Hardware Support for Stability in a Persistent Architecture

*F. A. Henskens, †D. M. Koch, *R. Jalili & *J. Rosenberg

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

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

Abstract

Persistent stores support uniform management of data objects regardless of their lifetimes and locations. Such stores typically maintain a self-consistent state even after failure of the host computer system. This property is termed stability, and may be achieved using operations called checkpoints. When objects in the store are modified, or modified objects are accessed, dependencies are created between the modifying processes and the objects. Directed graphs may be used to describe such dependencies. For the persistent store to maintain a consistent state, all dependent entities must be checkpointed together. In this paper we show that hardware support can assist in the construction of stable stores for which stability is based on dependency graphs. We then describe an implementation of such support in the Monads-MM computer.

1 Introduction

Persistent systems support mechanisms which allow programs to create and manipulate arbitrary data structures which outlive the execution of the program which created them [2]. This has many advantages from both a software engineering and an efficiency viewpoint. In particular it removes the necessity for the programmer 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 in that both the data and its interrelationships can be stored in their original form. In order to achieve this a uniform storage abstraction is required. Such an abstraction is often called a *persistent store*. A persistent store supports mechanisms for the storage and retrieval of objects and their interrelationships in a uniform manner regardless of their lifetime.

Persistent stores thus abstract over the distinction between primary and secondary storage. The state of the store at any instant is a combination of the contents of the volatile data held in main memory (RAM) and the more stable data held in secondary memory (disk). When a system unexpectedly shuts down, for instance as a result of hardware failure or loss of power, the contents of main memory are typically lost. As a result of such failures the data stored in secondary memory may be inconsistent or unreachable. Cockshott [5] and later Brown [3] proposed that the abstraction over storage should include transparent recovery from such store failures so that the store contents are guaranteed to be consistent even after unexpected store failure. Such stores

Published in: *Workshops in Computing: Proceedings of the Sixth International Workshop on Persistent Operating Systems*, Tarascon, France, ed M. Atkinson, D. Maier and V. Banzaken, Springer-Verlag and British Computer Society, ISBN 3-540-19912-8, pp. 387-399, 1994

are said to be stable, and move between stable states through a sequence of operations called checkpoints.

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 or associations [11] , and describe a scheme for expressing associations based on directed graphs. We then show in section 4 how appropriate hardware support facilitates the implementation of stable stores based on this scheme and introduce issues related to multiprocessor architectures. Finally in section 5 we describe the implementation of such hardware support in the latest generation of the Monads architecture.

2 Implementation 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. Techniques which achieve stability are typically based on the use of operations called checkpoints which commit all recent modifications to stable secondary storage. The act of checkpointing a store in effect flushes all modified data currently held in main memory to disk, and creates 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. If it could be guaranteed that the contents of disk were never modified between checkpoints, and that the checkpoint operation itself was atomic, then the contents of disk would always represent a stable state of the store. In fact, virtual memory management requires that from time to time main memory pages are re-assigned. Pages containing unmodified data may be safely re-used without disk access. Modified data, however, must firstly be saved before the page(s) containing the data can be re-used. This is typically achieved by flushing the entire page contents to disk. Writing such a page back to its original location on disk potentially leads to the disk representation of the store being inconsistent and therefore unstable. Shadow paging [12] is a technique that allows modified page discard without causing an inconsistent disk version of the store. The atomicity of checkpoint operations may be guaranteed using Challis' algorithm [4] . In the following discussion the term *object* is used to describe an arbitrarily large unit of logically related data.

2.1 Shadow paging

This technique maintains two forms of data which has been modified between checkpoint operations; the stable data as it existed at the last checkpoint (shadow data) and 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 the usual 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.

Shadow paging may be implemented using either before-look or after-look strategies. The before-look strategy takes an on-disk shadow copy of a page prior to its modification and allows discard of the current version page to the original page disk location. A checkpoint operation flushes modified main memory pages to their original locations on disk and causes the return of disk pages containing shadow copies to the pool of free disk space. Recovery from store failure involves copying shadow versions of pages onto their

original disk locations. This strategy has the advantage that it maintains the physical location of data on disk, and was implemented in Brown's stable store [3].

