The DataSafe Failure Recovery Mechanism in the Flask Architecture

S.J.G. Scheuerl, R.C.H. Connor, R. Morrison & D.S. Munro

School of Mathematical and Computational Sciences
University of St Andrews
North Haugh, St Andrews, Fife, KY16 9SS, Scotland

{stephan, richard, ron, dave}@dcs.st-and.ac.uk

Abstract

A major design goal of the Flask architecture is to separate the mechanisms of concurrency control and recovery management in database programming systems. This paper describes the DataSafe component of Flask, which is the second recovery mechanism to be implemented within the architecture and therefore provides a proof of concept. The DataSafe is closely based on the DB Cache mechanism, modified to fit into the Flask architecture. The major modification comprises the use of a separate safe map which allows pages of recovery data to be block aligned and affords opportunities for efficiency gains during recovery. The page-level locking implicit in the DB Cache is lifted from the DataSafe, permitting concurrency control and recovery to be independent.

Keywords concurrency, recovery, persistent stores

1 Introduction

Flask [11] is a layered architecture which has the flexibility to support different models of concurrency and different recovery mechanisms over the same data. The architecture has no fixed notion of concurrency control; instead it provides a framework in which models can be designed and supported. The Flask architecture is designed to work with processes or actions that maintain global cohesion under control of the concurrency control schemes. The significant events defined by a particular concurrency control scheme are generated from and reported to the higher layers of the architecture enabling these schemes to undertake conflict detection. This approach isolates the memory management from the onus of interference control and hence enables instantiations of lower layers of the architecture to be constructed independently of any particular concurrency model. An important design aim of Flask is that it has the flexibility to support different recovery mechanisms.

The importance of supporting several recovery mechanisms within a single architecture has been highlighted by analytical modelling using the MaStA I/O cost model [12, 14]. The MaStA model provides an analytical framework for comparing recovery mechanisms under a variety of different workloads and configurations. It has been shown that the cost of recovery mechanisms can be critical to the overall performance of data-intensive applications, with I/O bandwidth being a limiting factor. Many recovery mechanisms have been invented, each with different performance tradeoffs [8]. Each technique's cost involves not only the overhead of restoring data after failures but also the time and space overhead required to maintain sufficient recovery information during normal operation to ensure recovery. Under different workloads and configurations these crash recovery mechanisms exhibit different costs.

This paper details the DataSafe recovery mechanism which conforms to the Flask abstractions. DataSafe is based on the DB Cache recovery mechanism [6] but differs in two key aspects:

- It conforms to the Flask layering by using a meld propagation strategy to allow concurrent updates to a page and to avoid the low-level page-locking of the DB Cache and hence false conflicts.
- It uses a map to store the locations of pages of recovery information on disk instead of tagging the page headers. This allows pages of recovery information to be block aligned on non-volatile storage and affords opportunities for efficiency gains during recovery.

The description of the DataSafe given here does not attempt compare the mechanism against others since it is not the intention to design a recovery mechanism which works best for all applications. The goal is to provide an architecture in which the recovery mechanism and concurrency model may be chosen to suit the application. The DataSafe offers an alternative to the concurrent shadow paging of the original implementation of Flask. Thus within Flask either mechanism may be chosen to suit the characteristics of a particular application.

2 The Flask Architecture

Flask is a layered architecture which has the flexibility to support different models of concurrency, such as atomic [4, 5, 7] and nested transactions [13], over the same data. Flask also permits the use of different recovery mechanisms such as shadow paging and logging over the same data. The architecture does not have a built in notion of concurrency control. Instead Flask provides a framework in which concurrency models may be designed and supported, and in which recovery

Proceedings of the 19th ACSC Conference, Melbourne, Australia, January 31-February 2 1996.

mechanism may be engineered to best suit the persistent application.

At the top layer a concurrency control design method, such as CACS [15], is used to specify concurrency models. Global cohesion of the data operated on by concurrent activities is maintained by CACS and is expressed in terms of the visibility of data from actions. The global cohesion is modelled in CACS as the movement of data between a globally visible persistent store and conceptual stores called access sets, and the movement of data between access sets. Figure 1 illustrates access sets and a persistent store within a conceptual concurrent architecture.

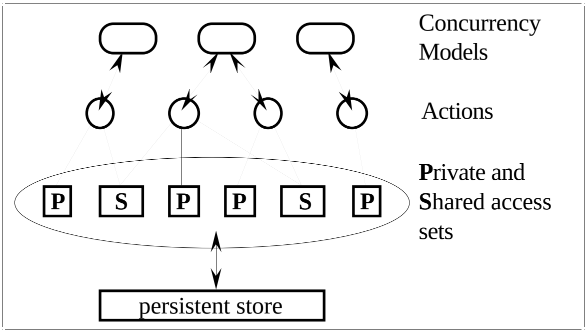


Figure 1: Conceptual Concurrent Architecture.

