task terminates) are not a practical alternative for our target systems.

IV. Application-Transparent Recovery

Application-transparent recovery techniques have the advantage that they can be used without the programmer having any knowledge of the fault tolerance characteristics of the system. There are two fundamental ways to avoid domino effect in transparent distributed checkpointing and rollback schemes: Message Logging[12, 2, 15, 6, 16, 7] and Coordinated Checkpointing [1, 18, 9, 19]

Message logging techniques checkpoint both process states and (log) interprocess messages onto stable storage. When a process is rolled back, its *message log* is played back to it, such that when the message log has been depleted, the process is in a state consistent with the rest of the non-failed processes in the system. In order for message logging to work correctly, application processes must be *deterministic*: given a process state and a sequence of inputs (message log), the process will generate the same outputs. Processes can be checkpointed independently, thus minimizing disruptions to the entire system, and a single process can be recovered without interfering with the operation of other processes.

Coordinated checkpointing techniques checkpoint a set of processes together (the entire system or a subset thereof) in such a way that each pair of process states *on disk* are consistent with each other. [3] The recovery algorithm is guaranteed to be able to find a recovery line and hence be able to recover a valid system state. Recovery and checkpointing are likely to disrupt more of the system with coordinated checkpointing as opposed to messages logging, since more than one process is involved. However, there is no need to store message logs and very little bookkeeping information is required.

Several, previously published, recovery techniques are described in the rest of the paper. They are judged based on a set of either highly desirable or necessary attributes for a recovery scheme to be appropriate for use in large multicomputers (Section II):

application-transparent

minimal restrictions on application behavior

low-overhead: as enumerated in Section IIIa.

n-fault-tolerant: since nodes may be multitasking, it is desirable for the recovery scheme to be n-fault-tolerant, i.e., be able to handle the case where n processes in the system fail simultaneously. Such a recovery technique can consequently also handle multiple simultaneous node failures.

distributed: the recovery scheme must not have a single point of failure, i.e., it cannot depend on any single resource.

suitable for a variety of applications: computation-intensive and communication-intensive applications.

A. Message Logging

Several papers on message logging have been published[12, 2, 15, 6, 16]. The basic idea behind message logging is to record all messages that a process sends along with their send and receive order, so that they can be ''played back'' to a failed, rolled-back process in order to bring that process back to a state consistent with that of the

rest of the system. The send and receive order is needed to ensure that the repeated computation following a rollback will be identical to the computation performed prior to the error. As previously mentioned this requires processes to be deterministic in their actions.

Two methods, Publishing[12] and Auragen[2], assume the use of a common-bus multiprocessor instead of a multicomputer architecture. In Publishing a ''recorder'' is attached to the shared-bus which can snoop and write to disk all messages which travel over it. Auragen, while not strictly a single bus architecture still requires the use of an atomic broadcast. In a large multicomputer architecture it is not practical to implement message transmission as an atomic broadcast and hence ''snooping'' is not a viable method. Auragen's method of error recovery can be called *pessimistic* (incurs synchronization delays) because both sender and receiver process must be suspended at each communication in order to log information equivalent to send and receive order[15].

In [15] Strom and Yemini proposed an *optimistic* message logging technique for non-shared-bus architectures where logging proceeds asynchronous of process computation. A process' current state *depends* on its latest checkpoint and on the disk's log of the messages which the process has received since that checkpoint. Since messages are logged to disk asynchronous of process computation, some messages might not yet be logged to disk when a process fails. In order to solve this problem sending a message involves the following steps (simplified for clarity):

1) A message M from sender S to receiver R is sent along with S's current *dependency vector* and a SSN (send sequence number). The dependency vector (composed of process ID and message number pairs) indicates which messages S's current state *depends* on that S does not yet know to be logged. S saves a copy of M until the R notifies S that the message has been logged.

2) R updates its own dependency vector with the dependency vector sent along with the message since R's state now depends on S's ability to resend M in the event of R failing.

3) R sends the message to disk along with the message's dependency vector and an RSN (receive sequence number).

4) R ''eventually'' receives notice that the message has been logged and then ''eventually'' notifies S the message has been logged. R updates its dependency vector to reflect the fact that it no longer depends on S's ability to resend M.

