Reducing the Extent of Cascadable Operations in Stable Distributed Persistent Stores

Rasool Jalili and Frans A. Henskens

Basser Department of Computer Science
The University of Sydney
N.S.W. 2006
Australia

{rasool,frans}@cs.su.oz.au

Abstract

The act of accessing data objects in a persistent store may result in the creation of state dependencies between the accessing processes and data objects. It is important for the logical integrity of the store that checkpoint or roll-back of all dependent entities (processes or objects) occurs as an atomic action. Reducing the number of dependent entities in the system leads to shortening of the period of suspension of processes during checkpoint and roll-back operations and thus improvement in system efficiency. In this paper we investigate a new approach to the representation of dependencies based on differentiating between dependencies created as a result of read accesses and of write accesses. Using directed graphs to maintain information about dependencies and separately defining the meaning of graph edges for checkpoint and roll-back operations, we show that it is possible to significantly reduce the cascade effects of these operations.

1 Introduction

Persistent systems provide mechanisms for the uniform management of data regardless of its lifetime [1]. Provision of such uniformity, using conventional two-level store computer architectures, requires the abstraction of a very large store which involves all data in the system and appears as a failure-free store. Such store is referred to as a *persistent store* and the property of providing the illusion of being failure-free is often called *store stability* [2].

A Distributed Persistent Store (DPS) is a network-wide persistent store which supports concurrent access of users for a distributed system [6]. Casper [12] and the Monads-DSM [5] are examples of DPSs. Both support the existence of persistent objects in a paged virtual storage space. Nodes provide access to the persistent store by maintaining a cache of virtual pages from the store in their local volatile memory. These virtual pages are provided to client nodes by a central page server (Casper) or by a group of page servers (the Monads-DSM). Pages are transmitted between client and server nodes as required by user processes, the page coherence protocols and the store stability protocols.

When a computer system fails, the contents of its volatile memory are typically lost, while the contents of its non-volatile memory (disk or tape) remain unchanged. As persistent stores provide uniform management of data, the transfer of such data between volatile and non-volatile memory is transparent to the user. At any instant in time the true state of a persistent store is represented by the combination of the contents of the volatile and non-volatile memories. Since the contents of volatile memory are typically lost after system failure, a stable persistent store must be able to revert to some consistent state described in non-volatile memory.

Techniques proposed to achieve this ([7, 10]) are typically based on flushing the volatile system state to non-volatile storage (*checkpointing*) and using this state on occurrence of a failure (or in response to the request of the user or higher level applications) to revert to the most recent checkpoint (*rolling back*). This requires the existence of at least one global consistent state at each point in time. Accordingly, checkpoint operations must be achieved as an atomic action in order to guarantee the existence of such a state even if some failure occurs during the checkpoint operations themselves. It is desirable that such a consistent state is close to the state at the time of failure, thus minimising the loss of modifications to the store. However, frequent recording of a global state is not normally acceptable due to the cost of such operations.

Checkpointing the whole system state in an atomic action may necessitate cessation or restriction of the system normal operations during the checkpoint. This is not so efficient for the system throughput, especially in the case of DPSs. To overcome this problem, mechanisms [7] have been proposed to checkpoint parts of the store independently; the global state of the system is the collection of these stable parts. Checkpointing parts

Published in *Proceedings, 18th Australian Computer Science Conference* , Adelaide, Australia, pp 227-236, 1995.

of the store independently however, creates the possibility of logical inconsistencies between data. Therefore, inter-relationships (dependencies) between such parts must be taken into account to ensure consistency.

As processes access data objects, they may become dependent on each other, due to the possible effect of the data on the way that the processes modify data in some other objects. It is important for the logical integrity of the store that checkpointing of all dependent entities occurs as an atomic action. Dependencies are detected and recorded in Casper using sets called *associations* [12].

The problem with large sets of dependent entities (especially in a distributed environment) is the degradation in system performance caused by the propagation of checkpoint and roll-back operations called the *cascade effect*. This cascade effect can be reduced by controlling the number of the independently checkpointed entities and the frequency of checkpoint operations. As we will show the extent of cascadable operation is reduced through the use of a new technique for describing dependencies.

In this paper, we investigate the use of directed graphs as a method of maintaining dependencies among entities in a DPS. We show that the use of directed graphs allows us to separately describe checkpoint and roll-back dependencies, thus improving store efficiency.

2 Entity dependency in distributed environments

Constructing a system-wide consistent state in distributed systems is not as straightforward as for single-node systems. Even if each node has its own individual consistent stable state, a collection of individual stable states cannot be considered as a global consistent state. This is because of the dependencies among entities belonging to different nodes and that such dependencies may traverse nodes.

Process communication in computer systems may lead to inter-entity dependencies. Processes in distributed systems may communicate through passing messages or accessing a (virtual) shared memory. The criterion for dependency is not similar for these methods of communication.

