A Checkpoint Protocol for an Entry Consistent Shared Memory System

Nuno Neves, Miguel Castro and Paulo Guedes

IST - INESC
R. Alves Redol 9, 1000 Lisboa PORTUGAL
email: (nuno, miguel, pjg)@inesc.pt

ABSTRACT

Workstation clusters are becoming an interesting alternative to dedicated multiprocessors. In this environment, the probability of a failure, during an application's execution, increases with the execution time and the number of workstations used. If no provision is made for handling failures, it is unlikely that long running applications will terminate successfully. One solution to this problem is process checkpointing.

This paper presents a checkpoint protocol for a multi-threaded distributed shared memory system based on the entry consistency memory model. The protocol allows transparent recovery from single node failures and, in some cases, from multiple node failures. A simple mechanism is used to determine if the system can be brought to a consistent state in the event of multiple machine crashes.

The protocol keeps a distributed log of shared data accesses in the volatile memory of the processes, taking advantage of the independent failure characteristics of workstation clusters. Periodically, or whenever the log reaches a high-water mark, each process checkpoints its state, independently from the others. The protocol needs no extra messages during the failure-free period, since all checkpoint control information is piggybacked on the memory coherence protocol messages.

1 INTRODUCTION

Workstation clusters are becoming an interesting alternative to dedicated multiprocessors, due to the improvements in network and processor technology [5,6,25]. Parallel applications based on the shared memory programming paradigm can be supported by software running on the workstation cluster. This software implements a distributed shared memory (DSM) system. In these systems the application's threads of execution see a single shared memory space, which is kept coherent according to a given memory model. The first DSM systems used strict models [10,16] resulting in poor performance. More recent systems [1,3,8] introduced relaxed consistency memory models, which are capable of delivering higher performance than strict models.

In workstation clusters, mean time to failure decreases with increasing number of nodes. Hence, the probability of a failure during the execution of a long running application can become intolerably high. One solution to this problem is process checkpointing. This paper presents a checkpoint protocol for DiSOM [11], a multi-threaded DSM system based on the entry consistency memory model [3]. The protocol allows transparent recovery from single node failures and, in some cases, from multiple node failures. It uses a combination of a distributed log of shared data accesses and uncoordinated checkpointing.

The logging mechanism is tightly integrated with the entry consistency memory coherence protocol. In the entry consistency memory model, shared data objects are associated with synchronization objects and all accesses to a shared data object must be enclosed between an acquire and a release on its associated synchronization object. Synchronization objects enforce a concurrent read exclusive write synchronization policy. Therefore, the entry consistency constraints ensure that a thread only observes updates to a shared data object when it acquires the associated synchronization object. These constraints also ensure that updates, performed by a thread, only become visible after the release.

The logging mechanism creates a log entry with a copy of the object state, whenever a release is issued by a thread that acquired the object for writing. The log entry is stored in the volatile memory of the process where the thread is executing. If a thread in another process subsequently acquires the object and later that process fails, the object version can be recovered from the process where the log entry was stored. This is always possible if we assume a single process per-node, a single node failure and that nodes fail independently. This assumption is usually verified in workstation clusters.

The logging overhead is minimal because the log is

A version of this paper appeared in the 13th ACM Symposium on Principles of Distributed Computing, August 1994.

stored in volatile memory and all checkpoint control information is piggybacked on the memory coherence protocol messages, avoiding extra messages during the failure-free period. Contrarily to the sequential consistency memory model [16], where each update operation must potentially be logged, our protocol takes advantage of the entry consistency constraints to avoid logging individual updates.

The checkpoint mechanism saves the state of each process on disk, periodically or when the process' log reaches a maximum size. We use an uncoordinated protocol [4,14,23] where each process checkpoints itself, independently of the others. This way we avoid the overhead of checkpoint coordination and the loss of process autonomy characteristic of coordinated schemes [7,15,20]. In uncoordinated checkpoint protocols the domino effect [21] must be prevented to avoid a cascading of roll backs. Our checkpoint protocol prevents the domino effect by using the distributed log. Furthermore, the protocol is pessimistic [4,14], i.e. no thread in a surviving process has to be rolled back if a failure occurs. Nevertheless, it keeps the efficiency of optimistic schemes [23], because logs are kept in the volatile memory of other processes.

The recovery mechanism starts by loading the failed process' last checkpoint in a free processor. Then, the recovering process obtains, from the other processes, all the object versions acquired by its threads between the checkpoint and the failure. Next, the recovering process threads re-execute, acquiring the same versions of the same objects as they did before the failure. This assumes that threads execute in a piece-wise deterministic manner [9]. After recovery, in the event of a single process failure, the system will be in a consistent state. On the other hand, if multiple process failures occurred it might be impossible to recover the system to a consistent state. The recovery mechanism detects this situation and aborts the application.

