Sādhanā, Vol. 25, Part 5, October 2000, pp. 489–510. © Printed in India

A survey of checkpointing algorithms for parallel and distributed computers

S KALAISELVI and V RAJARAMAN[a]

Supercomputer Education and Research Centre (SERC), Indian Institute of Science, Bangalore 560 012, India
[a]Also at Jawaharlal Nehru Centre for Advanced Scientific Research, Indian Institute of Science Campus, Bangalore 560 012, India
e-mail: rajaram@serc.iisc.ernet.in

MS received 27 August 1998; revised 8 June 2000

Abstract. *Checkpoint* is defined as a designated place in a program at which normal processing is interrupted specifically to preserve the status information necessary to allow resumption of processing at a later time. *Checkpointing* is the process of saving the status information. This paper surveys the algorithms which have been reported in the literature for checkpointing parallel/distributed systems. It has been observed that most of the algorithms published for checkpointing in message passing systems are based on the seminal article by Chandy and Lamport. A large number of articles have been published in this area by relaxing the assumptions made in this paper and by extending it to minimise the overheads of coordination and context saving. Checkpointing for shared memory systems primarily extend cache coherence protocols to maintain a consistent memory. All of them assume that the main memory is safe for storing the context. Recently algorithms have been published for distributed shared memory systems, which extend the cache coherence protocols used in shared memory systems. They however also include methods for storing the status of distributed memory in stable storage. Most of the algorithms assume that there is no knowledge about the programs being executed. It is however felt that in development of parallel programs the user has to do a fair amount of work in distributing tasks and this information can be effectively used to simplify checkpointing and rollback recovery.

Keywords. Checkpointing algorithms; parallel & distributed computing; shared memory systems; rollback recovery; fault-tolerant systems.

1. Introduction

Parallel computing with clusters of workstations (cluster computing) is being used extensively as they are cost-effective and scalable, and are able to meet the demands of high

performance computing. Increase in the number of components in such systems increases the failure probability. It is, thus, necessary to examine both hardware and software solutions to ensure fault tolerance of such parallel computers. (We refer to a cluster of workstations as a parallel computer.) To provide fault tolerance it is essential to understand the nature of the faults that occur in these systems. There are mainly two kinds of faults: permanent and transient. Permanent faults are caused by permanent damage to one or more components and transient faults are caused by changes in environmental conditions. Permanent faults can be rectified by repair or replacement of components. Transient faults remain for a short duration of time and are difficult to detect and deal with. Hence it is necessary to provide fault tolerance particularly for transient failures in parallel computers.

Fault-tolerant techniques enable a system to perform tasks in the presence of faults. Fault tolerance involves fault detection, fault location, fault containment and fault recovery. Fault tolerance can be provided in a parallel computer at three different levels (Zomaya 1996): hardware level, architecture level and application/system software level. In the hardware and architecture levels, importance is given to fault detection and replication of tasks. In the application/system software level, checkpointing techniques are used to provide fault tolerance. It is easier and more cost effective to provide software fault tolerance solutions than hardware solutions to cope with transient failures. Thus, checkpointing is an important technique to ensure software fault tolerance.

Checkpoint is defined as a designated place in a program at which normal processing is interrupted specifically to preserve the status information necessary to allow resumption of processing at a later time (Zomaya 1996). *Checkpointing* is the process of saving the status information. By periodically invoking the checkpointing process, one can save the status of a program at regular intervals. If there is a failure one may restart computation from the last checkpoint thereby avoiding repeating computation from the beginning. The process of resuming computation by rolling back to a saved state is called *rollback recovery.*

1.1 *Application of checkpointing*

Besides its use to recover from failures, checkpointing is also used in debugging distributed programs and migrating processes in a multiprocessor system. In debugging distributed programs state changes of a process during execution are monitored at various time instances. Checkpoints assist in such monitoring. To balance the load of processors in a distributed system, processes are moved from heavily loaded processors to lightly loaded ones. Checkpointing a process periodically provides the information necessary to move it from one processor to another. The main objective of this paper is to survey checkpointing algorithms used in error recovery.

1.2 *Checkpointing in uniprocessor systems*

Events in a uniprocessor are governed by a single clock, providing a total ordering of events with respect to this clock. Thus checkpointing may be performed at a specified clock time by stopping the execution of the process and saving the state of the process in a stable storage (e.g. a disk, Siewiorek & Swarz 1982). When an error is detected all the events after the last checkpoint are repeated.

1.3 *Checkpointing in distributed systems*

Systems with more than one processor are known as multiprocessor systems. We use distributed systems, parallel systems and multiprocessor systems interchangeably referring to all of them as multiprocessor systems. As the number of processors increase the probability of any one processor failing is high. It has been found in practice that over 80% of the failures in such systems are transient and intermittent (Ralston & Reily 1993). Checkpointing and rollback recovery are particularly useful in such situations. Checkpointing, however, is more difficult in multiprocessors as compared to uniprocessors. This is due to the fact that in multiprocessors there are multiple streams of execution and there is no global clock. The absence of a global clock makes it difficult to initiate checkpoints in all the streams of execution at the same time instance. We have to pick one checkpoint from each stream in such a way that the set of these checkpoints are ''concurrent''. Such a set of checkpoints permits a consistent rollback recovery. The concept of concurrency is defined based on the ''happens before'' relation defined by Lamport (Chandy & Ramamoorthy 1972). We will now review the methods used to select a set of checkpoints, one per process, which forms a consistent global checkpoint allowing rollback recovery from such a global state.

1.4 *Ordering of events in distributed systems*

When processors interact with each other by exchanging messages, dependency is introduced among the events of different processors, making it difficult to have a total ordering of events. Lamport pointed out this and he proposed a relation called 'happens before' to have a partial ordering of events in a distributed system. This is an irreflexive, antisymmetric, transitive relation.

DEFINITION 1 (Lamport's 'happens before' relation)
(i) If a and b are two events occurring in the same process and if a occurs before b, then $a \rightarrow b$. (ii) If a is the event of sending a message and b is the event of receiving the same message in another process then, $a \rightarrow b$.

DEFINITION 2 (Concurrent events)
Two events a and b are said to be concurrent iff $(a \not\rightarrow b)$ and $(b \not\rightarrow a)$.
We denote concurrency by $\perp$.

DEFINITION 3 (Local checkpoint)
Local checkpoint is an event that records the state of a process at a processor at a given instance.

DEFINITION 4 (Global checkpoint)
Global checkpoint is a collection of local checkpoints, one from each processor.

DEFINITION 5 (Consistent global checkpoint)
A global state is said to be consistent, if all the included events form a concurrent set. A consistent global checkpoint G_c is a collection of local checkpoints, one from every processor such that each local checkpoint is concurrent to every other local checkpoint.

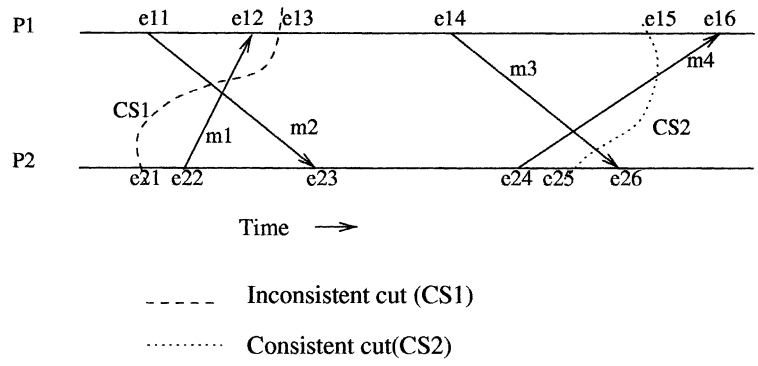

Figure 1. Process time diagram.