2.1 Dependency criteria in message passing systems

In message passing systems a process communicates with another process or invokes an object by sending and receiving messages. Such message passing results in the dependency of the receiver to the sender of a message. The sender may also have received messages from other processes, resulting in a cross-dependency of entities. Knowledge of the order (time) in which messages are sent and received is important in such systems. However, no system-wide physical clock exists in distributed systems. Lamport [9] proposed achievement of such ordering using the concept of logical time. Logical time describes the happened-before relation between local events in a process as well as the inter-process events such as sending and receiving messages. Having such a unique time and appending messages with the time, it is possible to specify criteria for dependency and global consistent state. According to [11] ,

1) Message M_i depends on message M_j if and only if M_i is produced in a site after receipt of M_j or after receipt of M_k which depends on M_j .
2) Message M is called an 'orphan message' if its receiver records the receipt in its stable state, but its sender reverts to a state which existed before its sending of the message. Message M is called a 'missing message' if its sender records its sending, but its receiver reverts to a state which existed before its receipt of the message. A global state which does not include any orphan or missing message is called a *global consistent* state.

The problem with message passing systems is that each message transmission is regarded as a source of dependency between the sender and receiver of the message. This is while some messages may include special system management information or read-only data which in fact result in no real dependency.

2.2 Dependency criteria in shared memory systems

Processes in Distributed Shared Memories (DSMs) communicate through accessing the same data objects. This sort of communication results in dependency of the processes and the object if the object has been modified since the last checkpoint operation. Similarly to sending and receiving messages in message-based systems, writing to and reading from an object are considered the two major events in DSMs. Writing on an object may be considered equivalent to sending a message to the object and reading from an object equivalent to receiving a message in terms of message passing systems [4] .

In contrast to message passing systems in which each message transmission results in a dependency, in DSMs,

- access to data (only transmission of messages including data in terms of message passing systems) is considered as the criterion of dependency, and

- operation behaviour can be taken into account to prevent false dependencies to be considered. For example, accessing an unmodified data results in no dependency; this may be detected using coherency mechanisms provided for the DSMs.

In summary, using DSM reduces inter-entity dependencies in comparison with message passing systems [8] .

3 Implications of dependencies in distributed persistent stores

Casper and the Monads-DSM are examples of DPS implementations which attempt to provide stability of their store through checkpointing individual parts of their store. They track and maintain dependencies for their defined granularity of stabilisation.

Casper employs the central server model of distributed memory management to provide the abstraction of a DPS. It considers the world as a set of clients served by a central server which provides access to shared objects and maintains the stability and coherency of the paged persistent store. Checkpoint operations occur at the client level and may be cascaded to other clients. Clients which have seen the same modified data since the last stable state, are deemed to be dependent on each other and are grouped into dynamic sets called *Associations* [12]. Each Association is accompanied by a list of persistent pages which contain the modified data which was accessed in common. Whenever a client obtains a copy of a modified page, Associations may change according to the following rules:

- The Association to which the client belongs is merged with the Associations of other clients dependent on the page, and
- The page is inserted into the list of persistent pages modified by members of the merged Association.

When any client belonging to an Association initiates a checkpoint operation, all clients in the Association are forced to checkpoint. Similarly, if any client in an Association rolls back to its last stable state, all clients in the Association must roll back. This results in consistency of the persistent store.

The granularity of checkpoint in Monads is a volume (logical disk partition) [10] . In a multi-volume single Monads node, or a network of Monads computers, it is possible to have cross references between volumes. Volumes are used to store both processes (process stacks) and objects and therefore, dependencies may develop between them. Monads-DSM represents such dependencies using dependency graphs of volumes. In order to ensure consistency, volumes containing cross references are checkpointed together and a dependency graph is maintained at each node to describe relationships between volumes. To ensure atomicity of checkpoint operations, a two-phase commit protocol is also proposed, in which a volume and all its dependent volumes are checkpointed together [7]

A problem with the original Monads-DSM approach is the granularity of stability. Using volumes as the granularity of stability leads to the incidence of checkpoint operations on unwilling data objects in a volume together with some other objects which must be checkpointed for consistency reasons. In single volume nodes, a checkpoint on that volume effectively results in a suspension of operation for that node.

4 Implications of dependency on performance and useability

Existence of a system-wide consistent stable state at recovery from a failure is ensured if checkpoint of an entity is propagated to all its dependent entities. This can be provided through the achievement of the following operations as an atomic action:

- the checkpoint of the entity, and
- the checkpoint of all other dependent entities to which the checkpoint is propagated.

