# Operating System Support for Object Dependencies in Persistent Object Stores

*Rasool Jalili, *Frans A. Henskens, †David. M. Koch & *John Rosenberg

*Department of Computer Science
University of Sydney, N.S.W., 2006, Australia
{rasool,frans,johnr}@cs.su.oz.au

†Department of Computer Science
University of Newcastle, N.S.W., 2308, Australia
dmk@cs.newcastle.edu.au

## Abstract

*Persistent object stores provide uniform management of short-term and long-term objects. Such stores ensure the integrity of the data even after occurrence of a failure, by guaranteeing the existence of some previous self-consistent stable state at each point in time. Maintaining a consistent state of a persistent store necessitates recording of inter-object dependencies and checkpointing of each object together with all its dependent objects. Directed graphs may be used to describe such dependencies. In this paper we describe eager and lazy construction of dependency graphs. We then address operating system and hardware support for management of dependencies.*

**Keywords**: Dependency, Stability, Persistent Systems, Fault-tolerant Systems, Checkpoint, Roll-back.

## 1 Introduction

Persistent systems support mechanisms to allow the creation and manipulation of arbitrary data structures which survive in their original form even after the termination of the program that created them [2]. As a result, programmers are not required to flatten data structures in order to store them permanently. In this sense a persistent system provides an alternative to a conventional file system for the storage of permanent data. This alternative is far more flexible as it allows both the data and its interrelationships to be stored in their original form. Achievement of this requires a uniform storage abstraction which is often called a *persistent store*.

A persistent store supports mechanisms for the uniform management of objects and their interrelationships regardless of their lifetime. Therefore, the distinction between primary and secondary storage has been abstracted over in persistent stores. The state of the store at any instant is a combination of the data held in main memory (RAM) and the more stable data held in secondary memory (disk). When a failure unexpectedly occurs, the contents of main memory are typically lost. As a result, the data stored in secondary memory may be inconsistent or unreachable. The abstraction over storage should include transparent recovery from such store failures so that the store contents are guaranteed to be consistent despite unexpected store failure. Stores supporting such a property are referred to as *stable stores*, and advancement of permanent and consistent state of the store is achieved through a sequence of operations called *checkpoints* [12].

The benefits of support for dual object sizes, providing both small objects corresponding to the logical units of data manipulated by programs (structures etc) and paged large objects comprising collections of logically related small objects, has been considered elsewhere [6]. In this paper we assume that either:

- the object store supports both small and large objects, with stability being implemented at the large object level, or
- the object store supports only paged objects.

We consider paged objects as the granularity of stabilisation and thus express dependencies in terms of these objects and processes. In the remainder of this paper we use the term object to refer to such paged objects and the term entities to refer to objects or processes. We assume that processes are persistent as their state is kept in the persistent store. Accordingly, by checkpoint and roll-back of a process, we mean the checkpoint and roll-back of its state. We also refer to data as being modified if it has been changed or created since the last time the object containing the data was checkpointed.

In section 2 of this paper we examine techniques used in the implementation of stable stores and introduce the concept of object level checkpoints. In section 3 we show that such checkpoints must consider logical relationships between objects (dependencies), and describe a scheme for expressing inter-entity dependencies using directed graphs. In section 4 we show how operating system support facilitates the implementation of stable stores based on this scheme. In sections 5 and 6, we describe eager and lazy construction of directed dependency graphs. Finally, in section 7, we describe the role of disk directories and the operating system kernel itself as

Published in: *Proceedings, International Workshop on Object-Oriented Real-Time Dependable Systems*, IEEE, Dana Point, California, U.S.A., pp. 21-29, 1994

special objects in the process of managing dependencies.

