Concurrent Shadow Paging in the Flask Architecture

D.S. Munro[1], R.C.H. Connor[1], R. Morrison[1], S. Scheuerl[1] & D.W. Stemple[2]

[1]Department of Mathematical and Computational Sciences, University of St Andrews,
North Haugh, St Andrews, Fife, KY16 9SS, Scotland

[2]Department of Computer Science, University of Massachusetts,
Amherst, Massachusetts, MA 01003, U.S.A.

Abstract

The differing requirements for concurrency models in programming languages and databases are widely diverse and often seemingly incompatible. The rigid provision of a particular concurrency control scheme in a persistent object system limits its usefulness to a particular class of application, in contrast to the generality intended by the provision of persistence. One solution is to provide a flexible system in which concurrency control schemes may be specified according to the particular task in hand, allowing the same data to be used in conjunction with different concurrency control schemes according to the needs of the application.

A major difficulty in the engineering of such a system lies in the building of generic mechanisms to provide the facilities of data visibility restriction, stability, and atomicity, *independently* of the combination of these employed by a particular concurrency control scheme. Flask is a architecture which is designed to achieve this goal by defining a layering whereby the memory management is not tied to any one concurrency control scheme operating above. This paper outlines the Flask architecture and focuses on an implementation based on concurrent shadow paging. The architecture described is currently being used as the basis for experimentation in generic concurrency specification.

1 Introduction

The notion of atomic transactions [Dav73, Dav78, EGL+76] can be identified as one of the first forms of concurrency control in databases. Transactions provide users with the so-called ACID properties, namely atomicity, consistency, isolation and durability, with which to understand computations. For concurrent access, transactions isolate the effects of one activity from another using a serializable schedule. The transaction concept was extended to enable nesting [Ree78, Mos81] whilst Sagas [GS87] provided a non-serializable concurrency control model as a potential solution to certain classes of problems caused by long-lived transactions. More recently for many applications, especially those in design and interactive systems [Sut91, EG90, NZ92], it has been suggested that the serializability constraint of the atomic transaction model is too restrictive. Such systems need the global cohesiveness of the transaction model but also require the competing transactions to interact with each other in a structured way because of their inter-dependence.

With an increase in the number and diversity of such models, there is a growing need within database and persistent programming systems to support different styles of concurrency control scheme operating in conjunction over the same data. For example, an airline company may employ an atomic transaction system for flight-bookings whereas a tour operator or travel agent, operating on the same data, may utilise a saga-style scheme when booking connecting flights through different carriers.

Flask is a layered architecture which has the flexibility to support different models of concurrency over the same data. The architecture eschews any fixed notion of concurrency control. Instead it provides a framework in which models can be defined and supported. One of the major difficulties in engineering such a system lies in the building of generic mechanisms to provide the facilities of data visibility restriction, stability, and atomicity, independently of the combination of these employed by a particular concurrency control scheme.

One of the key features of the Flask architecture is that as little as possible is built-in to the lower layers providing increased functionality moving up the levels of abstraction. This provides the necessary flexibility but may also permit efficiency gains since many optimisations are achievable at a higher-level. The approach is comparable with that taken in RISC architecture where a simpler, more understandable hardware interface can be exploited more readily by software construction to forge efficiency gains. Promoting some of the complexity to the higher layers simplifies the memory management allowing a number of possible implementations.

The Flask architecture challenges the orthodoxy whereby most database systems, such as System/R [CAB+81], Ingres [Sto86], O2 [Deu90] and Oracle build in specific models of concurrency, usually at the store level. This design decision is taken for reasons of simplicity of implementation and arguably trades efficiency for flexibility. A drawback with the approach is that the specification of a particular concurrency control model is often described in terms of its implementation. For example, the conceptual simplicity of a transaction model can often become confused with particular locking tactics. Such situations can add to complexity of understanding the semantics of a concurrency control scheme.

This paper presents an overview of Flask but concentrates on the low-level memory management implementation that is structured to be independent of the concurrency control schemes operating above. One such implementation, based on shadow-paging, is described in detail.

2 Overview of Flask

The framework of the Flask architecture is shown in Figure 1 as a “V-shaped” layered architecture to signify the minimal functionality built-in at the lower layers. At the top layer the specifications of the model are independent of the algorithms used to enforce them and can take advantage of the semantics of these algorithms to exploit potential concurrency [Gar83]. For example a particular specification may translate into an optimistic algorithm or alternatively a pessimistic one while the information they operate over remains the same. More importantly such an approach can accommodate different models of concurrency control.

The Flask architecture is designed to work with processes or actions that maintain global cohesion under control of the concurrency control schemes. In general, changes to data do not overlap except where this happens co-operatively. In Flask, the significant events defined by a particular concurrency control scheme are generated from and reported to the higher layers enabling these schemes to undertake conflict detection. This assumption frees the lower layers from the onus of interference management.

Two systems which meet these requirements and which are major influences on the Flask approach are Stemple and Morrison’s CACS system [SM92] and Krablin’s CPS-algol system [Kra87]. The CACS system provides a framework in which concurrency control schemes can be specified. CACS is a generic utility for providing concurrency control for applications. The system does not actually manipulate any objects, but instead maintains information about their pattern of usage. In order for the system to collate this information, the attached applications must issue signals regarding their intended use of their objects. In return the CACS system replies indicating whether or not the operation would violate the concurrency rules. CPS-algol is an extension to the standard PS-algol system [PS87] that includes language constructs to support and manage concurrent processes. The concurrency model is essentially co-operative with procedures executing as separate threads and synchronising through conditional critical regions. Krablin showed that with these primitives and the higher-order functions of PS-algol, a range of concurrency abstractions could be constructed including atomic and nested transactions as well as more co-operative models.

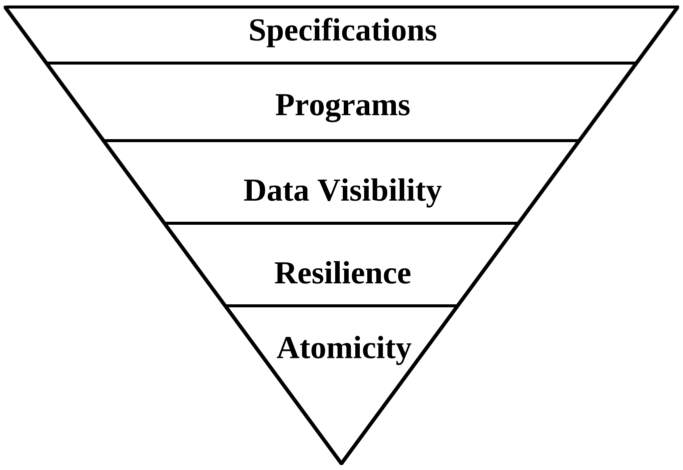

Figure 1: V-shaped layered architecture

This paper is not concerned with how the specifications are written or how the concurrency control algorithms initiate and receive events. Instead the main theme is concerned with low-level memory management implementations in the Flask architecture under the assumptions given above. The focus of interest then centres on the visibility of data from different actions. In Flask, the visibility is expressed in terms of the control of movement between a globally visible database and conceptual stores called access sets. Each action is associated with a local access set and may use other shared access sets. The interfaces to the data visibility are :-

- Start action
- End action
- Create and delete local access set
- Create and delete shared access set
- Copy from access set to access set
- remove object from access set
- Meld

At the lowest level the atomicity layer ensures consistent update to the access sets and the global database. The failure resilience layer utilises this atomicity to effect an action *meld*. The term *meld* is used to describe the action of making updates permanent rather than terms like *commit* or *stabilise* since they imply specific meanings in particular models.