The paper is organized as follows. The next section presents related work. The system model is described in section 3. In section 4 the checkpoint protocol and the distributed shared data log are explained. We draw our conclusions in section 5. In appendix A we present proof sketches of the theorems mentioned in the paper.

2 RELATED WORK

One way to tolerate faults in a distributed system is to use process checkpointing. In coordinated checkpoint schemes [7,15,20], processes coordinate to ensure that the set of process checkpoints represents a consistent state of the system. These systems tolerate multiple failures at the expense of checkpoint coordination. In uncoordinated checkpoint schemes [4,14,23], processes take checkpoints independently. After a failure a consistent state is reached by recovering the failed processes' checkpoints and possibly by rolling back some surviving processes. Logging can be used to prevent roll back propagation.

In a message passing system, message logging can be made in the receiver [23] or in the sender [14]. Our shared memory abstraction is implemented using messages, therefore we could use a message logging protocol to achieve fault tolerance. This solution would perform worse than our protocol because our protocol takes advantage of the memory model constraints to avoid logging all the information in all the messages.

Most of the previous work in checkpoint protocols for DSM systems on workstation clusters was based on the sequential consistency memory model [12,24]. The protocols proposed by Stumm and Zhou [24] only offer a partial solution to the process recovery problem, since only the state of shared pages is recovered. In their read-replication algorithm a process sends a copy of the dirty pages on every message send. Richard and Singhal [12] logged all the pages acquired in the volatile memory of the acquirer and saved the log in stable storage whenever a modified paged was transferred to another process. The only reference we found about checkpointing in relaxed consistent DSM was by Janssens and Fuchs [13]. In their protocol a process is checkpointed exactly before its updates become visible to the other processes. They used this characteristic to reduce the number of checkpoints needed for recovery. They presented results from trace-driven simulations showing a five- to ten-fold decrease in checkpoint overhead over sequential consistency based techniques.

Our checkpoint protocol is also based in a relaxed consistency memory model but it uses the releaser's volatile memory to log updates to shared data objects. The log is used to avoid roll back propagation. The checkpoint frequency is independent of the application's actions. Therefore, it can be chosen based only on recovery time constraints. Unlike most checkpoint protocols ours supports multiple-threads per process. Although the protocol only guarantees recovery from single failures, a simple detection mechanism is used to determine if the system can be brought to a consistent state, in the event of multiple failures.

3 SYSTEM MODEL

DiSOM is composed of a set of processes, one on each workstation in the cluster. Processes communicate only by message passing. Messages are delivered reliably and in FIFO order. Each process is viewed as a collection of resources, which provides an execution environment for multiple threads. These resources include an address space, where a subset of the shared objects is mapped.

Threads run in a piece-wise deterministic manner [9], i.e. the execution of a thread is divided into deterministic intervals started by nondeterministic events. A new interval starts when a thread acquires an object for reading or writing and ends at the next acquire. If the piece-wise determinism assumption holds, a process can be recovered by restoring its state from a checkpoint, letting its threads re-execute and ensuring that they re-acquire the same versions of the same objects. Each thread maintains a counter called *logical time*, lt, which is incremented whenever an acquire is issued, the acquire's logical time is the incremented value. The system assigns a unique identifier, tid, to each thread. The tid is composed of the process identifier and a local thread identifier. Therefore, the process identifier can be obtained from the tid. The tuple $<tid, lt>$ identifies a unique *execution point*, ep, in the system, i.e. it identifies a thread and a logical instant in its execution. We say that an operation was executed at $ep = <tid, lt>$ if tid executed the operation at lt. We also define the relations $\prec$ and $\preceq$ as follows : $ep_i \prec ep_j$ iff $tid_i = tid_j \wedge lt_i < lt_j$ and $ep_i \preceq ep_j$ iff $tid_i = tid_j \wedge lt_i \leq lt_j$.

We consider a fail-stop model [22], where a processor fails by halting and all surviving processors detect the node failure within bounded time. Network partitions are not tolerated. We do not rely on special hardware support, e.g. non-volatile RAM [2,17] or uninterruptible power supply [19].

3.1 MEMORY COHERENCE MODEL

A memory coherence protocol is best described as a contract between the system and the application program. If the program satisfies the contract's requirements the system guarantees that the program views a sequentially consistent memory [16], which is the model expected by most programmers. DiSOM uses the entry consistency memory coherence protocol introduced by Midway [3]. The contract imposed by entry consistency on the program is as follows. Firstly, the program has to identify the relationships between synchronization objects and data objects and explicitly associate them. Synchronization objects enforce concurrent read exclusive write synchronization and data objects are arbitrarily sized language-level data items. Secondly, all write accesses to a shared object must be enclosed between an `acquire-write` operation and a `release-write` operation, on the synchronization object associated with the data object. The updates only become visible in another process when a thread in that process subsequently acquires the object. A thread acquires an object when it executes either an `acquire-write` or `acquire-read` operation. Acquires are synchronous, i.e. a thread is blocked until the acquire completes. Thirdly, all read accesses must be enclosed between either an `acquire-read` and a `release-read` or an `acquire-write` and a `release-write`. The program should use `acquire-read` and `release-read` pairs to maximize concurrency and thus performance.