## 2 Provision of stability

A persistent store is said to be stable if it automatically recovers to a consistent state after a failure which has prevented orderly system shutdown. Stability in persistent stores is typically provided using operations called checkpoints which flush all modified data currently held in main memory to disk, and atomically create a snapshot of the store at that moment. Processing usually ceases on the store during such a checkpoint operation. Between checkpoint operations on a store, the state of the store is represented by the contents of disk plus the contents of modified data held in main memory. As the management of the store requires that from time to time a main memory page frame is re-assigned, modified data in such a page frame must be saved before the page frame can be re-used. Writing such page data back to its original location on disk potentially leads to the disk representation of the store being inconsistent and therefore unstable. Shadow paging [10] is a technique that allows modified pages to be discarded without causing an inconsistent disk version of the store. The atomicity of checkpoint operations may be guaranteed using Challis' algorithm [3].

Shadow paging maintains two copies of modified data between checkpoint operations; a copy of the stable data as it existed at the last checkpoint (shadow version data) and a copy of the latest version of the data (current version data). The scheme may be implemented for individual objects, but is more typically applied at the virtual page level. In a paged store, implementation of shadowing at the virtual page level minimises fragmentation by allowing more than one object to reside in the same page, and improves the efficiency of shadowing for objects that span multiple pages. A new disk block is allocated per modified page to store either the shadow version or the current version of the page depending on the method of implementation. System data structures are provided to point to disk blocks containing both the shadow version and the current version of the store. Occurrence of a failure may make the current version of the pages useless, but in such circumstances the shadow version of pages is still accessible.

Ensuring the consistency of a persistent store requires that checkpoint operations are atomic. Since writing to disk is a sequential operation, it is impossible to write a set of disk blocks in a single operation. Even the write of a single disk block may not be atomic in the case of failure. Challis [3] proposed an algorithm to make the update of a disk block appears to be a single action. Using Challis' algorithm, it is possible to achieve atomicity of checkpoint operations by committing them through the update of a single disk block. This requires that all virtual pages forming a distinct state of the store are ultimately accessible from this disk block. Viewing the store as a tree structure of pages and maintaining the information about its root entry in the disk block satisfies this requirement. We refer to the disk block as the root block and provide its update atomicity according to Challis' algorithm. By maintaining two root blocks referring to different states of the store, it is possible to preserve the consistency of the store even if some failure occurs while a root page is being written to disk [12].

The problem with the stability scheme as described is that the entire store must be checkpointed at the same time. Such operation requires processing either to be ceased or severely restricted during the checkpoint. In a multi-user store involving multiple nodes this would result in unacceptable degradation of performance. Accordingly, systems have been developed which checkpoint parts of the store independently [8, 13]. The stable state of such a store is the collection of these stable parts. Checkpointing parts of the store independently however, creates the possibility of logical inconsistencies between data. Modified data from one object may influence the way a process modifies data in some other object. As a result the two objects have a dependency relationship which must be considered when checkpointing either of them. Such dependencies have been described using sets of clients in Casper [13], and more recently using directed graphs of entities [9].

## 3 Object dependencies in persistent stores

Accessing of data objects by processes in a store may result in dependencies being created between the processes and the objects. Such dependencies are established as a result of write operations which modify data, and subsequent read operations on the modified data. It is important to note that data objects cannot become directly inter-dependent without processes, and that processes cannot become directly inter-dependent without data objects. Nevertheless, dependencies may be recorded on a per process (client) basis; a set of modified pages is associated with each set of dependent clients. It should also be noted that read operations on unmodified data do not create dependencies.

Ideally dependencies should be established based on knowledge of access to data at the basic unit of data reference (e.g. byte, word). In a paged store, however, it is excessively expensive to monitor access behaviour at this level. The basic unit of transfer of data between secondary and primary storage and of virtual to physical address mapping is the virtual page. Accordingly, a virtual memory system is required to maintain access behaviour knowledge at, and hardware support is optimised to, the virtual page level. It is prudent, therefore, to use the same granularity in determining inter-object dependencies. Such dependencies, while detected at the virtual page level, result in dependencies at the objects[1] level. Thus dependency information either is maintained at the object level, in object-based stores, or is maintained at the client level, in client-server models of the store. Dependencies are used to control checkpoint and roll-back operations and may be represented using a set called an *association* [13].