DEFINITION 6 (Rollback recovery)
It is a process of resuming/recovering a computation from a consistent global checkpoint.

The kth global checkpoint $G_c(k, n_p)$ in a system with n_p processors is defined as follows: $G_c(k, n_p) = \{L_c(i,l) | L_c(i,l) \perp L_c(j,m), (i \neq j)$ and $(1 \leq i,j \leq n_p), (1 \leq l,m \leq k)\}$, where $L_c(i,l)$ denotes the lth checkpoint of the ith process. Note that the kth consistent global checkpoint does not necessarily include the kth local checkpoints. In a parallel distributed system, an inconsistent global checkpoint changes the order of events leading to improper results.

Figure 1 shows a process time (PT) diagram, which depicts the events occurring in various processors of a distributed system. Message transactions and checkpoints are denoted as events. A recovery line (cut) in a (PT) diagram is a line connecting all the local checkpoints of a consistent global checkpoint. In figure 1, $\{e_{13}, e_{21}\}$, $\{e_{15}, e_{25}\}$ comprising a set are the checkpoints at different locations. Each set defines a cut as illustrated in the figure. In cut $CS1$, the checkpointing event $e21$ happens before the checkpointing event $e13$, (changing the ordering of events) making it an inconsistent cut. In CS2, checkpointing events $e25$ and $e15$ are concurrent, making the cut a consistent one.

Messages $m2$ and $m3$ are sent before the recovery line and received after the recovery line. Unless they are logged as part of the global checkpoint, they will not be available at recovery time. These messages are called *missing messages* (Lamport 1978). Message $m1$ is sent after the recovery line and received before the recovery line $CS1$. After recovery, $m1$ will be sent by $P2$ but $P1$ no longer needs this message. These kinds of messages are called *orphan messages* (Lamport 1978) and are not present in a consistent state. Orphan messages are responsible for creating inconsistent checkpoints. Netzer & Xu (1995) have given the necessary and sufficient condition for a set of checkpoints to be consistent, based on extensions of Lamport's relation (Tong *et al* 1992).

1.5 *Aspects of checkpointing*

Some of the aspects to be considered with checkpointing are (a) frequency of checkpointing, (b) contents of a checkpoint, and (c) methods of checkpointing.

1.5*a* *Frequency of checkpointing*: A checkpointing algorithm executes in parallel with the underlying computation. Therefore, the overheads introduced due to checkpointing

should be minimised. Checkpointing should enable a user to recover quickly and not lose substantial computation in case of an error, which necessitates frequent checkpointing and consequently significant overhead. The number of checkpoints initiated should be such that the cost of information loss due to failure is small and the overhead due to checkpointing is not significant. These depend on the failure probability and the importance of the computation. For example, in a transaction processing system where every transaction is important and information loss is not permitted, a checkpoint may be taken after every transaction, increasing the checkpointing overhead significantly.

1.5*b* *Contents of a checkpoint*: The state of a process has to be saved in stable storage so that the process can be restarted in case of an error. The state/context includes code, data and stack segments along with the environment and the register contents. Environment has the information about the various files currently in use and the file pointers. In case of message-passing systems, environment variables include those messages which are sent and not yet received.

1.5*c* *Methods of checkpointing*: The methodology used for checkpointing depends on the architecture of the system. Methods used in multiprocessor systems should incorporate explicit coordination unlike uniprocessor systems. In a message-passing system, the messages should be monitored and if necessary saved as part of the global context. The reason is that the messages introduce dependencies among the processors. On the other hand, a shared memory system communicates through shared variables which introduce dependency among the nodes and thus, at the time of checkpointing, the memory has to be in a consistent state to obtain a set of concurrent checkpoints.

1.6 *Overheads of a checkpointing algorithm*

During a failure-free run, every global checkpoint incurs coordination overhead and context saving overhead in a multiprocessor system. We define them as follows.

DEFINITION 7 (Coordination overhead)
In a parallel/distributed system, coordination among processes is needed to obtain a consistent global state. Special messages and piggy-backed information with regular messages are used to obtain coordination among processes. Coordination overhead is due to these special messages and piggy-backed information. The book-keeping operations necessary to maintain coordination also contribute to coordination overhead.

DEFINITION 8 (Context-saving overhead)
The time taken to save the global context[1] of a computation is defined as the context-saving overhead. This overhead is proportional to the size of the context. If stable storage is not available with every node in a multiprocessor system, the context is transferred over the network. Network transmission delay is also included in the overhead.

[1]The information that is necessary to resume a computation after it is pre-empted is called the context of that computation. In a distributed system, the computation is carried out on more than one node. Thus the global context is the collection of local contexts from each node.

1.7 *Organisation of the paper*

Section 2 describes the checkpointing algorithms of message-passing multiprocessors and § 3 those for shared memory systems. Section 4 discusses distributed shared-memory checkpointing algorithms. Section 5 describes our classification scheme. Section 6 is on future directions of research in checkpointing algorithms. Section 7 concludes the paper.

2. Checkpointing algorithms for message-passing systems

Chandy & Lamport (1985) proposed a global snapshot algorithm for distributed systems. We observe that every checkpointing algorithm proposed for message-passing (MP) systems uses Chandy & Lamport's (1985) algorithm as the base. We show that most of the algorithms proposed in the literature for checkpointing MP systems may be derived by relaxing various assumptions made by them and by modifying the way each step is carried out. As per Chandy and Lamport's model, a distributed system consists of a finite set of processors and a finite set of channels (for a detailed description of the model, refer Chandy & Lamport 1985). This section describes their algorithm, modifications possible at each step of the algorithm, followed by the features and their alternatives.

2.1 *Chandy and Lamport's algorithm*

Chandy and Lamport's CL algorithm is based on the following assumptions.

- The distributed system has a finite number of processors and a finite number of channels.
- The processors communicate with each other by exchanging messages through communication channels.
- The channels are fault-free.
- Communication delay is arbitrary but finite.
- The global state of the system includes the local states of the processors and the state of the communication channels.
- State of a channel refers to the set of messages sent along that channel and not yet received by the destination node from that channel.
- Buffers are of infinite capacity.
- Termination of the algorithm is ensured by fault-free communication.

2.1*a* *Algorithm*: The global state is constructed by coordinating all the processors and logging the channel states at the time of checkpointing. Special messages called *markers* are used for coordination and for identifying the messages originating at different checkpoint intervals. The algorithm is initiated by a centralised node. The steps followed after a checkpoint initiation, however, are the same in all the nodes except that a centralised node initiates checkpoint on its own and the other nodes initiate checkpoints as soon as they receive a marker. The steps are as below.

(1) Save the local context in stable storage;
(2) *for* $i = 1$ *to* all outgoing channels *do* Send markers along channel i;
(3) continue regular computation;
(4) *for* $i = 1$ *to* all incoming channels *do* Save incoming messages in channel i until a marker is received along that channel.

The details of each step and the possible alternatives are discussed in the following subsection.

2.2 *Modifications of Chandy and Lamport's algorithm*

Each step of the CL algorithm can be modified to accommodate some improvements in the basic global snapshot algorithm. In step one, a node saves its context in stable storage. The overhead associated with step one is *context-saving overhead*. The objective of saving the context in stable storage is to ensure its availability after a node failure. The overhead of context saving is proportional to the size of the context and the time taken to access the stable storage. Context-saving overhead can thus be reduced by (a) minimising the context size, and (b) overlapping context saving with computation. Various techniques that minimise context-saving overhead are discussed in detail in § 6.

