Staggered Consistent Checkpointing

Nitin H. Vaidya, Senior Member, IEEE

Abstract—A consistent checkpointing algorithm saves a consistent view of a distributed application’s state on stable storage. The traditional consistent checkpointing algorithms require different processes to save their state at about the same time. This causes contention for the stable storage, potentially resulting in large overheads. Staggering the checkpoints taken by various processes can reduce checkpoint overhead. This paper presents a simple approach to arbitrarily stagger the checkpoints. Our approach requires that the processes take consistent logical checkpoints, as compared to consistent physical checkpoints enforced by existing algorithms. Experimental results on nCube-2 are presented.

Index Terms—Staggered checkpoints, consistent recovery line, rollback recovery, stable storage contention, fault tolerance.

1 INTRODUCTION

APPLICATIONS executed on a large number of processors, either in a distributed environment or on multicore systems such as nCube or Intel Paragon, are subject to processor failures. Consistent checkpointing is a commonly used technique to prevent complete loss of computation upon a failure [2], [3], [7], [8], [12], [16], [18], [22]. A consistent checkpointing algorithm saves a consistent view of a distributed application’s state on stable storage (often, a disk is used as a stable storage). The loss of computation upon a failure is bounded by taking consistent checkpoints with adequate frequency.

The traditional consistent checkpointing algorithms require different application processes to save their state at about the same time. This causes contention for the stable storage when multiple processors share a stable storage, potentially resulting in significant performance degradation. Clearly, if each processor has access to a separate stable storage, such contention will not occur.

However, in many installations of multicomputers and distributed systems, although multiple stable storages (disks) may be available, the number of stable storages is often smaller than the number of processors. Thus, multiple processors are required to share a stable storage.

Staggering the checkpoints taken by various processes can reduce the overhead of consistent checkpointing by reducing stable storage contention, as observed by Plank [18]. Plank proposed some techniques for staggering the checkpoints [18], however, these techniques result in “limited” staggering in that not all processes’ checkpoints can be staggered. Moreover, the previous algorithms do not have much control on which checkpoints are staggered. Ideally, one would like to be able to stagger the checkpoints in a manner most appropriate for a given system. In particular, the nature of staggering would need to be different depending on whether the different processors share a single stable storage or multiple stable storages, as discussed later.

In systems where processors are able to make an “in-memory” copy of entire process state, checkpoint staggering is trivial. In such cases, each process can first take its checkpoint in-memory and then the checkpoints can be written to stable storage one process at a time [18] (thus, writing of the checkpoints would be staggered). Memory requirement of in-memory checkpointing can be reduced by using the copy-on-write optimization [15]. With the copy-on-write optimization, only the pages that are modified by a process since its checkpoint, but not yet written to stable storage, need to be explicitly copied in-memory. Although in-memory checkpointing, particularly using the copy-on-write optimization, can obviate the need for special checkpoint staggering algorithms, experiments by Plank et al. [18], [4] on Intel Paragon and DEC Firefly show that many applications do not have free memory necessary to take an in-memory checkpoint. The checkpoint staggering scheme presented in this paper is useful in such cases.

The objective of this paper is to show how checkpoints can be staggered in a controlled manner, independent of the application’s communication patterns. This paper presents a simple approach to arbitrarily stagger the checkpoints. Our approach requires that the processes take consistent logical checkpoints, as compared to consistent physical checkpoints enforced by existing algorithms for checkpoint staggering. As elaborated later, a physical checkpoint is a copy of a process’ state, and a logical checkpoint is obtained by saving sufficient information (e.g., messages) to recover a process’ state.