Each action is associated with a private access set which isolates its view of data from all others. Actions may also use shared access sets when the concurrency model permits co-operative work between actions. Atomicity of Flask ensures that the movement of data between access sets and the persistent store is performed as an atomic unit. This property is used by the recovery management layer of Flask to effect an atomic meld, the process of making changes made to data in access sets globally visible in the persistent store. The semantics of a meld may differ according to the concurrency model.

By separating out the issues concurrency control the data management layers of Flask are relieved from the burden of interference management, and may therefore make use of recovery mechanisms designed independently of the concurrency model.

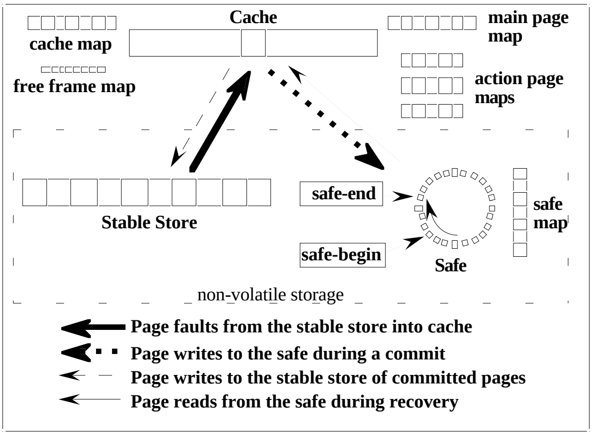
The interface to the recovery layer is sufficiently flexible to allow a number of mechanisms to be used. The interface permits the creation, deletion, and context switching between access sets. Reads and writes are performed by the higher layers of Flask with respect to the current access set. Melds may be performed on some set of access sets. When a meld is performed, all updates made with respect to the access sets are atomically and consistently propagated to the stable store. In many cases the meld process may be performed by the recovery mechanism without the need for call-backs to the higher layers, using logical operations.

In concurrency models which permit actions to share updates, data may also be accessed and updated within shared access sets. The consistency of melding a shared access set is the responsibility of the concurrency control at the top layer of Flask. A shared concurrency model may require a number of access sets to be melded as an atomic action (e.g. a private and a shared access set, used by some action).

In the following design the focus is not on specifying concurrency models but on the provision of access sets and the atomic update of a recoverable persistent store using the DataSafe recovery mechanism. It is assumed that a higher-layer concurrency control is performing object-level conflict detection with respect to access sets.

3 The DataSafe

The DataSafe is a page log recovery mechanism based on the DB Cache [6]. The DataSafe ensures the recoverability of a stable store by controlling the movement of pages of data among three areas of storage: the stable store, a safe and a cache. The layout of the mechanism is illustrated in Figure 2.



Cache
cache map
free frame map
main page map
action page maps
Stable Store
safe-end
safe-begin
Safe
safe map
non-volatile storage
Page faults from the stable store into cache
Page writes to the safe during a commit
Page writes to the stable store of committed pages
Page reads from the safe during recovery

Figure 2: Layout of the DataSafe.

The stable store is held on non-volatile storage and only contains pages which have been melded. The cache is held in volatile storage and contains pages which have been read, pages which have been updated, and pages which have been updated and melded but not yet written to the stable store. The safe is held on non-volatile storage and contains copies of melded cache pages which may not yet have been written the stable store. For the moment it is assumed that the cache is sufficiently large to hold all updated pages between melds.

Reads operate on data in the cache, faulting stable store pages into free cache frames as required. Writes are also performed on pages in the cache. Updated pages remain in the cache at least until the user melds or aborts. A meld operation involves writing updated cache pages which are part of the meld to contiguous pages in the safe. This ensures that in the event of a crash, melded cache pages which have not yet been written to the stable store are recoverable. In the case of a successful meld these pages either remain in the cache to be reused or are written to the stable store opportunistically.

A cache map in volatile storage is used to record the state information (original, melded or updated) of cache frames. A free frame map is used to indicate which cache frames are free and which are in use.

A main page map records the cache locations of store pages which have been faulted into the cache but not updated. Each action has its own private access set and is associated with an access set page map. Similarly an access set page map is created for each shared access set required by the concurrency model. Entries are added to these maps as cache pages are updated by actions.

A safe map on non-volatile storage holds the state information of the pages in the safe. A safe-begin-pointer and a safe-end-pointer also held on non-volatile storage designate the area of the safe which contains pages which are required for recovery.