One approach, presented here, marries the notion of data visibility as expressed by access sets to a concurrent shadow paging mechanism. Shadow paging [Lor77] has been seen by some [GMB+81, Kol87] as an inappropriate technology for concurrent systems and most database systems use logging techniques instead [OLS85, VDD+91]. It is argued that in moving the programming algebra and the concurrency control to higher layers many of the perceived difficulties concerning concurrent shadow paging have been abstracted out. Shadow paging is then seen as a mechanism worthy of consideration for such an architecture in essence because it can stabilise over arbitrary address ranges without requiring knowledge of objects and program behaviour.

3 Conceptual Concurrent Layered Architecture

The crux of the design of an architecture that can support different styles of concurrency is to define the global cohesion or understandability in terms of data visibility between concurrent activities. This is reflected in the design of a conceptual concurrent layered architecture in which visibility is defined and controlled by the movement of data between a hierarchy of conceptual address spaces. The architecture provides isolation by constraining data accessed by one activity to a separate address space from all others. Concurrency models using this architecture are defined in terms of data visibility through the sharing of address spaces and the control of movement between these spaces.

The conceptual concurrent architecture is layered in such a way that it separates the concurrency control mechanism, the atomicity and the persistence. These layers are described in terms of conceptual address spaces together with a protocol that controls the movement of data between these address spaces. By unbundling and distributing the concurrency intrinsics in this hierarchy, the architecture provides a generic layered store capable of supporting a number of different models of concurrency.

Figure 2 gives a diagram of the Flask architectural hierarchy shown as a layer of conceptual address spaces. Each layer implements a separate and independent property of concurrency. This permits any particular concurrency model the flexibility to choose a desired combination of intrinsics.

At the top level the concurrency control schemes contains a full definition of the concurrency control mechanism. No assumptions are made by the lower layers about the concurrency control and hence this leaves the implementor freedom to choose any desired scheme.

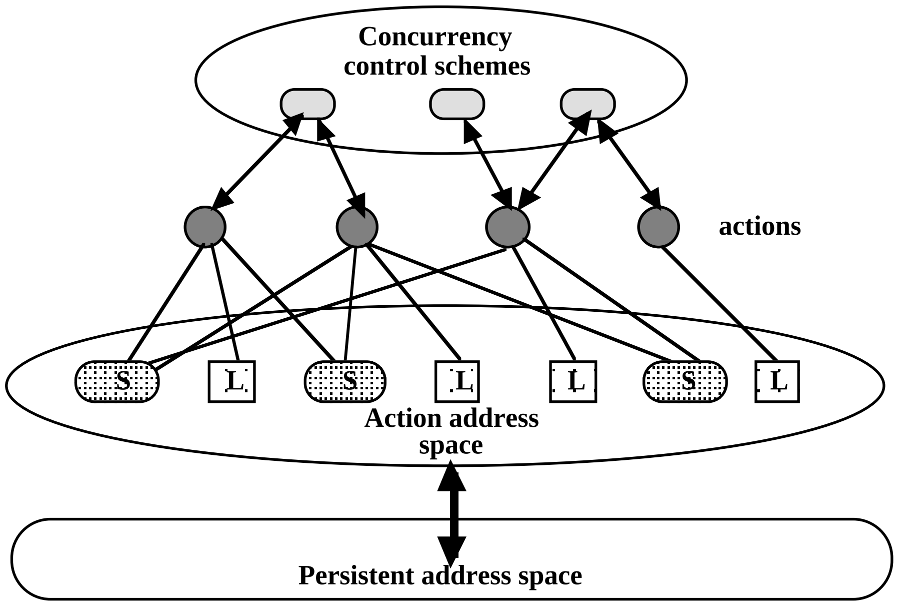

Figure 2: Flask Conceptual Concurrent Architecture

An action, in the Flask architecture, is an isolated thread of control that communicates with the concurrency control schemes and whose state is defined by the action address space. The action address space layer is a set of private and group address spaces. Each action has a private address space (marked “L” for local in Figure 2). In addition the action address space may house a number of group address spaces that are shared by a combination of actions (marked “S” for shared in Figure 2). The globally visible database is modelled by the persistent address space.

Schemes are defined in terms of action creation and abort along with the movement of data between address spaces. The different schemes control the visibility and interference among actions using them. The architecture ensures that all data movement is atomic. Movement of data from an action’s private address space or a group of actions’ group address space to the persistent address space is through a meld operation. The Flask architecture supports the atomic update of the persistent address space so that its data is permanent, recoverable and consistent. The meld operation is instantiated with different semantics dependent on the particular concurrency control scheme to ensure that data movement to the persistent address space becomes visible to all other actions. The effect of an action abort is to release the action’s private address space. The relationship between action abort and group address space is determined by the concurrency control scheme.

The Flask architecture provides the versatility to support a range of concurrency models. For example:-

- Support for atomic transactions can be provided in this architecture through a concurrency control specification that constrains each action’s updates to their private address space thereby isolating their effects. Transaction commit involves the atomic update of the persistent address space making the transaction’s changes permanent and globally visible. Transaction abort is a trivial matter of discarding the private address space.

- Co-operative concurrency is viewed in this architecture as a group of actions which do not require isolation control and interact co-operatively over a group address space.

- Designer transactions model can be accommodated by a combination of private and group address spaces and a concurrency control specification that defines their creation, interaction and the movement of objects. Thus the effects of operations may be shared among actions without their objects being committed to the persistent address space.

4 Instantiating the Architecture

The discussion here of how an instance of the Flask architecture is achievable focuses on the provision of data visibility, resilience and atomicity. It is assumed that the concurrency control specifications and algorithms are handled by a mechanism such as CACS. One possible implementation strategy arises from the observation that each layer need only maintain copies of the portions of the underlying layer’s address space that it has changed along with a table which provides a mapping between the original and the copy. Applying this strategy down the layers, the resulting architecture collapses into one flat address space with a hierarchy of mappings. One approach to providing such an address space is recoverable paged virtual memory.

