Recoverable Virtual Memory through the Multi-View Memory Computer System

P. Bodorik  
Faculty of Computer Science,  
Daltech, Dalhousie University  
Halifax, Nova Scotia, Canada  
bodorik@cs.dal.ca

D. Jutla  
Dept. of Finance and Mgmt. Science  
Faculty of Commerce  
Saint Mary’s University  
Halifax, Nova Scotia, Canada  
dawn.jutla@stmarys.ca

A. Agarwal  
IBM Canada Ltd.  
1B/020/1150/TOR  
1150 Eglinton Avenue East  
North York, Ontario, Canada  
agarwala@ca.ibm.com

Abstract

Recoverable virtual memory (RVM) is very useful for applications that require fault tolerance for persistent data structures. Updates to a recoverable region of memory are recorded so that recovery in case of faults is possible. Multi-view virtual memory (MVM) provides recoverability for a memory region by defining a view, consisting of access units, on that region. Finite state machines (FSMs) control access to the units of the view. The FSM states are defined to trap on the first write to an access unit in order to save the before-image or to simply change the state of the unit so that an after-image can be recorded at a commit point.

The recoverability provided by the MVM approach is compared to the hardware approach of Logged-RVM and to the software approach of Light-weight-RVM. The comparison is made qualitatively in terms of the interface and functionality, and also quantitatively by examining the delays using simple back of the envelope calculations.

1. Introduction

Recoverable virtual memory (RVM) is very useful to a wide variety of applications that require fault-tolerance for persistent data structures [Eppinger 1989, Eppinger 1991, Cheriton 1995, Satyanarayanan 1993]. Examples of applications that can benefit from RVM include distributed file systems, databases, object oriented repositories, CAD tools and CASE tools [Satyanarayanan 1993]. Other examples include parallel discrete-event simulators that use optimistic concurrency control, object oriented databases and debuggers that use reverse executions [Cheriton 1995]. Thus, it is indeed a wide variety of applications that can benefit from the use of RVM. It is for this reason that both software and hardware approaches have been attempted in providing RVM.

To provide recoverability on a memory region, updates to memory regions on which the recoverability property is to be enforced are recorded in log records so that recovery in case of faults is possible. In the software approaches it is the application programmer’s responsibility to annotate the updates so that the system could record appropriate log information. In the hardware approach, dedicated hardware incorporated directly in the memory subsystem itself is used to record log records about writes to recoverable memory regions.

Camelot [Eppinger 1989, Eppinger 1991] provides a general purpose transactional facility for construction of reliable distributed systems. The current practice is for each separate application, such as a DB or file system, to implement its own recoverability subsystem. Instead of each application providing its own recoverability mechanism, Camelot is used to provide logging, synchronization and commitment services on regions of virtual memory. It is very flexible in that it provides a choice of strategies for each function. For instance, both undo and redo strategies for recoverability are supported. To provide the flexibility, Camelot extensively utilizes external page management and inter-process communication of the Mach kernel.

In [Satyanarayanam 1993], Camelot has been used in the implementation of the two-phase optimistic replication protocol for the Coda File system [Kistler 1992, Satyanarayanam 1992]. It has been surmised, however, that the Camelot’s flexibility came at a price of poor performance, difficult maintenance, programming constraints and poor scalability [Satyanarayanam 1993]. The difficulties arose primarily due to the frequent heavy-weight context switching to user-level managers, such as an external pager, and extensive use of inter-process communication. It was decided that instead of a very flexible system providing various transactional services, an efficient system that provides RVM only should be built. The result was a light-weight recoverable virtual memory (Light-RVM) which, in its interface is similar to that of Camelot. The application programmer is still responsible to call routines that save appropriate back-up information...
in a log before each update to a recoverable memory region. Unlike in Camelot, the routines are linked directly into the application’s virtual address space, thus eliminating the heavy-weight context switching occurring in Camelot.

In [Cheriton 1995], it is advocated that because of the wide variety of applications that can benefit from the RVM, the mechanism for recoverability should be incorporated directly into the memory management unit. The developed prototype has a memory management unit (MMU) that provides logged recoverable virtual memory (Logged-RVM). The method was developed for a virtual memory system in which the write-through method is used to propagate updates from the first-level data cache to the physical memory itself. (Please note that any further unqualified reference to a data cache or simply a cache implies the first-level data cache.) Each write-memory operation to a recoverable segment is recognized by snooping hardware monitoring the write-through of updates from the cache to memory. In addition to updating the memory, the update is also recorded/logged in a hardware-based queue servicing the log. The log record includes the after-image, i.e., the new updated value, the identity of the writer (transaction ID), and the size of the data written by the memory write operation. Log records are continually removed from the queue and DMA-ed to a region of memory that serves as the log. Note that the size of the write here refers to the number of bytes written by the individual machine write operation and is thus a byte, or a small number of bytes forming a word. Advantages cited over the Light-RVM include performance and also the fact that it is no longer the programmer’s responsibility to make explicit calls to the recovery subsystem to log appropriate information — this is performed automatically by the memory hardware.