The paper is organized as follows: Section 2 discusses the related work. Section 3 discusses physical and logical checkpoints. Section 4 presents consistent checkpointing algorithms proposed by Chandy and Lamport [3] and Plank [18]. Section 5 presents the proposed algorithm. Section 6 presents experimental results. Some variations of the proposed scheme are discussed in Section 7. Section 8 summarizes the paper.
2 RELATED WORK

Two approaches may be used to checkpoint a distributed application: uncoordinated checkpointing and coordinated checkpointing. In uncoordinated checkpointing, different processes take checkpoints independently [6], [10], [12], [21]. In the event of failure, a consistent state is determined using the independently taken checkpoints. In the coordinated (or consistent) checkpointing approach, different processes in the application coordinate to take checkpoints that are mutually consistent [3], [14], [18].

With uncoordinated checkpointing, the checkpoints may often be staggered because different processes take checkpoints independently. Therefore, special checkpoint staggering algorithms may not be necessary in case of uncoordinated checkpointing. However, it has been shown that the uncoordinated checkpointing approach often performs poorly [24] because the number of checkpoints taken tends to be larger (as compared to when using coordinated checkpointing). This paper considers the coordinated checkpointing approach. Previous work has shown that, despite the overhead of control messages required for coordination, coordinated checkpointing schemes can be implemented with a low overhead [7], [6].

Plank [18] was the first to observe that stable storage contention can be a problem for coordinated (consistent) checkpointing and suggested checkpoint staggering as a solution. As explained later, the degree of staggering with Plank’s algorithm (based on the Chandy-Lamport algorithm [3]) is limited in that checkpoints of many processes are not staggered. In contrast, our algorithm allows arbitrary and controlled staggering of checkpoints. Plank [18] also presents another approach for staggering checkpoints that is applicable to wormhole routed networks. This algorithm also does not permit arbitrary/controlled staggering.

Fowler and Zwaenepoel [9] present an algorithm for determining causal breakpoints (for the purpose of debugging). As a part of the breakpoint algorithm, they establish consistent recovery lines using an algorithm similar to ours. Our approach can be considered to be a modification of the algorithm in [9] to facilitate checkpoint staggering. Because the algorithm in [9] was designed for debugging purposes, various possibilities for checkpoint staggering and different approaches for establishing checkpoints were not considered.

Long [16] discusses an evolutionary checkpointing approach that is similar to logical checkpointing. Our algorithm staggers the checkpoints, while the scheme in [16] does not allow staggering. Long [16] also assumes synchronized communication and an upper bound on communication delays; no such assumptions are made in the proposed scheme.

Wang et al. [23] introduced the term logical checkpoint. They present an algorithm to determine a recovery line consisting of consistent logical checkpoints after a failure occurs. This recovery line is used to recover from the failure. Their goal is to determine the “latest” consistent recovery line using the information saved on the stable storage. Message logging [1] and independent checkpointing schemes, such as [12], also, effectively, determine a recovery line consisting of consistent logical checkpoints after a failure occurs. In these schemes, during failure-free operation, each process is allowed to independently take checkpoints and log messages. On the other hand, our scheme coordinates logical checkpoints before a failure occurs. These logical checkpoints are used to recover from a future failure.

Staggering the checkpoints by various processes tends to increase the elapsed time (sometimes called checkpoint “latency” [20]) while the checkpointing algorithm is in progress. Our previous work [20] shows that a large increase in checkpoint latency is acceptable if it is accompanied by even a small decrease in checkpoint overhead. Therefore, techniques such as staggering are of interest even though they may result in greater checkpoint latency.

3 Physical and Logical Checkpoints

A process may take two types of checkpoints, namely, physical checkpoints and logical checkpoints. A process is said to have taken a physical checkpoint at some time \( t_1 \) if the process state at time \( t_1 \) is available on the stable storage. A process is said to have taken a logical checkpoint at time \( t_1 \) if adequate information is saved on the stable storage to allow the process state at time \( t_1 \) to be recovered. A physical checkpoint is trivially a logical checkpoint, however, the converse is not true.

A physical checkpoint can be taken in two different ways. One possibility is to save the entire process state on the stable storage. The second possibility is to take an incremental checkpoint [17]. In incremental checkpointing, only the difference between current process state and process state at the previous physical checkpoint is saved.