If there are no free cache frames available to fault a page from the stable store a cache frame containing a page which has been melded or only read is selected for replacement. If the selected frame contains a page which has not yet been written to the stable store then it is written to the stable store before being replaced. This ensures that all read faults operate on the stable store rather than on the safe.

If there are insufficient free pages on the safe to complete a meld, safe pages which are required for recovery are written from the cache to the stable store. This means that they are no longer required for recovery in safe and as such may be overwritten during the meld.

During recovery the safe pages required for recovery are read from the safe into the cache after which normal processing resumes.

3.1 The Safe

The safe is designed as a circular buffer to enable writes to the safe to be performed sequentially. Being circular, the safe also bounds the amount of data that is maintained to ensure recovery and therefore bounds recovery time. The safe must be at least as large as the cache to ensure that all pages updated in the cache may be written to the safe.

Since the same page may be updated and melded to the safe many times, the safe may contain more than one version of the same page. Only the latest version of a page in the safe is required for recovery and then only if the corresponding cache page has not yet been written to the stable store. Thus a safe page is free if the corresponding cache page has been written to the stable store or if a more up-to-date version of the page is in the safe.

The safe map contains the stable store locations of pages in the safe. The safe map also contains a tag for each page in the safe. These tags are used during normal processing to indicate which safe pages are required for recovery and which are free. The safe-begin-pointer and the safe-end-pointer designate the area of the safe that contains pages which are required for recovery; although not all pages in this area are necessarily required for recovery. The safe pointers therefore also indicate the active part of the safe map, a copy of which is held in main memory for efficiency.

3.2 The Cache

For the moment the cache is designed to fit into main memory to avoid operating system page swapping. The cache is composed of a number of page sized cache frames which are empty or contain cache pages (pages of data). Cache pages are categorised as originals, melded or updated. Originals are duplicates of stable store pages. Melded cache pages are pages which have been changed and melded but not yet written to the stable store. Updated pages are copies of original or melded cache pages and are pages which have been updated but not melded. The cache map is used to tag cache pages accordingly.

The DataSafe holds an action's updated pages in the cache at least until the action melds or aborts. This avoids the need to maintain redundant undo information since non-melded updates are never swapped to the stable store. It also avoids swapping pages to the safe which eliminates the need to read the safe during normal processing.

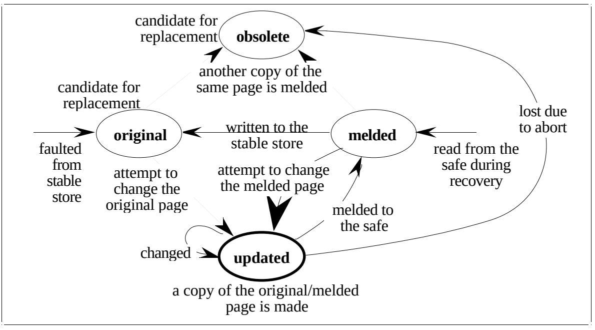

candidate for replacement
obsolete
another copy of the same page is melded
candidate for replacement
original
written to the stable store
melded
lost due to abort
faulted from stable store
attempt to change the original page
attempt to change the melded page
read from the safe during recovery
melded to the safe
changed
updated
a copy of the original/melded page is made

Figure 3: Cache Page State Diagram.

Figure 3 gives the state diagram of cache pages. When a page is faulted from the stable store the cache page is tagged as original to signify that the page has not been updated and that it may be selected for replacement when the cache becomes full. When a write is performed on an original cache page a copy of the original is made in a free cache frame. The write is performed on the copy and the copy is tagged as updated. An updated page may have further changes made to it, be lost due to an abort or system crash, or be written to the safe during a meld operation. When an updated page is written to the safe it is tagged as melded to signify that the page must be written to the stable store before being replaced.

During recovery, cache pages read from the safe are tagged as melded since it is assumed that the corresponding melded cache pages had not been written to the stable store before the crash. When a melded page is written to the stable store it is tagged as original to signify that the page may be replaced. When an attempt is made to change a melded page the update is performed on a copy of the page, made in a free cache frame, in a similar manner to an attempt to update an original. An original or melded page becomes obsolete if a copy of the page is melded.

3.3 Per-action Page Maps

A potential problem of concurrency which must be addressed by the DB Cache and other page based recovery mechanisms is that actions may make requests to modify the same page. When an action melds a modified page the page becomes the latest version. This includes the changes made by any non-melded actions, which may later abort, that modified objects on the same page.

