JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 6, July-August 2007

Graph-based Optimistic Transaction Management

Frans A. Henskens, University of Newcastle, N.S.W., Australia
Maurice G. Ashton, Avondale College, Cooranbong, N.S.W., Australia

Abstract

In this paper, we introduce and describe directed dependency graph-based transaction and concurrency control (DCC) for persistent (stable, single-level) object-based bulk data management systems. The new technique is optimistic and applicable across a wide range of store sizes, transaction sizes and multi-programming levels. It is also has potential for use in management of transactions in other contexts, for example web services.

1 INTRODUCTION

The use of directed graphs as a data structure for control of transaction-based concurrency control in object stores was motivated by the authors' observation of the similarity between the objectives of *stability* [3, 16, 32] in persistent object stores [2, 7, 29, 33] and of transaction management [6, 14, 15] in conventional database management systems. Previous work [16, 19] describing the use of directed graphs to maintain information about the inter-entity dependencies created during program activity in an object store appeared to provide the basis for the satisfaction of a similar need for transaction-based systems. This paper presents a directed-dependency-graph-based transaction mechanism for persistent object-based bulk data management systems.

2 PERSISTENT SYSTEMS

The descriptor, *persistent systems*, is applied to a wide range of systems offering a variety of features. Such systems are described and classified in [4, 27]. The work described in this paper is based on persistent systems that offer the following features:

1. Data structures and relationships are, by default, retained in their original form beyond the lifetime of the program or programs that created them.
2. Address spaces and processes are orthogonal. Processes execute in a logically single level object store. This addressing environment is typically shared by a

Cite this article as follows: F. A. Henskens and M. G. Ashton: "Graph-based Optimistic Transaction Management", in *Journal of Object Technology*, vol. 6, no. 6, July-August 2007, pp. 131-148 http://www.jot.fm/issues/issue_2007_07/article4

number of processes and consequently the normal protection mechanisms offered by conventional virtual memory management techniques are not applicable. Rather, protection is provided through the use of mechanisms such as capabilities [13].

Systems that provide such an environment include Monads [20-22, 30] and Grasshopper [11, 12, 25].

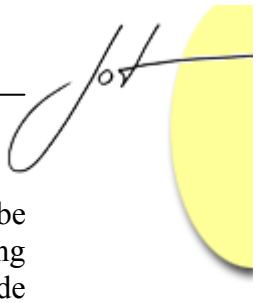
In a system where multiple concurrent processes execute in a (logically single-level) object store, and in the absence of any protection or concurrency control mechanism, any process may interact with the objects by either querying (reading) or mutating (writing) them. As previously described (e.g. in [19, 31]), objects in the store become dependent on each other through the activities of the processes.

3 STABILITY

Computational and file system memory use different mechanisms for naming objects, constructing/describing data structures and controlling access to the data stored in it. In particular, an object in computational memory is identified using its virtual memory address, and that address is in turn used as the reference or link to incorporate the object into a complex data structure such as a tree or list. This virtual address is meaningless when the object is copied into the file store and needs to be replaced by a symbolic link in a process known as *flattening* the structure. Persistent systems automate the movement of data structures between computational memory and the file system, performing link translation as required. For example systems such as Napier88 [28] identify and replace references as objects move between stores, while others such as Monads [23] use a single-level store in which the location of an object remains static over its lifetime.

When a system shuts down, the only information available on restart is the information that is available from durable storage. In a planned shutdown, data is transferred to the disk in an orderly manner and a consistent start-up state is thus maintained. In the event of a failure, such an orderly shutdown may not be possible and special techniques are necessary to ensure there is sufficient data stored on permanent storage to allow the system to start up in a self-consistent state. Conventional systems provide utilities such as fsck to rebuild system data structures that have been corrupted by failure and uncontrolled shutdown.

Failures present a special challenge for persistent systems because inconsistencies such as unreachable objects and dangling pointers can easily render the system useless. Persistent systems that can always find a consistent state from which to restart are said to be *stable.* In a single-level, shared persistent store, stability is of vital importance since the storage space is also computational space, not isolated from it as occurs with conventional systems. Additionally, user activity (and therefore failure-induced corruption of the store's representation of that activity) is not isolated from other store users so lack of consistence potentially affects all users.