A similar sequence should also be followed in the case of roll-back of an entity following a failure. The problem with such a sequence is the partial cessation of processing during the checkpoint or roll-back operation. Suspension of operation of a part of the system results not only in unuseability of the suspended part, but also inability of other (non-suspended) parts to communicate with the suspended part. This is more crucial in distributed systems as entities reside on different nodes may depend on each other and thus completion of a checkpoint may require transmission of some messages and wait for reply from remote nodes. Therefore, reducing the effect of dependent parts of the store is required to improve the system performance

Reducing the effect of dependent data on system performance requires that

1) Propagation of checkpoint and roll-back operations to unrequired parts of the store is prevented. Selection of entities as the granularity of checkpoint operations reduces such propagation.
2) The extent (number) of dependent parts of the store is reduced. To provide this, we need to rethink our basis for dependencies. By exploiting the behaviour of operations between processes and objects in terms of dependency, we will define a new model of propagation of checkpoint and roll-back operations.

5 A new representation of dependencies

Considering processes as the agents of change on data and objects as the repository of data in a persistent store, inter-entity dependencies are created as a result of the invocation of objects by processes. Neither processes nor objects can directly depend on each other, but they may depend on each other through an entity of the other kind.

We assume entities as the granularity of checkpoint (and roll-back) operations. Checkpoints are initiated for entities to advance their stable state. During a checkpoint operation for an object, all modified pages since the last checkpoint are written back to disk, becoming clean. We refer to such object as an *unmodified object* (unmodified since its most recent checkpoint) and to an object with some modified data since its most recent checkpoint as a *modified object*.

Previous work on stability in persistent systems [7, 13] considered that dependency is always a bidirectional relationship, i.e. when two entities are dependent, the checkpoint or roll-back of each of them is propagated to the other entity. In the new representation of dependencies, we distinguish between dependencies created as a result of different operations. Accordingly we propose that dependency is in fact a unidirectional relationship between two entities of different kind (processes and objects). By considering read and write as the main operations on objects, the following categories of operations may be considered in terms of dependency.

1) A process may read from an unmodified object. This results in no dependency between the reading process and the object as the current state of the object is stable.
2) A process may read from a modified object. This results in a unidirectional dependency between the reading process and the object, due to the unstability of the read data at the time of the read operation. The direction of the dependency (i.e. which side depends on the other side) is a matter which is specified when a *cascadable* operation (checkpoint or roll-back) is being propagated. Consider a process *P* which has read from a modified object *O*. Before *P* checkpoints its state, it must ensure that the read data is stable; *O* is not required to ensure about the stability of *P* when it is being checkpointed. As a result, the direction of dependency in the case of checkpoint is from *P* to *O*. A roll-back of *O* would result in an inconsistency with *P* and thus necessitates a roll-back of *P*. However, a roll-back of *P* does not affect *O*. Thus, the direction of dependency in the case of roll-back is from *O to P* (*O* depends on *P*).
3) A process may modify an object. This results in the dependency of both entities on each other and is represented by a pair of unidirectional dependencies between them. This pair of dependencies is necessary to prevent the occurrence of *missed object modification* or *orphan object modification* through propagation of checkpoint (roll-back) of each side to the other side. A missed object modification occurs when an object is modified and subsequently rolled back to the state prior of the modification. An orphan object modification occurs when an object is modified and subsequently the modifying process rolls back to the state prior to the modification.

Methods of representing dependencies in persistent systems which assume all dependencies are bidirectional are unable to represent directional dependencies. Directed graphs can be used to fulfil the requirements of the new representation. We refer to a directed graph which represents such dependencies as a *Directed Dependency Graph* (DDG). DDGs are also used to separately represent the checkpoint and roll-back dependencies between entities. This is described later in this paper.

One DDG is associated per entity, but different spanning-trees may be traversed depending on the kind of the operation (checkpoint or roll-back), initiated for the entity. As read operations in a computer system typically outnumber write operations [3], we believe that the cost of cascadable operations will decrease dramatically by use of the proposed directed graphs. This is because read operations on modified data now create a unidirectional dependency. The simulation results presented in section 7 confirm this claim.

5.1 Notation

We use $\rightarrow$ edge in order to specify the dependency relationship between two entities. $E_1 \rightarrow E_2$ means that E_1 depends on E_2. $\rightarrow$ is transitive, but not symmetric i.e.

if ($E_1 \rightarrow E_2$) and ($E_2 \rightarrow E_3$),
then it is implied that ($E_1 \rightarrow E_3$)
but, $E_1 \rightarrow E_2$ does not imply that $E_2 \rightarrow E_1$.

However, the right hand side of a $\rightarrow$ relation may depend on the left hand side

- through transitivity; the existence of a cycle in the directed dependency graph (e.g. for $E_1 \rightarrow E_2$, we may also have $E_2 \rightarrow E_3$ and $E_3 \rightarrow E_1$ which implies that $E_2 \rightarrow E_1$), or when
- a process has modified an object, which results in two unidirectional edges with different directions between the two entities.