When a process(es) fails some of the messages upon which it depends may not be fully logged. In Figure 3 process S has received two messages $m0$ and $m1$ both of which are *not fully* logged to disk when S fails. First S is rolled back to its checkpoint. When U and T learn (via a *recovery message* and local bookkeeping information) that S has rolled back and needs $m0$ and $m1$, the messages will be resent. However, no ordering information exists because the messages were not fully logged, hence S might process $m0$ and $m1$ in a different order and, as a result might not send $m2$ (or the message's contents may be different) to process R. This is an inconsistent system state. Hence process R, called an *orphan*, must roll back and *undo* the effects of $m2$.

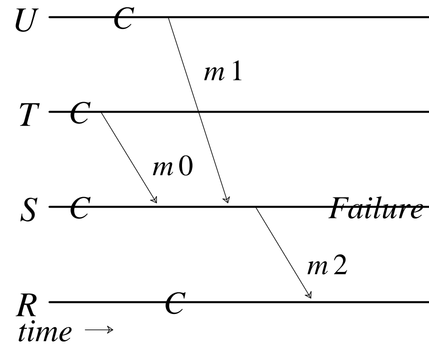

Figure 3: An example of an *orphan* in message logging techniques.

R learns that it must roll back when it receives the recovery message and, upon examining it's dependency vector, sees that its current state depends on S through message $m2$.

The overhead incurred in a system using message logging can be generally characterized as follows:

a) *computation time lost during checkpointing*: This overhead is ''minimized'' — checkpointing is asynchronous of process execution (uses volatile checkpoints) and involves only one process at a time.

b) *load on the communication network*: Includes checkpoints and the amount of data (messages) being sent between processes — this overhead is examined in the next subsection.

c) *disk bandwidth required*: Same arguments as (b).

d) *disk storage required*: Generally, the storage requirement would be one checkpoint state per process. Older checkpoints are discarded as soon as it can be determined that they will not be needed for either *state backout* (undoing the effects of messages) or for resending messages lost by a failed process.

e) *storage required in local memory*: Local storage requirements include volatile checkpoints, messages being held until they have been logged, dependency vectors, SSNs and RSNs, etc[15].

f) *Special processing during recovery*: the message log must be restored and played back to the recovering processes until they are brought up-to-date.

g) *lost computation time due to rollback*: All processes *directly* affected by a hardware failure must be rolled back, along with orphans, if any.

Optimistic recovery is a distributed, n-fault-tolerant, application-transparent scheme. The only restriction made on application processes is that they be deterministic.

The Cost of Logging Messages to Disk. Some application programs for multicomputers may require each process to send a short message (say, eight bytes) every 100 instructions [5, 4]. Such communication-intensive applications may be incompatible with message logging techniques since very large amounts of data may have to be logged. For example, consider a multicomputer with 512 10 MIPS processors running such an application. In this system a total of approximately **400 Mbytes** are sent as messages every second. Logging this much information is a major problem. Specifically, high-performance disk drives

have a maximum bandwidth of 3 Mbytes per second. Thus, if all messages are to be logged to disk, as required by Strom and Yemini's original scheme [15], more than *130* disk drives will have to be dedicated to logging messages.

As discussed above, message logging techniques also require bookkeeping information (dependency vectors, sequence numbers, etc) to be transmitted with each message. As a result, the load on the communication network and the complexity of send and receive operations, which have to manipulate this bookkeeping information, is increased. Some of this information must also be logged to disk, further increasing the number of disks required to support message logging techniques.

Other Message Logging Techniques. In sender-based message logging[6] messages are logged in local, volatile store and correct operation can be ensured only when a single process can fail at a time. In n-fault-tolerant logging[16] messages are logged in volatile store until it is determined that they can be discarded or spooled (logged to disk). Using the above example, volatilely logging messages in local memory involves storing an average of 686 Kbytes/sec (171 4K pages every sec) on every node. In both cases performance will be affected by having to copy messages to memory and because some fraction of main memory is needed to maintain message logs.

To reduce the local memory usage with sender-based message logging [6], processes may have to checkpoint frequently in order to limit the size of the message logs. In the case of n-fault-tolerant logging [16], the amount of information to be logged to disk is reduced by discarding some messages and predicting the arrival order of messages (using a deterministic heuristic). Performance will depend on how well the heuristic performs, on how many messages can be discarded, and on how much compute time and how many extra messages are required to run the heuristic.

B. Coordinated Checkpointing

The critical feature of coordinated checkpointing is that all interacting processes are checkpointed together, so that their checkpointed states can be guaranteed to be consistent with each other. Therefore, *domino effect* can not occur and it is sufficient to store only one ''generation'' of checkpoints on disk. Coordinated checkpointing techniques can be separated into Global Checkpointing and Process-Level Checkpointing.