Stability is implemented by ensuring that a consistent state can always be constructed using data stored in a durable medium (e.g. disk). This is achieved using techniques borrow heavily from those used in the database world, for instance to provide atomicity of durable store update (e.g. [9]) and parallel support for virtual memory discard and shadowing (e.g. [7]). Terminology has also been borrowed, with the term *checkpoint* being used to describe the act of stepping the durable store image between consistent states, and *rollback* being used to describe the act of reverting the durable and computational store to some previously-stored consistent state.

Stability Techniques

Stability in persistent systems is typically achieved using the *checkpoint and rollback recovery* technique [1]. The mechanisms developed for persistent systems use, for example, logging [8, 15, 17] and shadow paging [26] and may be classified as

1. Stop the world [7],
2. Incremental using *associations* [35], or
3. Incremental using *directed dependency* [18].

The easiest technique to implement is the *stop the world* approach, whereby all user activity on the store is suspended and the entire store is checkpointed at once (practically, only data that has been mutated since the previous checkpoint must be written to disk since a disk image will already exist for non-mutated data). Stop the world impacts badly on users, particularly if large amounts of mutated data are checkpointed at once.

An alternative is *incremental checkpointing* by which parts of the store are checkpointed in parallel with user access to other parts of the store. To ensure consistency, information about previous process activity, in particular the dependencies that activity has created, must be used to determine the extent of checkpoint operations. Ideally, dependencies would be detected and recorded at the object level, but this is usually not efficiently possible in practice so the virtual page granularity is typically used.

The *association* approach to maintenance of dependencies [36] maintains page sets (called associations) on a per-process basis. Each time a process accesses a modified page of the virtual space (either as mutator or reader), that page's identity is added to the process' page set. If a process accesses a page that already belongs to another process' page set, the sets are merged. Consistency is maintained by ensuring that associations are atomically checkpointed or rolled back. The use of associations can result in creation of false dependencies, resulting in larger-than-necessary checkpoint or roll back operations. In the case of checkpoint the extra size impacts on performance, and of roll back results in unnecessary undoing of achieved work through the domino effect [34].

The occurrence of false dependencies is alleviated through the description of dependencies using directed dependency graphs (DDGs) [16] as described in the next section.

Use of Directed Graphs

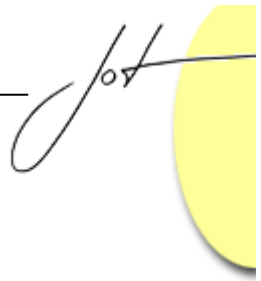
The critical observation of the DDG work was that schemes such as associations incorrectly assumed a bi-directional relationship between processes and objects. In other words, that a connection between a process and an object had the same meaning for checkpoint as for rollback. For example, consider processes P_1 and P_2 whose activities are linked through a common object O. If P_1 mutates O, after which O is read by P_2, then according to the association scheme all three entities would checkpoint or roll back as a unit (and all other objects associated with each process would also be affected). In fact P_1 and O could checkpoint independently of P_2, and P_2 could roll back independently of P_1 and O. Only a checkpoint of P_2 must propagate to P_1 and O.

This may be recorded using graph rather than set notation using the directed edges $\rightarrow$ and $\leftrightarrow$. When a process P modifies an object O, the edge $P \leftrightarrow O$ is added (if it does not already exist) to the DDG(s) including P and O. When a process P reads a modified object O, the edge $P \rightarrow O$ is added (if it does not already exist or if an $\leftrightarrow$ edge does not exist) to the DDG(s) including P and O. As implied, when a process belonging to a DDG reads a modified object or modifies an object that belongs to another DDG, the two DDGs are merged using one of the described edges to create a single larger graph. An $\rightarrow$ edge represents both $\overset{S}{\rightarrow}$ (i.e. in terms of checkpoint dependency) and $\overset{R}{\leftarrow}$ (i.e. in terms of roll back dependency). Thus $E_1 \rightarrow E_2$ implies that checkpoint of E_1 propagates to E_2 (but checkpoint of E_2 does not propagate to E_1) and that rollback of E_2 propagates to E_1 (but rollback of E_1 does not propagate to E_2). The $\leftrightarrow$ edge represents a union of $\leftarrow$ and $\rightarrow$ edges. A consequence of this is that if $E_1 \leftrightarrow E_2$, checkpoint and rollback of either entity propagates to the other.

A DDG shrinks when a set of dependent entities is checkpointed or reverts to its last stable state (rolls back). Once a checkpoint or rollback operation is initiated for an entity E, the operation propagates to each entity that is reachable from E in the DDG to which E belongs. Then, because each involved entity is now stable, all edges attached to them are removed.