In the case of a write operation which leads to a pair of dependencies, instead of indicating two unidirectional edges ($E_1 \rightarrow E_2$ and $E_2 \rightarrow E_1$), we use the notation $E_1 \leftrightarrow E_2$.

The construction and maintenance of DDGs are integrated into the management of the persistent store which typically utilises the operating system kernel services such as the virtual memory management.

5.2 Construction of directed dependency graphs

We assume that dependencies are recorded as soon as they occur. Update of DDGs is synchronously achieved with the operations cause the dependencies. A DDG grows or shrinks according to the following criteria.

- When a process reads an unmodified object, nothing is added to any DDG.
- When a process P_1 reads a modified object O_1, the edge $P_1 \rightarrow O_1$ is inserted into the DDG including P_1 or O_1, if at least one of the DDGs containing P_1 or O_1 includes only one entity; otherwise the edge joins the two DDGs.
- When a process P_1 modifies an object O_1, the edge $P_1 \leftrightarrow O_1$ is inserted into the DDG including P_1 or O_1, if at least one of the DDGs containing P_1 or O_1 includes only one entity; otherwise the edge joins the two DDGs.
- When a process belonging to a DDG reads an object or modifies an object which belongs to another DDG, the two DDGs are merged using one of the above edges to create a single larger graph.
- A graph shrinks when a set of dependent entities is checkpointed or reverts to their last stable state. Once a checkpoint or roll-back operation is initiated for an entity E, for each entity which is reachable from E in the DDG corresponding to the operation, the operation is applied on the entity. Then all edges attached to the entity are removed. Such removal of all edges is possible due to the stability of the entity[1].
- DDGs are not persistent. When a node crashes, all graphs stored on that node are lost.
- DDGs do not store any information about the order of occurrence of dependencies. Operations on an object may happen in different times, the issue which is unimportant during a time slice regarding dependency. Nevertheless, before the allocation of a new time slice to a process, it should be ensured that all dependencies till this stage have been recorded.

At any given time each entity belongs to one and only one dependency graph. To find the entities dependent on an entity, it is sufficient to find the location of the entity in its graph (subject to the kind of operation) and then traverse the directed graph starting from the entity. This may be different for each entity in the graph and thus may result in a different set of dependent entities.

[1] At this stage we assume that checkpoints are achieved atomically.

5.3 Description of dependencies in a multi-node environment using DDGs

So far we have implicitly assumed that entities belong to a single-node computing environment. It is crucial that the described model be able to describe dependencies between entities in a distributed environment where an entity in a node may depend on a remote entity. The described scheme is fully applicable to a distributed environment with the abstraction of shared virtual memory. It may also be applicable for message passing systems if page accesses are properly replaced by message transmission assuming that messages contain enough information regarding their operation behaviour.

Two major issues regarding the application of described DDGs for distributed virtual stores are the maintenance of distributed DDGs and the representation of edges which link entities residing on deferent nodes. A centralised or distributed approach may be applied to manage distributed DDGs regardless of the model of page serving. However, to gracefully integrate stability management with memory management, distributed graphs might be more harmonised with the model of page serving used in distributed virtual stores. Accordingly the following approaches can be taken:

1) Use of a central server for dependencies: In DPSs such as Casper which rely on a central page server, all objects are accessed through a central server. Such a server can also manage distributed dependency graphs. This method of maintaining graphs is simple, but it has the potential drawback of the server bottleneck.
2) Use of a distributed server for dependencies: Regardless of the page serving strategy this group of approaches relies on a distributed graph which is maintained by all processing sites. This does not force any relationship between page serving nodes and dependency maintaining nodes. For example, diskless nodes in a distributed system do not serve any data object, while they may host processes which access remote objects and thus insert some edges in the distributed dependency graph. Distribution of DDGs may be achieved either through replication of DDGs or decomposing DDGs into sub-graphs, as described below.

- **Replication**: A distributed DDG is replicated on all nodes which own at least one entity in the graph, even disk-less nodes. This overcomes the drawback of the server bottleneck in the central-server approach, but it requires the maintenance of consistency between graph replicas. In this approach, a node must be aware of all

dependencies happening on all nodes with common DDGs with the node; most of the dependencies may be locally unimportant. Moreover, any change in a DDG must be atomically multicast to all nodes residing a replica.

- **Sub-graphs**: A distributed DDG is decomposed into some sub-graphs, each maintained by a different node. Each sub-graph may include all local dependencies in a node as well as *direct dependencies*[2] of its entities on remote entities. As the majority of accesses in a node are typically local, each sub-graph is populated with local entities, but some edges connecting a local entity to a remote entity may also exist.

  This approach balances the load of maintaining dependency graphs gracefully over nodes, but finding all dependent entities on an entity may require traversing the graph over nodes which necessitates transmission of messages. The approach is appropriate for distributed persistent systems which are constructed using a true distributed shared memory. In comparison to the alternative approaches the cost of the approach is reasonable, but cross references of entities on different nodes is a problem. However, node crashes result in the lost of their sub-graph and thus roll-back of dependent entities.