One common solution and one used in the DB Cache is to use page-level locking whereby an action obtains an exclusive lock on a page before updating the page [1, 9]. Each action maintains its own page map and the locking guarantees that a page is never in more than one page map. The main drawbacks of this solution are firstly that it introduces false conflicts where two actions are prevented from modifying different parts of the same page. Secondly it employs a built-in concurrency control mechanism at a low level, a property which Flask avoids. Problems of deadlock must also to be addressed.

The DataSafe solution (similar to the concurrent shadow paging solution [11]) involves keeping a page map for each action and maintaining per-action page copies, i.e., an access set. Object conflicts are detected through the Flask architecture at the concurrency control layer removing the need for such interference management in the memory management layer. When an action first modifies a page a copy of the original or melded page is made in the cache and the modification made to the copy. An original is not updated directly to avoid re-faulting the page from the stable store should another action accesses the page. A melded cache page is not updated directly because the safe purge mechanism described later relies on melded pages which are still in the cache remaining unchanged. If no original or melded version of the page is present in the cache the page is first faulted from the stable store.

The updated page's mapping is then added to the action's page map. To resolve the address of a stable store read the action's page map is searched first and, if still unresolved, then the main page map. This mechanism ensures that the changes made by one action are isolated from other actions and the previously melded state.

3.4 Per-action Meld and Abort

Action abort involves freeing the cache frames containing the pages updated by the action, and removing the action's page map.

During a meld operation cache pages updated by the melding action are written to contiguous free pages in the safe at the location given by the safe-end-pointer. As pages are written to the safe an in-memory copy of the safe-end-pointer is advanced and an in-memory copy of the safe map is updated to record the stable store locations of the safe pages. The cache pages updated by the action are found using the action's page map.

When a cache page is melded any melded or original version of the same page present in the cache becomes obsolete. The obsolete version is found by searching the main page map. A cache frame containing an obsolete page is designated free using the free frame map. The main page map is then updated to record the cache location of the newly melded version of the page.

Once all the required meld writes to the safe have been performed the safe map and the safe-end-pointer are written atomically to non-volatile storage. The safe map is atomically updated using a mirroring technique similar to the concurrent shadow paged store of Flask. The safe map is composed logically of a number of safe map pages. Each page of the safe map is preallocated two blocks on non-volatile storage. A root block on non-volatile storage contains a safe map page table recording the mappings between the logical pages of the safe map and the blocks which contain them. A safe map page is always written to the block which contains the obsolete version of the page. The safe map page table in a cached version of the root page is then updated to indicate the new locations of the safe map pages. The safe-end-pointer also held on the root page is updated to record the position of the last data page written to the safe by this meld. The safe-begin-pointer is not advanced during a meld.

Once the safe map is written to the stable store the root page on non-volatile storage is atomically updated, thus atomically updating the safe map page table and the safe-end-pointer. Atomic update of the root page is achieved by mirroring the page on non-volatile storage. The page contains two date stamps, one at the start of the page and another at the end. These are incremented when the root page is written. The page is written to the block containing the obsolete version of the root page. The date stamps are used during recovery to determine which version of the root page is the most recent. They are also used to determine whether any corruption occurred while writing the root page. With the advent of modern disk controllers the atomicity of these actions is only guaranteed if the ordering of disk writes can be inferred. If they cannot, others schemes may be used to atomically update the root page.

If a crash occurs during a meld all pages written to the safe by the incomplete meld are ignored by the recovery process since the safe-end-pointer and the safe map have not yet been atomically updated. Atomicity of action meld is therefore attained by the atomic update of the root page containing the latest mappings for the safe map and the latest version of the safe-end-pointer.

During the meld the safe page at the location given by the safe-begin-pointer cannot be overwritten since it may be required for recovery. Hence the safe is said to be full when there are insufficient pages between the locations given by the safe-end-pointer and the safe-begin-pointer to complete a meld. In such a case a safe purge is performed to advance the safe-begin-pointer, before the meld begins, by a sufficient number of pages to allow the meld to complete.

When an action melds, the changes it has made become globally visible to all other actions. There is therefore a requirement to ensure that the changes made to a page by the melding action are propagated to any other action holding a copy of the same page.

3.5 Meld Propagation

Since the meld resolution is at a page-level the changes made by the melding action must be propagated to other actions' private copies of the same page. Suppose that two actions A and B share a page but modify different objects on that page. Because of the isolation provided by the DataSafe mechanism, action A can meld without affecting B. For B to subsequently meld it must retain the changes made by action A. A mechanism is required for B to ingest the changes made by action A. The algorithm that meld uses to propagate changes is dependent on the particular concurrency model in operation and is determined at a higher layer of the Flask architecture by a concurrency control design method such as CACS. Under the assumption that the higher-layer concurrency control can detect object-level conflicts there are a number of methods of achieving this.