The evolution of an object state can be seen as a sequence of versions $\mathcal{V} = V_0, V_1, ...$ The initial version, V_0, is the object's state at creation time. When a thread acquires an object for writing, a copy of the last object version is created. All updates performed by the thread, prior to the release, are reflected in this copy. A new version, based on the copy, is produced when the thread releases the object. The thread is called the *producer* of the version. If an object is acquired for reading and later released, no version is produced. The last version of an object is the last element in the sequence.

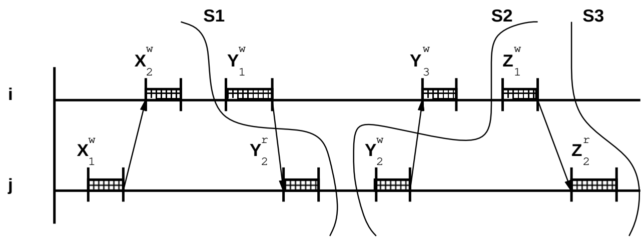

Figure 1: Three system states *S1*, *S2* and *S3*. System states *S1* and *S2* are inconsistent and *S3* is consistent.

A suffix of the sequence of versions may be lost due to a failure. A system state is *consistent* if all threads, holding objects, hold the last versions of those objects and no thread has acquired a version of an object that was lost due to a failure.

During the failure free period, the memory coherence protocol keeps shared memory in a consistent state. After a failure the system must be brought back to a consistent state. Figure 1 depicts the execution of two threads in different processes. The notation O_v^t is used to represent an acquire of the version v of object O. The type t of the acquire is either read, R, or write, W. *S1*, *S2* and *S3* are system states. System state *S3* is the only consistent state. *S1* is inconsistent because the acquire Y_2^r is included in the system state and the previous acquire Y_1^w is not. Similarly, *S2* is inconsistent.

4 THE CHECKPOINT PROTOCOL

The next section describes the data structures used by the protocol. The use of these structures is explained in sections 4.2 and 4.3. These sections describe the failure-free behavior and the recovery procedure. The garbage collection procedure applied to the protocol data structures is presented in section 4.4. The last section explains the detection of inconsistencies in the event of multiple failures.

4.1 DATA STRUCTURES

```
objId;          // object identifier
version;        // version number
probOwner;      // probable owner
status;         // object status
copySet;        // list of readers
epDep;          // ep dependency
```

Figure 2: Object data structure.

DiSOM uses a modified version of Li's dynamic distributed manager protocol [18] to keep the shared memory coherent and implement synchronization[1]. The proto-

[1] DiSOM uses distributed copy sets. In this paper, to clarify the description, we present a simplified version of the algorithm with centralized copy sets.

col maintains an instance of the data structure presented in figure 2 for each shared object. The first field, $objId$, is a system wide unique identifier. The $version$ field contains the version number of the local copy of the object. The protocol associates with each object the notion of ownership. The $owner$ of an object is the process that keeps the last version of the object, i.e. the last process from which a thread has acquired the object for writing. The field $probOwner$ is a hint to the identity of the object's owner. The system guarantees that the true owner of an object can be reached by following the distributed $probOwner$ chain. The $status$ field indicates how the object is being used, e.g. held for writing, reading or not held, and the accesses permitted on the object's local copy. Finally, the $copySet$ is a set whose elements are the identifiers of all the processes with a readable copy of the object. Whenever an object is acquired for reading by a thread, a read-only copy is kept in the process where the thread is executing. Threads in that process can successively re-acquire the object for reading, without interacting with any other process, until a writer invalidates the copy. When an object is acquired for writing, the new writer uses the $copySet$ to invalidate the read-only copies kept in the other processes.

We call an acquire $local$ when an object is re-acquired by a thread in the same process. This local acquire can occur when the process has an up-to-date version of the object, i.e. the process is the owner or has a read-only copy of the object. The order of local acquires prior to a failure must be reproduced during recovery. We use the field $epDep$ to achieve this goal. It records an execution point.

```
tid;         // identifier
lt;          // logical time
waitObj;     // requested object
depSet;      // dependencies
```

Figure 3: Thread data structure.