The after-look strategy allocates a new disk page for storage of the current version, and retains the previous unmodified version in its original location as a shadow copy. In effect the disk pages existing immediately after a checkpoint form shadow pages until the next checkpoint. A checkpoint operation flushes modified main memory pages to the current version disk locations and returns the disk space occupied by shadow copies to the pool of free disk pages. Since the data structures describing the disk are also stabilised at a checkpoint, recovery from store failure is automatic. This strategy has the advantage that it requires one less write operation for modified pages between each checkpoint but results in the random distribution of data on disk. It has been implemented for Monads [14] and Casper [17].

Both strategies require atomicity of checkpoint operations. Since writing to disk is a sequential operation, such atomicity can only be an abstraction. This is achieved by viewing the store as a structure accessible from a single point or root, and changing that root according to Challis' algorithm as the last step in a checkpoint operation.

2.2 Challis' algorithm

Challis proposed that atomicity could be achieved by starting and ending a root page with a timestamp. Such timestamps are never the same for consecutive versions of a root page. If a root-page write operation is successful, the disk version of the page will have identical timestamps; if it fails then the first timestamp will differ from the last, which will remain at the value prior to the failed write operation.

By maintaining two root pages in well known locations on disk, ensuring that there is always one valid root block and using a system of timestamps which allow determination of the most recent correct root page, it becomes possible to atomically commit a checkpoint operation. Further, because the root page either contains or points to structures which describe the store, it is possible after failure to determine which root points to the most recent stable state and to access that state.

The problem with the stability scheme as described is that the entire store must be checkpointed at the same time. Since user processing must either cease or be severely restricted during such an operation, a checkpoint involves 'stopping the world'. 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, 17]. The stable state of such a store is the collection of these stable parts.

While checkpointing parts of the store independently has a positive effect on performance, it 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 pages in Casper, and more recently using directed graphs of entities [9]. Other work based on message-passing research is currently investigating the maintenance of causality relationships to allow reconstruction of consistent states (which are not necessarily recreations of previous states) following failure [16] .

In the following section we refer to data as being modified if it has been changed or created since the last time the object containing the data was checkpointed. The term *entity* in this discussion refers to an object (as defined for the system) or a process.

3 Inter-relationships between system entities

As processes access data objects in a store, these accesses 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 such modified data. It is important to note that data objects cannot become inter-dependent without processes, and that processes cannot become inter-dependent without data objects. It should also be noted that read operations on unmodified data do not create dependencies.

By way of example of the creation of a dependency between objects, consider a motor car registration system in which a vehicle cannot have its registration renewed unless it has current insurance. Assume a system consisting of an insurance object, a registration object and a number of user-level processes. If one process updates the insurance object to indicate that the insurance on a particular vehicle has been renewed, and another process subsequently queries the insurance status of the vehicle and allows and records the vehicle's re-registration, this activity results in a dependency between the involved entities. If the registration object were checkpointed independently of the insurance object, a failure could result in the store recording that the vehicle was registered without being insured.

Ideally dependencies should be established based on knowledge of access to data at the basic unit of data reference (eg. byte, word). In a typical paged store, however, it is overly 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 to use the same granularity in determining inter-object dependencies. Such dependencies, while detected at the virtual page level, result in checkpoint dependencies at the large object level. Thus dependency information is maintained at the large object level for the purpose of controlling checkpoint operations. The dependency between such objects may be represented using a set called an *association* [11].

3.1 Describing dependencies using associations

As defined for Casper, associations are sets of dependent entities [11] . After it has been checkpointed, an entity belongs to an association of which it is the only member. Over time entities interact with other entities 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 entities with no inconsistent inter-relationships. It is apparent, however, that the use of a single structure to describe both the checkpoint and the roll-back relationships between entities 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. In the motor car registration example, for instance, it is not really necessary to checkpoint the registration object when the insurance object is checkpointed, but it is necessary to checkpoint the insurance object with the registration object. Accordingly it should not be necessary to roll-back the insurance object because of a failure resulting in the roll-back of the registration object. Such a situation may occur if the insurance and registration objects were stored on different nodes in a distributed store.