In concurrency models that require isolation, where the concurrency control ensures that two transactions do not modify the same object, it is possible to use logical operations for efficiency to propagate the changes. For example, in an atomic transaction model, suppose two actions A and B have changed different objects on the same page P and action A melds. The changes made by A to page P can be calculated by an xor of P onto the original page, i.e., as it was at the last meld. This derives a page of changes made by A to page P. These changes can now be xor'd onto action B's copy of page P. So the meld propagation formula can be written as :-

$$P_B = (P_A \textbf{ xor } P_O) \textbf{ xor } P_B$$

where P_A is action A's copy of page P, P_O is the page P as it was at the last meld and P_B is action B's copy of page P. Thus B's version of page P now includes the changes made by A. The propagation can be performed eagerly or lazily on demand. Eager propagation is performed when each action melds. Lazy propagation involves delaying propagating changes until an action accesses the melded updates of another action. Using lazy propagation in the case above means that propagation is not performed if B aborts.

This approach is not restricted to atomic transactions. In co-operative models where the actions agree to change an object to the same value majority rules logical operations can be used.

3.6 Recovery

The meld process in the DataSafe does not write updates to the stable store during a meld. This avoids non-melded updates appearing in the stable store should a system failure occur during the meld. Instead meld writes are performed to the safe and writes of melded pages to the stable store are performed opportunistically after the meld completes. Since these writes may be performed opportunistically some pages may not have been written to the stable store before the system failure.

Recovery involves reading into the cache the safe pages which had potentially not been written to the stable store before the crash. The safe-begin-pointer, the safe-end-pointer, and the active part of the safe map are read from non-volatile storage. Only the area of the safe between the safe pointers contains pages required for recovery. The safe map is scanned to determine the safe pages required for recovery to be read.

If the number of safe pages to be read is greater than the number of pages that fit into the cache a number of pages equal to the excess are read from the safe into the cache and then written to the stable store. The safe-begin-pointer on non-volatile storage is atomically updated to give the location of the next safe page to be read. This avoids re-reading the safe pages that have been written to the stable store should another crash now occur. The cache is then cleared of these pages and the remaining safe pages are read into the cache. Once all safe pages have been read normal processing resumes.

Cache pages read from the safe which are not written to the stable store are tagged as melded using the cache map to ensure that these pages are written to the stable store should they be chosen for replacement. The stable store locations held in the safe map are used to reconstruct the main page map as pages are read into the cache.

By either reading safe pages required for recovery into the cache or writing them to the stable store the DataSafe recovery process ensures that the latest version of every page is either in the cache or in the stable store and thus ensures that no read faults operate on the safe. This strategy ensures that all writes to the safe incur low sequential costs. If the safe were to be read during normal processing extra seek costs would be incurred during a meld to move the device head back to the end of the safe.

3.7 Cache Overflow

When there are no free cache frames available to either fault a page from the stable store or to make copies of cache pages, a cache frame containing either an original or a melded cache page is selected for replacement. Only originals and melded pages can be replaced since updated pages must by definition remain in the cache. A victim selection algorithm takes into account how recently cache pages have been accessed and whether these pages are originals or melded. Originals are given a higher probability of being chosen since choosing a melded page requires it to be written to the stable store before being replaced. In this implementation the victim selection strategy uses a LRU algorithm.

The DataSafe makes use of virtual memory techniques to accommodate workloads which temporarily overflow the cache. When the cache becomes full of updated pages, new update transactions are blocked and current transaction are allowed to complete, making use of some temporary storage to swap updated pages as required. Once all updated pages again fit into the cache new update transactions may begin and the temporary storage freed.

To reduce the probability of using virtual memory, new update transactions are blocked before the cache becomes full of updated pages. The threshold used in deciding when to block these transactions is derived dynamically based on the average number of pages updated by each transaction.

3.8 Safe Purge

Safe purging is the process of writing safe pages that are required for recovery to the stable store. A safe purge is performed automatically if there are insufficient free pages on the safe to write the updated cache pages which are part of a meld. Safe purging in this case advances the safe-begin-pointer by a sufficient number of pages to allow the meld to complete.

Since the area of the safe containing safe pages required for recovery is bounded by the safe pointers, the safe-begin-pointer may only be advanced past safe pages no longer required for recovery. An in-memory copy of the safe-begin-pointer is advanced to the first safe page required for recovery. If there are still insufficient free safe pages between the safe pointers the page at the safe-begin-pointer is written to the stable store and the safe-begin-pointer advanced to the next safe page required for recovery. This process is repeated until there are sufficient free pages between the safe pointers. The safe-begin-pointer on non-volatile storage is then atomically updated. This ensures that the meld does not write updated pages to the area of the safe that is read during recovery should a system crash occur during the meld. The meld may then be performed.