An instance of the data structure in figure 3 is kept by each thread. The field tid uniquely identifies the thread and lt is the thread's logical time. The field $waitObj$ is used by the protocol during process recovery to re-issue object acquires that could have been lost due to the failure. It has the form $<objId, type>$. If not null, it means that the thread has issued an acquire request of $type$ (R/W) for object $objId$, which has not completed. The set $depSet$ is used to record a description of the thread's dependencies. An acquiring thread, tid_{acq}, *depends* on a producer thread, tid_{prd}, if tid_{acq} acquired an object version produced by tid_{prd}. The $depSet$ has an entry per each acquire performed by the thread in the past. Each entry has the form $<objId, type, ep_{acq}, ep_{prd}, P>$. This tuple should be read as follows: a version of $objId$ was acquired for $type$ (R/W) when the acquiring thread's execution point was ep_{acq} and the producer thread's execution point was ep_{prd} and it is logged in process P. The P field usually stores the process identifier of the process where the producer thread was running. The exception occurs with local acquires as will be explained in the next section.

```
objId;         // object identifier
version;       // version number
objData;       // object data
tidPrd;        // producer
nextOwner;     // next owner process
threadSet;     // threads which accessed the object
```

Figure 4: Log entry data structure.

Each process maintains a list of log entries, see figure 4. A log entry is created when a `release-write` is issued. It records the object's identifier, its version number, its data, $tidPrd$, $nextOwner$ and the $threadSet$. The field $tidPrd$ contains the identifier of the thread that produced the logged object version. The $nextOwner$ field is initially null. When this object version is acquired for writing, it is filled with the identifier of the new owner process. The $threadSet$ elements are pairs of execution points, which represent acquires of the logged object version. In each pair, $<ep_{acq}, ep_{prd}>$, ep_{acq} is the execution point of the acquire and ep_{prd} is the producer thread's execution point when the acquire request was satisfied.

```
objId;         // object identifier
epAcq;         // acquire execution point
localDep;      // local dependency
Plog;          // process where it is logged
```

Figure 5: Dummy log entry data structure.

A dummy log entry, see figure 5, is created whenever a local acquire is issued. The entries are used to describe local acquires in order to reproduce them during recovery. The dummy entry cannot be stored in the local process' volatile memory because if the process fails it will be lost. Therefore the entry is sent to another process. Each process maintains a list of dummy log entries received from other processes. Each entry records the object's identifier and the execution point of the local acquire, $epAcq$, that produced the dummy log entry. The field $localDep$ records the execution point of the local event that precedes this acquire. The $Plog$ field is only used during recovery.

We call the log entries described in figure 4 *regular* log entries, to distinguish them from the dummy log entries.

4.2 FAILURE-FREE BEHAVIOR

During the failure-free period, threads execute their programs and acquire objects for reading or writing. The memory coherence protocol keeps shared objects consistent. The checkpoint protocol logs data necessary to recovery and piggybacks its control information on the memory coherence protocol messages. This section describes the checkpoint protocol and a simplified version of the memory coherence protocol[2].

Whenever a non-local acquire is issued the following steps are executed:

1. When a thread tries to acquire an object, a request is sent to the object's $probOwner$ with the form ([$objId$, $type$, P_{acq}], [ep_{acq}]). This request carries the type of the acquire, $type$, the local process identifier, P_{acq}, and the execution point of the acquire, ep_{acq}. Then the thread's $waitObj$ is filled with $objId$ and $type$. Next, the requester thread blocks waiting for the reply.

2. The request is forwarded along the $probOwner$ chain, until it reaches the owner process. When the owner process receives the request, if it is a read request and the object is currently held for writing, the request is queued. Similarly, a write request is queued if the object is held for reading or writing. When the request is allowed to proceed, the pair $<ep_{acq}, ep_{prd}>$ is added to the $threadSet$, in the object's last version in the log. The ep_{prd} is the current execution point of the producer thread. Then, a response with the form ([$objData$, P_{prd}], [ep_{prd}, $version$]) is returned. The response carries the identifier of the owner process, P_{prd}, and the object version obtained from object structure, $version$. Based on the type of the request, the owner also does the following updates to the system data structures:

   (a) Read request: P_{acq} is added to the $copySet$.

   (b) Write request: The ownership is moved to the new writer, along with the $copySet$, and the $probOwner$ field is updated to point to the new writer. The field $nextOwner$ in the log is made equal to P_{acq}.

3. The requesting thread receives the response. If it is the reply to a write request, it sends a message to all the processes in the $copySet$ invalidating their read copies. In both cases, the tuple $<objId, type, ep_{acq}, ep_{prd}, P_{prd}>$ is added to the thread's $depSet$, revealing the dependency from ep_{acq} to ep_{prd}, the $version$ in the object structure is set to the value of $version$ in the reply and $waitObject$ is nullified. The $epDep$ in the object is made equal to ep_{acq} and then the thread is allowed to resume execution.