At any instant each entity belongs to one and only one dependency graph. To find the set of entities dependent on any entity, it is sufficient to find the location of the entity in its graph and then, subject to the kind of operation, traverse the directed graph starting from the entity. Thus the set of dependent entities may differ for entities in the same DDG.

As described in [16], relevant process activity can be recorded by lazily adding new entries to the DDG as part of the system activity that achieves a process switch at the end of each process time slice.



4 TRANSACTIONS

Database management systems (DBMSs) use transaction mechanisms [10] to support the consistency of their managed data. A transaction is a logical unit of work that groups multiple physical operations into a single operation. From the user's point of view, a transaction is the unit of communication with a database. The user describes the operations to be included in the transaction and passes that transaction as a request to the DBMS. The transaction may be seen as a pair of "brackets" enclosing the required operations, indicating that they are to be performed as a collective unit. It is the responsibility of the user or application programmer to position these brackets so that they accurately represent or express the intended change to the universe of discourse modelled by the database. On the other hand the DBMS is responsible for the correct execution of a transaction, carrying out the intent of the user as represented by the user's program code [24]. In summary, transactions provide for database operations the ACID (atomicity, consistency, isolation, durability) properties required for correct, concurrent access and are used as the basis for both recovery and concurrency control.

Concurrency control mechanisms are classified as either *pessimistic* or *optimistic*. The pessimistic approach is predicated on the assumption that action must be taken for each and every data access to prevent breaches of transaction isolation. On the other hand the optimistic approach allows data access to occur in the absence of preventative action, and later performs checks for violation of transaction isolation (often immediately prior to transaction commit).

Stability and Transactions

DDG-based stability mechanisms for persistent systems appeared to offer some of the features required for database management in single-level persistent stores, namely:

1. A mechanism for recovering a consistent state in the event of failure. This mechanism appeared to satisfy the recovery aspects of database operation.
2. A mechanism for recording dependencies between processes that had the potential to be used for determining the isolation of transactions.

It may appear that the facility provided by DDG-based stability is all that is needed to provide transaction support in persistent systems. This is not so: while stability systems use terms such as checkpoint and rollback in common with conventional DBMS transaction systems, their meaning is different. Stability schemes provide durability and a level of atomicity and consistency, but they do not attempt to provide isolation. Additionally, stability is typically system-initiated, whereas transactions are defined in user-level software. Indeed it has been claimed that the very existence of transactions is incompatible with the principles of stability [5].

5 STABILITY-BASED TRANSACTIONS

The graph-based stability mechanism developed by Jalili [18] uses directed graphs to record dirty-read (read access to unstable mutated data) and write dependencies formed between processes and objects. This information is recorded in directed dependency graphs (DDGs) and used by the stability mechanism to determine the extent of checkpoint and rollback operations. There are similarities in the information recorded in stability DDGs and the information required for transaction management and concurrency control. However, there are also differences between stability and concurrency control requirements, namely: the DDG stability technique maintains dependency information on a per-process basis rather than a per-transaction basis; the DDG stability technique records information about dirty-read and write accesses, whereas transaction isolation also requires knowledge of clean-read accesses; stability checkpoints and transaction commits have different semantics. The effects of these differences are described in the following subsections.

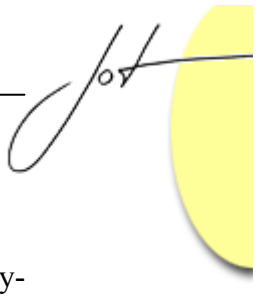
Transactions and Processes

A transaction is an abstract concept that includes the user-defined boundaries (BEGIN_TRX and COMMIT), the required data resources and accesses (that may include mutation) to that data. Processes are entities that execute the activities specified by transactions. A transaction may use a single process, a group of processes, or share a process (or processes) with other transactions to execute its activities. A process executing on behalf of many transactions executes instructions for only one transaction in any given timeslice [14].

Because of this relationship between transactions and processes, dependencies created by transaction activity are appropriately viewed as existing between transactions and other entities. Thus at the end of every process timeslice it is necessary to record the process-object dependency, including the identification of the transaction that used the process.

Read Accesses

For a transaction to be considered isolated it must be guaranteed that the transaction has not seen an inconsistent state of the data. By way of illustration, consider two concurrent transactions T_a and T_b. Transaction T_b modifies a set of entities E. Transaction T_a reads some of these entities before T_b has modified them and some of them after. This represents an inconsistent view of the data for T_a, compromising its isolation. In summary, if, during the time that transaction T_a is in progress, some other transaction T_b modifies a set E of one or more entities, then T_a is not compromised by T_b provided that when T_a reads from E, these read operations occur either before T_b's writes, or after T_b's writes, but not both. Thus for DDGs to be used for determining transaction isolation it is necessary to extend the recorded information to include transactions' clean-read accesses.