The safe purge mechanism only writes sufficient safe pages to the stable store to permit the meld to complete instead of writing all safe pages required for recovery. This is based on the assumption that applications running over data in the stable store exhibit some degree of locality, and that during a meld some pages already in the safe become obsolete before being written to the stable store due to the melding of newer versions of the same pages. If all safe pages were written to the stable store during a safe purge, unnecessary writes may be performed in flushing safe pages to the stable store which may have become obsolete during later melds.

As mentioned earlier melded pages in the cache are not updated directly. This ensures that when safe pages that are required for recovery are written to the stable store the pages need not be read from the safe since the melded pages are still present in the cache. Therefore writing a safe page to the stable store involves writing a cache page to the stable store.

Provided that the safe is larger than or equal to the size of the cache, safe purging guarantees that there are sufficient contiguous free safe pages starting at the safe-end-pointer to hold all pages that are to be melded. An extreme situation occurs when the cache is full of updated pages which are to be melded to the safe. In this case the cache contains no melded cache pages since they have been written to the stable store due to page replacement. Therefore no pages in the safe are required for recovery and so if the safe is greater than or equal to the size of the cache all updated cache pages fit into the safe.

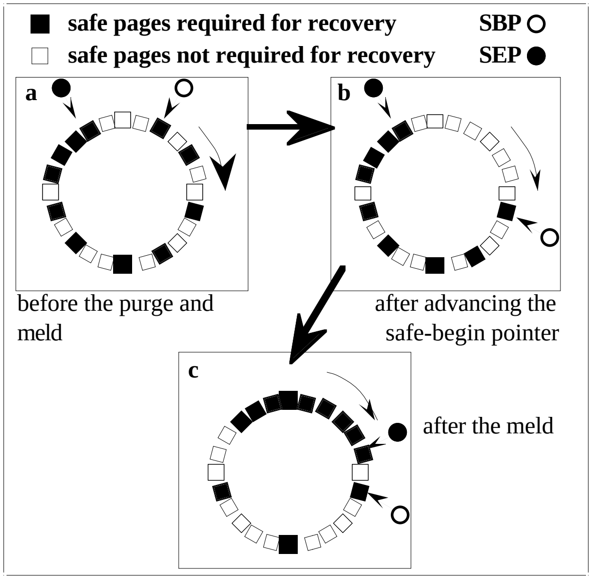

■ safe pages required for recovery
□ safe pages not required for recovery
SBP ○
SEP ●
a
b
c
before the purge and meld
after advancing the safe-begin pointer
after the meld

Figure 4: States of the Safe During a Purge and Meld.

Figure 4 gives an illustrated example of a safe purge and meld. The locations given by the safe-begin-pointer and the safe-end-pointer held on non-volatile storage are shown. In this example 7 updated cache pages are to be melded. Figure 4.a shows the state of the safe before the meld.

Before the meld begins, the mechanism ensures that there are sufficient free safe pages between the two safe pointers to allow the meld to complete. Since there are only 3 free pages between the safe-end-pointer and the safe-begin-pointer (Figure 4.a), a safe purge is performed to advance the safe-begin-pointer by at least 4 pages to provide at least 7 contiguous free pages. Figure 4.b illustrates the safe after writing 2 safe pages to the stable store and shows the new position given by the safe-begin-pointer.

The meld can now proceed. Figure 4.c shows the state of the safe after the meld and shows the new locations given by the safe pointers. This figure also illustrates that some safe pages are made no longer required for recovery through the melding of more recent versions of the pages. This enables the next safe purge to advance the safe-begin-pointer past these safe pages without writing them to the stable store.

3.9 Opportunistic Write Back

So far melded cache pages are only written to the stable store when the cache becomes full, or during a safe purge. Since the safe ensures that melded cache pages are recoverable they may be written to the stable store at any time. These writes may be performed opportunistically by writing melded cache pages to the stable store while no other page faults or writes are being performed. These writes may also be performed in such a way as to take advantage of the position of the head of the non-volatile storage device to minimise the average cost of performing writes to the stable store.

When a melded cache page is written opportunistically to the stable store the corresponding safe page becomes obsolete and is no longer required for recovery. Thus opportunistic writing of melded cache pages reduces the number of safe pages that must be written synchronously to the stable store by a safe purge and through page replacement.