4. The thread eventually releases the object. If the local process is the owner the $epDep$ field in the object structure is updated with the execution point of the release. If the object was acquired for writing, the version number in the object structure is incremented and an entry for this version is created in the log. This entry records the object's identifier, $objId$, the object version, $version$, the object's data associated with the version, $objData$, and the identifier of the producer thread, $tidPrd$.

[2] Throughout the description messages are represented by the tuple ([α], [β]) , where α and β are the information sent by the memory coherence protocol and the checkpoint protocol, respectively. Symbols with the "acq" subscript are related to the acquiring thread and symbols with the "prd" subscript are related to the thread that produced the object version.

The procedure outlined above assumes that the releasing thread and the acquiring thread execute in different processes, that fail independently. The assumption fails whenever there is a local acquire. Whenever a local acquire is issued the following steps are executed:

1. When a thread issues a local acquire a dummy log entry is created. The fields in the dummy entry are used in the following way: the identifier of the object is stored in the $objId$ field, the execution point of the acquire is saved in $epAcq$ and the value of $epDep$ from the object structure is recorded in the $localDep$ field. Additionally, the thread inserts a new dependency, $<objId, type, ep_{acq}, ep_{prd}, P>$, in the $depSet$. The value of the entry's ep_{prd} is set to the value $epDep$.

2. The release step is identical to the non-local acquire release step.

3. The dummy entry is kept locally until a message is sent by the memory coherence protocol. Then, the entry is sent with the message, to ensure that it will not be lost if the local process fails. The receiver process, P_i, saves the entry in its log and sets the entry's $Plog$ field equal to its identifier. The local process deletes the entry and stores P_i in the P field of the dummy dependency.

From time to time, each process checkpoints itself in an asynchronous way, independently from the others. The checkpoint is stored in stable storage. The size of the object log and the elapsed time since the last checkpoint are used to determine the moment to take the checkpoint. The checkpoint includes each thread's stack and machine state, the shared data and all system data structures (e.g. the log and per-thread data structures).

4.3 FAILURE RECOVERY

When a machine crash is detected, the process running on that node has to be recovered. Process recovery restores the shared objects and the system data structures to a state consistent with the rest of the system. When recovery is completed the system is able to recover from subsequent faults on any process. In the appendix we prove that the checkpoint protocol brings the system to a consistent state after a single process failure.

The first step to recover a process is to get its most recent checkpoint and reload it in a free processor. Then, the process state is reconstructed from the checkpoint in two steps. In the first step, the surviving processes collect all data necessary to recover the failed process. The entries kept in the log and dummy log are used to restore the state of the data objects in the failed process. The data contained in the dependencies is used to recover the log. In the second step, the failed threads re-execute their programs, acquiring the same versions of the objects as they originally did, until the recovery is completed.

4.3.1 DATA COLLECTION

The failed process, P_{ckp}, logically broadcasts a message asking for all information related to its threads. The request message contains the set $CkpSet$, whose elements are the execution points of P_{ckp}'s threads at checkpoint time. When a surviving process receives the message it executes the following steps, creating the sets $LogSet$, $DependSet$ and $DummySet$:

1. Determine the object versions acquired by the recovering threads which were produced locally. The log entries are inspected and the ones with $ep_{ckp} \prec ep_{acq}$ for each $ep_{ckp} \in CkpSet$ and each $<ep_{acq}, ep_{prd}> \in threadSet$, are added to the $LogSet$.
2. Determine which dummy log entries, created in the failed process, are stored locally. The dummy log entries are inspected and the ones with $ep_{ckp} \prec epAcq$ for each $ep_{ckp} \in CkpSet$ are added to the $LogSet$.
3. Determine which objects produced in the failed process were acquired by the local threads. The $depSet$ of each local thread is examined, and all entries with $ep_{ckp} \preceq ep_{prd}$ for each $ep_{ckp} \in CkpSet$, are added to the $DependSet$.
4. Determine if any dummy logged object, created locally, was saved in the failed process. The $depSet$ of each thread is examined and entries with ep_{prd} from a local thread and $P = P_{ckp}$ are added to the $DummySet$.
5. Pending acquire requests may have been lost due to the failure. The $waitObj$ of each blocked thread, in the surviving process, is used to re-issue pending acquire requests. Duplicate requests are detected and discarded by the memory coherence protocol.

The various sets are then sent to the recovering process and the surviving process has completed its contribution to the recovery.

4.3.2 LOG REPLAY

The recovering process collects all responses, merges the various $DummySet$ into a single $DummySet$ and organizes the data from the $LogSet$ and $DependSet$ in a group of lists. There are two lists per thread, the $LogList$ and the $DependList$, one for each type of set. Additionally the recovering process creates the $InvalidSet$, which is initially empty.