The use of directed graphs has been shown to minimise the extent of checkpoint and roll-back operations.

3.2 Describing dependencies using directed graphs

Using directed graphs it is possible to separately represent the checkpoint and roll-back dependencies between entities [9]. Graphs are created in a similar way to associations, however different graphs are traversed depending on the operation being performed. The $\rightarrow$ edge is used to specify the dependency between two entities. E1 $\rightarrow$ E2 means that E1 depends on E2. $\rightarrow$ is transitive, but not symmetric ie. if E1 depends on E2 (E1 $\rightarrow$ E2) then E2 does not necessarily have the same relationship with E1 ($\neg$ E2 $\rightarrow$ E1). The relationship E1 $\rightarrow$ E2 is established if E1 reads modified data from E2. Write operations lead to a pair of dependencies; instead of indicating two unilateral arrows (E1 $\rightarrow$ E2 and E2 $\rightarrow$ E1), the notation E1 $\leftrightarrow$ E2 is used. Note also that the expression E1 $\rightarrow$ E2 is congruent to the expression E2 $\leftarrow$ E1.

While a single graph can be used to describe both checkpoint and roll-back dependencies, the edges have different meanings for each purpose. Thus, in effect, a single dependency directed graph represents separate checkpoint and roll-back graphs. The relationship between the edges forming a dependency graph and their meanings in checkpoint and roll-back graphs are shown in figure 1.

Dependency Graph	Stabilising 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 1. The relationship between Dependency Graph, Stabilising Graph, and Roll-back Graph.

As described in section 2, a typical checkpoint operation in a paged persistent store results in the flushing to disk of main memory pages containing unstable modified data. Similarly dependencies between entities should only be established through accesses involving modified data. In the following section we show how explicit hardware support improves the efficiency of such stability-related operations.

4 Hardware support for stability

We have earlier argued the benefits of support for dual object sizes [7] 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.

In this discussion we assume that either:

- the object store supports both small and large objects as described above, with stability being implemented at the large object level, or
- the object store supports a single paged object type.

To enable implementation of the checkpoint operation for such a store it is necessary to be able to detect:

(1) which main memory pages have been modified by some process,

(2) which main memory pages have been accessed in this time-slice by the currently executing process, and

(3) which main memory pages have been modified in this time-slice by the currently executing process.

The need for these abilities, and features which provide support for them are discussed in the following sections.

4.1 Identification of modified main memory pages

This requirement 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 in their address translation unit (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.

The proposed *modified* ATU bit is used 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 and accessing processes. The modified bits for the pages of an object are, of course, cleared when the object is checkpointed.

Implementation of stability 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 would need to be loaded with the dirty bit set. As a result the page would be flushed again on its next discard or when its object was checkpointed. It is recognised that the modified bit duplicates information available from the shadow paging data structures; however these data structures are typically stored in main memory, and speed is of the essence given that the modified bit is checked on every memory access. It is thus necessary that separate dirty and modified bits are implemented in the ATU.

The implementation of the modified bit requires that the virtual memory page table(s) used to locate pages for loading into main memory must be extended to indicate whether non-resident pages have been modified since the last checkpoint. The page table is used to retrieve pages, and this extra information 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.

The features described in the next section serve to improve the efficiency of construction of dependency graphs by allowing them to be updated once per process time-slice.

4.2 Lazy dependency graph construction

The building of a dependency graph requires detection of process access to modified data and of process modification of object data. Such events may be detected and if necessary recorded either eagerly (i.e. after every access to an object) or lazily. Identification of critical accesses at the time of those accesses would cause an unacceptable deterioration in system performance.

The collection of appropriate data during a process' time-slice is proposed, thus allowing dependency graphs to be updated at the time of a process switch. Accordingly it is necessary to record the accesses and modifications performed by a process during its latest activation.

4.2.1 Identification of accessed main memory pages

Pages may remain in main memory for a period encompassing many process activations. The *m_accessed* bit maintained by the ATU allows detection of process access to 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 dependency graph at the conclusion of the process' period of activation. All m_accessed bits must be clear at the commencement of a process time-slice; this may be achieved in a single operation using appropriate hardware.