## 3.1 Describing dependencies using associations

As defined for Casper, associations are sets of dependent clients (processes) [13]. After it has been checkpointed, a process belongs to an association of which it is the only member. Over time processes access pages which have been modified by other processes causing their respective associations to merge. To ensure logical consistency it is necessary to checkpoint all members of an association together in an atomic operation. If such a checkpoint operation fails, or a system failure occurs, all members of an association roll back by reverting to their last stable states. The use of associations guarantees that such reversion results in processes with no inconsistent inter-relationships. It is apparent, however, that the lack of the consideration of behaviour of operations in describing dependencies often results in unnecessarily large checkpoint and roll-back operations. Moreover checkpoint operations are expensive and roll-back operations may result in unnecessary loss of data modifications. The use of directed graphs reduces the extent of checkpoint and roll-back operations.

## 3.2 Describing dependencies using directed graphs

The efficiency of checkpoint and roll-back operations can be improved by reducing the cascade of such operations. This can be achieved by separately representing the checkpoint and roll-back dependencies between entities and also exploiting operation behaviour to represent inter-entity dependencies. When a process reads a modified page of an object, its state will depend on the state of the object. Any write operation on a page of an object makes the object and the writer process dependent on each other.

By considering read and write as the two main operations on pages of objects, dependency can be expressed according to the interaction between processes and pages of objects. Operations which may be performed on a page by a process can be categorised as one of the following:

1) A process may read an unmodified page of an object. This results in no dependency between the reading process and the object.
2) A process may read a modified page of an object. This results in a unidirectional dependency between the reading process and the object. This is because the data which is read is unstable at the time of the read operation. The direction of the dependency depends on the type of cascadable operation (checkpoint or roll-back) which propagates to other entities. Only the checkpoint of a process P which has read a modified page of an object O should propagate to O, not vice versa. Similarly a roll-back of O would result in an inconsistency with P and thus necessitates a roll-back of P; a roll-back of P does not affect O. Thus, the direction of dependency in the case of checkpoint is from P to O, while it is from O to P in the case of roll-back.
3) A process may modify a page of an object. This results in a pair of unidirectional dependencies between the modifying process and the modified object (i.e. the process is dependent on the object and vice-versa).

Directed graphs are used to represent such dependencies [9]. The $\rightarrow$ edge is used to specify the dependency 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 $E_1$ depends on $E_2$ ($E_1 \rightarrow E_2$) then $E_2$ does not necessarily have the same relationship with $E_1$. The relationship $E_1 \rightarrow E_2$ is established if $E_1$ reads modified data from $E_2$. Write operations lead to a pair of dependencies; instead of indicating two unidirectional arrows ($E_1 \rightarrow E_2$ and $E_2 \rightarrow E_1$), the notation $E_1 \leftrightarrow E_2$ is used. We refer to the resultant graph as a directed dependency graph or DDG.

Each entity is associated with one and only one DDG. After creation of an entity and after each checkpoint for the entity, its corresponding DDG contains the entity itself as the root and the only vertex. A DDG is extended and merged with other DDGs as objects are modified or accessed. The creation and maintenance of DDGs is integrated into virtual memory management and is achieved either eagerly or lazily. In eager management of dependencies, the operating system kernel updates DDGs as soon as a clean page is modified or a modified page is accessed. Lazy management of DDGs delays recording of dependencies and requires some hardware support.

[1] As mentioned earlier, we suppose that objects (as used in this discussion) are page aligned.

## 4 Kernel support for dependency detection

As mentioned above, modifying a page or accessing a modified page results in dependencies between entities. To record dependencies, it is required to detect which virtual memory pages have been modified by some process since the last checkpoint. Completion of access to an object by a process may require further accesses to system-related data such as the disk directory and the table of free disk blocks. Identification of such accesses at the time of accessing the pure data would cause an unacceptable deterioration in system performance and thus we do not consider them explicitly. Instead, we implicitly include some system-related objects with all DDGs as described in section 7.