In step two, *markers* are sent along all the outgoing channels. The purpose of a marker is

(1) to inform the receiving node that a new checkpoint has to be taken;
(2) to separate the messages of the previous and the current checkpoint interval.

At the time of checkpointing the centralised node informs all the nodes to initiate checkpoints through this marker message. CL algorithm sends markers along every channel to inform the nodes to log all transit messages onto stable storage. It is not necessary to send markers along all the channels as they may be safely eliminated along those channels in which there was no message exchange between the previous and the current checkpoint (Koo & Toueg 1987; Venkatesan 1989; Li *et al* 1991). This is achieved at the cost of monitoring all the messages exchanged along various channels from each node. Coordination through markers can also be achieved in two phases by delaying the message transmission between the two phases (Kim & Park 1993).

Checkpointing can be coordinated without using markers by sending with regular messages a header which has the checkpoint interval number in which the message originated. The simplest would be a one-bit header, which toggles between one and zero indicating the consecutive checkpoint intervals (Lai & Yang 1987). Note that the marker overhead has now become *header overhead*; overhead due to appending headers with regular messages. When a message is received with a header value different from that of the receiving node, either a new checkpoint is initiated or the message is logged depending on whether the message is an orphan message or a missing message. This one-bit header complicates checkpoint initiation when out-of-sequence messages are encountered. Message sequence numbers along with checkpoint interval number in the message header can help in controlling the number of checkpoints along with logging of missing messages and elimination of orphan messages (Venkatesh *et al* 1987). The cost of this approach is the size of the header for maintaining the message sequence numbers and checkpoint interval number. Algorithms that use only headers for coordination, require the nodes to have the ability to initiate checkpoints on their own. When nodes initiate checkpoints on their own, it is called *distributed checkpointing*. If checkpoint initiation completely depends on the header of regular messages, those nodes which have not communicated with other nodes between consecutive checkpoints cannot participate in a global checkpoint. One can also use markers just to inform about checkpoints; markers take care of coordination and headers take care of message logging (Leu & Bhargava 1988; Li *et al* 1991).

In all the schemes mentioned above, coordination is achieved at runtime and a consistent global state is always maintained in stable storage. At recovery time, the global state is

restored from stable storage and execution continues from the restored state. The next major alternative called *independent checkpointing* eliminates coordination overhead at runtime and forms a consistent global state only when it is needed, i.e only at recovery time. Instead of coordinating the nodes during every global checkpoint, nodes can be coordinated once at recovery time to form a consistent global state. When there is no coordination, nodes should be able to initiate checkpoints independently on their own. To form a consistent global state at recovery time, nodes have to maintain multiple checkpoints and messages in stable storage. The advantages of this independent checkpointing are that (i) coordination and thereby the use of markers is eliminated; (ii) nodes can initiate checkpoints at their convenience without being forced to initiate by the receipt of marker messages. The disadvantage is the maintenance of multiple checkpoints and message logs. Multiple checkpoints occupy more space and garbage collection algorithms can be run periodically to reclaim the space occupied by unwanted checkpoints (Wang *et al* 1995). Consistent global state is constructed periodically and all the checkpoints which do not belong to the recovery line are declared unwanted checkpoints. Though special messages are used for identifying the recovery line, the frequency of usage is lower when compared with a coordinated algorithm based on markers. The other significant overhead in independent checkpointing is due to logging of messages (*logging overhead*) since it has to log all the messages received. *Pessimistic logging approach* has the advantage of faster recovery since it logs a message as and when it is received (Borg *et al* 1989). By grouping the messages over a period and logging them once in a while *optimistic logging* approaches reduce the stable storage access overhead (Strom & Yomini 1985; Juang & Venkatesan 1991). If sufficient messages are not logged, multiple rollbacks (*domino effect*, Randell 1975) are possible in optimistic logging schemes. One can also send sufficient information with regular messages so that messages can be logged selectively (Wang & Fuchs 1992) thereby reducing the message logging overhead. Optimistic schemes need a complicated recovery procedure. The advantages of pessimistic and optimistic schemes can be combined to achieve minimum logging overhead with faster recovery (Elnozahy & Zwaenepoel 1992; Alvisi *et al* 1993). Further modification of independent checkpointing algorithm is possible depending on where a message is logged; at places other than the receiver (Johnson & Zwaenepoel 1987). The advantage is that the messages need not be logged onto stable storage (Young & Chiu 1994).

Yet another mode of coordination is to synchronise the clocks and initiate the checkpoints approximately at the same time in all the nodes (Cristian & Jahanian 1991). To account for the differences in the clock values, message sending can either be delayed during checkpointing or headers can be used with messages (Tong *et al* 1992).

Step three of the CL algorithm allows regular processing to proceed without waiting for the channel state recording and consequently the checkpoint operation to be completed. This is a good way of reducing the intrusion of a checkpointing algorithm but a better approach would be to overlap the context-saving process with regular computation.

Step four of CL algorithm logs those messages which cannot be generated at recovery time. The purpose served by markers in identifying these messages can also be fulfilled by headers and this was mentioned while discussing step two.

2.3 *Modifications in the features of CL algorithm*

There are certain other aspects of CL algorithm which can be improved.

2.3*a* *FIFO vs. non-FIFO channels*: CL algorithm works with FIFO channels only. If a message $m1$ followed by $m2$ is sent from p_i to p_j, $m1$ reaches before $m2$ when the channels are FIFO. Advantage of a FIFO channel is that without explicitly sending any message sequence numbers with messages, it is possible to arrange the messages in a sequence. Non-FIFO channels necessitate headers with regular messages to ensure correct ordering of messages (Silva & Silva 1992). Headers should contain sequence numbers of regular messages. The possibility of a non-FIFO channel is justified in a distributed environment, since it is possible for messages to be routed through different channels and reach the destination out of order.

2.3*b* *Centralised vs. distributed checkpoint initiation*: In a centralised algorithm like CL, there is one node which always initiates the checkpoints and coordinates the participating nodes. The disadvantage of a centralised algorithm is that all other nodes have to initiate checkpoints whenever the centralised node decides to checkpoint. Nodes can be given autonomy in initiating checkpoints by allowing any node in the system to initiate checkpoints. Such a distributed checkpointing algorithm can either support complete checkpointing (Lai & Yang 1987) or selective checkpointing (Koo & Toueg 1987).

2.3*c* *Complete vs. selective checkpointing/rollback*: In complete checkpointing, nodes have to participate in every global checkpoint and a consistent global state is readily available in the stable storage (Lai & Yang 1987). In selective checkpointing, a group of nodes which are dependent on each other participate in a checkpointing process (Leu & Bhargava 1988; Kim & Park 1993). The procedure for coordinating nodes in selective checkpointing is difficult because it has to keep track of dependencies and resolve the conflicts when multiple checkpoint requests come to a node. CL algorithm supports complete rollback because consistent global state is always maintained in stable storage. Complete rollback forces all the nodes in the system to rollback and restart. It is not, however, necessary for all the nodes to rollback and repeat the computation. Only those nodes which are dependent on each other need to rollback and the rest can continue with their computation. If the nodes communicate sparsely with each other selective rollback is better. When there is frequent communication among nodes, complete rollback is better because of message dependencies among nodes. Selective rollback needs one to maintain dependency information and to construct the recovery set after a failure (Elnozahy & Zwaenepoel 1992; Tong *et al* 1992). This is the cost associated with selective rollback.