There is a trade-off between writing melded cache pages to the stable store opportunistically and writing the pages synchronously only when required through page replacement or during a safe purge. An opportunistic write policy may be adopted on the assumption that the pages are eventually written to the stable store through page replacement and safe purging, and by performing these writes asynchronously the overall cost of writing to the stable store is reduced. On the other hand by adopting a pessimistic write policy by which melded cache pages are only written to the stable store through page replacement or during safe purging, the mechanism allows melded cache pages to become obsolete and therefore avoids writing some pages which would have been written by an opportunistic policy.

In this implementation an opportunistic writing strategy is adopted which makes use of the LRU algorithm used for page replacement victim selection. When a melded cache page is written to the stable store through page replacement or safe purging the mechanism searches the main page map for entries corresponding to melded cached versions of adjacent stable store pages. If such pages are found to be older than the average age of all cache pages then the pages are also written to the stable store. By only writing older melded cache pages opportunistically the mechanism allows pages which are frequently updated and melded to become obsolete and thus avoids writing them unnecessarily.

4 Related Work

The DataSafe is a variation of the DB Cache [6], a recovery mechanism designed to provide high throughput of short transactions and designed to reduce the cost of recovery. Both the DB Cache and the DataSafe are comparable to deferred page log mechanisms in the manner in which they utilise a sequential log called the safe during normal processing to reduce the cost of melding cache pages, and to ensure that melded pages that have not yet been written to the stable store are recoverable after a crash. Melded pages in the cache may be written opportunistically to the stable store, thus potentially reducing the cost of performing such writes.

Instead of a safe map, the DB Cache uses safe page headers to record the stable store locations of the safe pages. Two bits held in each page header are used to determine which pages are associated with successful melds and therefore which are required for recovery. A safe-begin-pointer on non-volatile storage indicates the first of these safe pages. Since there is no safe map to indicate which safe pages are required for recovery the DB Cache must read all safe pages starting at the safe-begin-pointer during recovery. If a page occurs more than once on the safe the older version in the cache is overwritten with the new version. Pages read from the safe which are found to be part of an unsuccessful meld are discarded.

If the cache becomes full of updated pages, the DB Cache swaps updated pages from the cache back to the database and makes use of a separate before image log to ensure that inconsistencies can be undone after a crash. Once these pages are melded to the safe, the log is discarded. This mechanism is unsuitable for the DataSafe since its cache may contain multiple copies of the same logical page, which cannot be swapped to the same physical database page. Instead the DataSafe makes use virtual memory as described in section 3.7.

To isolate actions from one another the DB Cache uses page locking to ensure that only one concurrent action can update a given page.

5 Evaluation

In contrast to the DB Cache, the DataSafe incurs the added cost of maintaining a safe map. The cost of reading this map during recovery is offset by the fact that once the safe map has been scanned, only those pages which are required for recovery are read from the safe rather than all safe pages. Since few pages are required to hold the active part of the safe map (the part corresponding to safe pages between safe pointers), the extra cost of reading the safe map at recovery time is more than compensated for by reading fewer safe pages.

The DataSafe incurs the added cost of updating the safe map at meld time. In this design each safe page requires a one word entry in the safe map. In a system with pages of size N words, approximately P/N safe map pages are written during each meld, where P is the average number of pages updated by a transaction.

Through the use of a safe the DataSafe and the DB Cache ensure the recoverability of melded pages which have not yet been written to the stable store. Thus these mechanisms are free to opportunistically write melded cache pages to the stable store. With small transactions and by maintaining sufficient large pool of melded pages to be written to the stable store the cost of performing these writes can be largely subsumed to reading and synchronous writing i.e. the opportunistic writes are performed when the device heads are suitably positioned due to a read or some other write. When pages are eventually written back to the stable store they are written in place. These mechanisms therefore maintain the original clustering of data within the stable store, a property often deemed important in databases and a property lost by recovery mechanisms such as shadow paging.

6 Conclusions

The characteristics of the DataSafe are that it maintains the original clustering of the data, writes recovery information sequentially and utilises opportunistic write back of melded pages.

An assumption of the DataSafe is that the cache can hold all pages updated by all running actions. While this may be achievable is some systems it is clearly unrealistic in others, particularly those with high degrees of multi-programming. Thus the DataSafe is only guaranteed to work efficiently where the amount of uncommitted data can be bounded, and uses virtual memory otherwise.

By ensuring recoverability and by having no fixed notion of concurrency control the DataSafe offers an alternative to the concurrent shadow paging of the original implementation of Flask. As such the Flask architecture is shown to be flexible in that it may be instantiated with a number of different recovery mechanisms.

Experiments with the MaStA I/O cost model [14] have illustrated that the total I/O costs of recovery mechanisms vary qualitatively between different application workloads. This enforces the need for a flexible architecture such as Flask in which the recovery mechanism may be chosen to best suit the particular application.