Stability Checkpoint vs Commit

Stability mechanisms provide the abstractions: a durable computational store; a logically-consistent store restart state at all times; concurrency control at process level. Full transaction support requires these properties to be augmented as follows:

1. Support for transaction-based events associated with the programming language key words (e.g. BEGIN-TRX and COMMIT-TRX) used to define the extent of each transaction.
2. The transaction extent defines an atomic unit of work that is isolated from any other concurrent activity.
3. The means for managing concurrency should be flexible enough to cope with run-time determination of the temporal extent and physical granularity of interaction.

A consequence of these requirements is that the transaction management system must have control over the timing of checkpoints that correspond to transaction commits.

6 GRAPH-BASED CONCURRENCY CONTROL

This section describes how the graph-based notation used to maintain dependency information required for stability is extended to provide support for transaction management and hence for transaction-based concurrency control.

Graph-based Representation

The DDG-based transaction manager creates edges between graph nodes representing transactions and accessed entities as follows:

1. A clean-read edge is recorded as "—". T — E indicates that transaction T has read an unmodified entity E.
2. A dirty-read edge is recorded as "→". $T \rightarrow E$ records that process T has read an entity E that had been previously modified since its most recent checkpoint.
3. A write edge is recorded as "↔". $T \leftrightarrow E$ indicates that process T has modified entity E since it was last check-pointed.

Transaction concurrency control is incorporated into the existing stability system as follows:

- At the commencement of a transaction the initiating process must exist in a single-node DDG. To achieve this it may be necessary for the process to initiate a checkpoint operation, with isolation being the consequence. The process is then part of a DDG associated by the system with the fledgling transaction and is under transactional control.
- As the process (and any parallel processes incorporated in the transaction) interacts with entities in the store, →, ↔ and — edges are used to incorporate the entities into the transaction DDG. Construction of the graph is achieved lazily on process switch using access data collected during each process timeslice.

- Edges have a precedence order —, →, ↔ with the rule that insertion of an edge to the right in this order will replace an edge to the left. An edge to the right will not be replaced by an edge to the left; indeed, an edge to the left will not be inserted if it occurs after an edge to the right.
- If there are no existing edges between any transaction node and the accessed entity node, the appropriate edge is added and the entity belongs to (and becomes a node in) the same DDG as the transaction.
- If all prior edge(s) between other transaction nodes and the node representing the accessed entity are to nodes in the same DDG as the process, the appropriate edge is inserted subject to the precedence rule.
- If one or more edges exist between other transaction nodes and the node representing the accessed entity, and these are to nodes belonging to a different DDG to the current process, the system inserts the appropriate edge.
- During each transaction DDG update, the system analyses any graph merge operations and determines if the merge causes a violation of transaction isolation. Transactions that have violated the isolation rules are immediately aborted.
- A transaction that completes, i.e. whose DDG could be constructed without a need for transaction rollback, commits by checkpointing its transaction DDG.
- A transaction that aborts has its transaction DDG rolled back.

Determining Isolation Using DDGs

In [14] Gray and Reuter define transaction isolation as follows:

"… transaction T is isolated from other transactions if:

0. *T does not overwrite dirty data of other transactions.*
1. *T's writes are neither read nor written by other transactions until COMMIT WORK.*
2. *T does not read dirty data from other transactions.*
3. *Other transactions do not write (dirty) any data read by T before T commits."*

Following on from this definition, a transaction is not isolated if it can be shown that it executes both before and after some other transaction. A transaction with such an execution history is known as a *wormhole transaction*.

In conventional lock-based (pessimistic) concurrency control the appropriate use of locks precludes those events that lead to loss of isolation. In the DDG-based approach prevention of isolation violations is not possible because the graphs are created after the transaction's actions have occurred. Concurrency control in these circumstances must rely on detecting isolation violations, aborting the affected transaction or transactions, and rolling back their actions.

In the following discussion, it is shown that all violations of the isolation rules may be detected by an inspection of a transaction's DDG. Furthermore, a transaction's violation of the isolation rules may be detected before a transaction has completed its intended activities, allowing the transaction to be aborted early, avoiding the execution of unproductive work. Because process activity is retrospectively analysed to determine

possible breaches of isolation, this technique is optimistic (though it differs from conventional optimistic techniques because transactions can be aborted prior to executing all of their component operations).