Global Checkpointing. In[18] the entire system is checkpointed and recovered (global checkpoints). Checkpointing involves taking a *snapshot* of the entire system state, while recovery restores all nodes to their last checkpoint. The problem with the global checkpoints technique is that checkpointing is expensive since it requires saving the state of the entire system. Thus, for performance reasons, the time between checkpoints is relatively long (e.g. on the order of thirty minutes [18]). Hence, the system can only be used for batch applications, such as large numerical computations, where the possibility of losing thirty minutes of computation during recovery is an acceptable price for the resulting low overhead (a few percent [18]).

Process-Level Checkpointing. The global checkpointing technique has been extended to perform checkpointing and recovery of sets of interacting processes rather than of the entire system[19, 9]. An *interacting set of processes* is defined as the set of processes which have communicated directly or indirectly since their last checkpoints[1]. Checkpointing results in the saving of a consistent snapshot of the states of an interacting set of processes in such a way that a valid global checkpoint of the system state is maintained in stable storage. When a node fails, the interacting set(s) of all failed process(es) are rolled back to their checkpoints on disk.

In order to support process-level checkpointing, the system must maintain, for each process, *dynamic communication information*, which is the list of all processes with whom there has been direct communication since that process' last checkpoint session. With this information a *communication tree* can be dynamically formed during checkpoint/recovery sessions where the ''root'' of the tree is the initiator and coordinator of the session.

To ensure that the checkpoints of all processes in the interacting set are consistent, all messages in transit between those processes must be flushed to their destinations and saved with the checkpoints. The checkpoints of all other processes in the system (on disk or ''concurrently'' being taken) are ''automatically'' consistent with the checkpoints of the processes within the interacting set. This is because no messages have been sent to/from processes within the interacting set to/from processes outside of it (by definition). During a checkpoint session the interacting set of processes are suspended from normal operation until the communication tree has been formed, all messages have been flushed to their destinations, and all process states have been copied in local memory. Processes may then continue normal operation while their checkpoints are sent to the remote disks. Recovery proceeds in a similar fashion except that messages are discarded after they are flushed and process state(s) are restored from disk.

Unlike other error recovery schemes, this method and global checkpointing [18] do not assume that the message delivery system is made reliable via sliding windows (or two-phase commit protocols) and error detection bits for two reasons. First, the use of sliding windows has the stringent requirement that message transmission delays must be bounded while the use of two-phase commit protocols imposes a high level overhead on every message. Secondly, communication errors occur very infrequently relative to the frequency at which messages are sent. Hence, it is not desirable to penalize the system continuously by checking for errors on every message. Instead, detection of communication errors (and recovery from) is incorporated into the checkpointing/recovery algorithms[18, 19].

In[19] ''normal'' interprocess messages have no error detection bits, no bookkeeping information, and require no acknowledgements. Instead *signatures* (accumulated by Linear Feedback Shift Registers), which are updated in parallel with message transmission, are maintained for each pair of communicating processes. For example, if virtual circuits are used for interprocess communication then signatures are accumulated for each incoming and outgoing

virtual circuit. Communication between processes in the interacting set can be checked during a checkpoint session by performing ''end-to-end'' checks on all the virtual circuits between processes in that set. If a ''mismatch'' is detected then the interacting set containing the processes on both ends of the virtual circuit are rolled back. Messages which are used to coordinate the creation of checkpoints and for error recovery must be verified before they are used and hence carry error detection bits. This reduces the total overhead incurred by the error recovery scheme during normal operation.

The overhead of process-level recovery can be generally characterized as follows:

a) *computation time lost when checkpointing is performed*: The interacting set is suspended until the checkpointing is terminated. If the nodes are time-shared, they may continue to execute other processes.

b) *load on the communication network for checkpointing*: This overhead is minimized since i) no additional check bits are added to normal messages, and ii) there is no need to send message logs to stable storage.

c) *disk bandwidth required*: Only process checkpoints are sent to disk. Timers can be used to fine tune this overhead on a ''per application'' basis. This bandwidth is thus entirely dependent on application size and checkpoint frequency.

d) *storage required on disk*: Less than two checkpoints per process [9].

e) *local memory usage*: ''Minimal'': only dynamic communication information needs to be stored.

f) *special processing during recovery*: Loading the state of the interacting set of processes from stable storage.

g) *computations lost due to recovery*: All processes directly affected by an error *and* their interacting set must be rolled back.