Multi-view virtual memory (MVM) can also be used to provide recoverable memory regions. Like in the Logged-RVM, the programmer need not make explicit calls to the recovery subsystem to ensure that logging is performed. There is hardware assistance provided by the memory subsystem, but unlike in the Logged-RVM approach, the hardware is not dedicated to provide RVM only; it can serve many functions, RVM is only one of them.

MVM enables a view to be defined on a region of memory. A view has units on which access control protocols can be defined in terms of finite state machines (FSMs). The size of access units is user-defined and although it is fixed within a view it varies between views. The MVM can be used to provide recoverable virtual memory by specifying an appropriate FSM. Consider the case when the before-image, that is the value before the update, is required. The FSM causes a trap on the first write to the access unit in order to save the before-image in a log record. If an after-image is required, the FSM changes the state of the access unit so that an after-image can be recorded at a commit point. In comparison to the Logged-RVM, which logs an after-image of every single memory write operation, the before-image of an access unit is captured only on the first write to that unit or at a commit point. Furthermore, the size of access units in a view is defined by the application. In comparison to the Light-RVM, no programmer annotation of writes is required.

This paper is another step in demonstrating the versatility of the complex MVM model. It is complex because the project spans the areas of computer architecture, operating systems, and various applications that benefit from MVM. Although we are concentrating on the use of MVM in providing recoverable memory regions, this is only one application of MVM; others can be found in [Jutla 1997a] and [Bodorik 1998]. The performance of a database locking application using the MVM model is detailed in [Bodorik 1998, Jutla 1997a, Jutla 1998]. The performance of recoverable virtual memory implemented through MVM is presented here. Showing the suitability of the Multiview Memory Model to a range of applications is one of our goals toward establishing that MVM model is a feasible investment.

In addition to presenting how the MVM provides recoverable virtual memory regions, the paper also compares MVM to the logged and light-weight RVM approaches both qualitatively and quantitatively. It is organized as follows. The second section overviews the MVM model and its protection architecture. How the MVM supports recoverable memory regions is shown in the third section. The fourth section compares the method of providing recoverable memory regions by the MVM, referred to as Multiview-RVM, to those of the logged and light RVM methods. The last two sections review related work and provide a summary with conclusions, respectively.

2. Multi-view Memory Model and Protection Architecture

The Multiview Memory model is briefly overviewed in this section only to provide reader context. The description closely follows that found in [Bodorik 1998] and [Jutla 1998]. It is assumed that virtual address spaces are flat and that threads and tasks have the usual meanings in the context of process management: one or more threads execute in an environment of a single task which has one virtual address space (shared by all of the task's threads).

2.1. Multi-view memory model

The sparse address space consists of memory regions that have different access control views. The concept of different views for a region is shown in Figure 2.1. Access
control units, or access units for short, have the same size within a single view, but the size can vary across views. Examples of different types of access units include locking, coherency and recovery units. Applications can thus be serviced with various sizes of units across views, although the size of units is fixed within a view.

Each view of a memory region has its own state which is independent of the states of the other views defined on the same or another region. For a memory operation to succeed, access must be permitted for each view defined on the data in the line (see Figure 2.2). Instead of including the Virtual Address space ID (VAID), each valid/invalid bit, and the W bit indicating if only a read operation is permitted, access for one protection domain, its PDID is recorded in the state. If the definition of a state indicates so, a thread is permitted to access a unit having an exclusive state only if the PDID it is associated with matches the PDID recorded in the state.

Access control is enforced by the kernel using the FSM definition is modified in the MVM model by storing/recording a PDID together with state information for each access unit. For a state representing exclusive access for one protection domain, its PDID is recorded in the state. If the definition of a state indicates so, a thread is permitted to access a unit having an exclusive state only if the PDID it is associated with matches the PDID recorded in the state.

Access control for a view utilizes the standard concept of an access matrix in which each subject has a row while each object has a column. Each matrix entry A[S,O] is a finite state machine (FSM) which defines the access rights of the subject S to an object O depending on the current state of access and the desired operation/access. The FSMs are identical in terms of the defined states and transitions; they differ only in their current state of access. Therefore, there is only one definition of the FSM states and transitions and a matrix entry A[S,O] represents the current state of access of S to O. Objects are access units of a view and subjects define the access rights for threads, or sets of threads that actually issue the reads and writes, that is execute respective memory read and write operations. Each view of a memory region has its own state which is independent of the states of the other views defined on the same or another region. For a memory operation to succeed, access must be permitted for each view defined on the referenced memory location. Once access is permitted for each view, the state changes independently for each view -- the state changes independently for each one as defined by the FSM state transition definition.

Since access control is specified for subjects while memory operations are issued by threads, a thread must be bound to (associated with) a subject. For an operation to be permitted in a view, a thread must be bound to a subject that has appropriate access rights to the referenced access unit of that view.

The access matrix method is an elegant and convenient paradigm for protection purposes. Architectural support for protection is typically based on the object manager approach, e.g., [Koldinger, 1992] and [Wilkes, 1991]. The MVM model also adopts this approach in that the state of access is kept for each access unit. Furthermore, the computer architecture term of the protection domain is used instead of the term subject (as is done in [Koldinger, 1992] and [Wilkes, 1991]). A protection domain is uniquely identified by a protection domain ID (PDID). A thread executes in one protection domain only at any one time, or paraphrasing it, a thread has only one PDID at any one time, but the PDID may change throughout the thread’s execution. If two threads of an application task do not have the same PDID then they do not have the same access rights to access units of views. Any two threads with the same PDID have the same access rights to views. Thus synchronization of threads within the same task may be achieved without their explicit synchronization.