Each thread's $LogList$ contains regular log entries and dummy log entries. These entries correspond to acquires performed by the thread. The recovering process creates the $LogList$ of thread tid using the elements from the various $LogSet$ in the following way:

- If the element is a regular log entry and the value of ep_{acq} in one of the pairs in the element's $threadSet$ is an execution point of thread tid, then the element is inserted in tid's $LogList$.
- If the element is a dummy log entry and the value of the element's $epAcq$ field is an execution point of tid, then the element is inserted in tid's $LogList$.

The $LogList$ is ordered by ascending execution points, i.e. the regular entries are ordered by the value of ep_{acq} and dummy entries are ordered by the value of $epAcq$.

The $DependList$ contains the description of the acquires, performed by surviving threads, of object versions produced by the thread. The $DependList$ of thread tid contains the dependency entries from the $DependSet$ where ep_{prd} is an execution point of thread tid. The list is ordered by ascending values of ep_{prd}.

Recovering threads begin to re-execute when all data is organized. During the log replay, the acquires issued by the recovering threads are trapped by the system. Instead of the usual acquire algorithm the thread obtains the object versions locally from the $LogList$, without exchanging any message with the other processes. Since the $LogList$ is ordered by ascending execution points, the object versions are arranged in the order by which they were originally acquired.

The shared objects state and the thread's $depSet$ are recovered by executing the following sequence of steps whenever a thread issues an acquire:

1. It removes the first element from its $LogList$. If this element is a regular log entry, the thread waits until all the previous versions of the object are acquired by the other threads. When the thread is allowed to proceed it updates the object state with the logged object data. Otherwise, if this element is a dummy log entry, the thread waits until the event with execution point $localDep$ is reproduced. Note that during recovery no dummy entries are created.
2. The thread adds a new dependency, $<objId, type, ep_{acq}, ep_{prd}, P>$, to its $depSet$. In the case of a regular log entry the value of ep_{prd} is retrieved from the entry's $threadSet$ and P is the identifier of the process where the version was produced. Otherwise, ep_{prd} and P are set to the values of $localDep$ and $Plog$ from the dummy entry.
3. In order to recover the $probOwner$ and $status$ field in the object structure, every time an object is acquired, if the $nextOwner$ field in the regular log entry is not null, the pair $<objId, nextOwner>$ is added to the

$InvalidSet$. On the other hand, if $nextOwner$ is null, the pair is removed from the $InvalidSet$.

The protocol recovers log entries as follows. When a thread issues a `release-write` it creates a regular log entry. The entry's $threadSet$ is recovered using the elements from the $DependList$. The $nextOwner$ field in the log entry is left null, unless there is an element in $DependList$ representing an acquire for writing. In that case the element's ep_{acq} is used to update the $nextOwner$ field. If there is no such element, then the process was still the object owner when the failure occured. In this case, the object's $copySet$ is recovered using the $threadSet$.

When all thread lists are empty, the $probOwner$ and $status$ fields in each object structure are recovered. For each pair in $InvalidSet$ the local object version is invalidated, i.e. $status$ is set to no-access, and the $probOwner$ is set to the value of $nextOwner$ in the pair.

The dummy log entries that were kept in the failed process are recovered using the elements in the $DummySet$. For each element in the $DummySet$, $<objId, type, ep_{acq}, ep_{prd}, P>$, a new dummy log entry is added to the process' list of dummy log entries. The dummy entry has the fields $objId$, $epAcq$, $localDep$ and $Plog$ equal to the fields $objId$, ep_{acq}, ep_{prd} and P from the $DummySet$ element. Recovery is finally complete and requests received during recovery, which were blocked, are replied to and the normal execution mode is resumed.

The duration of the recovery period grows proportionally to the elapsed time, starting at the last checkpoint and ending at the failure. In an environment where failures are rare, checkpoints can be made less frequently, allowing a more efficient behavior during the failure-free period. Nevertheless, the protocol tries to reduce interference between the surviving processes and the recovering process. Surviving threads do not have to roll back and after sending the information needed for recovery, they only have to wait for the recovering threads, if they need an object which is being reconstructed.

4.4 GARBAGE COLLECTION

The log, the dummy log and the $depSet$ data structures must be garbage collected to avoid memory exhaustion.

We call an object version *old* if it is not the last object version and we call the corresponding log entry an *old* log entry. Old log entries which were not acquired by remote threads, are not needed for the recovery of any process. After a process, P_{ckp}, checkpoints its state it deletes these entries, i.e. old entries with an empty $threadSet$.

The object versions logged in the other processes which were accessed by P_{ckp}'s threads prior to the checkpoint are no longer needed for its recovery. After the checkpoint, P_{ckp} creates a set, $CkpSet$, with the execution points of its threads at checkpoint time. Next, it logically broadcasts the $CkpSet$ to the other processes. The receiver processes traverse their logs and remove pairs, $<ep_{acq}, ep_{prd}>$, from the various $threadSet$ according to the condition $ep_{acq} \prec ep_{ckp}$ for each $ep_{ckp} \in CkpSet$. An empty $threadSet$ means that the logged object version is no longer needed for the recovery of any thread and, if it is an old object version, it can be deleted.