Now, we summarize three approaches for taking a logical checkpoint at time \( t_1 \).

Approach 1. One approach for establishing a logical checkpoint at time \( t_1 \) is to take a physical checkpoint at some time \( t_0 \leq t_1 \) and log (on stable storage) all messages delivered to the process between \( t_0 \) and \( t_1 \). This approach is essentially identical to that presented by Wang et al. [23]. Fig. 1 presents an example wherein process \( P \) takes a physical checkpoint at time \( t_0 \). Messages \( M_1, M_2, \) and \( M_3 \) are delivered to process \( P \) by time \( t_1 \). To establish a logical checkpoint of process \( P \) at time \( t_1 \), messages \( M_1, M_2, \) and \( M_3 \) are logged on the stable storage. We summarize this approach as:

\[
\text{physical checkpoint + message log = logical checkpoint.}
\]

A process is said to be deterministic if its state depends only on its initial state and the messages delivered to it [12], [19]; otherwise, the process is said to be nondeterministic. Approach 1 for taking a logical checkpoint may only be used for deterministic processes, because it implements a logical checkpoint using a physical checkpoint and the message log.

Approach 2. The essential purpose behind saving the messages above is to be able to recreate the state at time \( t_1 \). This may also be achieved by taking a physical checkpoint at time \( t_0 \) and taking an incremental check-
Approach 3. The above two approaches take a physical checkpoint prior to the desired logical checkpoint followed by logging of additional information (either messages or incremental state change). The third approach is the converse of the above two approaches. Here, the physical checkpoint is taken at a time \( t_2 \), where \( t_2 > t_1 \). In addition, enough information is saved to undo the effect of messages received between time \( t_1 \) and \( t_2 \). For each relevant message (whose effect must be undone), an anti-message is saved on the stable storage. The notion of an anti-message here is similar to that used in time warp mechanism [11] or that of UNDO records [5] in database systems. Anti-message \( M^* \) corresponding to a message \( M \) can be used to undo the state change caused by message \( M \).

Fig. 2 illustrates this approach. A logical checkpoint of process \( P \) is to be established at time \( t_1 \). Process \( P \) is delivered messages \( M_4 \) and \( M_5 \) between time \( t_1 \) and \( t_2 \). A physical checkpoint is taken at time \( t_2 \) and anti-messages corresponding to messages \( M_4 \) and \( M_5 \) are logged on the stable storage. The anti-messages are named \( M_4^* \) and \( M_5^* \), respectively.

To recover the state of process \( P \) at time \( t_1 \), the process is initialized to the physical checkpoint taken at time \( t_2 \) and then anti-messages \( M_5^* \) and \( M_4^* \) are sent to the process. The order in which the anti-messages are delivered is reverse the order in which the messages were delivered. We summarize this approach as:

\[
\text{anti-message log + physical checkpoint = logical checkpoint.}
\]

If the overhead of creating the anti-messages is high, then this approach is not suitable. One approach to create anti-messages is that the anti-messages may be formed by the application itself, taking the application characteristics into account. Another approach is to implement an anti-message \( M^* \) as the set unmodified versions of the pages that are modified after receiving message \( M \), but before the next message is received. Similar to copy-on-write, this approach will keep track of the old state that is changed on receipt of a particular message. For instance, if only pages \( X \) and \( Y \) are modified after receiving message \( M \) (but before receiving the next message), then \( M^* \) consists of copies of pages \( X \) and \( Y \) at the time message \( M \) is received. These old pages can be used to undo the state change caused by message \( M \), starting from a future physical checkpoint. This approach is essentially equivalent to the copy-on-write optimization.

4 Chandy-Lamport Algorithm
Chandy and Lamport [3] presented an algorithm for taking a consistent checkpoint of a distributed system. Assume that the processes communicate with each other using first-in-first-out (FIFO) unidirectional communication channels; a bidirectional channel can be modeled as two unidirectional channels. For simplicity, we assume that the communication graph is fully connected. The algorithm presented next is essentially identical to Chandy-Lamport [3], [18] and assumes that a certain process (named \( P_0 \)) is designated as the checkpoint coordinator.