The FSM definition is modified in the MVM model by storing/recording a PDID together with state information for each access unit. For a state representing exclusive access for one protection domain, its PDID is recorded in the state. If the definition of a state indicates so, a thread is permitted to access a unit having an exclusive state only if the PDID it is associated with matches the PDID recorded in the state.

Access control is enforced by the kernel using the FSM in cooperation with a View manager, or manager for short. (Any unqualified reference to a manager from now on refers to the view manager.) If a view definition specifies so, any state changes effected by a thread through accessing units of a view are communicated to the manager. When the thread is suspended, state changes it effected are communicated to the manager by the kernel. The manager may also instruct the kernel to change states of specific access units; these are referred to as forced state changes.

The following section shows the FSM state definitions in order to achieve a recoverable memory region and also describes the use of views in order to clarify the MVM model concepts.

2.2. Protection architecture

The protection architecture that utilizes a protection control unit (PCU) for enforcement of access control protocols is briefly described here. Further details can be found in [Jutla 1997a].

The architecture assumes a virtually addressed, physically tagged, data cache that is separate from the instruction cache and thus memory access/operations are read and write, not execute. In addition to the data itself, each data cache line includes the clean/modified bit, the valid/invalid bit, and the W bit indicating if only a read operation or both the read and write operations are permitted on the data in the line (see Figure 2.2). Instead of including the Virtual Address space ID (VAID), each
line includes a Protection Domain ID (PDID) that is used in the following way.

Consider a thread’s read/write memory operation. If the referenced data cache line is not valid or the thread’s PDID does not match the PDID stored in the data line then a data cache miss occurs. Assume that the line is valid. If the PDIDs match then a read operation can proceed, while a write operation can proceed only if the W bit indicates that the write operation is permitted otherwise a write access fault occurs. In case of a write access fault or a cache miss, the protection subsystem, in particular its PCU, is invoked to determine if the access can proceed. If there is a view defined on the referenced line, the PCU examines the current state of the access unit (containing the address issued by the memory read/write operation) and determines whether the operation can proceed. If the write access can proceed, then the read/write bit in the line is set to indicate that the write access is permitted so that a subsequent write access does not invoke the PCU again. For both read and write operation, the thread’s PDID supplied by the CPU is recorded in the line. Finally, the new state of access to the protection unit of the view is recorded. If the PCU determines that the access cannot proceed, an access control fault occurs and the thread is suspended until it is bound to a protection domain that has appropriate access rights to the access unit.

It should be noted that the protection subsystem is not on the critical path of instruction execution in that it is not invoked on every memory access. Also, the architecture is distinguished from current architectures in two respects: one is the replacement of the VAID with the PDID while the other one is that the protection subsystem is invoked on a data cache miss and write access fault.

In [1997a, Jutla 1997b and Tong 1997] a PCU design in described and evaluated in which all activities are performed by a hardware controller accessing information stored in caches. It is this PCU design that shall be assumed in the section that compares the three approaches to provision of RVM.

3. Recoverable Memory Regions by MVM

This section describes how the MVM provides recoverable regions of memory. Because we want to make comparisons to the light and logged RVMs, and Light-RVM concentrates on the recoverability for transactions, our presentation is also targeted for the transactional environment. We shall first describe functional requirements for provision of recoverable memory regions. How the MVM model provides recoverable memory regions through usage of FSMs and views is described first for the redo method, then for the undo methods, and then for the combination of both.

3.1 Requirements

Systems that provide transactional guarantees may perform recovery using (i) a redo method only, (ii) an undo method only, or (iii) both the redo and undo methods. This depends on whether an update is immediate or deferred, in-place or out-of-place, and whether updated data is (or is not) flushed on commit [Navathe 1995, Ozsu 1991]. Description of when the undo and redo methods are used is outside the scope of this paper.

Simplifying, it can be stated that an undo method records updates in logs and on recovery it uses the log records to restore the updated data items to some previous state prior to a fault – hence an undo method. Similarly, a redo method also records the updates in logs, but on recovery it uses the logs to redo updates made (since the last checkpoint) prior and up to the fault. In transactional systems, there are rules in terms of when log records must be propagated from memory to the stable storage (disk) in relation to propagating updates to data items from memory to the stable storage and also rules on how log buffers are handled on commitment of transactions. What is important is to note that the problem at hand is to record the updates, made to data items in virtual memory, and to log buffers that are also stored in memory (either virtual or physical). How the log records are used depends on the application. For transactional recoverable virtual memory, the log buffers must be carefully propagated from virtual memory to the stable storage, but for other applications the use of the log records is quite different (see [Cheriton 1995] for the use of log records for various applications).
To summarize, to support recovery, before-images of data items updated by a transaction must be recorded in log records in order to undo a transaction while after-images of data items must be recorded in order to redo a transaction. Note that the Logged-RVM provides only for recording of after-images, as will be elaborated upon later, while Light-RVM can be used with both the undo and redo methods.