There are a number of different mechanisms, such as write-ahead logging or shadow paging, which can provide the necessary resilience to support stable virtual memory. Many, if not most, databases and persistent systems, such as O2 [VDD+91], System/R [GMB+81] and Argus [Kol87], tend to favour a form of logging to provide failure recovery after crashes. The arguments against a shadow paging solution centre around efficiency issues and the mechanism's ability to handle concurrent access. In the instantiation of the Flask architecture described here an extended form of shadow paging has been developed. It is argued here that this concurrent shadow paging mechanism, under the assumptions described above, is not only a viable mechanism but also a potentially efficient solution.

4.1 Shadow Paging

A virtual memory system controls the movement of pages of the virtual memory on backing store to and from the page frames on main store and maintains a page table, the main memory page table, which records the allocation of page frames of main memory to pages of the virtual memory. The page table is transient since soft failure results in the loss of the main store. The operating system typically allocates the pages of the virtual memory to a contiguous range of disk blocks on backing store and records the disk address of the first disk block, the base address. Virtual memory addresses are usually relative offsets from the first page and hence the *i*th page can be found at the *i*th disk block from the base because the pages and disk blocks are always in one-to-one correspondence.

A shadow-paged virtual memory system is similar to virtual memory where the pages reside on backing store and a main-memory page table records the allocation of pages to physical memory page frames. The essential difference is that the system ensures that before a modified page is written back to non-volatile store that there is always a retrievable copy of the original page on non-volatile storage. This scheme then destroys the contiguous correspondence between the pages of the virtual memory and disk blocks and so shadow paged virtual memory requires another table, the *disk* page table, to record the mapping between pages of the virtual memory and disk blocks on backing store. There are two varieties of shadow paging :-

4.1.1 After-look Shadow Paging

With an after-look shadow paged scheme the mechanism makes sure that a modified page is never written back to non-volatile store to the same place it was read from. When a modified page is written back to non-volatile store an unused disk block is found and the disk page table updated to reflect the new mapping. This is analogous to deferred-write logging [Dav73, GMB+81]. Figure 3 illustrates the after-look scheme showing the modified pages being shadowed to a different disk page.

The system uses a root block which resides at a known disk address. The root block is stable by mirroring and from this the disk page table can be located. At system startup the disk page table is interrogated to re-establish the state of the address space. In fact two versions of the disk page table are maintained; the version held on disk which reflects the stable state and another in main memory which is updated by the shadowing mechanism. This *transient* disk page table reflects the current state of the address space. Figure 4 illustrates the mechanism. The stable disk page table records the mappings from the last consistent state whilst the transient disk page table in volatile store records the current mappings. The diagram shows that the third and fourth page have been shadowed to unused disk blocks. When the user melds all the modified pages are flushed to disk and then the in-memory version of the disk page table atomically replaces the disk version. The atomic update depends on the root block being written correctly. This can be performed using any mirroring technique such as Challis' algorithm [Cha78].

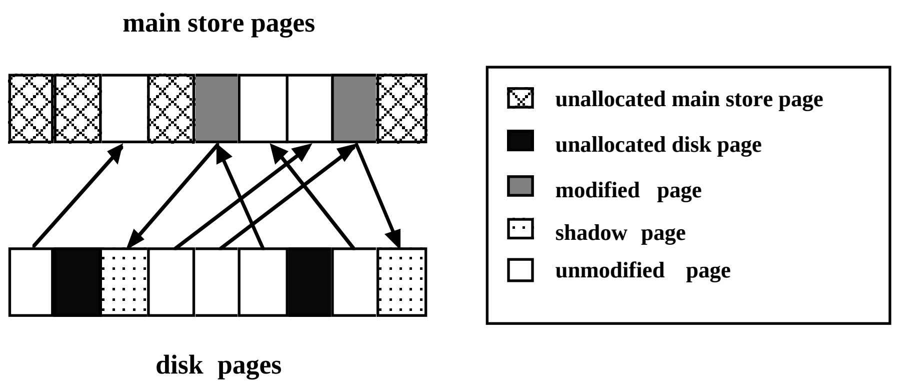

Figure 3: After-look shadow paging

Challis' algorithm uses two fixed root blocks with known disk addresses. The root blocks contain information that allows the system to find the mapping table for a consistent state. From the root blocks then the two previous consistent states n-1 and n can be found. Each root block also contains a version number that enables the system to determine which contains the most recent state. This version number is written twice as the first and last word of the block. The atomic update operation entails overwriting the root block with oldest version number, in this case n-1, with a new version number, n+1, and a pointer to the new updated mapping table. The space occupied by the old consistent state n-1 may now be reused.

Challis' algorithm depends upon two critical points for safety. Firstly an error in an atomic update can only occur if the root block is written incorrectly. It is expected that if a disk write operation fails during the atomic update it will inform the system which can then take appropriate action immediately. If, however, the failure is more serious, the technique depends upon the version numbers at the start and end of the root block being different in order to detect failure.

On system startup the root blocks are inspected. If the version numbers are consistent within the root blocks, the most up-to-date version of the system can be found. If not, only one root block may have different version numbers at any one time unless a catastrophic failure has occurred, which in any case would have other implications for the integrity of the data. Thus, subject to the above proviso, the correct stable data can be identified.