The dummy log entries created by P_{ckp} before the checkpoint can be discarded. Those entries which are still stored in P_{ckp} are simply deleted. An entry which is stored in another process is deleted when that process receives the message with the $CkpSet$, if it meets the condition $epAcq \prec ep_{ckp}$ for each $ep_{ckp} \in CkpSet$.

After a process checkpoint, dependencies in preceding intervals are no longer needed to recover that process' log. Therefore, the $depSet$ entries can be garbage collected in the same way as the log entries. When a process receives a message with a $CkpSet$, it traverses the $depSet$ of its threads and removes entries with $ep_{prd} \prec ep_{ckp}$, for any $ep_{ckp} \in CkpSet$.

4.5 MULTIPLE FAILURES

The protocol ensures that the system is brought to a consistent state after a single node failure. However, if more than one processor fails, it might be impossible to restore the system to a consistent state. The protocol detects these situations with a conservative mechanism. The mechanism detects all situations that can lead to an inconsistent state but in some circumstances it can be pessimistic. In the appendix we prove that in the event of multiple failures, either the system is brought to a consistent state or the application is aborted.

The detection is accomplished after the receipt of the responses from all the processes, including the recovering ones (each process knows the identifiers of all the other processes involved in the computation). A recovering process replies as soon as its checkpoint is loaded. The detection mechanism traverses the per-thread $LogList$ searching for a maximum length prefix, that includes an element for each logical time since the logical time at checkpoint. If a logged object version was lost due to a failure the prefix will be a proper prefix and the rest of the list is discarded. It might be impossible to recover the system to a consistent state when there is an element in the thread's $DependList$ with a logical time larger than the logical time of the last element in the prefix. This situation occurs when a process has acquired a version of an object that might not be recovered. In this situation the application is aborted. Otherwise, recovery proceeds as described in the previous section.

5 CONCLUSIONS

We presented a checkpoint protocol for parallel applications, running in a workstation cluster. Applications are composed of multiple threads of execution that communicate by sharing memory. Shared memory is kept coherent according to the entry consistency memory model. The protocol allows transparent recovery from single node failures and, in some cases, from multiple node failures. In the latter case, the protocol guarantees that the application is either brought to a consistent state or an inconsistency is detected and the application aborted.

The protocol's main design goal is an efficient behavior during the failure-free period. It takes advantage of the independent failure characteristics of workstation clusters to log shared memory accesses in the volatile memory of the cooperating processes. Each process checkpoints its state periodically or when the log reaches a maximum size. The use of distributed logs permits a choice of checkpoint frequency independent of the application's actions, considering only recovery time constraints. Hence, in a cluster where failures are rare, checkpoints can be made less frequently. The checkpoint protocol is tightly integrated with the entry consistency memory coherence protocol. It uses the constraints imposed by the memory model to reduce the number of shared data accesses that need to be logged. Furthermore, no extra messages are necessary during the failure-free period, since all checkpoint control information is piggybacked on the memory coherence protocol messages.

Currently we are studying ways to generalize the protocol, by applying it to other relaxed consistency memory models. We are also exploring ways to integrate other forms of synchronization with the checkpoint protocol.

ACKNOWLEDGEMENTS

Special thanks go to Ellen Siegel for her careful reading of this manuscript and for her many suggestions that led to its improvement. We would like to thank Paulo Verissimo, David Matos, Luis Rodrigues and the anonymous referees for their comments on early versions of this manuscript. We would also like to thank to Paulo Meneses for his work on DiSOM's compiler.

REFERENCES

[1] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[2] Mary Baker and Marl Sullivan. The Recovery Box: Using fast recovery to provide high availability in the UNIX environment. In *Proceedings of the Summer 1992 USENIX Conference*, pages 31–44, June 1992.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. In *Proceedings of the 93 COMPCON Conference*, pages 528–537, February 1993.

[4] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 90–99, October 1983.

[5] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th Symposium on Operating System P rinciples*, pages 152–164, October 1991.

[6] Miguel Castro, Nuno Neves, Pedro Trancoso, and Pedro Sousa. MIKE: A distributed object-oriented programming platform on top of the Mach micro-kernel. In *Proceedings of the USENIX Mach Conference*, pages 253–273, April 1993.

[7] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, pages 3(1):63–75, February 1985.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 16th Annual Symposium on Computer Architecture*, pages 15–26, May 1989.

[9] A. Goldberg, A. Gopal, K. Li, R. Strom, and D. Bacon. Transparent recovery of Mach applications. In *Proceedings of the Usenix Mach Workshop*, pages 169–184, July 1990.