The before and after-images are collected in log records, that is in buffers stored in a memory region that is different from the recoverable region. A log record for an update contains the transaction ID, the start address and the number of bytes that were updated, and the before or after-image.

Consider the case when before-images are required. Prior to writing a number of consecutive bytes to a segment of recoverable memory, that is prior to updating a data item, the before-image must be recorded in a log buffer. If an after-image is required, a copy of the updated data item must be recorded in a log buffer; however, this does not have to be performed immediately after the update, but i at the convenience of the recovery mechanism. In the Multiview-RVM (MVM approach to provision of RVM), as in the Light-RVM, this is performed during the transaction’s commit. Thus, for recording of after-images, the recovery system must record which data items have been updated by the transaction so that after-images can be recorded in log records at commit time.

3.2 FSMs and views for recoverability

To achieve logging of before or after-images, a view is defined on a memory segment that is to be recoverable. The access units of the view are the data units and are the units of recoverability: their size, static/fixd within the view, varies across views and is defined by the application. The view’s FSM, which synchronizes access to the units of the view, is defined to enable the capture of before and/or after-images. If a before-image is required, the FSM is defined to raise a (write access control fault) exception on the first write to an access unit so that the before-image can be recorded in a log buffer before the unit is updated. If an after-image is required, the state of access to the unit changes on the first write to the unit and this new state is recorded. The state indicates which unit has been written to and the identity of the writer, the thread’s PDID that serves as a transaction ID. All such states are collected and upon completion of the transaction they are used to identify which access units require after-images to be recorded in the log buffers. Thus the required after-images are collected at the transaction’s commit.

When a region of memory that is to be recoverable is mapped to the application’s address space, a view to provide recoverability must be defined on it, a view that uses appropriate FSM definition. Thus, the FSM must be defined first. The FSM for recoverability purposes is extremely simple, consisting of only two states as is shown in Table 3.1 for the redo method. (The table appears at the end of the paper.)

Because we are not concerned with locking or other transactional properties besides recoverability, the PDID’s of the threads are not relevant in determining the FSM transitions and thus the corresponding FSM entries are all N/A. PDID’s would be relevant, for instance, in an FSM defined for locking of units [Bodorik, 1998]. If a thread locks a unit in an exclusive mode, the thread’s PDID is recorded with the state of access. Subsequently, a thread would be allowed access to that unit only if its PDID matches the PDID that is recorded in the unit’s state of access.

The FSM for recoverability has two states only: Clean, the initial state for all units, and Dirty. When a thread issues a memory read operation to a byte(s) of the unit, there is no transition in the unit’s state. A write to an access unit in a Clean state, however, causes a state transition from Clean to Dirty. To stress, the state transition occurs only on the first write to the unit; thus, writes to a unit that is in a dirty state simply proceed without any action or changes to the state of access to that unit.

The FSM shown in the table is defined for the case when only after-images are required. Thus, on a first write to an access unit, a state transition from Clean to Dirty is recorded for that unit and the thread may proceed. The case of before-images will be discussed shortly.

Once the FSM is defined, a view is defined by the application on the recoverable segment. The view specifies which FSM governs access to the units of the view and the fixed size of units for the view. Also specified is the identity of the manager so that the kernel can communicate with it.

Consider now threads accessing the recoverable memory region on which a view for recoverability is defined. Recall that the initial state for each unit of the view is Clean. A read access to a unit by a thread proceeds ahead without any state transitions. A write to a unit that is in a Dirty state also proceeds ahead. A write to a unit in a Clean state, however, causes a state transition from Clean to Dirty. The state transition is recorded by the kernel in its internal buffers and the thread proceeds with the write access. If the kernel’s buffer is full of state transitions, the transitions are removed from the buffer and are forwarded to the manager.

When a thread finishes, it informs the manager and it, in turn, informs the kernel to pass on all state transitions caused by the thread to units of the recoverable memory region. Once the manager receives the state transitions, it ensures that the after-images of the affected units are collected in the log buffers. It also instructs the kernel to set the state of these units to Clean.

To achieve logging of before or after-images, a view is defined on a memory segment that is to be recoverable. The access units of the view are the data units and are the units of recoverability: their size, static/fixd within the view, varies across views and is defined by the application. The view’s FSM, which synchronizes access to the units of the view, is defined to enable the capture of before and/or after-images. If a before-image is required, the FSM is defined to raise a (write access control fault) exception on the first write to an access unit so that the before-image can be recorded in a log buffer before the unit is updated. If an after-image is required, the state of access to the unit changes on the first write to the unit and this new state is recorded. The state indicates which unit has been written to and the identity of the writer, the thread’s PDID that serves as a transaction ID. All such states are collected and upon completion of the transaction they are used to identify which access units require after-images to be recorded in the log buffers. Thus the required after-images are collected at the transaction’s commit.

When a region of memory that is to be recoverable is mapped to the application’s address space, a view to provide recoverability must be defined on it, a view that uses appropriate FSM definition. Thus, the FSM must be defined first. The FSM for recoverability purposes is extremely simple, consisting of only two states as is shown in Table 3.1 for the redo method. (The table appears at the end of the paper.)