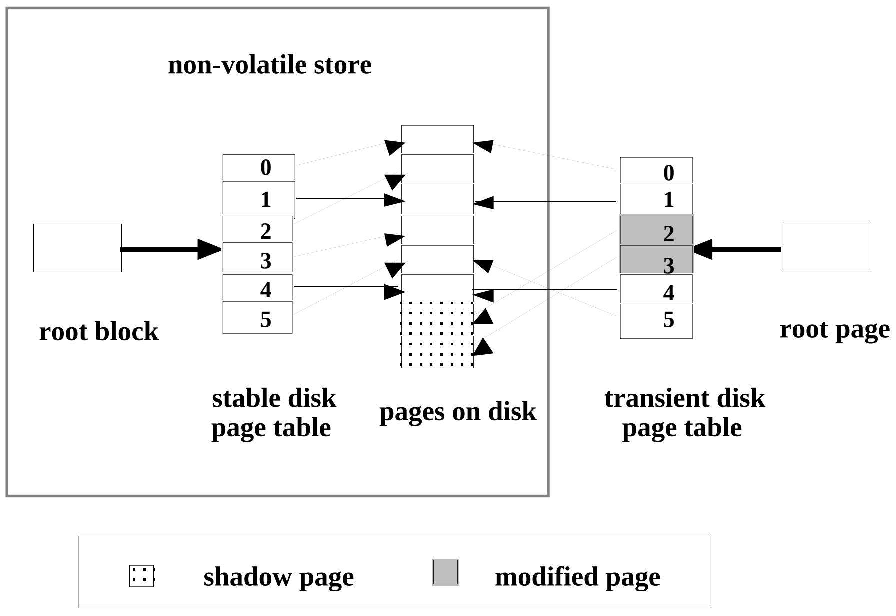

Figure 4: Layout of after-look shadow paging

For efficiency the transient disk page table is often kept in main store. The disk page table may, however, be too large and would in that case be stored in a transient area of disk. This causes difficulties in the disk mapping as there is now a transient disk area which has to be mapped in some manner and the disk area mapped by the disk page table. A solution to the above is to incorporate the disk page table in the virtual address space itself. This means that the page table is paged and that updates to it create shadow copies. The advantage of this is that there is only one paging mechanism and that changes to all data, including the page table, are incremental. The only requirement for such an organisation to work is that the address of the first page of the disk page table must be held at a fixed location, the root block, within the virtual address space.

On crash recovery the root block is read and the disk page table is recovered. This is used to re-establish the contents of the pages from their associated disk blocks. It will include the program state if it is considered part of the data. If this is the case then once the recovery manager has reconstructed the data the computation will automatically carry on from the last meld point. No changes to data in this mechanism get undone or rewritten and hence the after-look shadow paging recovery algorithm can be classified as a **no undo / no redo** algorithm.

4.1.2 Before-look Shadow Paging

With before-look shadow paging [BR91], the first modification to a page causes a copy to be written to a new block on non-volatile store, i.e., its shadow page. In contrast to after-look shadow paging modifications then take place in the original. The disk page table is used to record the location of the shadow pages and must itself be on non-volatile store before any updates reach non-volatile store. This is similar to logging with immediate writes. The before-look scheme is illustrated in Figure 5 where a modified page is written back in place after a shadow copy of the original has been taken.

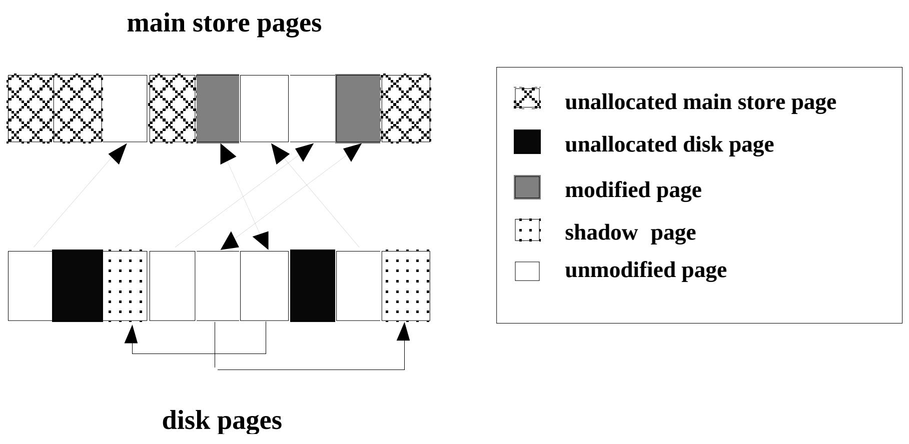

Figure 5: Before-look shadow paging

These shadow pages must be locatable after a crash and effectively form a coarse-grain log of the previous values of the modified pages. On meld the modified pages are written to disk and a new consistent state is established. This log of previous values is then discarded.

Recovery from a system crash occurring before a meld involves using this “log” to overwrite the pages that have been modified since the last meld with their previous values. The system is thereby established to the same state as it was at the last meld. This undo of the pages is clearly idempotent. If the program state is considered part of the data then once the recovery manager has completed the undo operation the computation will automatically proceed from the last meld point. Before-look shadow paging can be classified as being **undo / no redo**.

5 The Flask Concurrent Shadow Paged Store

The main problem of concurrency and shadow paging is that transactions may make conflicting requests to modify the same page. When one of the transactions melds the modified page is written to its shadow page on non-volatile store. This of course will include the changes made by any unmelded transaction that modified objects on the same page.

One common solution is to use page-level locking whereby an atomic transaction obtains an exclusive lock on a page before shadowing the page [AD85b, Lor77]. Each transaction maintains its own page table of shadows and the locking guarantees that a page is never in more than one page table. The main drawbacks of this solution are firstly that it introduces phantom locking where two atomic actions are prevented from modifying different parts of the same page. Secondly it employs a built-in concurrency control mechanism at a low level. Problems of deadlock also have to be addressed.

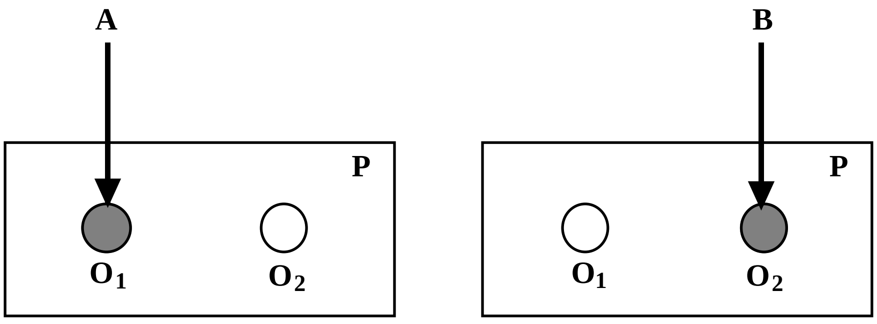

Figure 6: Flask shadow paging

The Flask solution involves not only keeping a disk page table of shadows for each process but to also maintain per-process shadow pages, i.e., an access set. Flask also maintains a global disk page table of pages which constitute the last meld point. When a process first modifies a page a shadow copy is made and its address is added to the process's shadow page table. To resolve a disk address on a page fault the process shadow page table is searched first and, if still unresolved, then the global disk page table. This mechanism ensures that the changes made by one process are completely isolated from other processes and the global state. Action abort is thus a trivial matter of discarding a process's shadow pages. Figure 6 illustrates the Flask shadow paging showing that both action A and action B can freely modify objects

O_1 and O_2 respectively from the same page P since each action has a shadow copy of page P.