There are two cases to consider concerning construction of DDGs: pages not in main memory and pages which are memory resident at the time of access. Pages which are not in main memory and have not been modified since the last checkpoint are of no interest. Pages which are not in main memory and have been modified (and subsequently discarded to disk) can result in dependencies. Accessing of such pages results in a page fault and subsequent page retrieval. As a part of handling the page fault, it can be determined that the page has been modified since the last checkpoint and that therefore, the access should result in a new dependency. Pages resident in main memory at the time of access can be classified as belonging to one of the following groups.

1) Clean/unmodified pages which have remained in main memory from previous time-slices.
2) Dirty pages which have been modified in previous time-slices and have remained in main memory or have been modified in this time-slice.
3) Clean/modified pages which have been modified and subsequently flushed to disk prior to this time-slice.

We assume that conventional address translation hardware supports mechanisms to apply access protection over pages. This is typically provided through the allocation of two status bits (read-only and dirty) per entry in the hardware-supported Address Translation Unit (ATU). We use these facilities to determine whether a memory-resident page is modified. For pages not in main memory, a separate mechanism is required to specify if a page has been modified since the last checkpoint.

Corresponding to the above groups of pages in main memory, detection of dependencies are discussed.

Access to clean/unmodified pages is unimportant for the management of dependencies. Modification of such pages results in a write fault exception; the page is then marked as a dirty page in main memory.

Access to dirty pages inherited from previous time-slices and even modification of such pages does not result in any kernel interference. This means that no record is made of a dependency between the current process and the object containing the page. Suppose that a virtual page $V_1$ was modified by a process $P_1$ in a previous time slice and it has not been discarded from main memory. The page has remained in main memory as dirty. Access by the current process ($P_2$) to such a page, as well as its modification, is achieved without raising of any access violation or page fault. As a result, the dependency remains undetected and this potentially results in an inconsistent state. A possible solution is to force all dirty pages to be discarded as a part of a context-switch operation. Further access to such pages in later time-slices would then result in page faults so that dependencies could be managed. This is, however, inefficient as a considerable amount of CPU time is consumed for unnecessary page swapping.

An optimisation to this approach is to invalidate all dirty pages in the ATU as a part of context-switch operations. Further access to such pages in subsequent time slices results in a *dummy page fault* which requires no page retrieval. Handling of a dummy page fault includes only validating the entry in the ATU and re-mapping the page in as read-only. Management of dependencies can also be achieved as a part of handling the dummy page faults. Conventional architectures which utilise Translation Lookahead Buffers (TLBs) [5], have been equipped with such facility, as all entries in a TLB are usually cleared as a part of context-switch. In architectures without such a facility (e.g. Monads [1, 11]), invalidation of dirty pages during a context-switch may be achieved through a simple loop over dirty pages.

Access to dirty pages modified in the current time-slice is unimportant as the dependency would already have been recorded in the time slice.

Clean/modified pages are normally mapped in main memory as clean read-only. Conventional computers typically consider a page as dirty only if it has been modified since the most recent time it was mapped into main memory. Later modification of such pages results in write faults and thus detection and record of dependencies can take place. If no modification occurs for pages which were modified on previous occasions when they were in main memory, all read accesses on the pages will be considered incorrectly identical to accessing of clean/unmodified pages and therefore the dependency between the process and the modifier becomes transparent. This problem is due to

- the lack of difference between disk blocks containing modified and unmodified virtual pages in typical computer systems which overwrite a modified page on its original disk location at discard time, and also

- the lack of difference between unmodified pages and modified/discarded pages prior to this time-slice, in main memory.

To overcome the problem, it is required that the virtual memory management software be aware of modified pages. Virtual memory page table(s) used to locate pages for loading into main memory can be extended to indicate whether non-memory-resident pages have been modified since the last checkpoint. Such an extension has been already provided in computer systems which implemented shadow paging (e.g. [12]) as their method of stability. In Monads, a separate data structure has been used to determine if a page has been shadowed (modified) since the last checkpoint. We use this feature to manage dependencies in both the eager and lazy methods of dependency management described in sections 5 and 6.