Algorithm: The coordinator process \( P_0 \) initiates the consistent checkpointing algorithm by sending marker messages on each channel, incident on, and directed away from \( P_0 \) and immediately takes a checkpoint. (This is a physical checkpoint.) A process, say \( Q \), on receiving a marker message along a channel \( c \) takes the following steps:

\begin{verbatim}
if Q has not taken a checkpoint then
    begin
    Q sends a marker on each channel, incident on, and directed away from Q.
    Q takes a checkpoint.
    Q records the state of channel c as being empty.
    end
else Q records the state of channel c as the sequence of messages received along c, after Q had taken a checkpoint and before Q received the marker along c.
\end{verbatim}

4.1 Plank’s Staggering Scheme
Plank [18] suggested that the processes should send markers after taking their checkpoints, rather than before taking the checkpoint (unlike the algorithm above). This simple modification introduces some staggering of checkpoints. However, not all checkpoints can be staggered.

In our experiments, we use the Chandy-Lamport algorithm that incorporates Plank’s modification. In the rest of

2. Note that the Chandy-Lamport algorithm is applicable to any strongly connected graph. Our algorithm can also be generalized to strongly connected graphs.
this paper, this modified algorithm will be referred to as the Chandy-Lamport/Plank algorithm, or CL/P for brevity.

In CL/P, while the coordinator process takes a checkpoint, no other process takes a checkpoint because the coordinator does not send markers until its checkpoint is stored. However, when the coordinator sends the markers to other processes, those processes typically receive the markers at about the same time and, therefore, their checkpoints tend to overlap in time. Therefore, CL/P algorithm achieves only a limited degree of staggering when the number of processes exceeds 2.

Plank [18] observed that his staggering scheme works better than the original “nonstaggered” algorithm when 1) degree of synchronization among the processes is relatively small and 2) the message volume is relatively small (message volume is the amount of information communicated by messages). If degree of synchronization is high and message volume is large, then checkpoint staggering may actually result in worse performance, as also observed in [18]. The reasons behind these observations are summarized below.

The motivation behind checkpoint staggering is to reduce the contention for the stable storage, so that the checkpoint overhead is reduced. However, consider the case when a process Q is waiting to receive a message from another process P that is busy taking a checkpoint. Assume that checkpoint staggering ensures that process Q does not take a checkpoint while P is taking a checkpoint. Although this reduces stable storage contention, it does not improve performance because process Q must remain idle waiting for a message from P—process P will send the message only after completing its checkpoint. Thus, when processes frequently synchronize, the benefits of checkpoint staggering are offset by the overhead due to the idle time.

When checkpoints are staggered, the number of messages logged tends to be larger than that when the checkpoints are not staggered. Therefore, if message volume is large, the benefit of checkpoint staggering is offset by the overhead of additional message logging.

5 STAGGERED CONSISTENT CHECKPOINTING

We initially assume that all processors share a single stable storage. In Section 7, we consider the case when multiple stable storages are available. The proposed checkpoint staggering algorithm can stagger the checkpoints in any desired manner. Here, we assume that the objective is to stagger all checkpoints. Also, in the following, we only consider approach 1 for taking a logical checkpoint, as presented in Section 3. However, the proposed algorithm can be easily extended to use the other approaches as well. Note that Approach 1 can only be used with deterministic processes.

The proposed algorithm, named STAGGER, coordinates logical checkpoints rather than physical checkpoints. The STAGGER algorithm may be summarized as follows:

\[
\text{staggered physical checkpoints + consistent logical checkpoints = staggered consistent checkpoints}
\]

For the purpose of this discussion, assume that the checkpoint coordinator is named \( P_0 \) and other processes are named \( P_1 \) through \( P_{n-1} \), \( n \) being the number of processes. We now present the proposed algorithm (consisting of two phases), followed by an illustration in Fig. 3.