Edge Insertion Rules

In the following description of DDG edge insertion, a transaction T_a accesses an entity X, creating a new edge. Transaction T_b is another transaction that has already accessed X. Decisions on the transaction's isolation resulting from T_a's edge-producing access are made by considering the edge to be inserted with respect to each individual existing edge between the nodes representing X and each other concurrent transaction T_b. This discussion assumes that the system has already determined that there is no existing edge of higher or equal priority than the new edge between the nodes representing X and T_a.

1. If there is no edge between the nodes representing any T_b and X the new edge is inserted. It is either a clean read or a write edge, as illustrated in Figure 1. An insertion of either a read or write edge in this situation has no effect on the transaction's isolation.

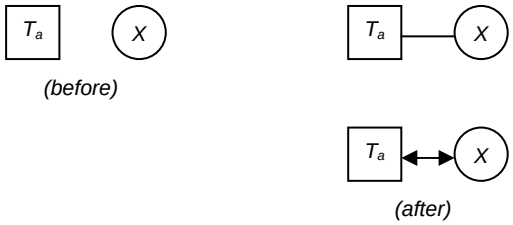


Figure 1: Addition of a new read or write edge.

2. If there is an existing — edge between the nodes representing T_b and X and the access by T_a is a read, a new — edge is inserted between the nodes representing T_a and X, as illustrated in Figure 2. This edge insertion does not affect the isolation of either transaction.

Figure 2: Adding a read edge when there is an existing read edge.

3. If there is an existing — edge between the nodes representing Tb and X and the access by Ta is a write, a new ↔ edge is inserted between the nodes representing Ta and X, as illustrated in Figure 3. The addition of this edge in itself does not preclude either transaction from being isolated. However the action leading to the insertion of this edge creates the potential for a wormhole. This is discussed in Section 6.5.

Figure 3: Adding a write edge where there is an existing read edge.

4. If there is an existing ↔ edge between the nodes representing T_b and X and the access by T_a is a read, a new → edge is inserted between the nodes representing T_a and X, as illustrated in Figure 4. As with the insertion described in (3) the insertion of this edge does not preclude either transaction from being isolated but creates the potential for a wormhole. This is also discussed in Section 6.5.

Figure 4: Adding a read edge where there is an existing write edge.

5. If there is an existing ↔ edge between the nodes representing T_b and X and the access by T_a is a write, an ↔ edge is added between the nodes representing T_a and X as illustrated in Figure 5. This action potentially corrupts the working copy of X and both transactions must be aborted.

Figure 5: Adding a write edge where there is an existing write edge.

Isolation at Commit

As described in the previous section the DDG is analysed as each edge is added to determine the isolation of the affected transactions. At the validation stage a transaction's DDG is finally analysed to determine its isolation

1. Transaction T_a's DDG has no edges connecting to the DDG of any other transaction T_b as illustrated in Figure 6. Transactions T_a and T_b have accessed (and modified) mutually exclusive sets of entities. Such transactions are isolated by definition.

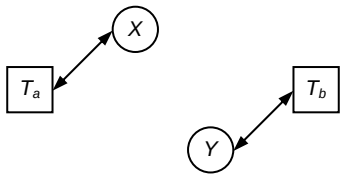


Figure 6: Two isolated transactions

2. Transactions T_a and T_b have read edges to the same entity X as illustrated in Figure **7.** Since neither transaction has modified entity X both transactions are isolated and both may commit.

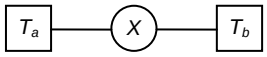


Figure 7: Two transactions read the same entity

3. As shown in Figure 8, transaction Ta has modified an object X that has been clean read by transaction Tb. Transaction Ta is isolated and may commit. However transaction Tb has the potential to access (read or write) X and any other entity modified by Ta after Ta has committed. This represents a wormhole situation. Since prevention is not an option in the persistent system described here, mechanisms must be used to ensure these accesses are detected.

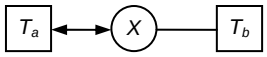


Figure 8: A transaction modifies an entity read by another transaction

4. These inconsistent accesses are detected by adding a blocked edge (denoted ⊣—) between Tb and X where X has been modified by Ta, as shown in Figure 9. A blocked edge is not a dependency edge in the same sense as a dirty-read or a write edge, and does not need to be considered in interactions with other transactions. These blocked edges remain in effect until Tb is committed or rolled back.