2.3*d* *Periodic vs. nonperiodic checkpointing*: Periodic checkpointing algorithms ensure that the maximum information loss cannot exceed the period between consecutive global checkpoints. Aperiodic algorithms do not force the nodes to initiate checkpoints at predetermined times. Aperiodic algorithms are helpful in situations where nodes should not be interrupted for checkpointing at certain time instances. Aperiodic algorithms are also helpful if advancing or delaying the checkpointing process minimises the context-saving overhead. The cost incurred in aperiodic algorithms is once again in terms of constructing global consistent state. Non-periodic algorithms will require nodes to initiate checkpoints independently and so all the problems of independent checkpointing algorithms will be part of this option.

2.3*e* *Static vs. dynamic checkpointing*: Chandy and Lamport's work does not assume any knowledge about programs being executed. At runtime, periodically the checkpointing

algorithm is initiated and the checkpoints are taken. It is a dynamic checkpointing algorithm. An alternative is to identify the checkpointing locations statically before executing the programs. Static approach is widely used in uniprocessor checkpointing (Chandy & Ramamoorthy 1972). One can employ a static approach only when knowledge about the program is available. Further discussion about static approaches is given in § 6.

2.4 *Summary of checkpointing algorithms for MP systems*

We have examined Chandy & Lamport's (1985) work along with various checkpointing algorithms in the literature and how they all relate to Chandy's work. We have summarised them based on the parameters that have been taken for comparison and their features. The comparison is listed in tables 1 and 2.

3. Checkpointing algorithms for shared-memory systems

Shared-memory (SM) systems have a global address space and nodes communicate using shared variables. To reduce the memory latency, shared-memory systems use cache memory which in turn requires coherence of caches. Cache coherence protocols (Archibald & Baer 1986) help in maintaining a consistent memory. Checkpointing algorithms are incorporated as part of the cache protocols because at the time of checkpointing all cache lines must be updated and at the time of recovery no cache line should be updated more than once. The shared variables of a SM system are equivalent to the messages of a message passing system as they introduce dependency among the processes in the same way the messages introduce dependency among the processes in a message passing system.

3.1 *Cache-based checkpointing algorithms*

The basic checkpointing algorithm for a SM system is given below.

(1) At the time of checkpointing, make the main memory consistent using the cache coherence protocols.
(2) Save the process contexts in the memory.
(3) Save the global state in secondary storage.

A checkpoint can be taken whenever the memory is consistent and the state of all processes are available. Existing checkpointing algorithms for shared-memory systems initiate checkpoints based on the cache line modifications (*cache-based algorithms*). In the simplest case, a node initiates a global checkpoint whenever the number of dirty cache lines exceed a threshold (Ahmed *et al* 1990). This requires all the nodes to participate in the global checkpoint. Limiting the participation to only those nodes which are dependent on each other improves the algorithm. Another alternative is to initiate a global checkpoint whenever the effect of a modified cache is made visible to other nodes. The above three approaches were proposed by Ahmed *et al* (1990).

The cache-based recovery algorithm (Wu *et al* 1989) assumes the presence of a special memory called *recovery stack*, to reduce the time taken for context saving. when a cache line is updated, it is written onto recovery stack instead of the main memory. The main memory is updated either when the recovery stack is full or when a new checkpoint is

Table 1. Comparison of coordinated checkpointing algorithms.
p – number of processors; m – number of regular messages exchanged; c – number of communication channels; p_s – number of processors interacting between consecutive checkpoints ($p_s \leq p$).

Algorithm by	Marker overhead	Header overhead	Centralised/ distributed approach	Selective/ compelete rollback	Comments
Kim & Park (1993)	$\propto 2p_s$	Nil	Distributed	Selective	Messages delays during checkpointing two-phase protocol
Silva & Silva (1992)	$\propto p$	$\propto m$	Centralised	Complete	Non-FIFO message handling
Tong *et al* (1992)	$\propto p$ (once in few checkpoints)	$\propto 2m$	Distributed	Selective	Coordination by bounding the clock drift
Li *et al* (1991)	$\propto p$	$\propto m$	Centralised	Complete	
(a) Tag bit method					
(b) Marker based	$\propto (p + c)$	Nil	Centralised	Complete	Minimum no. of marker messages
Cristian & Jahanian (1991)	Nil	$\propto m$	Distributed	Complete	Use of time stamps to detect communication failure
Venkatesan (1989)	$\propto 2p_s$	$\propto m$	Centralised	Selective	Minimum no. of marker messages
Leu & Bhargava (1988)	$\propto 2p_s$	$\propto m$	Distributed	Selective	Non-FIFO channel handling, concurrent checkpoints/rollback
Venkatesh *et al* (1987)	Nil	$\propto m$	Distributed	Selective	Vector information with regular messages
Lai & Yang (1987)	Nil	$\propto m$	Distributed	Complete	Single-bit information with regular messages
Koo & Toueg (1987)	$\propto 2p_s$	$\propto m$	Distributed	Selective	Two-phase protocol
Chandy & Lamport (1985)	$\propto c$	Nil	Centralised	Complete	Global snapshot algorithm

Table 2. Comparison of independent checkpointing algorithms.
p – number of processors; m – number of regular messages exchanged; p_s – number of processors interacting between consecutive checkpoints ($p_s \leq p$).

Algorithm	Header overhead	Logging overhead	Recovery line construction overhead	Comments
Young & Chiu (1994)	$\propto m$ (separate vector is circulated for each message)	Nil (logged in volatile stores of other nodes)	$\propto p$	Requires a separate logical ring for circulating control messages, selective checkpoint maintenance, optimistic logging, selective rollback, FIFO requirement, quasi-synchronous approach
Leong & Agrawal (1994)	$\propto m$ (state interval of the message)	All messages with dependencies	$\propto p$	Messages are selectively eliminated and maximum recoverable state is constructed at recovery time, optimistic logging
Alvisi *et al* (1993)	$\propto m$ (vector with every message)	Vector with five entries for every message	$\propto (p + m)$	Optimistic logging of messages, acknowledgments for every message, selective rollback, no checkpoint in stable store, adaptive logging
Wang & Fuchs (1992)	$\propto m$ (vector with two entries)	Selective (dependency information with every message)	$\propto p$	Selective checkpoint maintenance, optimistic selective receiver logging, selective rollback
Elnoazahy & Zwaenepoel (1992)	$\propto m$ (antecedence graph with every message)	Selective (dependency information with each message)	$\propto p$ (logged info. is sent to recovering node)	Selective checkpoints, sender based logging, forces checkpoints for fast output commit
Strom & Yemini (1985)	$\propto m$ (time stamps of various nodes)	Selective (dependency information with every message)	$\propto p_s$	Optimistic logging, selective checkpoint maintenance receiver based logging

initiated. The size of the recovery stack and the pattern of interaction among the processes determine the frequency of checkpointing.

Analytical studies (Wu *et al* 1989) showed that the use of special memory improved the performance only when the cache lines were frequently modified. The reason is that the special memory acts as another layer in the memory hierarchy and efficiently overlaps the updation process. It is also shown that the effect of modifying the cache protocols for checkpointing does not significantly degrade the performance of the cache. Janssens & Fuchs (1994) studied the impact of various checkpointing and recovery algorithms on the performance of cache memory. Simulation shows that the effect of these algorithms are insignificant on the performance of cache memory and the use of special memory is helpful in reducing the cache write back traffic.