\textbf{Algorithm STAGGER}

1. \textit{Physical checkpointing phase:} Checkpoint coordinator \( P_0 \) takes a physical checkpoint and then sends a take_checkpoint message to process \( P_1 \).

   When a process \( P_i, i > 0 \), receives a take_checkpoint message, it takes a physical checkpoint and then sends a take_checkpoint message to process \( P_{j}, j = (i + 1) \mod n \).

   When process \( P_0 \) receives a take_checkpoint message from process \( P_{n-1} \), it initiates the second phase of the algorithm (named consistent logical checkpointing phase). After a process takes the physical checkpoint, it continues execution. Each message delivered to the process, after taking the physical checkpoint (but before the completion of the next phase), is logged in the stable storage.

   The above procedure ensures that physical checkpoints taken by the processes are staggered because only one process takes a physical checkpoint at any time. The physical checkpoints taken by the processes are not necessarily consistent.

   The physical checkpointing phase is illustrated in Fig. 3, assuming that the application consists of three processes.
2. **Consistent logical checkpointing phase**: This phase is very similar to the Chandy-Lamport algorithm. The difference between Chandy-Lamport algorithm and this phase is that when the original Chandy-Lamport algorithm requires a process to take a “checkpoint,” our processes take a logical checkpoint (not a physical checkpoint as in the Chandy-Lamport algorithm). A logical checkpoint is taken by ensuring that the messages delivered since the physical checkpoint (taken in the previous phase) are logged on stable storage. The exact algorithm for this phase is provided below:

**Initiation**: The coordinator \( P_0 \) initiates this phase on receipt of the `take_checkpoint` message from process \( P_n \). Process \( P_0 \) sends marker messages on each channel, incident on, and directed away from \( P_0 \). Also, \( P_0 \) takes a logical checkpoint by ensuring that all messages delivered to it since its physical checkpoint are logged. (The number of messages logged can be somewhat reduced, as discussed later.)

A process, say \( Q \), on receiving a marker message along a channel \( c \) takes the following steps:

```plaintext
if \( Q \) has not taken a logical checkpoint then
begin
  \( Q \) sends a marker on each channel, incident on, and directed away from \( Q \).
  \( Q \) takes a logical checkpoint by ensuring that all messages delivered to it (on any channel) after \( Q \)'s recent physical checkpoint have been logged.
end
else \( Q \) ensures that all messages received on channel \( c \) since its recent logical checkpoint are logged.
```

The above algorithm establishes a consistent recovery line consisting of one logical checkpoint per process. Similar to the Chandy-Lamport algorithm, messages sent by a process before its logical checkpoint but not received before the receiver’s logical checkpoint are logged as part of the channel state. Note that a message \( M \) that is logged to establish a logical checkpoint may be logged any time from the instant it is received until the time when the logical checkpoint is to be established. In our implementation, such messages are logged immediately on receipt.

In Fig. 3, when process \( P_0 \) receives `take_checkpoint` message from process \( P_2 \), it initiates the consistent logical checkpointing phase. Process \( P_0 \) sends marker messages to \( P_1 \) and \( P_2 \) and then takes a logical checkpoint by ensuring that messages \( M_0 \) and \( M_2 \) are logged on the stable storage. When process \( P_1 \) receives the marker message from process \( P_0 \), it sends markers to \( P_0 \) and \( P_2 \) and then takes a logical checkpoint by logging message \( M_1 \) on the stable storage. Similarly, process \( P_2 \) takes a logical checkpoint by logging message \( M_3 \) on the stable storage. Messages \( M_4 \) and \( M_5 \) are also logged during the second phase (as they represent the channel state). Message \( M_6 \) is not logged.

**Proof of correctness.** The correctness follows directly from the proof of correctness of the Chandy-Lamport algorithm [3].

**Recovery**: After a failure, each process rolls back to its recent physical checkpoint and reexecutes (using the logged messages) to restore the process state to the logical checkpoint that belongs to the most recent consistent recovery line.