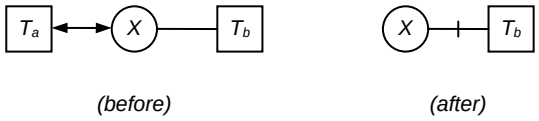


Figure 9: Detecting inconsistent reads after transaction committal.

5. As depicted in Figure 10, a write edge connects the nodes representing Ta and X and a dirty read edge connects the nodes representing Tb and X.

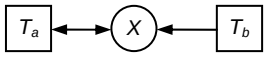


Figure 10: A transaction reads an entity after it has been modified by another transaction

If T_a attempts to commit it may do so. T_b's dirty read becomes a clean read because entity X is now stable (the modified version has been committed).

On the other hand T_b cannot commit because it depends on uncommitted modifications by T_a. There are two possible policies that may be used:

a. Wait until T_a has committed. This policy leaves the T_b open to compromise of its isolation by the actions of other transactions and the possibility that it may later need to be rolled back.
b. Immediately roll back the transaction. This policy is pessimistic in the sense that it precludes the possibility that the writing transaction eventually commits, rendering the waiting transaction isolated and therefore able to commit.

The appropriate policy can be dynamically chosen, using runtime monitoring to measure throughput and suggest the policy based on the prevailing situation.

1. Transactions Ta and Tb both have write edges to X. This situation, shown in Figure 11, should not exist at the validation stage because the situation is always detected during edge insertion and the transactions aborted.

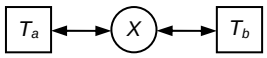


Figure 11: A transaction modifies an entity modified by another transaction

The analysis so far has concentrated on those cases where isolation may be compromised by the single actions of two transactions on a single entity. Isolation may also be compromised where transactions access entities more than once or where two (or more) transactions access a set of entities in common. Such interactions have the potential to form wormholes and thus compromise the isolation of transactions. The following section describes how wormholes may be detected using DDGs.

Wormhole Detection

A wormhole is defined as a transaction that appears to run both before and after another transaction (see Section 6.2). Wormholes appear in DDGs as cycles. For example, a simple cycle may occur when a transaction T_a reads an entity X before and after it is modified by another transaction T_b as illustrated in Figure 12. Transaction T_a is aborted in this situation.

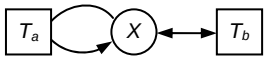


Figure 12: A simple wormhole

Cycles may involve more than two transactions. Consider the sequence of operations:

T_a reads X
T_b reads Y
T_c reads Z
T_a writes Y
T_b writes Z
T_c writes X

This is illustrated in Figure 13.

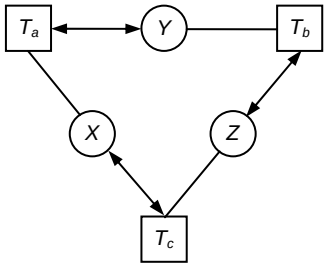


Figure 13: A wormhole represented as a cycle in a DDG

This cycle may be resolved by rolling back one of the transactions in the cycle allowing the other transactions to continue, subject to resolving their other dependencies.

A more costly cycle occurs where transactions become deadlocked as illustrated in Figure 14

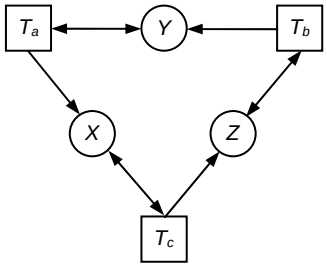


Figure 14: A deadlock cycle

Two or more transactions waiting for one another to commit in a cycle form a deadlock situation. The DDG management system detects the formation of such cycles and aborts sufficient transactions to break the deadlock. There is the potential to form very large cycles of deadlocked transactions, resulting in a consequent loss of work and performance when they are rolled back. However, during the simulation experiments carried out by the authors to evaluate the DDG-based transaction technique the largest observed cycle involved two transactions. This observation supports the view that such cycles are rarely large [14].

Transaction Rollback

When a transaction aborts, its rollback removes the DDG edges between the rolled-back transaction and the entities it has modified. Other in-progress transactions may have formed dependencies on a rolled back transaction by optimistically depending on an uncommitted state of the transaction. Such transactions must also be rolled back. Specifically, transactions with dirty-read edges to entities modified by the rolled back transaction are also rolled back because they are dependent on the uncommitted state of the entity.