One important point to observe in the cache-based algorithms is that the context is not saved in stable storage. The algorithms are meant only for *soft error recovery* (Ahmed *et al* 1990). The context is either maintained in a special memory or in the main memory. It is assumed that the memory is safe and the state of the processors can be restored from memory. The advantages of soft recovery approach are: (a) time taken for checkpointing is considerably less than the checkpointing which moves the state to stable store, (b) large number of checkpoints can be initiated and the time interval between consecutive checkpoints can be small. The disadvantage is that memory failure will require the system to restart from the beginning, making the efforts expended on checkpointing useless. In this kind of soft error recovery algorithm, the checkpointing overhead is mainly due to the large number of checkpoints. Moreover, when there is no control over the number of checkpoints, it is not advisable to take a checkpointing approach that saves context in stable storage. There should be assurance that the time between consecutive checkpoints is large enough for saving the context in stable storage. From the above observation it is clear that the frequency of checkpointing cannot be predicted in any cache-based algorithm and is influenced significantly by the pattern of interaction among the nodes.

4. Checkpointing algorithms for distributed shared-memory systems

Distributed shared memory (DSM) systems have global address space like shared-memory systems but the memory is distributed across all the nodes. It is a software layer that provides the appearance of a shared-memory system to the user and internally communicates through messages. Caches are present to minimise latency in data access and the programming paradigm is shared-memory paradigm. Since DSM systems are similar to SM systems, the checkpointing algorithms for DSM systems should concentrate on making the memory consistent at the time of checkpointing. DSM systems cannot, however, use checkpointing algorithms identical to SM systems because a node failure makes a portion of the global memory not available, unlike an SM system where memory is separate and is assumed to be safe. To tolerate node failures, checkpoints should be maintained in stable storage like MP systems.

Considering the facts mentioned above, a basic DSM checkpointing algorithm is given below.

(1) At the time of checkpointing, make the distributed main memory consistent through the memory management protocols.
(2) Save the process contexts in the memory.
(3) Save the global state in secondary storage.

Step three of the algorithm cannot be omitted in DSM systems unlike as in SM systems. Techniques have been proposed to bring down the time taken for context saving. Wu & Fuchs (1989) proposed an algorithm for recovery in DSM systems in which checkpoints are initiated when a modified page of the memory is communicated to other nodes. Because it takes a long time to write pages onto secondary storage, a technique called 'Twin Paging' (Reuter 1980) is used to overlap the context-saving time with regular computation. Twin-paging technique allots two pages for every address and checkpointing algorithm flips between the pages and efficiently overlaps the context-saving overhead by overlapping the checkpointing activity with computation. The disadvantage of this algorithm is that if the nodes frequently update pages, many checkpoints will be initiated. It is even possible for the checkpoint frequency to become so high that the context saving may not be over before the initiation of the next checkpoint.

Tam & Hsu (1990) proposed an algorithm for the retrieval of page table information in a DSM system. Brown & Wu (1994) proposed an algorithm to retrieve the lost pages of a faulty node with the help of a snooping protocol. The above two techniques have to be combined with other techniques that save the context of the processes to make them suitable for resuming the computation of a failed process.

By examining the few algorithms available in the literature for DSM systems, it is clear that the approaches are extensions of cache-based algorithms because the dependency among the nodes is created through sharing of memory. We can call them *memory-based algorithms* because they are incorporated as part of memory coherence protocols. Note that these algorithms resort to techniques for overlapping the context-saving process with computation because the time taken to write the context in stable storage is significant.

5. Classification of checkpointing algorithms

Having seen the various algorithms available in the literature, we propose a classification scheme. We also give a list of desirable features of a checkpointing algorithm.

5.1 *Classification tree*

The classification tree is given in figure 2. The first level of classification is based on the availability of a global clock. Absence of a global clock has led to a variety of checkpointing algorithms in multiprocessor systems. In both uniprocessor and multiprocessor systems two major approaches known as *static* and *dynamic* approaches are used to identify checkpointing locations. In uniprocessors, static approaches identify the checkpointing locations prior to program execution using either a task graph of the program (Chandy & Ramamoorthy 1972) or by the compiler analysing the program (Li & Fuchs 1990) the dynamic approach identifies the checkpointing locations at runtime.

In multiprocessor systems, the static approach has not been widely used except for a recent algorithm based on task graphs (Kalaiselvi & Rajaraman 2000). Almost all the algorithms we have seen for distributed systems are dynamic. Dynamic algorithms for multiprocessors are classified based on the architecture of the system as it determines the way dependencies are introduced among the communicating processes which in turn influence the way checkpointing has to be carried out. Shared memory systems use cache-based algorithms and they are meant only for soft error recovery. Distributed shared memory systems use memory-based schemes that can withstand node failures.