In handling a page fault, the table of modified pages is consulted; if the page has been modified, an edge is inserted in the DDG. Then the page is retrieved and mapped into the ATU as dirty/read-only. Mapping a modified page into the ATU as dirty has the disadvantage of allowing unnecessary page discards.

Assuming the kernel support for dependency detection, dependencies may be recorded eagerly.

## 5 Eager dependency graph construction

In eager construction of DDGs, a DDG is modified as soon as the kernel realises that a new dependency is created. In terms of the possible operations which a process may perform on a page described in section 3.2, a DDG grows or shrinks according to the following criteria.

- When a process $P_1$ reads a modified page of an object $O_1$, the edge $P_1 \rightarrow O_1$ is inserted into the DDG.
- When a process $P_1$ modifies a page of the object $O_1$, the edge $P_1 \leftrightarrow O_1$ is inserted into the DDG.
- When a process in one DDG reads some modified page of an object or modifies an object which is associated with another DDG, the two DDGs are merged using one of the above edges to create a single larger DDG.
- A DDG shrinks when a set of dependent entities is checkpointed or entities revert to their last stable state.

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

We illustrate the construction and reduction of a DDG in figure 1. 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 ($P_1$, $P_2$, and $P_3$) are accessing four objects ($O_1$, $O_2$, $O_3$, and $O_4$). To demonstrate access to different pages of an object, we also assume that $O_1$ and $O_2$ have two pages each, $O_3$ has one and $O_4$ has three pages. Processes are shown by circles in the figure, while objects are shown by a set of pages; each page is represented by a blank (unmodified) or shaded (modified) rectangle. Each object with at least one modified page is considered to be a modified object. For simplicity, we do not consider system-related information maintained on a per object basis. Figure 1(a) depicts the resultant DDG.

Figures 1(b) and 1(c) depict the effect of using DDGs on the propagation of checkpoint and roll-back operations. The checkpoint of $P_1$ in the resultant DDG only propagated to $O_1$ and $O_2$. The roll-back of $P_3$ in the resultant DDG only propagated to $O_3$. As read operations outnumber write operations in a typical computer system [4], the described DDG is normally populated by unidirectional edges. This results in a considerable portion of the DDG being unaffected by the propagation of checkpoint and roll-back operations resulting in improvement in the system performance. Our simulation results confirm this claim.

## 6 Lazy dependency graph construction

By lazy construction of DDGs, inter-entity dependencies made in a time slice are only recorded during the process switch at the end of the time slice. Accordingly, it is necessary to record the accesses and modifications performed by a process during its latest activation. This facility is not supported by conventional address translation units. To enable lazy construction of DDGs, it is necessary to be able to detect:

1) which virtual pages have been modified by some process since the objects containing the pages were last checkpointed,
2) which virtual pages have been accessed in this time-slice by the currently executing process, and
3) which virtual pages have been modified in this time-slice by the currently executing process.

The first requirement is identical to that for eager construction of DDGs and may be provided with no special hardware support. Nevertheless, support of hardware to detect modified pages results in an efficiency improvement in both methods of DDG construction.