4.2.2 Identification of written main memory pages

The inclusion of a *written* bit maintained by the ATU allows detection of object data modifications made by the current process. This bit is distinct from the modified bit described in section 4.1 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 dependency graph 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 $\leftrightarrow$ dependency graph edge. The operation of the ATU with respect to the described status bits is shown in figure 2.

OPERATION	DIRTY	MODIFIED	M_ACCESSED	WRITTEN
Unmodified page retrieved	Cleared	Cleared	Cleared	Cleared
Modified page retrieved	Cleared	Set	Cleared	Cleared
Process reads data from page	Unchanged	Unchanged	Copy *modified*	Unchanged
Process writes to page	Set	Set	Set	Set
End of process time-slice	Unchanged	Unchanged	Cleared	Cleared
Page flushed	Cleared	Unchanged	Unchanged	Unchanged
Object checkpoint	Cleared	Cleared	Unchanged	Unchanged

Figure 2. Effect of operations on page status bits.

In the following section we discuss the implications of m_accessed and written bits when used in multiprocessor machines.

4.3 Multiprocessor architectures

In a multiprocessor machine multiple processes are able to execute in parallel. It is thus necessary to maintain m_accessed and written bits for a page frame on a *per process* (and thus per processor) basis. Thus for any page frame there is an array of m_accessed bits and an array of written bits. One element of each of these arrays represents each attached processor.

Multiprocessor computers increase the complexity of the described operations because they introduce the aspect of causal ordering of events. Such ordering is important because of:

(1) its impact on the discard of pages during virtual memory operation and

(2) the impact of a checkpoint involving one process on other executing processes.

4.3.1 Parallel page discard

Lazy dependency graph construction, as described, is dependent on the important condition that no page is discarded while its m_accessed or written bits are set. For a single processor machine this is not an issue because page discard occurs synchronously with process activation. A multi-processor computer allows several processes to execute simultaneously, and thus page discard may occur in parallel with process activation.

When a page is selected for discard, the arrays of m_accessed and written bits for the page are scanned, and the process executing on any processor for which either bit is set is forced to end its current time-slice (i.e. there is a re-schedule on the processor). This results in merging of the dependency graphs containing the processes. To minimise this imposition on executing processes, the discard algorithm attempts to select pages for which neither bit is set. The number of such pages is minimised by ensuring that

processes checkpoint regularly; the effect of this is similar to that of the Unix *sync* operation [13].

4.3.2 Parallel checkpoint and process execution

In a multiprocessor a checkpoint on an object may occur in parallel with execution of other processes. This raises the possibility of an object being checkpointed whilst it is being accessed by a process active on another processor. Such a process must be forced to suspend, causing its dependency graph to be updated. This may result in the process being checkpointed.

The authors are currently investigating multiprocessor and distribution aspects of support for stability and will report their findings in a later paper.

5 An implementation example

This section describes the addition of the above-mentioned hardware support to the Monads-MM architecture [15]. The Monads-MM supports a network of nodes each of which may comprise multiple processors. Each processor generates 128 bit virtual addresses which reference a persistent global distributed shared memory (DSM).

Each node has a single ATU which translates virtual addresses to physical addresses for all of the processors attached to that node. The global address space is structured to assist in determining the physical location of data stored in it [6]. The store provides explicit support for small and large objects called *segments* and *modules* respectively, with segments being positioned orthogonally to page boundaries and logically related segments being collected to form paged modules [7]. Thus many segments may exist within a single virtual page or a segment may span several virtual pages. Modules are self-describing; each module contains internal data describing its own location on disk. As a result there is no need for a central DSM page table.

A two level capability-based protection scheme controls access to the interface procedures of modules and addressing of the data stored in segments [10]. A set of capability registers is provided for the purpose of addressing segments. All addressing is of the form <capability register>+<offset>, providing dynamic bounds and access-rights checked access. Segment capabilities define the identity of the encompassing module; as a result the module is identified with every data access, facilitating collection of dependency information.

The ATU itself is organised as an inverted hash table [1]. To provide the support described above each cell of this table must be extended to provide modified, m_accessed and written bits for the appropriate page frame in addition to the dirty bit already provided. In the Monads-MM implementation we have taken advantage of the machine's architecture by maintaining m_accessed and written information with each processor rather than at the ATU. Moreover such information is maintained at the module level and is thus of the granularity required by the dependency graph.