Representation of edges which link two remote entities is similar to that of edges which link local entities in both centralised maintenance of DDGs and replication of DDGs. In the case of distributed maintenance of DDGs using sub-graphs, however, this is not so straightforward as an edge should link two entities which reside on different sub-graphs on different nodes. When a process *P* attempts to access a modified page of a remote object *R*, a page fault is raised locally. In handling the fault, the local kernel realises that the fault is for a remote page and therefore sends a message to its owner node, requesting to supply the page. To ensure that the local kernel and the kernel in the owner node are aware of possible dependency as a result of the access, a solution is to duplicate such edge and let both kernels to insert the edge into their sub-graphs. Inserting the edge in both kernels require the existence of a vertex representing the remote entity.

We propose *pseudo entities* to provide the local representation of remote entities. A pseudo entity is the ghost of a remote entity and contributes in local sub-graphs of a DDG behalf of the remote entity. To distinguish entities which correspond to remote pseudo entities, we refer to them as *real* entities and to other entities as *non-pseudo* entities. Pseudo entities are identified with the identity of their corresponding real entities. Pseudo entities act similar to real entities regarding the propagation of cascadable operations except that a message should also be sent to the node owning their corresponding real entities to propagate an operation. We assume that each entity's identifier encloses its host node identifier.

Each link connecting two entities residing on different nodes in fact is represented by two edges, each links a pseudo entity to a real entity in the same way as for local entities. To cope with this arrangement, the local kernel in the scenario of accessing a modified page of a remote object mentioned above, has the knowledge of the local process (*P*) and the remote object (*R*) containing the requested page, the remote kernel also must have the knowledge of both *R* and *P* (as a remote process). Therefore, as a part of its requesting message, the local kernel must enclose the identifier of *P*.

Consider the distributed DDG shown in figure 1 in which process P_{12} reads a modified page of the object O_{21}. P_{12} accesses the object through a virtual address. Because the corresponding page is not in local memory, a page fault occurs and consequently the page-fault-handler sends a request to the server for the page. Eventually,

- a copy of the page is transferred into the local memory of N_1,
- N_1 inserts the edge $P_{12} \rightarrow O'_{21}$ into its dependency graph (O'_{21} is a pseudo entity for O_{21}), and
- N_2 inserts the edge $P'_{12} \rightarrow O_{21}$ into its dependency graph (P'_{12} is a pseudo entity for P_{12}).

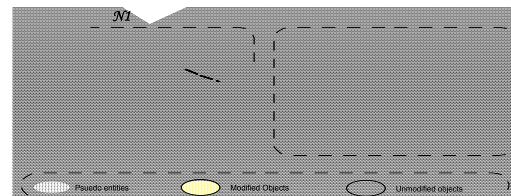

Figure 1 P12 reads from a remote object O21.

6 Examples of implications of this new representation

In order to illustrate the impact of the new scheme in single-node stores, we present, in figure 2, a scenario in which the effect of operations on DDGs is demonstrated. We then show the effects of checkpoint and roll-back operations on the resultant graph.

The figure depicts a sequence of operations performed in a store starting from an initial state (e.g. after system restart). We assume that three processes (P1, P2, and P3) are accessing four objects (O1, O2, O3, and O4). Processes are shown by circles in the figure, while objects are shown by

---

[2] An entity directly depends on another entity, if the path connecting them has only one edge.

blank (unmodified) or shaded (modified) rectangles. For simplicity, we do not consider system-related information maintained on a per object basis.