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

5.1 Concurrency control and per-action melding

In the Flask architecture the access sets are implemented by paged address spaces. Because the meld resolution is at a page-level then the changes made by the melding action must be propagated to other actions' per-process shadows. Suppose that two actions A and B share a page but modify different objects on that page. Because of the isolation of the concurrent shadow paging mechanism A can meld without affecting B. For B to subsequently meld it must retain the changes that A made. And so a mechanism is required for B to ingest the changes made by A. The algorithm that meld uses to propagate changes is dependent on the particular concurrency model in operation. Under the assumption that the higher-layer concurrency control can detect object-level conflicts there 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 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 :-

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

where P_A is the action A's shadow page P, P_O is the page P as it was at the last meld and P_B is action B's shadow page P. Thus B's version of page P now includes the changes made by A. This approach is not restricted to atomic transactions. In co-operative models where the actions agree to change an object to the same value using majority rules logical operations can also be used.

One other possible method of providing meld propagation is for an action to record the address ranges that it has modified for each page in its shadow page table as the changes are made. This could then be used to copy the modifications to other transactions holding a copy of the same page.

Whilst the cost of propagation may seem high it should be borne in mind that this is only necessary when an action melds. In the case of long-lived actions such a cost may be less burdensome. Furthermore, a meld propagation only affects actions that have accessed shared pages. On stock hardware logical operations are typically very fast machine instructions making this shadow page scheme not only viable but also potentially efficient.

5.2 The Flask Architecture

The concurrent shadow paging scheme works much as the single-user shadow paging described above whereby the virtual address space is mapped to non-volatile storage through a disk page table. Modified pages are shadowed and the transient disk page table reflects the current global state of the address space. In addition a separate disk page table is created for each private and group action address space used. Each action has its own private address space and so a disk-page table is created for each action. Similarly a disk page table is created for each group address space required by the model. Entries to the disk page tables are added for each page modified by the action. When an action first modifies a page a shadow copy is made and the action works on the copy. The concurrency control specification dictates whether the update is a private or group one. Hence the changes made by an action to its private address space are totally isolated from other actions' private address spaces. Also the group address spaces are isolated from each other and from the private address spaces.

The mechanism is illustrated in Figure 7 and shows that the transient disk page table and the action disk page tables are accessible from the root page. As in the single threaded case the transient disk page table and the stable disk page table maintain a page table entry for each page of the address space. When a page, modified by an action, is written out to non-volatile store it is written to its shadow and the mapping recorded in the per-action page table. The action page table only has entries for pages modified by that action and not the complete address space. The illustration in Figure 7 shows that there are five pages in the virtual address space and that there are currently two actions A and B. The disk page table for action A shows that A has modified pages 0 and 2 in its private address space and that action B has modified pages 0, 1 and 3 in its private address space. Note that page 0 has been modified by both actions but that the shadow page mechanism isolates their modifications. The third disk page table reflects a group address space that shows action A and B are working on a shared copy of page 2 and 4. A has a local object in page 2 and a shared object with B. The particular concurrency control scheme will disambiguate action A's access to page 2 since it knows which objects are shared and which are not.

The scheduler for this concurrent system must ensure that the correct mappings are established on a page fault. For example when action A accesses a page that results in a page fault, the system must search A's disk page table for the page (or a group that A is currently in). If there is no entry for the page in A’s disk page table then the transient disk page table is searched.

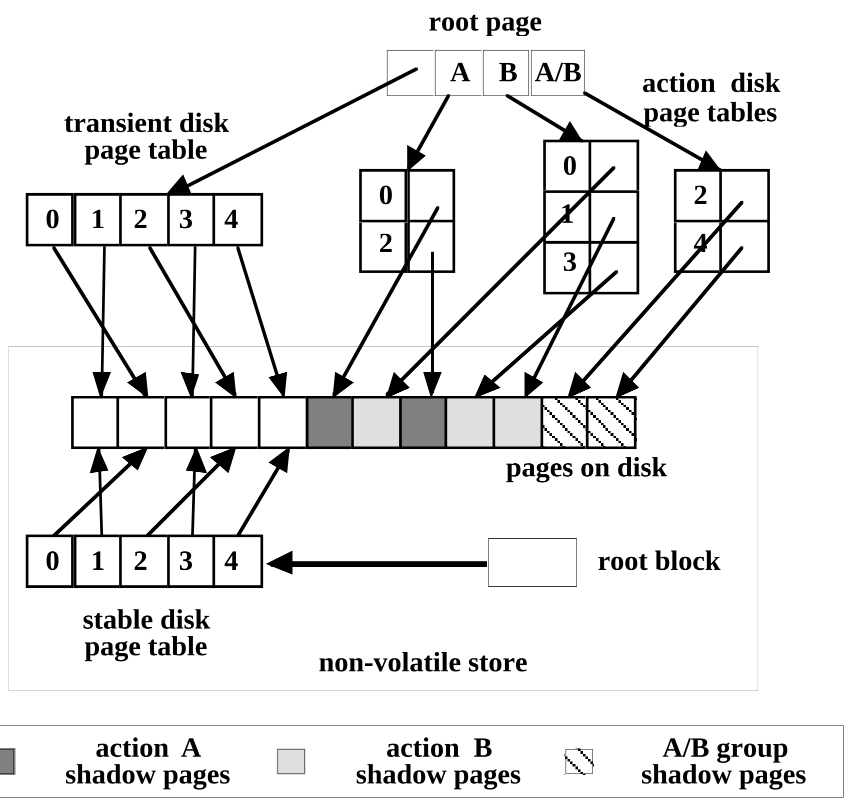

Figure 7: Concurrent shadow-paged architecture

A meld mechanism is provided on a per-action basis so that changes made by an action can become part of a new recoverable, consistent state and then these changes are made visible to other actions. It is possible for a number of actions to be working on shadow copies of the same page and the meld propagates the differences between the shadowed pages modified by the action and the originals through to any other action working on copies of the same original pages.

To establish a new consistent state all pages modified by the action are written to their shadows. Then the entries for these pages in the transient disk page table are updated to record the same mappings. For example, if a melding action had modified page P and its shadow page was disk block D then the transient disk page table entry for P must also record that it is mapped to D. To ensure atomicity of the meld the updating of the transient disk page table will involve shadowing of the page encompassing the transient disk page table entry. Once the transient disk page table reflects the new consistent state it atomically replaces the stable disk page table.

6 Evaluation

Two criticisms commonly associated with after-look shadow paging are its effects on the clustering of data and the cost of disk page table lookups. Two logically adjacent pages in the conceptual store may be allocated physically distributed shadow pages causing increased seek time. The effect of this can be reduced by using physical clustering techniques such as suggested by Lorie [Lor77] where shadow pages are allocated within a cylinder where possible. With after-looks, shadow pages need only be allocated disk blocks when the user melds or when main memory is exhausted and pages must be written back. Hence this provides an opportunity to use such a clustering scheme.