Note that the above STAGGER algorithm was designed assuming that it is desirable to stagger all checkpoints. If some other pattern of staggering is more desirable, the above algorithm can be easily modified to achieve that pattern. Section 7 illustrates this with an example.

### 6 Performance Evaluation

We implemented the proposed algorithm STAGGER and the Chandy-Lamport/Plank (CL/P) algorithm on an nCube-2 multicomputer with a single disk (stable storage). In our implementation of CL/P and STAGGER, the markers sent by process 0 are sent asynchronously using interrupts (or signals)—sufficient care is taken to ensure that the markers appear in first-in-first-out (FIFO) order with respect to other messages, even though the markers are sent asynchronously. Markers sent by other processes are sent without using interrupts. If no markers are sent asynchronously, the checkpointing algorithm may not make progress in the cases where synchronization (or communication) is infrequent. As staggering is most beneficial...
under these circumstances, it is necessary to ensure that the algorithm progresses without any explicit communication by application processes. Therefore, process 0 sends asynchronous markers.

The first application used for evaluation of STAGGER is a synthetic program, named sync-loop, similar to a program used by Plank [18]. The pseudo-code for the program is presented below using a C-like syntax.

```c
sync-loop(int iter, int size, int M) {
    char state[size];
    repeat (iter) times {
        perform M floating-point multiplications;
        synchronize with all other processes;
    }
}
```

Process state size (and checkpoint size) is controlled by the size parameter. For the size chosen for our experiments, checkpoint size for each process of sync-loop is approximately 2.1 Mbyte. Synchronization is achieved by means of an all-to-all message exchange. By choosing a very large value for M, the frequency of synchronization in the program is reduced, and vice-versa.

Fig. 4 presents experimental results for STAGGER and CL/P schemes. Synchronization interval in this figure is the time between two consecutive synchronizations of the processes. The synchronization interval on the horizontal axis in Fig. 4 is obtained by dividing by iter the execution time of sync-loop without taking any checkpoints. Checkpoint overhead is obtained as:

\[
\text{execution time with } S \text{ consistent checkpoints} - \text{execution time without any checkpoints} \over S
\]

For our measurements, five checkpoints were taken per execution of the program (i.e., \(S = 5\)). Each instance of the sync-loop application was executed five times and checkpoint overhead was averaged over these five executions.

Fig. 4 presents overhead measurements for experiments on cubes of dimension \(d = 1, 2, 3,\) and \(4\). In Fig. 4, observe that, for a fixed dimension \(d\), as the synchronization interval becomes smaller, the checkpoint overhead grows for both schemes. For very small synchronization intervals, STAGGER does not perform much better than CL/P. However, when synchronization interval is large, the proposed scheme achieves significant improvements for \(d > 1\). For dimension \(d = 1\), the two schemes achieve essentially identical performance, since CL/P also staggers the checkpoints completely when the number of processes is 2.

Observe in Fig. 4 that, for a given instance of the application, when the dimension is increased, the overhead for STAGGER, as well as CL/P, increases. However, the increase in the overhead of CL/P is much greater than that of STAGGER.

The measurements presented above imply that when the parallel application has a large granularity (thus, requiring infrequent synchronization), the proposed STAGGER algorithm can perform well. As an example of an application with coarse-grain parallelism, Fig. 5 presents measurements for an embarrassingly parallel simulation program (SIM). State size for each process in SIM is approximately 34 Kbyte. The simulation program is completely parallelized and the processes synchronize only at the beginning and at the completion of the simulation. This synchronization pattern represents the best possible scenario for staggered checkpointing. As seen from Fig. 5, the checkpoint overhead for STAGGER remains constant independent of the dimension because synchronization is essentially nonexistent. In the absence of synchronization, each process can take a checkpoint without idling any other process or having to log any messages, therefore, the checkpoint overhead is independent of the dimension. On the other hand, the overhead for CL/P increases with the dimension because CL/P does not completely stagger the checkpoints (when number of processes is greater than 2) and contention for the stable storage increases when the number of processes is increased.