5.1 Maintenance of access status

Examination of the information content of the proposed modified, m_accessed and written bits reveals that the former conveys page-level information on a between checkpoints basis and the latter two are used to construct module-level information for the current time-slice. Accordingly, and since the appropriate module is identified by every access, it is possible to maintain a cache of module status information with each processor.

This is called the *Process Active Object Cache* (PAOC). Data is collected in the PAOC during processing using information provided by the ATU.

A key field attached to each capability register is loaded the first time that register is used in a time-slice. This field points to the appropriate PAOC cell and is used for subsequent accesses to the cache during this process activation. Since capability registers are used to address segments, multiple PAOC key fields may point to any single cache cell. A comparator is used when the PAOC key is invalid to detect whether the required cache entry already exists. If the appropriate entry does exist the PAOC key is set to point to it; if it does not exist a cache entry is created and the key is set.

The contents of the PAOC are used at the completion of a process time-slice to appropriately update dependency information. Finally the cache cells are cleared and capability register key fields are marked invalid in preparation for the next process. This structure is shown in figure 3.

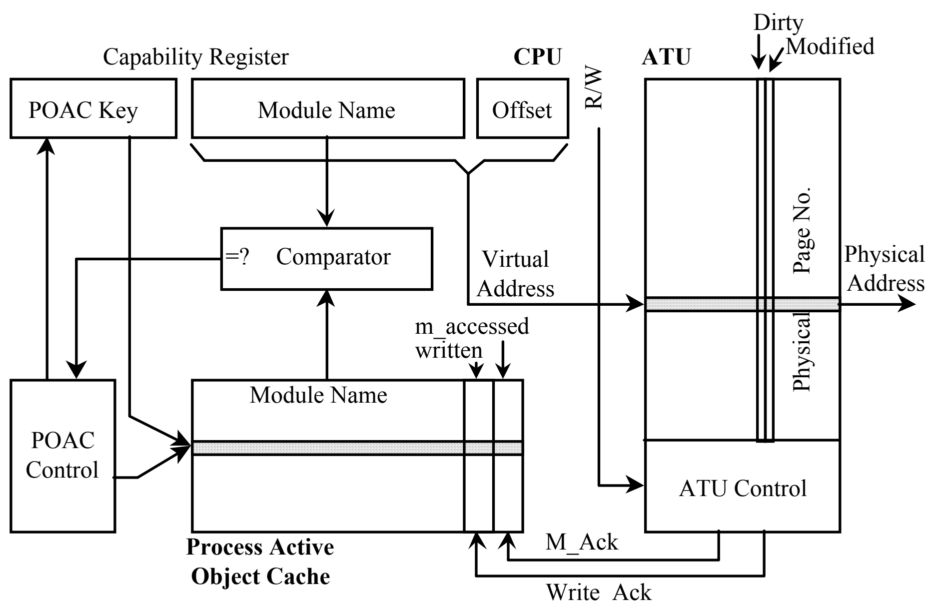

Figure 3. Cache of process related status correlated with module name.

Maintenance of the per-processor module access information separately from the ATU has the advantages that:

- m_accessed and written information is automatically collated at the required module-level granularity during system operation, reducing the cost of the dependency graph update performed at the conclusion of a process' time-slice,
- less hardware is required because m_accessed and written information for each attached processor is not maintained for every main memory page frame and
- it facilitates clearing of the status information at each process switch (compared with selectively clearing ATU bits in a multiprocessor machine).

5.2 Maintenance of page modification status

The provision of a modified bit for each ATU cell allows identification of virtual pages containing data modified since the last checkpoint operation. Information stored by the ATU is ordinarily lost when the page is discarded during virtual memory operation. As described in section 4.1, this situation requires the extension of the virtual memory page table to include modified information. When the object containing a page is checkpointed, the modified bit for the page must be cleared, involving a scan of all page table entries for the object.