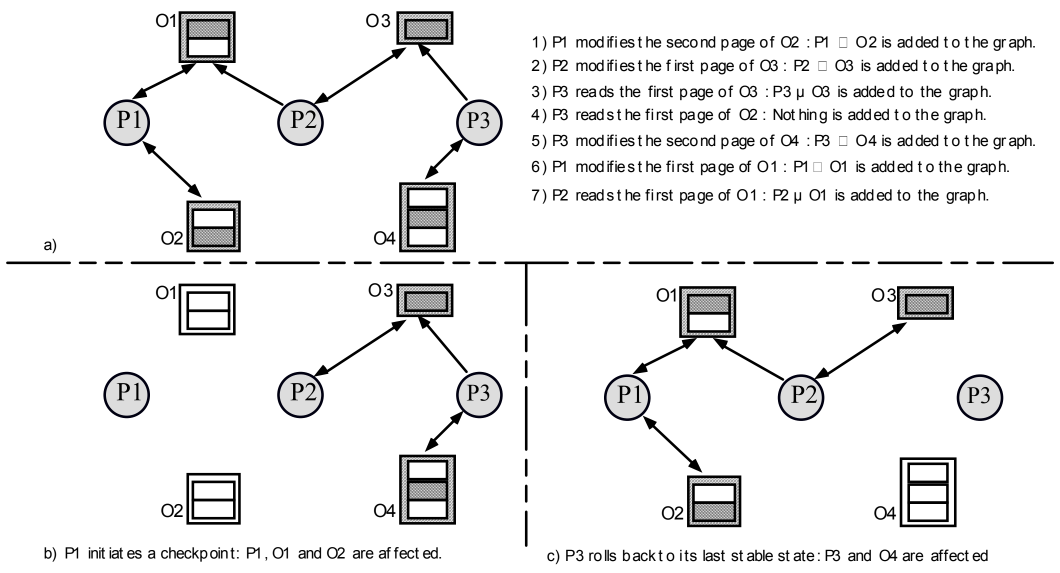
O1
O3
P1
P2
P3
O2
O4
a)
1) P1 modifies the second page of O2 : P1 □ O2 is added to the graph.
2) P2 modifies the first page of O3 : P2 □ O3 is added to the graph.
3) P3 reads the first page of O3 : P3 µ O3 is added to the graph.
4) P3 reads the first page of O2 : Nothing is added to the graph.
5) P3 modifies the second page of O4 : P3 □ O4 is added to the graph.
6) P1 modifies the first page of O1 : P1 □ O1 is added to the graph.
7) P2 reads the first page of O1 : P2 µ O1 is added to the graph.
b) P1 initiates a checkpoint: P1, O1 and O2 are affected.
c) P3 rolls back to its last stable state: P3 and O4 are affected

**Figure 1** Construction and reduction of DDGs.

The requirement of specifying modified pages should not be confused with the ability to identify dirty pages which is essential to virtual memory management. Conventional architectures typically provide that ability through the implementation of a dirty bit per entry in their ATU. On page discard the dirty bit is queried and accordingly the page-frame is immediately re-allocated if clean, or is flushed prior to re-allocation. Such dirty bits are used in exactly the same way for management of the proposed store.

A *modified* bit per entry in the ATU is proposed to indicate that the contents of this page frame have been modified by some process since the object containing the page was last checkpointed. As described in section 3, subsequent access by another process to such a page creates a dependency situation involving the object containing the page and the modifying or accessing process. The modified bits for the pages of an object are, of course, cleared when the object is checkpointed.

Determination of dependencies without this bit involves using the dirty bit for two purposes:

1) for virtual memory page discard decisions, and
2) to detect subsequent accesses to modified pages.

This is inefficient because a dirty page which is discarded as part of virtual memory management and later retrieved for read access will be flushed again on its next discard or when its object is checkpointed. It is thus recommended that two bits be implemented, one for each function.

The implementation of the modified bit requires that the virtual memory management software distinguishes between non-resident modified and unmodified pages as described in section 4. The page table is used to retrieve pages and extra information which is used to appropriately set the modified bit when the page is mapped in to the ATU. The ATU dirty bit for the page is not set, ensuring that the page may be later discarded without being flushed to disk (unless, of course, it is subsequently further modified). Subject to the same caveat the page will not be flushed when its object is next checkpointed.

There are two additional hardware features which can be provided to improve the efficiency of construction of DDGs by allowing them to be updated once per process time-slice. These are the *m_accessed* bit and the *written* bit.