A further contribution of the DataSafe is in extending the functionality of the DB Cache to allow more than one concurrent transaction to update a given page. This eliminates possible false conflicts incurred by traditional page locking techniques. Thus the frequency of transaction conflict may be reduced and the throughput of transactions may be higher than that of DB Cache. The meld propagation function used to propagate the changes of one action to all others is dependent on the particular concurrency scheme in operation. In many concurrency models the meld propagation function can be implemented using logical operations increasing the potential efficiency of the mechanism.

The DataSafe has been implemented and provides recovery for the Napier88 [10] system.

7 Hybrid Systems

The MaStA model indicates that different recovery mechanisms exhibit different read/write costs on the disk which radically effect the overall performance of the system. To use the Flask architecture effectively, the transaction workload must be matched to the recovery technique and indeed the flexibility of the system is designed to achieve just that.

One possibility is to use the DataSafe as part of a hybrid system. For example, before-image shadow paging [2, 3] maintains the original clustering of data and writes the recovery data sequentially. Since the data is paged there is no restriction in the amount of uncommitted data. However, analysis of before-image shadow paging, by the MaStA model, indicates a high cost for writing the page maps. But the page maps are effectively bounded and therefore it is possible to think of a system where the before-image shadow paging is used to ensure recoverability of the persistent data and the DataSafe for the recoverability of the page maps. This overcomes the need to hold all uncommitted data pages in the cache and the high cost of writing page maps.

It is a matter for further research to postulate other hybrid systems.

References

[1] Agrawal, R. & DeWitt, D. “Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation”. ACM Transactions on Database Systems, 10,4 (1985) pp 529-564.

[2] Brown, A.L. & Rosenberg, J. “Persistent Object Stores: An Implementation Technique”. In Dearle, Shaw, Zdonik (eds.), Implementing Persistent Object Bases, Principles and Practice, Morgan Kaufmann, 1991 pp 199-212.

[3] Brown, A.L. “Persistent Object Stores”. Ph.D. Thesis, University of St Andrews (1989).

[4] Davies, C.T. “Recovery Semantics for a DB/DC System”. In Proc. ACM Annual Conference (1973) pp 136-141.

[5] Davies, C.T. “Data Processing Spheres of Control”. IBM Systems Journal, 17, 2 (1978) pp 179-198.

[6] Elhardt, K. & Bayer, R. "A Database Cache for High Performance and Fast Restart in Database Systems". ACM Transactions on Database Systems, Vol. 9, No. 4, December 1984, pp 503-525.

[7] Eswaran, K.P., Gray, J.N., Lorie, R.A. & Traiger, I.L. “The Notions of Consistency and Predicate Locks in a Database System”. CACM 19,11 (1976) pp 624-633.

[8] Haerder T. & Reuter, A. "Principles of Transaction-Oriented Database Recovery". Computing Surveys, Vol. 15, No. 4, Dec. 1983, Pages 287-317.

[9] Lorie, R.A., "Physical Integrity in a Large Segmented Database". ACM Transactions on Database Systems, Vol. 2, No. 1, March 1977, pp 91-104.

[10] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “The Napier88 Reference

Manual”. University of St Andrews Technical Report PPRR-77-89 (1989).

[11] Munro, D.S., Connor, R.C.H., Morrison, R., Scheuerl, S. & Stemple, D.W. "Flask - A Flexible Layered Architecture for Supporting Concurrency Control Schemes". 6th International Workshop on Persistent Object Systems, Tarascon, France (September 1994). In Persistent Object Systems (Eds. M.P.Atkinson, V.Benzaken & D.Maier). Springer-Verlag, pp 16-42.

[12] Munro, D.S., Connor, R.C.H., Morrison, R., Moss, J.E.B & Scheuerl, S.J.G. "Validating the MaStA I/O Cost Model for Database Crash Recovery Mechanisms". In the Proceedings of the OOPSLA'95 Workshop on Object Database Behaviour, Benchmarks and Performance, Austin Texas (October 1995).

[13] Moss, J.E.B. “Nested Transactions: An Approach to Reliable Distributed Computing”. University of MIT (1981).

[14] Scheuerl, S.J.G., Connor, R.C.H., Morrison, R., Moss, J.E.B. & Munro, D.S. "The MaStA I/O Cost Model and its Validation Strategy". In the Proceedings of the Second International Workshop on Advances in Databases and Information Systems (ADBIS'95), Moscow, June 27-30 1995, Volume 1, pp 165-175.

[15] Stemple, D. & Morrison, R. "Specifying Flexible Concurrency Control Schemes: An abstract Operational Approach". Australian Computer Science Conference 15, Tasmania (1992) pp 873-891.