The cost of disk page table access can be reduced by maintaining the table in main store but where the disk page tables are themselves paged then extra disk hits are unavoidable. An evaluation of the worst case scenario of the Flask concurrent shadow paging mechanism is given below.

When a read request on page P by action A causes a page fault then the mechanism first searches action A’s disk page table since action A may have already shadowed the page. This may result in a disk access to bring the table into memory. If page P is not in action A’s disk page table then the transient disk page table is searched. This too may involve a disk hit to access the table entry. Finally there is a disk access for the original page fault for page P. In the worst case then a read request could result in three disk hits.

A write request on page P may also, in the worst case, require three accesses for the same reasons as a read request. In addition a first write to a page will require a shadow copy of the page and an update to the actions disk page table. However taking a shadow copy of P does not necessarily incur a disk write since it need only involve allocating a free disk block for the shadow.

On a meld, the propagation algorithm is dependent on the particular concurrency model being executed. However regardless of which scheme is in operation a meld requires that a new stable state is established before propagating the changes. This involves flushing each page of an action’s disk page table to disk. Then the stable disk page table must be atomically updated to reflect the new global state.

As an example of the cost of meld propagation suppose that an atomic transaction model was implemented using the double xor meld propagation formula given above. Each page P that has been modified by the melding action A must propagate its changes to all other actions that have also been shadowed page P. For each such page P then P is xor'd onto the original O and this is then xor'd onto each of the actions that have shadow copies of P. Note that these propagations do not need to be written back to disk.

In the worst cases then this mechanism appears to require a heavy disk access cost. However, this cost is greatly reduced where the actions exhibit a high degree of locality of reference. In addition efficiency gains can be sought in implementations by using features such as memory mapping or tailoring an external pager. As an example, an instance of the Flask architecture has been implemented to support an atomic transaction package written in Napier88. This makes use of the SunOS memory-mapping facilities to enhanced performance since it uses the operating system's page-replacement mechanism and utilises the Sun memory management hardware to perform the address translations [Mun93].

7 Related Work

Attempts have been made at analysing and comparing the cost of different recovery schemes [AD85a, KGC85]. The results of these efforts do not produce a clear winner. Other research [GMB+81, Kol87, AD85b] would suggest that logging is a better technique especially when the system needs to support conflicting actions. Implementations of shadow paging are not widespread and it is believed by some to be an inappropriate technology for database applications. The implementors of System R used a complex combination of shadow paging and logging and claim that in hindsight they would have adopted a purely log-based recovery scheme. Furthermore they stated they were unable to conceive of an appropriate architecture based purely on shadows that could support transactions.

Agrawal and DeWitt produced a complex cost model used for comparing shadow paging with logging using a variety of concurrency control techniques. Their approach was purely analytical and their database simulations did not account for the costs of buffer management. The results for shadow paging in these simulations were poor when compared with logging. However closer inspection of their model reveals an unexplained assumption. In the logging case it is assumed that the size of the records that are written to the log for each page modified by a transaction is 10% of the page size. So if during a transaction's execution data is modified on 10 pages the assumption is that the size of the log records for that transaction amount to 1 page. This assumption may be valid in some models of computation. However if the transactions are generated from language systems that frequently update large objects, such as graphical objects, or higher order functions the assumption may not be sustainable.

In contrast the Predator project [KGC85] took an empirical approach to comparing the two methods. A realistic transaction-based database was constructed and logging and shadow paging recovery mechanisms implemented on stock hardware. A variety of transaction experiments were carried out using both recovery techniques and the results compared. The performance metrics were based on transaction throughput and mean response time. Their first observation is that there is no one best mechanism and that the choice of recovery method is application dependent. They concluded that shadow paging works best when there is locality of reference and where the page table cache is large. By using meld batching, shadow paging outperformed logging as the number of simultaneous transactions increased. Another interesting observation they made was that the shadow paging imposes a more evenly balanced I/O load than logging. Under load a noticeable performance drop was observed in the logging scheme as the system is divided between requests for sequential disk writes for the log and page reads and writes for the database.

Most of the objections to shadow paging performance are founded on a belief that the cost of writing a journal of updates to a log will almost always be more efficient than the maintenance of shadow pages. This may be true for a class of problems but may not be true in general. Many of the measurements that this notion was founded on were based on simulations or were taken from tests run on machine architectures and configurations that are now obsolete. It may be fair to suggest that the results of the comparisons related to the limitations of technology and systems available at the time. For example the overhead of page-table lookups in shadow paging was considered very costly. However the size and speed of memory in an average workstation have risen dramatically over the last few years so that the page table even for a large store could reside in main memory.

Computational models too have changed, not least with the rise in popularity of database programming languages and persistent systems. These systems make different demands on a database or stable store with different patterns of use from conventional database accesses. For example programs and data are not treated differently in orthogonal persistent systems. It is relatively straightforward to efficiently record program state in a shadow paged system by including the process state and stack frames in the address space that is shadowed since shadow paging requires no knowledge of objects or process behaviour. In contrast logging program state change would probably require a special case to handle stack frames. These arguments suggest that the decision on a superior model of recovery is not so clear cut. It may be that shadow paging is a better alternative. Certainly it is clear that shadow paging implementations can get significant performance improvements from an operating system that provides an external pager or memory-mapping support.

This support seems more forthcoming in stock systems than explicit support for fast logging [GMS87, ABB+86].

It has been argued [AD85b] that on a small scale, locality of reference would seem to favour a log-based solution since the amount of information that requires to be written to the log is small compared with the overhead of copying a whole page. Furthermore with logging there is no requirement to write back modified pages after a meld and hence a frequently modified page can reside in main store through a number of transactions. Kent’s [KGC85] experimental observations suggest the exact opposite. As locality increases the page table overhead in shadow paging is soon amortised. With logging the amount of modified data that must be saved increases. There quickly comes a point where a lot of locality, especially within one page, along with frequent updates to objects on the page tips the balance in favour of a solution that makes a one-off copy of a page rather than maintains a journal of changes. Furthermore if the objects themselves are fairly big then frequent modifications to them will have an adverse effect on the log size but not on a shadow page. This kind of locality is exactly the type of behaviour that might be exhibited in persistent systems with higher-order functions.

8 Conclusions

The recent growth in the number and variety of concurrency control schemes in database and persistent systems has led to an increased need for such systems to support different schemes over the same data. The Flask architecture, described in this paper, presents one approach to providing a platform in which different concurrency control schemes can be supported.

The main feature of the Flask architecture is that as little as possible is built-in to the lower layers enabling the specification of a concurrency model to be separated from its implementation. This is very different from convention where traditional systems such as O2, Ingres and System/R use fixed models incorporated at the store level. Moving the concurrency control specifications and the programs that implement them to the higher layers relieves the constraints in handling data visibility, atomicity and resilience.