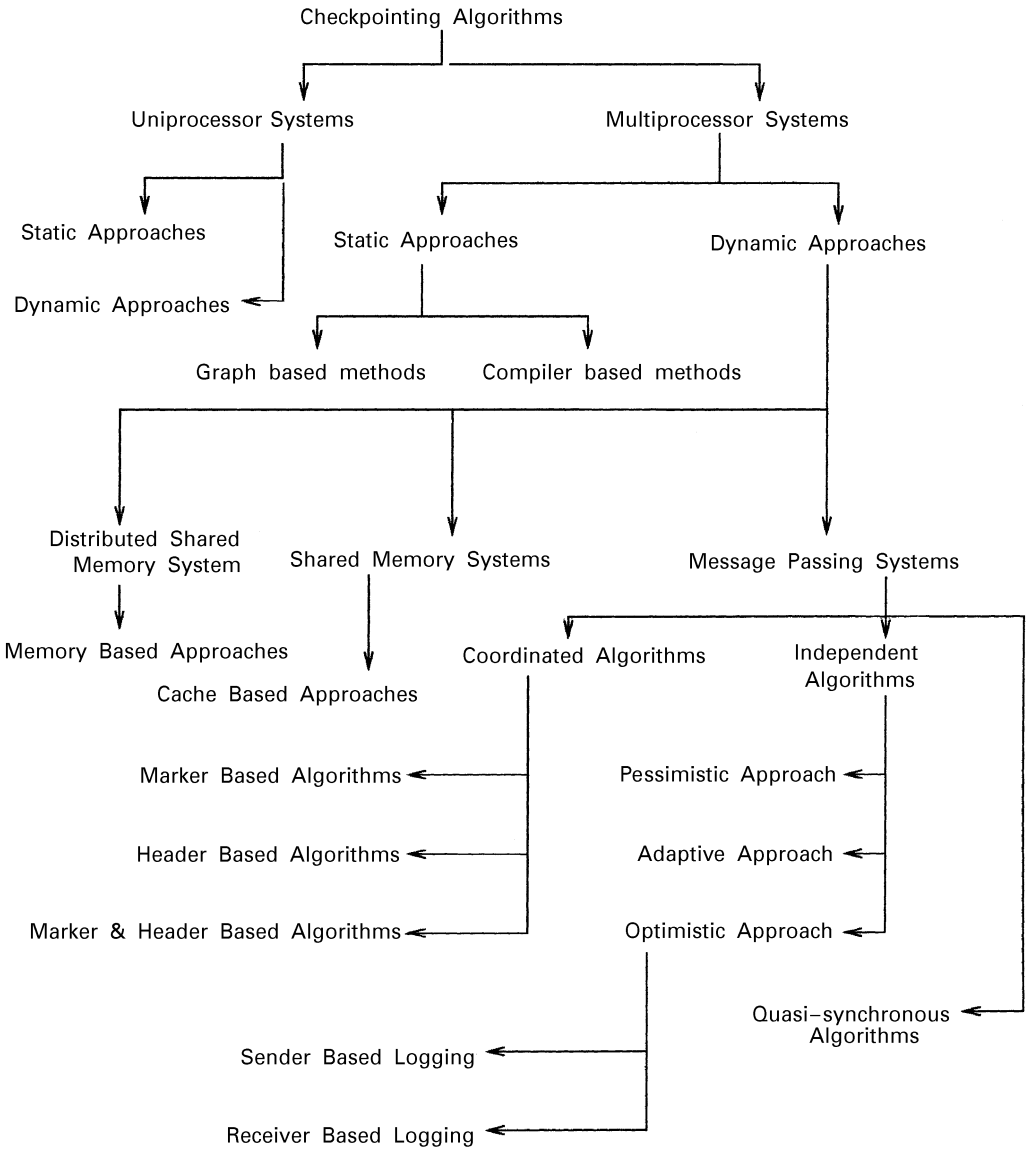

Figure 2. Classification of checkpointing algorithms.

In message-passing systems, the major difference in algorithms is based on whether coordination is done at runtime or at recovery time. Coordinated approaches coordinate the nodes and form a consistent state at runtime, whereas independent algorithms form a consistent state only at recovery time. The recent approach is to coordinate the checkpoints partially and is called quasi-synchronous approach (Manivannan & Singhal 1996). In coordinated algorithms, coordination is achieved through markers, headers or both. The main distinction between approaches used in independent schemes is the method used to log messages. They follow pessimistic, optimistic or a combination of pessimistic and

optimistic logging methods. Optimistic methods can further be classified based on where the messages are logged: sender-based logging or receiver-based logging.

5.2 *Desirable features of a checkpointing algorithm*

(1) The time taken by the checkpointing algorithm should be minimum during a failure-free run. In other words, increase in total execution time due to checkpointing should not be significant.
(2) Recovery should be fast in the event of a failure. Availability of a consistent global state in stable storage expedites recovery.
(3) Domino effect or rollback propagation should be eliminated completely. Cascading rollbacks of processes due to dependencies among them is termed the domino effect.
(4) Selective rollback should be possible.
(5) Dependency on the cache/memory coherence protocols in shared-memory systems should be minimum.
(6) Resource requirements (memory and processor) for checkpointing should be minimum.
(7) Modifications introduced on the network transmission protocols in case of message-passing systems and cache/memory coherence protocols in case of shared-memory systems should be minimum.

6. Future direction of research in checkpointing

In the first subsection (§ 6.1) we summarise shared and distributed shared-memory checkpointing algorithms and their current status. Section 6.2 summarises message-passing algorithms and their current status, while § 6.3 considers the potential for improvement if one concentrates on minimising context-saving overhead. Section 6.4 points out the advantage of adopting static approaches in multiprocessor systems which minimise the runtime overhead of a checkpointing algorithm.

6.1 *Shared and distributed shared-memory systems*

From the discussion about shared-memory checkpointing algorithms, it is clear that cache coherence protocols already maintain consistent memory and that checkpointing algorithms are part of them. Only the number of checkpoints needs to be controlled and this depends on the node interactions, which in turn depend on the program behaviour. Distributed shared-memory (DSM) systems are considered to be better than the conventional bus-based shared-memory systems as they are scalable. DSM systems should allow sufficient time between checkpoints to complete the context-saving process. We saw that DSM systems should save the context in stable storage, if they have to withstand node failures. Periodic checkpointing algorithms would be suitable for DSM systems since they have control over the number of checkpoints unlike algorithms which depend on the interactions among nodes. One such attempt was made recently for DSM systems based on Scalable Coherent Interface (SCI) (IEEE 1992) standards. The algorithm is initiated periodically to maintain consistency in memory and to maintain the consistent state in stable storage.

One may proceed to develop improved checkpointing algorithms for DSM systems by considering the following: (a) Control over number of checkpoints, and (b) scalability of the checkpointing algorithm along with system scalability.

6.2 *Message passing systems*

In message passing systems, earlier performance studies by Bhargava *et al* (1990) showed that coordinated checkpointing algorithms are costlier because they incur extra communication. Later simulation studies by Elnozahy *et al* (1992), however, revealed that coordinated algorithms are better than independent algorithms. The cost of coordination is much lower when compared with the cost of maintaining multiple checkpoints and logging messages.

One can follow an approach that is a combination of coordinated and independent checkpointing algorithms to reap the benefits of both the approaches. Recently, Elnozahy & Zwaenepoel (1994) proposed one such algorithm. In between two coordinated checkpoints, messages are logged like the independent approach so that the rollback recovery is restricted to just the faulty processor. Periodically checkpoints are initiated to maintain a consistent global state in stable storage at all times. Manivannan & Singhal (1996) have suggested a quasi-synchronous approach recently. Like the coordinated approach a consistent recovery line is always maintained in stable storage by selectively logging messages and initiating checkpoints when necessary. Instead of the usual garbage collection techniques followed in independent approaches, this algorithm always maintains the latest checkpoint and makes sure that the rollback will not go beyond the latest checkpoint of a node.

We have seen all possible modifications that can be made in checkpointing algorithms of MP systems. The algorithms concentrate on different ways to handle coordination. From the recent algorithms, it is clear that further improvement is possible by combining the existing approaches. We feel that future algorithms should concentrate possibly on static approaches for further improvement. Detailed discussion about static approaches is given subsequently.

6.3 *Minimising context-saving overhead*

The interesting observation in all the algorithms discussed so far is that they all attempt to minimise the coordination and message-logging overheads. A better performance can be achieved by concentrating on minimising the context-saving overhead which constitutes a significant portion of the checkpointing overhead.

Context-saving overheads can be reduced by (a) reducing the context size and (b) overlapping context-saving with computation. One way to reduce the context size is to go for *incremental checkpointing* Elzonahy *et al* (1992). Instead of moving all the pages to stable storage, it is sufficient to move only those pages which were modified in the current checkpoint interval. Unmodified pages can be obtained from the previous checkpoint. Results show that incremental checkpointing is certainly better than moving all the pages of the process to secondary storage (Elzonahy *et al* 1992). The second method to reduce the context size is to use *compression techniques* (Plank & Li 1994) on the context. Fast compression algorithms should be used for significant improvement in context-saving time.

The above techniques reduce the size of the context but copying the context to stable storage takes significant time. The time spent in context-saving cannot be eliminated but can be overlapped with computation. Overlapping hides the context-saving overhead and reduces the context-saving time. In the *pre-copying* technique (Theimer *et al* 1985), a portion of the main memory is reserved for checkpointing. At the time of checkpointing, the pages to be written to stable storage are moved to this special memory area. The special memory pages are moved to stable storage when computation proceeds. When the number

of pages to be written to stable storage is more than the capacity of this special memory, multiple precopying phases are needed. Any attempt made to modify a page when the special memory is full, suspends the process, moves the page to stable storage and resumes the process. This technique is advantageous compared to other techniques when the number of pages to be written is small. An extension of this technique is *main-memory* checkpointing (Plank 1993) where the size of the special memory is as big as the memory itself. The entire address space is first copied into special memory and the contents are moved to stable storage when computation proceeds. The advantage of this method over the pre-copying technique is that it avoids page faults, but the memory requirements are high. With incremental checkpointing pre-copying is a good technique. In *copy-on-write* (Fitzgerald & Rashid 1986) technique all the modified pages are just write-protected at the time of checkpointing and later moved to stable storage. Computation proceeds in parallel with context saving. Once a page is moved to the special memory area write-protect is removed from that page and it is available for further modification. Any attempt to modify a write-protected page results in a page fault. The page is first copied onto a special memory area and write-protect is removed from the page. From special memory, the page is moved to stable storage. Zwaenepoel's experiment (Elnozahy *et al* 1992) shows that the overlapping techniques mentioned above help in minimising the context-saving overhead. Bowen & Pradhan (1992) suggested a technique called *virtual checkpointing*, which can be directly included in the virtual memory management protocol. This can be included in shared-memory systems, message-passing systems and in DSM systems and needs extensive support from memory management protocols.