To be fair, we should note that STAGGER does not always outperform CL/P. An algorithm that staggers more would tend to perform poorly when degree of synchronization and message volume is large. To illustrate this, Fig. 6a presents measurements for a program named FFT-15 that
repeatedly evaluates fast Fourier transform of 215 data points and has frequent interaction between processes. Checkpoint size for each process is approximately 1.85 Mbyte. For this application, the overhead of STAGGER is larger than that of CL/P.

The performance of STAGGER can be improved by reducing the amount of information logged, using an optimization similar to that in [9]. Unlike in the original STAGGER algorithm, it is not necessary to log a message’s data content if it was sent by a process after taking its physical checkpoint—for such a message, it is sufficient to log its order information (i.e., send and receive sequence numbers and sender and receiver identifiers). During recovery, such a message is always reproduced by the sender process. Therefore, logging of order information is sufficient. Fig. 6b plots overhead of the STAGGER algorithm modified to implement the above optimization. The overhead of the modified algorithm is lower than the original STAGGER algorithm (see Fig. 6a), however, the overhead is still not much better than CL/P. Because the FFT-15 application performs frequent communication with large messages, it is hard to achieve overhead better than CL/P. In fact, in such cases, the basic Chandy-Lamport algorithm may perform as well as or better than CL/P and STAGGER both (in our work, we did not experimentally evaluate the basic Chandy-Lamport algorithm).

7 VARIATIONS ON THE THEME

7.1 Process Clustering to Exploit Multiple Stable Storages

The algorithm STAGGER presented above assumes that all processes share a single stable storage. However, in some systems, the processes may share multiple stable storages. For instance, the number of processes may be 16 and the number of stable storages may be 4. For such systems, we modify the proposed STAGGER algorithm to make use of all stable storages while minimizing contention for each stable storage. To achieve this we partition the processes into clusters, the number of clusters being identical to the number of stable storages. Each cluster is associated with a unique stable storage; processes within a cluster access only the associated stable storage [13].

The algorithm STAGGER, modified to use multiple stable storages, differs from the original STAGGER algorithm only in the first phase (i.e., staggered checkpointing phase). We illustrate the modified staggered checkpointing phase with an example. Consider a system consisting of six processes and two stable storages. The processes are divided into two clusters containing three processes each. Cluster $i$ ($i \neq 0; 1$) contains processes $P_{i0}$, $P_{i1}$, and $P_{i2}$. Process $P_{i0}$ in cluster $i$ is identified as the checkpoint coordinator for cluster $i$ and process $P_{00}$ is also identified as the global checkpoint coordinator. Fig. 7 depicts the first phase of the modified algorithm.

The global checkpoint coordinator $P_{00}$ initiates phase 1 of the algorithm (i.e., staggered physical checkpointing phase) by sending take_checkpoint messages to the checkpoint coordinator.
coordinators in all other clusters. Process $P_{00}$ then takes a physical checkpoint and sends a \textit{take_checkpoint} message to process $P_{01}$.

When a process $P_{ij}$, $ij \neq 00$, receives a \textit{take_checkpoint} message, it takes a physical checkpoint and sends a \textit{take_checkpoint} message to process $P_{km}$, where $m = (j + 1) \mod (\text{cluster size})$ and $k$ is equal to 0 if $m = 0$, and $i$ otherwise. When the global coordinator $P_{00}$ receives one \textit{take_checkpoint} message from a process in each cluster, it initiates the \textit{consistent logical checkpointing} phase (this phase is identical to the second phase of the original STAGGER algorithm).

Essentially, the above procedure guarantees that at most one process accesses each stable storage at any time during the first phase and that all stable storages are used for saving physical checkpoints. We expect that, similar to the measurements in the previous section, the STAGGER algorithm will reduce checkpoint overhead when used with multiple stable storages as well. Because the nCube-2 machine used in our experiments had access to only one disk, we are unable to provide experimental results for this case.