[10] J. Goodman and P. Woest. The Wisconsin Multicube: A new large-scale cache coherent multiprocessor. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 422–431, June 1988.

[11] Paulo Guedes and Miguel Castro. Distributed shared object memory. In *Proceedings of the 4th Workshop on Workstation Operating Systems*, pages 142–149, October 1993.

[12] Golden G. Richard III and Mukesh Singhal. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 86–95, Princeton, New Jersey, October 1993.

[13] Bob Janssens and W. Kent Fuchs. Relaxing consistency in recoverable distributed shared memory. In *The Twenty-Third Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 155–163, June 1993.

[14] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 14–19, July 1987.

[15] R. Koo and S. Toueg. Checkpointing and roolback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[16] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[17] Eliezer Levy and Avi Silberschatz. Incremental recovery in main memory database systems. Technical Report 01, Dept. of Computer Science, University of Texas at Austin, January 1992.

[18] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 229–239, August 1986.

[19] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 226–238, October 1991.

[20] J. S. Plank. *Efficient checkpointing on MIMD architectures*. PhD thesis, Princeton University, June 1993.

[21] B. Randel. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[22] R.D. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[23] Robert E. Strom and Shaula A. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[24] Michael Stumm and Songnian Zhou. Fault tolerant distributed shared memory algorithms. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 719–724, December 1990.

[25] V. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4), October 1991.

A THEOREM PROOFS

Theorem 1 *The checkpoint protocol brings the system to a consistent state after a single process failure.*

Proof sketch: The system is in a consistent state, after recovery, if all object versions acquired by surviving threads are recovered by the protocol. Consider object version V_n that was acquired by thread tid_{sur}, in a surviving process, and created by thread tid_{rec}, in the recovering process. If tid_{rec} created V_n, then it acquired V_{n-1} for writing. The version V_{n-1} was acquired (1) locally in the failed process or (2) from a remote process. In either case, a log of this acquire is left in a surviving process because we are assuming that only one process fails. In case (1) a dummy log entry is created in the first process that interacted with the failed process, through the memory coherence protocol. There is always one such process, because at least the process running tid_{sur} interacted with the failed process. Case (2) corresponds to the usual log creation.

During recovery, thread tid_{rec} will try to re-acquire version V_{n-1} of the object and then create version V_n. In case (1), the thread may have to wait until V_{n-1} is recreated by the other recovering threads. In case (2), it simply uses the version V_{n-1}, sent by one of the surviving processes. Therefore, version V_n is recovered by the protocol.

□

Theorem 2 *In the event of multiple failures, either the system is brought to a consistent state or the application is aborted.*

Proof sketch: The proof that the system can be brought to a consistent state is established by an argument similar to the one used in the previous theorem. We will concentrate on proving that the system conservatively detects inconsistencies. An inconsistency occurs when a thread has acquired an object version from a failed process, which the failed process cannot recover.

Assume that thread tid_1 acquired object version V_n. An entry describing this access was created in the thread's $depSet$. The entry has the value $<O1, type, ep_1, ep_2^2, P_2>$ with $ep_1 = <tid_1, lt_1>$ and $ep_2^2 = <tid_2, lt_2^2>$. The entry means that thread tid_1 acquired version V_n at logical time lt_1; version V_n was produced by thread tid_2 and lt_2^2 is the logical time of thread tid_2 when the acquire request was satisfied. This version was produced by a `release-write` issued by thread tid_2 at logical time lt_2^1 and $lt_2^1 \preceq lt_2^2$.

Assume that the process, P_2, where tid_2 was executing fails, then (1) the information in tid_1's $depSet$ will be transmitted to P_2 during recovery, unless (2) the process where tid_1 was executing failed before the $depSet$ was saved in a checkpoint.

Consider case (1). The information from the $depSet$ will be integrated into tid_2's $DependList$. During the detection process, thread tid_2 traverses the $LogList$, which is ordered by ascending execution points. The detection mechanism finds a maximum length prefix of the sequence of entries in the $LogList$, that includes an element for each logical time since the logical time at checkpoint. If some version were lost due to multiple failures the prefix will be a proper prefix and it discards the rest of the list.

Since thread tid_1 acquired version V_n, thread tid_2 will find an entry in the $DependList$ for the acquire that was satisfied at lt_2^2. If lt_2^2 is smaller than the logical time of the last element in the prefix, version V_n can be recovered because it was produced at lt_2^1 and $lt_2^1 \preceq lt_2^2$. Otherwise, the detection mechanism conservatively considers that V_n cannot be recovered and aborts the application.

In case (2) thread tid_2 will not find tid_1's dependency on version V_n. Therefore, if no other thread depends on version V_n the detection mechanism will not consider the system inconsistent. In fact tid_1 rolled back to a logical time previous to the acquire.

□