Plank & Li (1994) suggested *diskless checkpointing* technique, which eliminates access to stable storage completely. A dedicated checkpoint processor maintains copies of all the processors' checkpoints by XOR-ing their memory contents. Since the copy is maintained in the memory of the extra processor, frequency of checkpointing can be higher. The checkpoint processor periodically saves its memory in stable storage. The idea of using the parities and saving the contents is similar to the one used by RAID (Chen *et al* 1994) architectures. In addition to the extra processors this technique demands special memory in each processor.

All these techniques for minimising the context-saving overhead can be followed by the existing checkpointing algorithms, when assistance is provided by the memory management protocols. This certainly improves the algorithms. One can consider how efficiently these techniques can be integrated with existing algorithms and can explore the compatibility of a checkpointing algorithm with a context-minimising technique.

6.4 *Static approaches*

Multiprocessor algorithms discussed so far in this paper are dynamic which identify the checkpointing locations at run time. As mentioned in §1, they assume that they do not have any knowledge about the interactions among the various modules of a problem being solved. This is the main reason for coordinating the checkpoints in coordinated approach and maintaining multiple checkpoints and messages in case of independent checkpointing approach. In reality, a thorough understanding of the various modules of the program is necessary for developing a parallel program. This information can be used effectively to identify the checkpointing locations statically before running the program.

Chandy & Ramamoorthy (1972) proposed one of the earliest static checkpointing algorithms for uniprocessor system. Their objective was to locate the optimal places to

checkpoint so that the recovery overhead is minimum. They assume knowledge about the execution and recovery times of various modules of the program. Upadhyaya & Saluja (1986) suggested that by using cache for recovery, the context saving time can be reduced. Mishra *et al* (1991) suggested an algorithm that identifies optimal locations of recovery points in an inverted binary tree structured task graph. Chen *et al* (1989) suggested solutions for recovery point selection in uniprocessor and multiprocessor systems. In single-processor systems, solutions for optimal placement of recovery points and in multiprocessor systems, analyses for placement strategy are given by them. They have not considered the effect of maintaining a consistent recovery line in multiprocessor systems.

Li & Fuchs (1990) suggested a compiler assisted checkpointing technique. Their claim is that it is easier to include checkpoints by analysing the code generated by a compiler than by generating graphs of the program and identifying checkpointing locations. This technique analyses the code and locates the optimal checkpointing places. Based on this idea and a technique called *user-directed checkpointing*, Plank *et al* (1995), and Beck *et al* (1994) suggested a compiler-assisted checkpointing method that uses many techniques. Incremental checkpointing technique identifies the dirty pages of the memory and moves them to stable storage, and cannot identify the lifetime of the variables. With user-assistance the lifetime of variables can be identified and those variables which are no longer in use (dead) can be safely eliminated from the context. There may be some read-only variables which can also be eliminated from the context because they can be restored along with the code. Since identification is done by a user the technique is called user-directed checkpointing. Performance studies show that user-directed checkpointing is better in many cases (Beck *et al* 1994).

The advantage of a static approach is that it eliminates coordination overhead at run time. Compiler-assisted and user-assisted static techniques help in minimising the context size too. The overheads incurred by static algorithms are preprocessing overheads and do not increase the runtime overheads. In a parallel/distributed environment, one cannot follow the approaches as such because the interactions among the nodes introduce dependencies among the multiple streams of execution.

A recent approach based on directed acyclic task graph being the computation model is proposed for static checkpointing (Kalaiselvi & Rajaraman 2000). From the task graph, a consistent global checkpoint is identified statically in $O(m)$ time where m is the number of edges present in the task graph. Unlike other static techniques, this is meant for multi-processor environment. The coordination overhead is eliminated and context-saving overhead is minimized by introducing checkpoints at places that incur minimum overhead. One more advantage to note is that since the checkpoints are not initiated at the same time in all nodes, there will not be contention for accessing the stable storage when checkpoints are saved in a single disk.

One can further investigate this algorithm considering the scheduling effects and the programming paradigms. Other static methods can also be investigated because they have the advantage of minimum runtime overhead.

7. Conclusions

A survey of the literate on checkpointing algorithms show that a large number of papers have been published on checkpointing message-passing distributed computers. A majority of these algorithms are based on the seminal article by Chandy & Lamport (1985) and have

been obtained by relaxing many of the assumptions made by them; the main aim of improving the earlier extensions of the Chandy & Lamport (1985) algorithms was to minimise the overhead of coordination between processes in a multiprocessor system. More recent published work attempts to minimise the context-saving overhead.

A smaller number of algorithms have been proposed to checkpoint shared-memory multiprocessors. These algorithms primarily extend cache coherence protocols to maintain a consistent memory. These algorithms assume the main memory to be safe and do not save context in disk.

More recently, algorithms have been proposed for distributed shared-memory systems. In these systems also maintenance of cache coherence of the logical global memory is important for checkpoints. As the physical memory is distributed it is necessary to save main memory contents in the disk. Thus context saving overhead is higher when compared to shared-memory systems.

We also see that most of the algorithms assume no prior knowledge on the structure of programs meant for execution on multiprocessors. In practice, considerable information about such programs is available. We suggest that use of this knowledge by checkpointing algorithms can considerably reduce the coordination and context-saving overheads of such algorithms.

References

Ahmed R E, Frazier R C, Marinos P N 1990 Cache aided rollback error recovery (CARER) algorithms for shared memory multiprocessor systems. *Proc. IEEE 20th Int. Symp. on Fault Tolerant Computing* pp 82–88

Alvisi L, Hoppe B, Marzullo K 1993 Nonblocking and orphan free message logging protocols. *Proc. IEEE 23rd Int. Symp. Fault Tolerant Computing* pp 145–154

Archibald J, Baer J-L 1986 Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.* 4: 273–298

Beck M, Plank J S, Kingsley G 1994 Compiler-assisted checkpointing. Technical Report, CS-94-269, University of Tennessee, Knoxville, TN

Bhargava B, Lian S-R, Leu P-J 1990 Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms. *Proc. IEEE 6th Int. Conf. on Data Eng.* pp 182–189

Borg A, Blau W, Gratsch W, Herrman H, Oberle W 1989 Fault tolerance under UNIX. *ACM Trans. Comput. Syst.* 7: 1–24

Bowen N S, Pradhan K 1992 Virtual checkpoints: Architecture and performance. *IEEE Trans. Comput.* 41: 516–525

Brown L, Wu J 1994 Dynamic snooping in a fault-tolerant distributed shared memory. *Proc. IEEE 14th Int. Conf. on Distributed Computing Syst.* pp 218–226

Chandy K M, Lamport L 1985 Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3: 63–75