### 7.2 Asynchronous Markers

Arrival of an asynchronous marker is informed to the destination process by means of an interrupt (or signal). In spite of the asynchronous nature, a marker should appear in its appropriate position on the FIFO channel on which it is sent. We call a marker that is not sent with an interrupt a “synchronous” marker (for the lack of a better terminology). While an asynchronous marker can be processed as soon as it arrives, a synchronous marker may not be processed for a long time—particularly if the destination process does not need any messages on the corresponding channel.

Which markers (if any) are sent asynchronously can affect performance of STAGGER and CL/P algorithms. As noted previously, in our implementation, markers sent by process 0 are asynchronous, other markers are synchronous.

Plank [18] does not address the distinction between asynchronous and synchronous markers. One variation that can make CL/P imitate STAGGER, particularly for applications with infrequent synchronization (communication), is as follows: In CL/P algorithm, ensure that the marker sent by process $i$ to process $j$ is asynchronous if and only if $j = i + 1 \mod \text{(number of processes)}$. Thus, each process will take checkpoint, and the algorithm will make progress, even if the processes are not communicating with each other. Also, because each process sends only one asynchronous marker, the algorithm would tend to reduce contention for the stable storage.

With infrequent synchronization (communication), the above rule will tend to stagger checkpoints by different processes.

The above variation could also be used to reduce stable storage contention during the \textit{consistent logical checkpointing} phase of STAGGER algorithm.

### 8 Summary

This paper presents an algorithm for taking consistent logical checkpoints. The proposed algorithm can ensure that physical checkpoints taken by various processes are staggered to minimize contention in accessing the stable storage. Experimental results on nCube-2 suggest that the proposed scheme can improve performance as compared to an existing staggering technique, particularly when processes synchronize infrequently and message sizes are not very large. The paper also suggests a few variations of the proposed scheme, including an approach for staggering checkpoints when multiple stable storages are available.

Further work is needed to experimentally evaluate the process clustering approach when multiple stable storages are available. The impact of selectively sending asynchronous markers needs to be evaluated. The measurements presented in this paper were performed on nCube-2 multicomputer. Experimental evaluation of the proposed approach on a distributed system is also a topic for further work.

### Acknowledgments

We thank the referees for their detailed and helpful comments. We also thank James Plank and Yi-Min Wang.
for their comments on an earlier draft of this paper. The FFT program used for experiments was written by Akhilesh Kumar. Vidya Iyer wrote parts of the checkpointing layer used for experiments. This research is supported in part by U.S. National Science Foundation Grant MIP-9502563 and Texas Advanced Technology Program Grant 009741-052-C. This paper was presented in part at the IEEE Symposium on Parallel and Distributed Processing (SPDP), October 1996.

References


Nitin Vaidya received the MS and PhD degrees from the University of Massachusetts at Amherst in 1991 and 1992, respectively. He also received the ME degree from the Indian Institute of Science, Bangalore, in 1988 and the BE (Honors) degree from the Birla Institute of Technology and Science, Pilani, in 1986. He is currently an associate professor of computer science at Texas A&M University. His research interests include wireless networking and mobile computing, and fault-tolerant computing.

Dr. Vaidya is a recipient of the 1995 CAREER award from the U.S. National Science Foundation. He is a coauthor of the paper which received the 1998 Best Student Paper Award from MOBICOM ’98. He is presently a speaker for the Distinguished Visitor Program of the IEEE Computer Society. Dr. Vaidya has served on the program committees of several conferences and workshops, including the 1999 ACM Symposium on Principles of Distributed Computing (PODC), the 1999 International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communication (DIAL-M), and the 1999 IEEE International Symposium on Fault-Tolerant Computing (FTCS). He has served as the editor of the newsletter for IEEE Technical Committee on Fault-Tolerant Computing and as a guest editor for a special issue of Computer magazine on fault-tolerant computing. Dr. Vaidya is a member of the ACM and the IEEE.