Monads modules are self-defining, with each module storing its own page table. We propose that, rather than extending the existing page table entries, an extra bit list is added to the internal data maintained by every module. Each bit would logically form part of a page table entry and would store that page's modified status. The bit list would be checked every time a page from the module was loaded into main memory, and would be updated with every page discard. Segregation of the modified bits in this way facilitates clearing after a checkpoint operation on the module.

6 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 access to data. Separately recording these dependencies allows a reduction in the extent of checkpoint and roll-back operations.

Maintaining dependency information of read/write granularity appears to require a dependency graph update immediately after every store access. Such expensive updates may be avoided if the ATU maintains extra status information about main memory accesses, allowing a single dependency graph update at the conclusion of each process activation.

Structuring the store to support large objects comprising collections of logically related small objects, and then appropriately naming such large objects, allows the maintenance of process access history at the large object level. This has two significant advantages. Firstly it reduces the quantity of access history data stored and thus the hardware required for such storage. Secondly it removes the necessity to scan the page-related data maintained in the ATU by dynamically collecting an object-level history during process execution for use when updating dependency graphs.

Acknowledgments

Part of this work was supported by a University of Sydney research grant. The authors would also like to acknowledge the contributions of R. M. Wilkinson and L. D. Bacardi to discussions related to this work.

References

1. Abramson, D. A. "Hardware Management of a Large Virtual Memory", *Proc. 4th Australian Computer Science Conference*, pp. 1-13, 1981.

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

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

4. Challis, M. F. "Database Consistency and Integrity in a Multi-user Environment", *Databases: Improving Useability and Responsiveness*, pp. 245-270, 1978.

5. Cockshott, W. P. "Orthogonal Persistence", Ph.D Thesis, University of Edingurgh, 1983.

6. Henskens, F. A. "Addressing Moved Modules in a Capability-based Distributed Shared Memory", *Proceedings of the 25th Hawaii International Conference on System Sciences*, vol 1, ed V. Milutinovic and B. D. Shriver, IEEE Computer Society Press, Hawaii, U. S. A., pp. 769-778, 1992.

7. Henskens, F. A., Brössler, P., Keedy, J. L. and Rosenberg, J. "Coarse and Fine Grain Objects in a Distributed Persistent Store", *Proceedings, Third International Workshop on Object Orientation in Operating Systems*, Ashville, North Carolina, pp. 116-123, 1993.

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*, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 246-256, 1990.

9. Jalili, R. and Henskens, F. A. "Minimising the Extent of Cascadable Operations in Stable Distributed Stores", *Submitted for Publication*, 1994.

10. Keedy, J. L. and Rosenberg, J. "Support for Objects in the MONADS Architecture", *Proceedings of the International Workshop on Persistent Object Systems*, ed J. Rosenberg and D. M. Koch, Springer-Verlag, 1989.

11. Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. and Barter, C. "Cache Coherence and Storage Management in a Persistent Object System", *Proceedings, The Fourth International Workshop on Persistent Object Systems*, pp. 99-109, 1990.

12. Lorie, R. A. "Physical Integrity in a Large Segmented Database", *ACM Transactions on Database Systems*, 2(1) pp. 91-104, 1977.

13. Ritchie, D. M. and Thompson, K. "The UNIX Time-Sharing System", *The Bell System Technical Journal*, 63(6), pp. 1905-1930, 1978.

14. Rosenberg, J., Henskens, F. A., Brown, A. L., Morrison, R. and Munro, D. "Stability in a Persistent Store Based on a Large Virtual Memory", *Proceedings of the International Workshop on Architectural Support for Security and Persistence of Information*, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 229-245, 1990.

15. Rosenberg, J., Koch, D. M. and Keedy, J. L. "A Massive Memory Supercomputer", *Proc. 22nd Hawaii International Conference on System Sciences*, vol 1, pp. 338-345, 1989.

16. Vaughan, F., Dearle, A., Cao, J., di Bona, R., Farrow, J. M., Henskens, F. A., Lindström, A. and Rosenberg, J. "Causality Considerations in Distributed Persistent Operating Systems", *Proceedings, 17th Australian Computer Science Conference*, Australian Computer Society, Christchurch, New Zealand, pp. 409-420, 1994.

17. 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.