All coordinated checkpointing schemes [19, 9, 18, 3, 1] are distributed, application-transparent, n-fault-tolerant (except for [1]), and make no restrictions on application behavior.

V. Summary and Conclusions

We have described several application-transparent error-recovery techniques which can be used to provide fault tolerance for large general-purpose multicomputers. Message Logging techniques have the advantage of fast checkpoint and recovery sessions but may be inappropriate for high-performance multicomputer systems running communication-intensive applications due to the overhead of logging messages and bookkeeping information to disk (or local memory). Furthermore, Message Logging can only support processes which are deterministic in their actions. Global Checkpointing techniques are likely to incur the lowest level of overhead during normal operation but are only appropriate for use with applications that can tolerate the possible loss of many minutes of computation during recovery (e.g. batch applications). Coordinated Process-Level Checkpointing techniques are a ''compromise'' between Message Logging and Global Checkpointing and therefore show the most promise in providing error recovery for high-performance multicomputers that run both communication- and computation-intensive applications. Process-Level Checkpointing techniques minimize the overhead incurred during normal operation at the cost of a potentially expensive recovery. In the worst case, the entire system may need to be rolled back as a result of the failure of one node. However, the overhead incurred during checkpointing is dependent almost entirely on the checkpoint frequency which can be tuned to the specific needs of each application process.

References

1. G. Barigazzi and L. Strigini, ''Application-Transparent Setting of Recovery Points,'' *13th Fault-Tolerant Computing Symposium*, Milano, Italy, pp. 48-55 (June 1983).
2. A. Borg, J. Baumbach, and S. Glazer, ''A Message System Supporting Fault Tolerance,'' *9th Symposium on Operating Systems Principles*, Bretton Woods, NH, pp. 90-99 (October 1983).
3. K. Mani Chandy and Leslie Lamport, ''Distributed Snapshots: Determining Global States of Distributed Systems,'' *ACM Transactions on Computer Systems* **3**(1), pp. 63-75 (February 1985).
4. W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills, ''Architecture of a Message-Driven Processor,'' *14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 189-196 (June 1987).
5. W. J. Jager and W. M. Loucks, ''The P-MACHine: A Hardware Message Accelerator for a Multiprocessor System,'' *1987 International Conference on Parallel Processing*, St. Charles, IL, pp. 600-609 (August 1987).
6. David B. Johnson and Willy Zwaenepoel, ''Sender-Based Message Logging,'' *17th Fault-Tolerant Computing Symposium*, Pittsburgh, PA, pp. 14-19 (July 1987).
7. David B. Johnson and Willy Zwaenepoel, ''Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing,'' *7th Annual ACM Symposium on Principles of Distributed Computing*, Toronto, Canada, pp. 171-181 (August 1988).
8. 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).
9. Richard Koo and Sam Toueg, ''Checkpointing and Rollback-Recovery for Distributed Systems,'' *IEEE Transactions on Software Engineering* **SE-13**(1), pp. 23-31 (January 1987).
10. B. W. Lampson and H. E. Sturgis, ''Crash Recovery in a Distributed Storage System,'' Technical Report, Xerox PARC, Palo Alto, CA (April 1979).
11. G. Peattie, ''Quality Control for ICs,'' *IEEE Spectrum* **18**(10), pp. 93-97 (October 1981).
12. 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).
13. B. Randell, P. A. Lee, and P. C. Treleaven, ''Reliability Issues in Computing System Design,'' *Computing Surveys* **10**(2), pp. 123-165 (June 1978).
14. F. B. Schneider, ''Byzantine Generals in Action: Implementing Fail-Stop Processors,'' *ACM Transactions on Computer Systems* **2**(2), pp. 145-154 (May 1984).
15. R. E. Strom and S. Yemini, ''Optimistic Recovery in Distributed Systems,'' *ACM Transactions on Computer Systems* **3**(3), pp. 204-226 (August 1985).
16. R. E. Strom, D. F. Bacon, and S. A. Yemini, ''Volatile Logging in n-Fault-Tolerant Distributed Systems,'' *18th Fault-Tolerant Computing Symposium*, Tokyo, Japan, pp. 44-49 (June 1988).
17. 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).
18. 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).
19. Yuval Tamir and Tiffany M. Frazier, ''Application-Transparent Process-Level Error Recovery for Multicomputers,'' *Hawaii International Conference on System Sciences-22*, Kailua-Kona, Hawaii, pp. 296-305, Vol I (January 1989).
20. A. S. Tanenbaum, *Computer Networks,* Prentice Hall (1981).