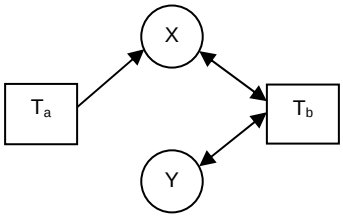


Figure 15: Transaction Rollback.

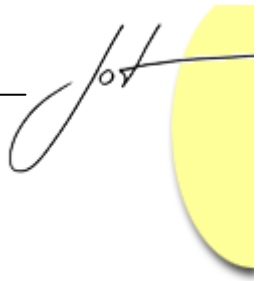
For example, in Figure 15, if T_b rolls back, the action must cascade to T_a because T_a is dependent on the uncommitted state of X. On the other hand, if T_a is rolled back, it does not cause either X or T_b to roll back.

7 CONCLUSION

The techniques described in this paper describe extensions to the directed-graph-based stability scheme for single-level stable object stores. These extensions provide graph-based support for transaction-based concurrency control in persistent object stores.

The presented technique is optimistic and, while space restrictions prevented it being fully described here, supports separate management for concurrent transaction-managed and stability-managed activities that co-exist in the object store. This support ensures that transactions are aborted if they are compromised either by transaction-managed or stability-managed activity. Importantly, assessment of each transaction's potential for success is made at the completion of each involved process time slice, so that if the ACID properties of the transaction have been compromised this is discovered earlier than with other optimistic schemes.

The graph-based transaction management scheme has been extensively evaluated and measured by simulation experiments. A detailed report of those experiments exceeds the scope of this paper, and will be reported separately. The results may be summarised as showing that the graph-based transaction management scheme has performance consistent with that of conventional pessimistic and optimistic bulk data management systems. Significantly, the simulation results show that the newly-presented technique provides excellent general-purpose performance across a wide range of transaction sizes, levels of concurrency and object store sizes.



REFERENCES

[1] Ahamad, M. and Lin, L., Using Checkpoints to Localize the Effects of Faults in Distributed Systems, in *Proc., The Eighth Symposium on Reliable Distributed Systems*, (1989), IEEE Computer Society Press, 2-11.

[2] Atkinson, M. and Morrison, R., Persistent System Architectures, in *Proc., Third International Workshop of Persistent Object Systems*, (Newcastle, Australia, 1988), Springer Verlag, 73-97.

[3] Bem, E.A., Global Stability and Resilience in a Persistent Operating System, PhD Thesis, *Basser Department of Computer Science*, University of Sydney, 1999.

[4] Bem, E.A., Issues in Persistent Systems, in *Proc., The 6th IDEA Workshop*, (Rutherglen, Victoria, Australia, 1999).

[5] Blackburn, S.M., Zigman, J.N.,. Concurrency — The fly in the ointment?, in *Proc., The Third International Workshop on Persistence and Java*, (Tiburon, CA, USA, 1999), Morgan Kaufmann, 250 - 258.

[6] Brössler, P. and Rosenberg, J., Transactions in a Segmented Single Level Store Architecture, in *Proc., International Workshop on Computer Architectures to Support Security and Persistence of Information*, (Bremen, Germany, 1990), Springer-Verlag, 319-338.

[7] Brown, A.L., Persistent Object Stores, PhD Thesis, *Faculty of Mathematics and Computational Science*, Universities Of St Andrews and Glasgow, St Andrews, 1989.

[8] Brown, A.L., A Prototype Log Structured Object Store, in *Proc., The Fourth IDEA International Workshop*, (Magnetic Island, Queensland, Australia, 1997).

[9] Challis, M.F., Database Consistency and Integrity in a Multi-user Environment. *Databases: Improving Useability and Responsiveness*. 245-270.

[10] Date, C.J., *An Introduction to Database Systems*. Addison-Wesley Publishing Co, Reading, MA, USA, 1999.

[11] Dearle, A., di Bona, R., Lindstrom, A., Rosenberg, J. and Vaughan, F., User-level Management of Peristent Data in the Grasshopper Operating System, Universities of Adelaide and Sydney, 1994.

[12] Dearle, A., Di Bona, R., Farow, J., Henskens, F., Lindström, A., Rosenberg, J., Vaughan, F., Grasshopper: An Orthogonally Persistent Operating System. *Computing Systems*, *7* (3). 289-312.

[13] Fabry, R.S., Capability Based Addressing. *Communications of the ACM*, *17(7)*. 403-412.

[14] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kauffmann Publishers, San Mateo, CA, 1993.