Because we are not concerned with locking or other transactional properties besides recoverability, the PDID’s of the threads are not relevant in determining the FSM transitions and thus the corresponding FSM entries are all N/A. PDID’s would be relevant, for instance, in an FSM defined for locking of units [Bodorik, 1998]. If a thread locks a unit in an exclusive mode, the thread’s PDID is recorded with the state of access. Subsequently, a thread would be allowed access to that unit only if its PDID matches the PDID that is recorded in the unit’s state of access.

The FSM for recoverability has two states only: Clean, the initial state for all units, and Dirty. When a thread issues a memory read operation to a byte(s) of the unit, there is no transition in the unit’s state. A write to an access unit in a Clean state, however, causes a state transition from Clean to Dirty. To stress, the state transition occurs only on the first write to the unit; thus, writes to a unit that is in a dirty state simply proceed without any action or changes to the state of access to that unit.

The FSM shown in the table is defined for the case when only after-images are required. Thus, on a first write to an access unit, a state transition from Clean to Dirty is recorded for that unit and the thread may proceed. The case of before-images will be discussed shortly.

Once the FSM is defined, a view is defined by the application on the recoverable segment. The view specifies which FSM governs access to the units of the view and the fixed size of units for the view. Also specified is the identity of the manager so that the kernel can communicate with it.

Consider now threads accessing the recoverable memory region on which a view for recoverability is defined. Recall that the initial state for each unit of the view is Clean. A read access to a unit by a thread proceeds ahead without any state transitions. A write to a unit that is in a Dirty state also proceeds ahead. A write to a unit in a Clean state, however, causes a state transition from Clean to Dirty. The state transition is recorded by the kernel in its internal buffers and the thread proceeds with the write access. If the kernel’s buffer is full of state transitions, the transitions are removed from the buffer and are forwarded to the manager.

When a thread finishes, it informs the manager and it, in turn, informs the kernel to pass on all state transitions caused by the thread to units of the recoverable memory region. Once the manager receives the state transitions, it ensures that the after-images of the affected units are collected in the log buffers. It also instructs the kernel to set the state of these units to Clean.
In summary, to collect after-images, affected units are identified by their state transitions. Any write to a unit that is in a Clean state causes a transition to a Dirty state and this transition is recorded. The record of these transitions is used to identify the units that were written to and thus after-images need to be recorded in log buffers. It is the manager that records after-images in the log buffers.

Consider now the case when before-images are required. On the first write to a unit in a Clean state, there is an access control fault and the thread is suspended. The kernel informs the manager regarding the cause of the fault, the address of the access unit, operation (write), the current state of access, and the identity of the thread that was suspended. The manager then records the before-image in a log record and instructs the kernel to resume the thread and also to change the state of access to the unit to Dirty. In terms of the FSM definition shown in Table 3.1, the result of a write to a Clean unit would be (an access control) Fault instead of Proceed. When a thread is finished with accessing the recoverable region, it informs the manager and it, in turn, instructs the kernel to forward it all state transitions recorded for units of the recoverable region. Once the manager receives this information it collects after-images (if they are required) – thus before and after images are collected for recovery methods that perform both redo and undo operations. Finally, the manager instructs the kernel to re-set the state of affected units to Clean.

4. Evaluation

In this section, the MVM approach to the provision of recoverable memory is compared to the approaches taken by Logged-RVM and Light-RVM. The fundamental differences are analyzed in order to determine advantages and disadvantages of the approaches. Camelot is not considered here as Light-RVM was developed as a dedicated approach to provision of RVM that is more efficient than the RVM provided by Camelot [Satyanarayanan 1993]. Furthermore, comparison of the logged and light RVMs has been performed in [Cheriton 1995]. Before proceeding with comparison of the three systems, at the risk of redundancy, the requirements on the virtual memory in providing recoverable segments are reviewed first. Following this, the methods are compared in qualitative terms. Finally, a quantitative comparison of the three methods is performed. This section also serves as a review of the logged and light RVM methods.

4.1 Requirements

To provide recoverable memory, the main task is to record the before and/or after-images of bytes written to a recoverable memory region, that is, the main function is the creation of log records. The log record contains either the before or after-image, together with the transaction ID, address and the size. If an after-image is required, it can be obtained when the application is finished as long as which addresses in the recoverable region were written to are recorded. When the application (transaction) is finished, the required after-images must be collected into the log buffers for further processing. If a before-image is required, the system must ensure that a before-image is captured in a log record prior to writing taking place. Note that this is not a strict requirement for transaction processing when the data is assumed to exist on a secondary storage. In such a case, it is sufficient to note the addresses of writes while the before-images can be retrieved later from the secondary storage. Nevertheless, even in such a case, the log records must be created that specify which data has been updated.

Furthermore, there are two aspects to creation of the log record. The triggering of a log record creation and making the log record itself. Recall that for the purposes of this paper we only consider how log records are created and not how they are processed. For the transaction environment, for instance, the log records must be written to stable storage – how that is done is not considered here as the same methods can be used in all three approaches.

4.2 Qualitative evaluation

We compare the methods in terms of how logging is triggered and how the records are made in log buffers.