This approach has led to a re-examination of shadow paging as a viable mechanism to satisfy the requirements of the lower-levels of the Flask architecture. The advantage of shadow paging in this architecture over the more widespread logging mechanisms is that it can effect a stable state without requiring knowledge of the structure of the objects or program behaviour.

The shadow paging mechanism does not readily extend to accommodate concurrent operation. The problem of two actions wishing to modify parts of the same page has traditionally been addressed using page-level locking. This approach, however, introduces phantom locking. The mechanism presented here circumvents most of the common objections to concurrent shadow paging by creating shadow pages on a per-action basis together with a method of propagating the changes made by a melding action to other actions. The actual meld propagation function 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.

Future work in this research includes using the Flask architecture to develop a suitable platform for measuring and comparing concurrent shadow paging with a log-based solution.

9 Initial Performance Evaluation

An initial evaluation of the performance of the FLASK store was carried out to determine the effects of different transaction workloads. A test suite of program simulations were written directly on top of the object store interface. Simulations at this low level enabled the experiments to have accurate control of: accesses and updates to the store; the size and number of objects being manipulated and the management of transaction creation, commitment, abortion and the interleaving of transactions.

The experiments were carried out on a Sun SPARCStation ELC with 40 MB main memory, 500MB SCSI disk and running SunOS 4.1.3. All blocks and pages are 8192 bytes. To suppress operating system cache interference the experiments were carried out on a cold single-user system. This involved flushing the memory of store pages and flushing the IO buffers before every experiment.

Experiment timings were obtained using the *getrusage()* system library function and the UNIX *time* command both of which gave the real time of process execution. The results include the execution time of relevant system processes on the machine, namely the swapper and pagedaemon.

The following experiments were carried out:

Experiment 1 - In the FLASK store, the meld propagation function for concurrent transactions uses the page-level double xor function to propagate changes to other transactions which use the same page. This experiment calculates the time to xor one memory resident page onto another memory resident page. The experiment chooses two pages and initially accesses both pages to make sure that they are resident in main memory in order to remove IO costs. An xor of one page onto another involves performing an xor on every pair of words in the pages. The experiment records the time to execute the page xor function 10000 times. **Result**: A single page xor = 2.3 milliseconds.

Experiment 2 - The FLASK implementation uses the memory mapping facilities of SunOS to control the mappings from pages to shadow disk blocks and is used by the FLASK store to re-map pages onto shadow disk blocks. This experiment gives the execution time of a SunOS *mmap()* system call. The experiment performs 20000 calls to *mmap()* using the same page and alternates between two different disk blocks to ensure the system updates its page mappings on every call. **Result**: A single *mmap()* call = 325 microseconds.

Experiment 3 - This experiment illustrates the effects of temporal and spatial locality of reference on the store. The experiment executes a transaction which performs 200 updates. On every execution, a different range of pages is updated with the range varying from 16 to 2048 pages. With a small range, the same pages are updated multiple times. With a large range (≥ 200 pages) 200 pages are updated.

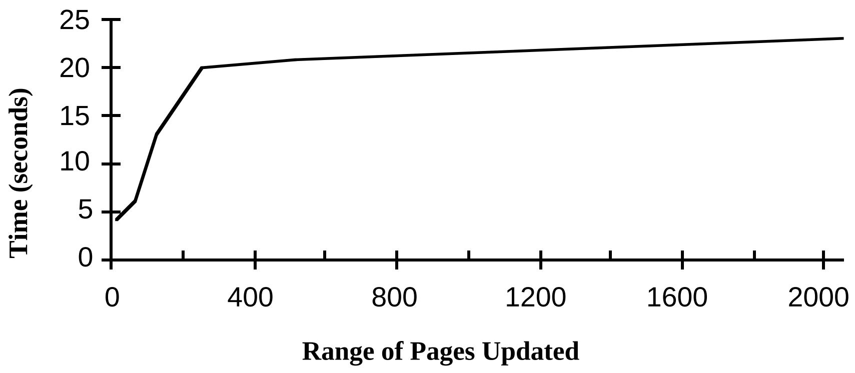

Figure 8: Effects of Locality

Figure 8 illustrates two results from the experiment. The first is shown by the increase in execution time as the range of updated pages expands to 200 pages. This is caused by the increased IO of updating increasing number of pages and highlights the sensitivity of the store to the locality of updates within pages.

The second result illustrates the increase in disk seek times. With page ranges greater than 200, the number of pages updated remains at a constant 200 but execution time slowly increases. As the range of pages increases, the cost of reading random pages over a larger area of disk increases, bearing in mind that pages are read before being updated.

This experiment is executed from a cold start. Experiment 4 gives the results of a hot experiment of multiple updates to the same pages.

Experiment 4 - This experiment illustrates the effect of performing multiple accesses to the same pages in memory. The experiment executes transactions which read 2000 pages and other transactions which read and update 2000 pages. The number of accesses performed on every page is varied.

	1 access per page	5 accesses per page	10 accesses per page	100 accesses per page
read only	23	23	23	25
read & write	148	148	148	151

Table 1: Experiment 4 (execution times in seconds)

Table 1 illustrates that the execution time of transactions does not increase greatly as the number of updates per page increases. The majority of the cost of updating a page is in shadowing the page on its first update, shown in column one. Subtracting the column one figures from the figures in the other columns gives the hot execution times of performing multiple updates on pages which have already been shadowed.

The difference in execution times between the rows is caused by the current FLASK implementation performing 2 page writes per updated page. The first write is required by the SunOS *mmap()* function to allocate a new shadow block onto which the updated page is mapped. The second is at commit time, when the updated page is written to the shadow block.

Experiment 5 - This experiment illustrates the effect of multi-programming on the store and highlights the cost of interleaving transactions. A range of concurrent transactions, from 1 to 100, are executed simultaneously. Every transaction performs a total of 40960 updates to the same 32 pages of the store. The degree of interleaving of the transactions is varied by adjusting the number of time slices per transaction. A time slice is the number of operations a transaction executes before context is switched to another transaction. Time slices per transaction vary from 1 (transactions are executed serially) to 10 (transactions are interleaved, with time slices of 4096 operations). Transactions commit at the end of their last time slice. This experiment, with more than one time slice per transaction, is a worst case example due the fact that all the pages are shared and shadowed by all transactions.

As Figure 9 illustrates, with a large number of transactions, there is a significant difference in execution time between serial and interleaved transactions. This is caused by two factors. The first is context switching. Due to the limited user control of the paging system in SunOS, the FLASK implementation requires all pages updated within a transaction's time slice to be written out to disk on a context switch. In a system with more flexible control over paging, these pages could be written back opportunistically [OS94], thus reducing context switch time.