Chandy K M, Ramamoorthy C V 1972 Rollback and recovery strategies for computer programs. *IEEE Trans. Comput.* C-21: 546–556

Chen P M, Lee E K, Gibson G A, Katz R H, Patterson D A 1994 RAID-High performance, reliable secondary storage. *ACM Comput. Surv.* 26: 145–185

Chen S-K, Tsai W T, Thuraisingham M B 1989 Recovery point selection on a reverse binary tree task model. *IEEE Trans. Software Eng.* 15: 963–976

Cristian F, Jahanian F 1991 A timestamp-based checkpointing protocol for long-lived distributed computations. *Proc. IEEE Conf. on Reliable Distributed Syst.* pp 12–20

Elnozahy E N, Zwaenepoel W 1992 Manetho: Transparent rollback recovery with low overhead, limited rollback and fast output commit. *IEEE Trans. Comput.* 41: 526–531

Elnozahy E N, Zwaenepoel W 1994 On the use and implementation of message logging. *Proc. IEEE Int. Symp. on Fault Tolerant Computing* pp 298–307

Elnozahy E N, Johnson D B, Zwaenepoel W 1992 The performance of consistent checkpointing. *Proc. IEEE 11th Symp. on Reliable Distributed Syst.* pp 39–47

Fitzgerald R, Rashid R F 1986 The integration of virtual memory management and interprocess communication in accent. *ACM Trans. Comput. Syst.* 4: 147–177

IEEE 1992 Std. 1596–1992. *IEEE Scalable Coherent Interface (SCI)* (Piscataway, NJ: IEEE)

Janssens B, Fuchs W K 1994 The performance of cache based error recovery in multiprocessors. *IEEE Trans. Parallel Distributed Syst.* 5: 1033–1043

Johnson D B, Zwaenepoel W 1987 Sender-based message logging. *Proc. IEEE Int. Symp. Fault Tolerant Comput.* pp. 14–19

Juang T T-Y, Venkatesan S 1991 Crash recovery with little overhead. *Proc. IEEE 11th Int. Conf. on Distributed Comput. Syst.* pp 454–461

Kalaiselvi S, Rajaraman V 2000 Task graph based checkpointing in parallel/distributed systems. *J. Parallel Distributed Comput.* (submitted)

Kim J L, Park T 1993 An efficient protocol for checkpointing recovery in distributed systems. *IEEE Trans. Parallel Distributed Syst.* 4: 955–960

Koo R, Toueg S 1987 Checkpointing and rollback recovery for distributed systems. *IEEE Trans. Software Eng.* SE-13: 23–31

Lai T H, Yang T H 1987 On distributed snapshots. *Inf. Process. Lett.* 25: 153–158

Lamport L 1978 Time clocks and the ordering of events in a distributed system. *Commun. ACM* 21: 558–565

Leong H V, Agrawal D 1994 Using message semantics to reduce rollback in optimistic message logging recovery schemes. *Proc. IEEE 14th Conf. on Distributed Computing Syst.* pp 227–234

Leu P-J, Bhargava B 1988 Concurrent robust checkpointing and recovery in distributed systems. *Proc. Int. Conf. on Data Engineering* pp 154–163

Li C-C J, Fuchs W K 1990 CATCH-Compiler-assisted techniques for checkpointing. *Proc. 1990 Int. Symp. on Fault Tolerant Computing* pp 74–81

Li K, Naughton J F, Plank S 1991 Checkpointing multicomputer applications. *Proc. IEEE Conf. on Reliable Distributed Syst.* pp 2–11

Manivannan D, Singhal M 1996 A low-overhead recovery technique using quasi-synchronous checkpointing. *Proc. IEEE Int. Conf. on Distributed Computing Syst.* pp 100–107

Mishra S K, Raghavan V V, Tzeng N-F 1991 Efficient algorithms for selection of recovery points in task models. *IEEE Trans. Software Eng.* 17: 731–734

Netzer R H B, Xu J 1995 Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel Distributed Syst.* 6: 165–169

Plank J S 1993 *Efficient checkpointing on MIMD architectures*. Doctoral dissertation, Dept. of Computer Science, Princeton University, Princeton, NJ

Plank J S, Li K 1994a ickp: A consistent checkpointer for multicomputers. *IEEE Parallel Distributed Technol.* 2: 62–67

Plank J S, Li K 1994b Faster checkpointing with $N + 1$ parity. *Proc. IEEE Int. Symp. on Fault Tolerant Computing* pp 288–297

Plank J S, Beck M, Kingsley G, Li K 1995 Libckpt: Transparent checkpointing under unix. *USENIX Winter Technical Conference*, New Orleans, Louisiana

Ralston A, Reily E D 1993 *Encyclopedia of computer science* 3rd edn (New York: IEEE Press)

Randell B 1975 System structure for software fault tolerance. *IEEE Trans. Software Eng.* SE-1: 220–232

Reuter A 1980 A fast transaction-oriented logging scheme for undo recovery. *IEEE Trans. Software Eng.* SE-6: 348–356

Siewiorek D P, Swarz S 1982 *The theory and practice of reliable system design* (Cambridge, MA: Digital Press)

Silva L, Silva J 1992 Global checkpointing for distributed programs. *Proc. IEEE 11th Symp. on Reliable Distributed Syst.* pp 155–162

Strom R E, Yemini S 1985 Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.* 3: 204–226

Tam V-O, Hsu M 1990 Fast recovery in distributed shared virtual memory systems. *Proc. IEEE 10th Int. Conf. on Distributed Computing Syst.* pp 38–45

Theimer M, Lantz K, Cheriton D R 1985 Preemptable remote execution facilities in the V-system. *Proc. of the 10th ACM Symp. on Operating Syst. Principles* pp 2–11

Tong Z, Kain R Y, Tsai W T 1992 Rollback recovery in distributed systems using loosely synchronized clocks. *IEEE Trans. Parallel Distributed Syst.* 3: 246–251

Upadhyaya J S, Saluja K K 1986 A watchdog processor based general rollback technique with multiple retries. *IEEE Trans. Software Eng.* SE-12: 87–95

Venkatesan S 1989 Message optimal incremental snapshots. *Proc. IEEE 9th Int. Conf. Distributed Comput. Syst.* pp 53–60

Venkatesh K, Radhakrishnan T, Li H F 1987 Optimal checkpointing and local recording for domino-free rollback recovery. *Inf. Process. Lett.* 25: 295–303

Wang Y-M, Fuchs W K 1992 Optimistic message logging for independent checkpointing in message passing systems. *Proc. IEEE 11th Symp. on Reliable Distributed Syst.* pp 147–154

Wang Y-M, Chung P-Y, Lin I-J, Fuchs W K 1995 Checkpoint space reclamation for uncoordinated checkpointing in message passing systems. *IEEE Trans. Parallel Distributed Syst.* 6: 546–554

Wu K-L, Fuchs W K 1989 Recoverable distributed shared virtual memory: Memory coherence and storage structures. *Proc. 1989 Int. Symp. on Fault Tolerant Computing* pp 520–527

Wu K-L, Fuchs W K, Patel J H 1989 Cache based error recovery for shared memory multiprocessor systems. *Proc. Int. Conf. on Parallel Processing* pp I159–I166

Young C, Chiu G 1994 A crash recovery technique in distributed computing systems. *Proc. IEEE 14th Int. Conf. on Distributed Computing Syst.* pp 235–242

Zomaya A Y H 1996 *Parallel and distributed computing handbook* (New York: McGraw-Hill)