4.2.1. Light-RVM. In Light-RVM, making of a log record is triggered by an explicit call by a programmer. It is thus the programmer’s responsibility to annotate each write to a recoverable region of memory by an explicit call to a library routine with parameters indicating the address and the size of data to be written/updated. The invoked routine records the parameters and the before-image in a log buffer, if the before-image is required. If an after-image is required, the address and the size are noted and the image is collected upon transaction’s commitment (or prior to writing a log record from the log buffer to the secondary storage). Creation of a log record is thus a simple loop to copy the portion of the recoverable region, annotated by the programmer, into a log buffer. The main problem is that the programmer must explicitly state what is to be logged.

4.2.2. Logged-RVM. In Logged-RVM, each and every write memory operation is intercepted on a bus by hardware. The address and the size (byte/word) are thus known and are recorded, together with the transaction ID and an after-image, in a 16-byte log record. Recall that this is achieved by a special purpose memory write-through mechanism that snoops on the bus for writes to the cache in order to propagate them to physical memory. In addition to writing the data into memory, the information listed
above is stored in a FIFO queue. Items are removed from the queue by hardware that DMAs the log records into physical memory, the log buffer. Because the FIFO queue has a finite number of entries, there is a danger of overflow if many consecutive writes are performed to a recoverable memory region. Furthermore, if the application repeatedly writes to the same address, a log record is created for each write even though one final after-image suffices. In addition to the specialized write-through hardware the bus itself must be modified so that the snooping hardware can recognize which addresses belong to recoverable memory regions and which do not so that writes to non-recoverable regions are not logged. Note that the before-image cannot be captured because the hardware mechanism snoops on the bus for writes and thus sees only the updated value and not the value before the update.

4.2.3 Multi-view RVM

In MVM, like in Logged-RMV, recording of logged records is triggered by the memory write operations themselves – programmer annotations of writes to recoverable memory regions are not required. Log records are created for access units of views in a manner similar to Light-RVM, in that once the creation of the log record is triggered, a copy of the whole data item is made in a log record, not just a byte/word as is the case in Logged-RVM. Although the access units have fixed size within a view, they vary across views and are defined by the application. When a before-image is required, there is a write access control fault on the very first write to the unit so that the log record with a before-image is created. If an after-image is required, the state of the access unit changes on the first write to indicate that a write occurred to the unit but the application thread proceeds without being interrupted. The states are collected in a hardware FIFO queue and are forwarded by the kernel to the manager when the thread is suspended (or the buffer becomes full). On transaction’s completion, these states are used to create the logs containing after-images of the updated records.

Recall that the MVM can be used for synchronization of either threads within the same address space/task or tasks/processes themselves in which case the manager is a separate process. For comparison with Light and Logged RVM’s, however, the manager executes in the same task/process as the application requiring recoverable memory regions in order to avoid heavy-weight context switches. This is also the approach used in Light RVM.

4.2.4 Summary. In summary, Logged-RVM utilizes dedicated hardware that monitors every write operation and records after-images in a FIFO queue. The method does not, however, provide for recording of before-images and thus cannot support the undo recovery methods. Finally, if there are frequent writes to recoverable memory regions, there is a danger of overflow because the FIFO queue is limited. In case of overflow log records are lost.

Light-RVM should be a very efficient software approach because the routines that perform logging are mapped into the application’s address space. The difficulty with this method is that the programmer must invoke the logging routines explicitly.

Multiview-RVM does not require a programmer to make explicit calls to logging routines – logging is performed automatically. Log records are created for updates to data units that are access units of a view. Although their size varies across views, it is fixed within a view.

4.3 Quantitative evaluation

Light-RVM was evaluated in [Satyanarayanan 1993] and improvements in performance over Camelot were reported. Both Camelot and the experimental Light-RVM software were compared using the TPC-A benchmark and performance was reported in transaction throughput per second. The exact details on the hardware platform, operating system, and other system and application parameters, however, are not available.

Performance of Logged-RVM was compared to that of Light-RVM in [Cheriton 1995] also using the TCP-A benchmark with results reported in transaction throughput per second. Both were evaluated on the same platform, but, as in the previous case, the detailed hardware platform, operating system and application parameters are not available.

As a consequence of the above, we cannot try to estimate the throughput achievable when recoverable memory is provided by the MVM and compare it to that reported in [Satyanarayanan 1993] or [Cheriton 1995]. Furthermore, because neither Logged-RVM nor Light-RVM is available to us, we cannot obtain their transaction throughput under controlled environment in which we have control over all relevant parameters. Also the TCP-A benchmark is now obsolete. Results from TCP-A benchmarks are no longer validated by the Transaction Processing Council.

In face of these difficulties, we chose to compare the three methods using the observation made in the qualitative evaluation in that the recording of a log record consists of two subtasks: triggering of the logging process and actually making the log record containing the before and/or after-image. Because the Logged-RVM captures after images only in log records, we also consider logging of after-images before logging of before-images.

Instead of trying to derive the throughput of transactions using recoverable regions for each of the three methods, we attempt to estimate the delay to make a log record. The estimates will be made using a number of assumptions and thus we attempt only to determine whether the delays in
the same order of magnitude. For simplicity, we shall exclude delays due to page faults in all three cases.

Because the Logged-RVM is used for recording of after-images only, we ignore logging of before-images in this subsection.