[15] Härder, T., Reuter, A., Principles of Transaction-oriented Database Recovery. *ACM Computing Surveys*, *15* (4). 287 - 317.

[16] Henskens, F.A., Koch, D.M., Jalili, R., Rosenberg, J., Hardware Support for Stability in a Persistent Architecture, in *Proc., The Sixth International Worskshop on Persistent Operating Systems*, (Tarascon, France, 1994), Springer-Verlag and British Computing Society, 387-399.

[17] Hulse, D., Dearle, A. A Log-Structured Persistent Store, University of Sydney, 1997.

[18] Jalili, R., A Failure Transparent Distributed Persistent Store, PhD Thesis, *Basser Department of Computer Science*, University of Sydney, Sydney, 1995.

[19] Jalili, R., Henskens, F.A., Entity Dependency in Stable Persistent Stores, in *Proc., The 28th Hawaii International Conference on System Sciences*, (Hawaii, U.S.A, 1995), IEEE, 665 - 674.

[20] Keedy, J.L., The MONADS-PC: A Programmer's Overview, University of Bremmen, Germany, 1989.

[21] Keedy, J.L., Projects Associated with the Department of Computer Structures: The Monads Project, University of Ulm, 1997.

[22] Keedy, J.L. and Brossler, P., Implementing Databases in the Monads Virtual Memory, in *Proc., The 5th International Workshop on Persistent Object Systems*, (San Miniato, 1992), Springer-Verlag, 318-338.

[23] Keedy, J.L. and Rosenberg, J., Support for Objects in the MONADS Architecture", in *Proc., International Workshop on Persistent Object Systems*, (Newcastle, Australia, 1989), Springer-Verlag.

[24] Kumar, V.J., *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice-Hall Inc, Edgewood Cliffs, New Jersey, USA, 1996.

[25] Lindström, A.G., User-level Memory Management and Kernel Persistence in the Grasshopper Operating System, PhD Thesis, *Basser Department of Computer Science*, University of Sydney, Sydney, 1996.

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

[27] Morrison, R. and Atkinson, M.P., Persistent Languages and Architectures, in *Proc., The International Workshop on Computer Architectures to Support*

Security and Persistence of Information, (1990), Springer-Verlag and the British Computer Society, 9-28.

[28] Morrison, R., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Munro, D.S., Atkinson, M.P., The Napier88 Persistent Programming Language and Environment, School of Mathematical and Computational Sciences, University of St Andrews, St Andrews, 1999.

[29] Moss, J.E.B., Munro, D.S. and Hudson, R.L., PMOS: A Complete and Course-Grained Incremental Garbage Collector for Persistent Object Stores, in *Proc., 7th International Workshop on Persistent Object Systems (POS7)*, (1996), 140 - 150.

[30] Rosenberg, J., The MONADS Architecture A Layered View, in *Proc., The Fourth International Workshop on Persistent Object Systems*, (Martha's Vineyard, Mass, USA, 1990), Morgan Kaufmann Publishers Inc, 215-225.

[31] Rosenberg, J., Dearle, A., Hulse, D., Lindstrom, A. and Norris, S., Operating System Support for Persistent and Recoverable Computations. *Communications of the ACM*, *39* (9). 62-69.

[32] Rosenberg, J. and Henskens, F., Stability in a Persistent Store Based on a Large Virtual Memory, in *Proc., International Workshop on Computer Architectures to Support Security and Persistence of Information*, (Bremen, Germany, 1990), Springer-Verlag, 229-245.

[33] Sjoberg, D.I.K., Cutts, Q., Welland, R. and Atkinson, M.P., Analysing Persistent Language Applications, in *The 6th International Workshop on Persistent Object Systems*, (1994), 227-247.

[34] Strom, R.E., Yemini, S.A., Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, *3* (3). 204-226.

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

[36] Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C., Barter, C., A Persistent Distributed Architecture Supported by the Mach Operating System, in *Proc., The First USENIX Conference on the Mach Operating System*, (1990), 123 - 140.

About the authors

Frans A. Henskens is Assistant Dean of the Faculty of Engineering & Built Environment at the University of Newcastle, Australia. His research interests centre on: distributed and grid computing; engineering of flexible software systems; programming language design; resilience and availability in object-based data stores. His email address is frans.henskens@newcastle.edu.au

Maurice G. Ashton is a Lecturer at Avondale College in the Fculty of Business and Information Technology. His research interests include transaction systems, object stores, and enterprise information management. His email address is maurice.ashton@avondale.edu.au