Pages may remain in main memory for a period encompassing many process activations. The allocation of the m_accessed bit per entry in the ATU allows detection of a process accessing modified object data during the current time slice. This bit is set for a page if the page is accessed while the modified bit for the page is set. Dependencies between a process and the objects containing pages with the m_accessed bit set, are represented by the addition of appropriate $\rightarrow$ edges to the DDG at the conclusion of the process' period of activation. All m_accessed bits must be cleared at the commencement of a process time slice; this may be achieved in a single operation using appropriate hardware.

The inclusion of a written bit per entry in the ATU allows detection of object data modifications made by the current process. This bit is distinct from the modified bit described above because it describes the modification behaviour of the current process only rather than the status of the virtual page itself. The written bit is set together with the modified and dirty bits, but is cleared as part of the DDG update at the conclusion of the process time slice. In contrast the modified bit is cleared at the next object checkpoint and the dirty bit is cleared when the page is flushed to disk. Pages with the written bit set cause the inclusion of an appropriate $\leftrightarrow$ DDG edge.

Lazy construction of DDGs is more efficient than eager construction. A process may read a modified page and later on modify it. In eager construction of DDGs, this results in two modifications of the DDG. First an edge is inserted into the DDG and then the edge is upgraded. Lazy construction of DDGs reduces this to the insertion of only one (upgraded) edge. Nevertheless, with the lazy method of constructing DDGs, initialisation of a checkpoint or roll-back operation in multi-processor machines potentially misses some unrecorded dependencies which have been created since the start of the respective current time-slices. Discard of a page whose corresponding m_accessed or written bit is set, is not an issue for single-processor machines as a page discard is synchronised with process activation. A multi-processor computer allows several processes to execute simultaneously, and thus page discard may occur in parallel with process activation. We described this in [7].

M_accessed and written bits are proposed per page in main memory. These in fact are required per object. Support of such bits per object requires further data structures and mechanisms to be provided by the address translation unit. Nevertheless, having these bits per object facilitates invocation of the dependency manager once per object instead of per page.

## 7 Critical Objects in DDGs

As mentioned earlier, each normal entity belongs to one and only one DDG. This results in autonomy of DDGs in terms of checkpoint and roll-back. However, there are special entities which belong to more than one DDG.

The kernel in each node can be considered as an object including data structures, manipulated frequently to record the system state. Accordingly, the current user process and the kernel are made dependent on each other. This in turn leads to the dependency of all entities on the kernel and vice versa. For simplicity, we refer to the entire kernel as a single entity which belongs to all DDGs.

Due to the lack of a one-to-one correspondence between virtual address space and physical disk address space, a mapping is required to translate object identifiers into their physical addresses. We assume that a mapping table exists per disk to perform such mapping for all objects on the same disk. We refer to this mapping table as the disk directory and allocate a special object with a well-known address per disk. The object is called the *root object* for the disk and contains all information required for management of the disk, including the disk directory and the disk free-list. Each access to an object located on a disk requires some references and probably modification to the root object. For example, any disk page allocation requires the modification of the disk free-list.

Each object necessarily depends on the root object of its accommodating disk and vice versa. Therefore, each network-wide DDG may contain one or more kernel entities and also one or more disk root objects. Disk root objects and kernel entities are called *critical objects* as they can broadcast each checkpoint or roll-back operation through the whole store if they are considered as normal entities.

To improve efficiency, however, critical objects are considered differently from normal entities and are not necessarily included in DDGs. They are considered as permanent entities of each DDG and are restricted in propagation of operations. Critical objects act as obstacles in propagation of checkpoint and roll-back operations. Otherwise at each point in time all entities would belong to the only DDG in the system.

As an example, consider the three DDGs spread over two disks $D_1$ and $D_2$ shown in figure 2. All DDGs on a node depends on the kernel object and also all DDGs with entities belonging to a disk depend on the disk root object. To provide autonomy of DDGs, critical entities are assumed to have an instance per DDG. Moreover, critical objects should be checkpointed such that the possibility of the roll-back of other existing non-stable DDGs is guaranteed.