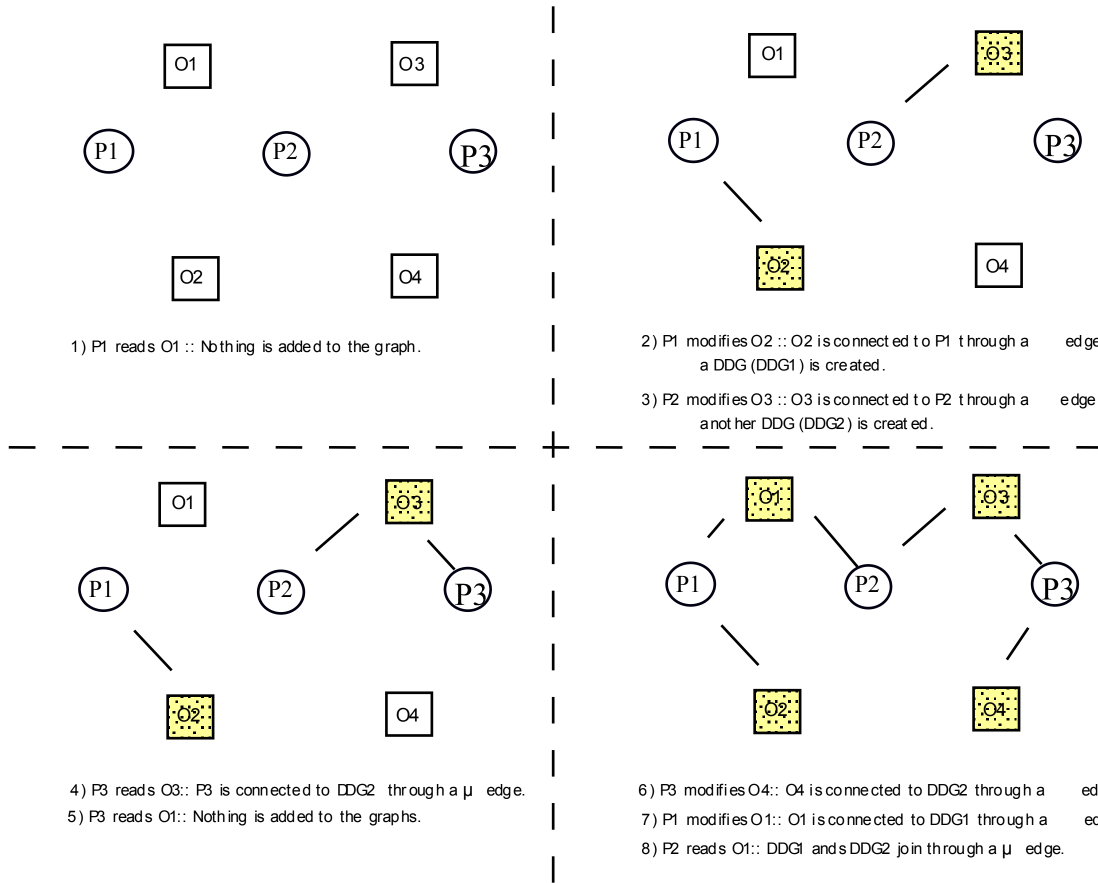

Figure 2 The scenario of merging directed dependency graphs due to read and write operations

6.1 Implication of new representation on cascadable operations

DDGs are used to identify entities for which a checkpoint or roll-back operation should be propagated. However, the path which should be traversed for cascading roll-back is different from that for cascading checkpoint operations as a result of the unidirectional edges in the graph. We use a single dependency graph and then apply separate checkpoint and roll-back algorithms to propagate the operation.

To this end, the relation ∅ in the dependency graph has different meanings in terms of cascading checkpoint or roll-back operation. To distinguish between our different meanings of the symbol, two further symbols are introduced to specify the dependency between two entities in terms of checkpoint or roll-back. By $E_1 \overset{S}{\rightarrow} E_2$ we mean that when E_1 is checkpointed, E_2 also should be checkpointed and therefore E_1 depends on E_2 in terms of checkpoint. Likewise, $E_1 \overset{R}{\leftarrow} E_2$ means that E_1 depends on E_2 in terms of roll-back. Figure 3 shows the relationship between the edges forming a dependency graph and their meaning in checkpoint and roll-back graphs. Note that the expression $E_1 \rightarrow E2$ is congruent to the expression $E_2 \leftarrow E_1$.

Dependency Graph	Checkpoint Graph	Roll-back Graph
$\rightarrow$	$\overset{S}{\rightarrow}$	$\overset{R}{\leftarrow}$
$\leftarrow$	$\overset{S}{\leftarrow}$	$\overset{R}{\rightarrow}$
$\leftrightarrow$	$\overset{S}{\leftarrow}$ and $\overset{S}{\rightarrow}$	$\overset{R}{\rightarrow}$ and $\overset{R}{\leftarrow}$

Figure 3 The relationship between Dependency Graph, Checkpoint Graph, and Roll-back Graph

As we have shown, read and write operations have different effects in terms of dependency. While the write operation makes both sides of the operation dependent on each other, the read operation on modified data makes only the reader process dependent to the read object. As the ratio of read to write operations in usual applications is high, we can prevent the propagation of checkpoint and roll-back in some parts of the graph.

To clarify the distinction between the implication of DDGs on checkpoint and roll-back

operations, consider a scenario in which a process P_1 reads from a modified object O_1. Until the next checkpoint operation is commenced for process P_1, all actions taken by P_1 based on what it has read from O_1 is unstable. During such period of unstability, either of O_1 or P_1 may be checkpointed or rolled back.

1) If O_1 is checkpointed, P_1 does not have to be checkpointed. The worst case is that after checkpointing O_1, P_1 is rolled back to its last stable state. This results in no inconsistency as the stable data in O_1 may be read by P_1 again.
2) If P_1 is checkpointed, O_1 must also be checkpointed. Otherwise, in the case of O_1's roll-back, there is an inconsistent state in which an orphan page modification has been read by P_1.
3) If O_1 is rolled back, P_1 has to be rolled back. Otherwise, P_1 has read an orphan object modification and thus an inconsistent state.
4) If P_1 is rolled back, O_1 does not have to be rolled back. This is due to the possibility of the repeat of the lost read operation.