4.3.1. Logged-RVM. In [Cheriton 1995] it is reported that a single recoverable write takes 16 cycles in Logged-RVM. This includes the cost of performing the single write operation and adding a record to the log buffer. The data-cache line size was 16 bytes. The 68040-based system has write instructions that can write at most 2 bytes. Thus, to make a single write to a recoverable region of memory (i.e., to write one word (2 bytes)), 16 cycles are taken to make a corresponding record in the log buffer. It should be noted that the log record is in a buffer in the physical memory.

4.3.2. Light-RVM. In Light-RVM, the programmer explicitly invokes a routine to make a note of the address and the size of the data item that requires the after-image to be taken. Because it is assumed that this routine is linked to the application, the delay to perform this is equivalent to the delay of making a simple library routing call plus the delay of recording the address and the size in local data structures. For simplicity, we shall assume that this recording is equivalent to 3 reads and 3 writes. Two reads and two writes are used to save the address and the size in a queue and a single read and write are used to update the queues’ rear pointer. The creation of the log records containing after images is performed on transaction’s commit. This entails a loop to go through the queue containing the record of the transaction’s updates. For each entry (containing the address and size), the updated data is moved to the log buffer together with the transaction’s ID and the address and the size of the update.

There are two components to writing a log record: fixed and variable that depends on the size of the updated data item, that is, on the number of bytes written by the application. Thus for each log record the following fixed overhead is assumed: 3 reads and 3 writes to record the transaction ID, address and size. An additional 3 reads, 3 increments, 3 decisions and 3 writes are required for looping and moving the pointers to move to the next item in the queue and the next log record. Finally, additional read, write and decision are required for the controlling loop to record all required log records. Thus, for each log record there are 20 operations (reads, writes, and decisions). The variable component of the work per log record entails recording of the after-image. If it is assumed that there are X bytes to be recorded, then X/2 reads and writes (when a word has 2 bytes) are need to record the after-image plus X/2 increments and decisions for the controlling loop.

We make a simplifying assumption in that each read, write and decision making instruction takes one cycle [Hennessey 96], as was done in determining cycle delays in [Cheriton 1995]. Thus, for each log record containing an after image of size X bytes there are A + B * X/2 cycles expended, where A = 21 cycles is the fixed overhead while B * X is the variable overhead, where B = 2 cycles and X is the size of an after-image in bytes. Note that A does not include the cost of the library call which we shall deal with later.

The above estimation assumes no misses on the data cache(s). Because the cycle delay for the Logged-RVM includes the delay to move the log record to the physical memory, we need to include it also in delays for the other methods. For comparability, we assume the cache line size of 16 bytes (8 words) and memory latency of 6 cycles as was reported in [Cheriton 1995] for Logged-RVM. Thus, to move a log record from the cache to physical memory, one cache line containing the address, the size of data and the transaction ID needs to be written from cache to memory incurring a latency of 48 cycles. In addition, if the after-image has X bytes, X/16 * 48 cycles are needed to write X/16 cache lines to memory.

In summary, 21 + 2 * X cycles are required to store one log record in virtual memory, a log record that contains an after-image. To this latency of X/16 * 48 cycles is added to propagate the log records to physical memory. Recall again, that the cost of a library call must be added to this.

4.3.3. Multi-view RVM. Recall that when read/write access is made to a recoverable memory region on which a view is defined, the PCU is invoked on each data cache miss or write access control fault. The PCU determines whether the thread can proceed with the access and also records a transition in the state of access to the unit (if any).

In the PCU design under consideration, caches are used to store relevant information. The minimum delay was estimated to be 19 cycles to make the decision whether to proceed or access control exception is to be raised plus to record the state transition (if any). Note that all caches, with the exception of the cache to store the state information, are assumed to be sufficiently large not to have misses. The minimum delay of 19 cycles is under the assumption that there are no misses on the state cache. If we assume the same miss rate for this state cache as for a (first level) data cache, reported in [Hennessay 1996] to be in the range from 0.5% to 10%, then the cycle delay increases to between 24-37 cycles. Instead of a range, we use average of the two values in subsequent presentation, that is 31 cycles. To this, we must add the cost of making the actual log records from the buffered state information. Fortunately, this is the same operation and thus delay as for the Light-RVM, a delay of 21 + 2 * X cycles, where X is
the size of the after-image. The total delay is thus estimated to be 50+ 2*X cycles to create a log record with an after-image in virtual memory plus the delay to propagate the record to physical memory, X/16 * 48 cycles.

4.3.4. Comparison. Before we can proceed, the delay of the library call for Light-RVM must be determined. Recall that the job of the library routine is to record the transaction ID, address of the update, and the size. In addition, if the after image is required, it also has a loop which records the before image in the log buffer. Only few registers are needed for this task and thus little overhead is required in saving/restoring the context of the of the application when the Light-RVM library routine is invoked. We shall thus assume overhead of 12 cycles for round trip (invocation plus return) which must be added to the Logged-RVM delays calculated in a previous subsection.

The three methods are compared in Table 4.1 for one update of X bytes by the application. The cache line size is assumed to be 16 bytes and memory latency 6 cycles.