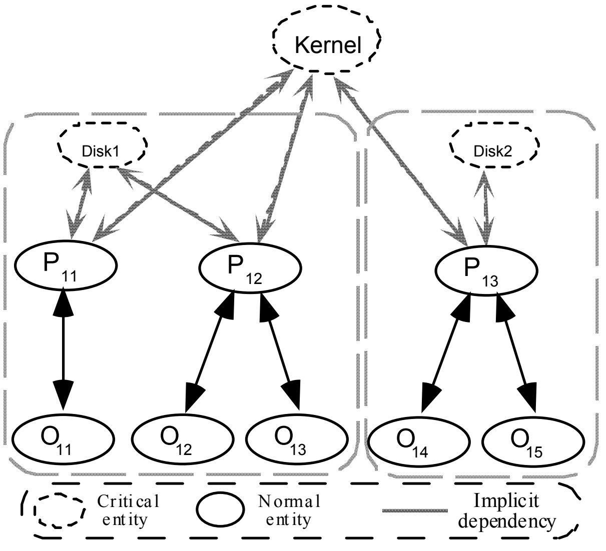

**Figure 2** Dependency of DDGs on critical entities.

## 8 Conclusion

Stability of persistent object stores may be achieved by checkpointing dependent entities together. Dependencies between entities are created during processing of the data held in the store, and may be recorded using directed graphs. It has been shown that different dependencies are created by read and write accesses to data. Distinguishing modification of a page from accessing a modified page allows a reduction in the extent of checkpoint and roll-back operations.

Maintaining dependency information requires operating system intervention to detect and update dependencies. This service is integrated with virtual memory management and utilises protection mechanisms provided by the ATU. Recording dependencies as soon as they happen is possible using the conventional ATU services, but it is not very efficient. By the provision of further support in the ATU to determine modified pages, modified-accessed pages and written pages in each time-slice, it is possible to lazily record dependencies at the end of each time-slice.

The techniques described in this paper have been evaluated by simulation and shown to result in significant stability-related performance improvements. Subsequently a new version of the Monads architecture which incorporates hardware support for the construction of directed dependency graphs has been designed and is currently being implemented [7].

## Acknowledgment

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

## References

[1] Abramson, D. A. and Rosenberg, J. “The Micro-Architecture of Capability-Based Computer”, *Proceedings of the 19th workshop on microprogramming (MICRO.19)*, pp. 138-145, 1986.

[2] 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.

[3] Challis, M. F. “Database Consistency and Integrity in a Multi-User Environment”, *Databases: Improving Usability and Responsiveness*, Academic Press, pp. 245-270, 1978.

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

[5] Hennessy, J. L. and Patterson, D. A. “Computer Architecture: A Quantitative Approach”, Morgan Kaufmann Publishers, Inc., 1990.

[6] Henskens, F. A., Brossler, P., Keedy, J. L. and Rosenberg, J. “Coarse and Fine Grain Objects in a Distributed Persistent Store”, *Processings of the Third International Workshop on Object Orientation in Operating Systems*, IEEE Computer Society Press, Ashville, North Carolina, 1993.

[7] Henskens, F. A., Koch, D. M., Jalili, R. and Rosenberg, J. “Hardware Support for Stability in a Persistent Architecture”, *Preceedings of the 6th International Workshop on Persistent Object Stores*, Tarascon, France, pp. 381-393, 1994.

[8] 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.

[9] Jalili, R. and Henskens, F. A. “Reducing the Extent of Cascadable Operations in Stable Distributed Persistent Stores”, *In Proceedings of the 18th Australian Computer Science Conference (ACSC’95)*, to appear, Adelaide, Australia, 1995.

[10] Lorie, R. A. “Physical Integrity in a Large Segmented Database”, *ACM Transactions on Database Systems*, 2(1):91-104, 1977.

[11] Rosenberg, J. and Abramson, D. “Monads-PC: A Capability-Based Workstation to Support Software Engineering”, *Proceeding of the 18th Annual Hawaii International Conference on System Sciences*, pp. 222-231, 1985.

[12] 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.

[13] 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.