6.2 Checkpoint propagation using DDGs

The checkpoint of an entity requires the checkpoint of the entity itself, all of its dependents (if any exist), and the system related data structures to be achieved as an atomic action. We review the impact of DDGs on the propagation of checkpoint operations in this section. To checkpoint an entity E, E is located in a DDG and the operation is resumed based on the following algorithm.

```
Procedure Stabilise (E: entity)
begin
   if E has already set 'visited'
      then return;
      else set E as 'visited';
   for all Ei connected to E do
   begin
      if E -S-> Ei
         then Stabilise (Ei );
      delete the edge between E and Ei;
   end;
   EntityStabilise (E);
end.
```

For example, consider the application of the algorithm to checkpoint P_1 in the resultant graph from the scenario in figure 2. The operation is propagated to only O_1 and O_2, as shown in figure 4. This restricted propagation represents a significant improvement in performance of the checkpoint operation in comparison with using Associations; in that case all entities in the scenario must be checkpointed. This is more crucial in the case of distributed dependency graphs where the propagation of a checkpoint operation may result in the blocking of all or some entities in the coordinator node until participant nodes reply.

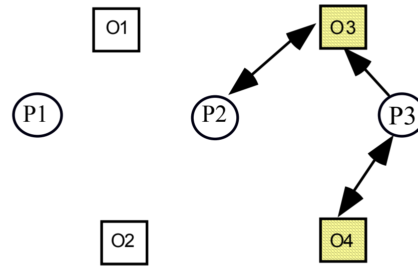

Figure 4 The state of entities after the checkpoint of p_1

6.3 Roll-back propagation using DDGs

A roll-back operation is initiated for an entity whenever the entity recovers following a system failure or the abortion of a transaction in an application. An entity may also be required to roll back due to the roll-back of a dependent entity, or even due to the occurrence of a failure in the node contains some entities dependent on the entity (in terms of roll-back graph). With minor differences, the steps described in section 6.2 are followed for the propagation of the roll-back of an entity E. After locating the entity E in the graph, operations resume according to the following algorithm.

```
Procedure Rollback (E: Entity)
begin
   if E has already 'visited'
      then return;
      else set E as 'visited';
   for all Ei connected to E do
   begin
      if E -R-> Ei
         then Rollback (Ei )
      delete the edge between E and Ei;
   end;
   EntityRollback (E);
end.
```

For example, consider the application of the algorithm to roll back P_3 in the graph resulted from the scenario in figure 2. This leads to the propagation of the roll-back to only O_3 as shown in figure 6. Such a roll-back would result in the roll-back of all entities if Associations were used.

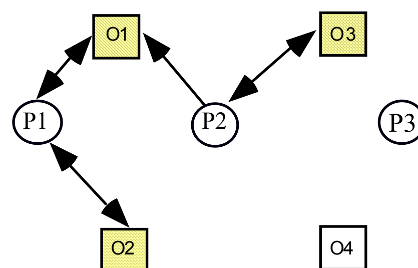

Figure 6 The state of entities after the roll-back of P_3

7 Simulation results

To evaluate the effect of using DDGs, we simulated the Monads persistent store when either Associations or DDGs are used to manage dependencies. Events considered in the simulation were: *process creation*, *process termination*, *object opening*, *object closing*, *entity checkpoint*, *entity roll-back*, *read access*, *write access*, and *process switch*. Parameters in the study were: *objects size*, *processes lifetime*, *checkpoint operations interval*, *roll-back operations interval*, *locality of accesses*, and *the rate of load/store operations*. For the results depicted in figures 6 and 7, we selected 20 seconds as the mean interval of checkpoint operations, 360 seconds as the mean interval of roll-back operations, 4 as the mean rate of load/store operations, and 10 as the mean locality of accesses. As there was no difference between a single-node and multi-node persistent environment to compare the two methods, we simulated a single-node environment.

The results show that using Associations about 65% more entities are checkpointed (possibly through propagation from other entities) than when DDGs are used. Figure 6 shows the cumulative number of entities checkpointed using Associations and directed graphs. Using Associations, the number of rolled back entities is 90% higher than when DDGs are used. Figure 7 compares Associations and directed graphs in terms of the number of rolled back entities.

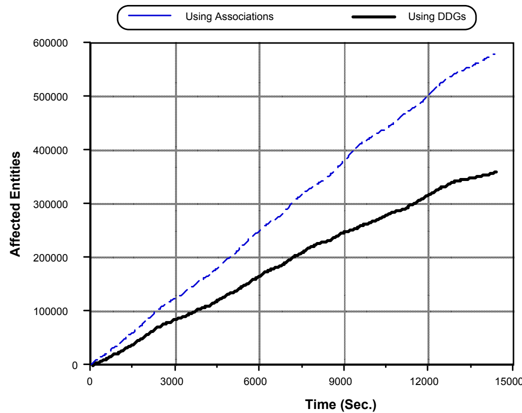

Figure 6 Number of checkpointed entities (cumulative) using associations and DDGs.