<table>
<thead>
<tr>
<th>Table 4.1</th>
<th>Delays (in cycles) to create one log record with an after-image of X bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logged-RVM</td>
<td>8 * X</td>
</tr>
<tr>
<td>Light-RVM</td>
<td>33 + 5 * X</td>
</tr>
<tr>
<td>Multiview-RVM</td>
<td>50 + 5 * X</td>
</tr>
</tbody>
</table>

The table shows that the methods have delays of the same order of magnitude. The software method (Light-RVM) is an efficient alternative to hardware methods. It should be noted that the estimation of the number of cycles required for the software was very optimistic and higher delays would occur in reality. For instance, only 12 cycles for invocation/return were used. Furthermore, for the Logged-method, the delay is primarily due to the memory latencies and it is by now much lower.

4.4. Evaluation summary

The estimated delays show that it is not likely that the hardware support for the purposes of efficiency in achieving recoverable memory would be justifiable. Perhaps the most important argument towards the hardware approaches is that they remove the responsibility from the programmer to annotate the updates in order to invoke the logging software.

5. Related Work

Camelot, light-weight recoverable virtual memory and logged recoverable virtual memory were already discussed in detail in the introduction and in the previous section and are not discussed here any further.

The protection schemes found in the Hewlett Packard's PA-RISC [Wilkes 1992] and the PLB protection organization [Koldinger 1992] are also closely related to our work in that they show a separation of protection information from address translation. Both the PA-RISC and PLB protection schemes are designed for the 64-bit single address space operating systems in which protection is on access units being pages. Our architecture is structured to provide for variable sized access units within multiple address spaces or within a single virtual address space of one application, but it can be extended easily to the 64-bit single-address space operating system environment. It is further distinguished by support of control strategies through the FSM supported protocols on the access units.

[Chang 1998] reports on one of the first complete attempts to incorporate transaction management functions in a MMU within the IBM's 801 storage architecture. A hardware locking mechanism monitors the read and write references of individual transactions and provides for locking of 128-bytes units of data for writes and pages or 128-byte size units for reads. It thus provides a fine granularity of locking. Furthermore, the scheme also provides for lock acquisition in hardware without software intervention. As in the MVM scheme, information about units automatically locked by a transaction must be collected periodically. The IBM Power architecture does include record-level hardware protection, but this was excluded from the PowerPC architecture.

6. Summary and Conclusions

This paper presented how the Multi-view virtual memory provides recoverable regions of memory. When a recoverable memory region is mapped to the application’s virtual address space, a view is defined on it. The view defines the size of access units and also an FSM that is used to control access to the units of views by application’s threads. If before-images are required, the FSM raises a write access exception on the first write to a unit so that the before-image can be logged. Subsequent writes to the unit proceed without interruption. If an after-image is required, the FSM records a state transition from Clean to Dirty on the first write to the unit. The state transitions are recorded and when a transaction is completed, they are used to create log records with after-images.

The method was compared to light and logged RVMs. In comparison to Light-RVM, the application programmer is not required to annotate updates to ensure the creation of log records. In comparison to Logged-RVM, both before and after-images can be captured in log records and thus both redo and undo methods can be supported.

The delays incurred by all three methods are of the same order of magnitude. The Light-RVM thus appears to be an
attractive alternative because it is software-based and thus does not require expensive hardware.

Whether Logged-RVM will be adopted in computer systems in the future remains to be seen. In [Chang 1988], transaction management functions were incorporated in a MMU. The IBM Power architecture does include record-level hardware protection, but this was excluded from the PowerPC architecture. It thus appears that there is reluctance to incur expenditure on hardware support that can be utilized by one application only.

The protection schemes found in the Hewlett Packard's PA-RISC [Wilkes 1992] and the PLB protection organization [Koldinger 1992] show a separation of protection information from address translation. Thus, protection hardware support applicable to many applications is deemed to be a worthwhile expenditure. It is thus the versatility of the MVM that is important. The size of access units is not fixed, but is customizable to applications. The MVM model provides support for many access control protocols, recoverability is only one of them. Because in the MVM model approach the access control protocols are specified by FSMs with their states and transitions stored in tables, any protocols that can be represented by FSMs can be supported. Various protocols supported by the MVM model can be found in [Jutla 1997a]. The performance of a database locking application using the MVM model is detailed in [Bodorik 1998, Jutla 1997a]. The performance of a database locking application using the MVM model is detailed in [Bodorik 1998, Jutla 1997a, Jutla 1998]. We are currently completing a performance evaluation when MVM is used for locking and recoverability and also investigating the applicability of MVM in provision of coherence in Distributed Shared Memory systems. Our goal is to show that hardware support provided by MVM is shared by many applications and is thus a feasible investment.

References


<table>
<thead>
<tr>
<th>Memory Operation</th>
<th>Current State</th>
<th>PDIDs irrelevant</th>
<th>PDIDs match</th>
<th>PDIDs do not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Clean</td>
<td>Proceed</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Read Dirty</td>
<td>Proceed</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Write Clean</td>
<td>Proceed Dirty</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Write Dirty</td>
<td>Proceed</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 3.1 State Transition Table for Recoverability
(An asterisk (*) indicates not applicable. A dash(-) indicates no state transition.)