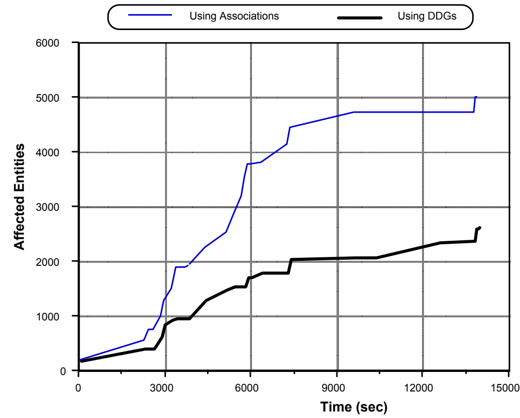

Figure 7 Number of rolled back entities (cumulative) using associations and DDGs.

Conclusion

In this paper we described the shortcomings inherent in the use of Associations to describe dependency relationships between processes and objects in a distributed persistent store. In particular we showed that the cascade effect of checkpoint operations resulted in larger checkpoint operations than absolutely necessary. We also showed that the cascade effect in roll-back operations resulted in unnecessary loss of store modification.

We presented an alternate method for describing entity inter-relationships. This alternate representation uses directed graphs. By further separately defining the meaning of graph edges for checkpoint and roll-back operations, we showed that it is possible to significantly reduce the cascade effects of these operations. As a result, checkpoint and roll-back operations are improved in terms of efficiency, and the loss of data caused by roll-back operations is reduced. This is a significant achievement, because only those modifications which it is absolutely necessary to reverse are lost as a result of the roll-back operation. The included simulation results confirm this claim.

By integrating the management of dependencies into the virtual memory management, checking to see whether a DDG should be updated is not expensive. As virtual pages are the unit of data transfer in a typical memory management scheme, detecting dependencies at the level of virtual pages can be achieved when page faults and write-fault exceptions are being handled.

Acknowledgements

This research has been supported by the Ministry of Culture and Higher Educations, the Government of the I. R. of Iran.

References

[1] Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, P. W. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, 26(4):360-365, 1983.

[2] Brown, A. L. "Persistent Object Stores", Universities of St. Andrews and Glasgow, Persistent Programming Report 71, 1989.

[3] Cvetanovic, Z. and Bhandarkar, D. "Characterization of Alpha AXP Performance Using TP and Spec Workloads", *IEEE Computer Architecture News*, 22(2):60-70, 1994.

[4] Gunaseelan, L., Richard, J. and LeBlanc, J. "Event Ordering in a Shared Memory Distributed System", *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, Pennsylvania, 1993.

[5] Henskens, F. A. "A Capability-based Persistent Distributed Shared Memory", PhD Thesis, University of Newcastle, N.S.W., Australia, 1991.

[6] Henskens, F. A. and Rosenberg, J. "Distributed Persistent Stores", Microprocessors and Microsystems, 17(3):147-159, 1993.

[7] Henskens, F. A., Rosenberg, J. and Hannaford, M. R. "Stability in a Network of MONADS-PC Computers", *Proceedings of the International Workshop on Computer Architectures to support Security and Persistence of Information*, Springer-Verlag and British Computer Society, pp. 246-256, 1990.

[8] Janssens, B. and Fuchs, W. K. "Reducing Interprocessor Dependence in Recoverable Distributed Shared Memory", *In Proceedings of the 13th Symposium on Reliable Distributed Systems*, IEEE Computer Society Press, Dana Point, California, pp. 34-41, 1994.

[9] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed Systems", *Communications of the ACM*, 21(7):558:565, 1978.

[10] Rosenberg, J., Henskens, F. A., Brown, A. L., Morrison, R. and Munro, D. "Stability in a Persistent Store Based on a Large Virtual Memory", Springer-Verlag and British Computer Society, pp. 229-245, 1990.

[11] Tong, Z., Kain, R. Y. and Tasi, W. T. "A Low Overhead Checkpointing and Rollback Recovery Scheme for Distributed Systems", Proceedings of the Eighth Symposium on Reliable Distributed Systems, pp. 12-20, 1989.

[12] Vaughan, F., Basso, T. L., Dearle, A., Marlin, C. and Barter, C. "Casper: a Cached Architecture Supporting Persistence", *Computing Systems*, 5(3):337-359, 1992.

[13] Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. "A Persistent Distributed Architecture Supported by the Mach Operating System", *Proceedings of the 1st USENIX Conference on the Mach Operating System*, pp. 123-140, 1990.