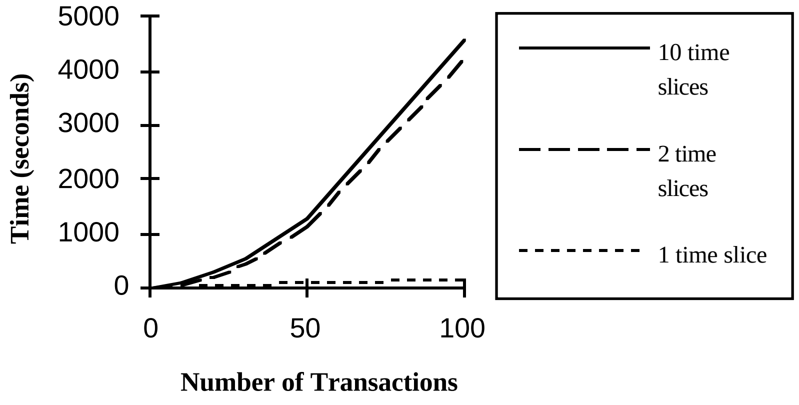

Figure 9: Worst case multi-programming

The second factor is the cost of installation reads and writes when propagating changes at commit time. In the current FLASK store, this involves reading from disk shadow pages onto which updates are being propagated. Once the changes have been made, the pages are written back to disk. Since all transactions shadow all the pages in this experiment, propagation adds a significant cost. Again, with greater control over the paging system, this cost could be reduced by keeping pages resident in memory and writing the pages opportunistically.

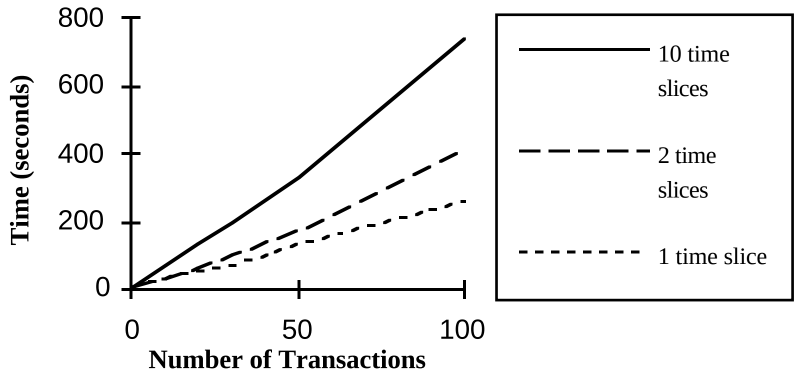

Figure 10: Less extreme multi-programming

Experiment 6 - This experiment illustrates the effects of multi-programming on the store using a less extreme transaction workload. The experiment is similar to Experiment 5 where transactions update 32 pages, performing 40960 updates. In this experiment, of the 32 pages updated, only one page is shared with one other transaction, while the other 31 pages are updated by that transaction alone. Context switching is varied as before.

This experiment reduces commit time, relative to Experiment 5, by having fewer pages shadowed by more than one transaction and therefore reduces the overall execution times. Execution times still increase from low to high interleaved transactions since context switch costs are still present.

9.1 Experiment Conclusions

The results illustrate that the cost of the xor propagation function in the FLASK model is relatively cheap compared to the other costs of transaction execution. With the present implementation, a large proportion of the execution time is spent performing IO in context switching and performing installation reads when propagating updates on commit. Therefore one method of increasing the performance of the FLASK store would be to reduce the amount of IO by keeping pages in memory, assuming that memory is large enough. This could be accomplished by building FLASK on an operating system which gives greater control over the paged memory system. With reduced IO for context switching and propagation, interleaved transaction execution times could be reduced and could be closer to the execution times of serial transactions.

The results also illustrate that the FLASK store performs better with high locality updates, since transactions performing multiple updates to the same pages do not produce more IO than transactions updating the pages once. FLASK is therefore well suited to persistent systems that perform a large number of updates which exhibit high temporal and spatial locality.

The major result of this experiment shows the effect of SunOS interference on the idealised FLASK model.

10 Acknowledgements

The authors wish to thank Chris Barter and Fred Brown for many helpful suggestions and comments on this work. The work was supported in part by SERC grant GR/G 17578 and ESPRIT III Basic Research Action 6309 - FIDE$_2$.

11 References

[ABB+86] Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. & Young, M. “Mach: A New Kernel Foundation for Unix Development”. USENIX (1986) pp 93-112.

[AD85a] Agrawal, R. & DeWitt, D. “Recovery Architectures for Multiprocessor Database Machines”. In SIGMOD International Conference on Management of Data, (1985) pp 131-147.

[AD85b] 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.

[BR91] 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.

[CAB+81] Chamberlin, D.D., Astrahan, M.M., Blasgen, M.W., Gray, J.N., King, W.F., Lindsay, B.G., Lorie, R.A., Mehl, J.W., Price, T.G., Selinger, P.G., Schkolnick, M., Slutz, D.R., Traiger, I.L., Wade, B.W. & Yost, R.A. “A History and Evaluation of System R”. CACM 24,10 (1981) pp 632-646.

[Cha78] Challis, M.P. “Data Consistency and Integrity in a Multi-User Environment”. Databases: Improving Usability and Responsiveness, Academic Press, 1978.

[Dav73] Davies, C.T. “Recovery semantics for a DB/DC System”. In Proc. ACM Annual Conference, (1973) pp 136-141.

[Dav78] Davies, C.T. “Data Processing Spheres of Control”. IBM Systems Journal 17,2 (1978) pp 179-198.

[Deu90] Deux, O. “The Story of O2”. IEEE Transactions on data and knowledge engineering. March 1990.

[EG90] Ellis, C.A. & Gibbs, S.J. “Concurrency Control in Groupware Systems”. In Proc. SIGMOD International Conference on Management of Data. (1990) pp 399-407.

[EGL+76] 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.

[Gar83] Garcia-Molina, H. “Using Semantic Knowledge for Transaction Processing in a Distributed Database”. ACM Transactions on Database Systems, 8,2 (1983) pp 186-213.

[GMB+81] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F. & Traiger, I. “The Recovery Manager of the System R Database Manager”. ACM Computing Surveys, vol. 13, no. 2, June 1981 pp 223-242.

[GMS87] Gingell, R.A., Moran, J.P. & Shannon, W.A. “Virtual Memory Architecture in SunOS”. USENIX Summer Conference Proceedings, Phoenix 1987.

[GS87] Garcia-Molina, H. & Salem, K. “Sagas”. In Proc. SIGMOD International Conference on Management of Data. (1987) pp 249-259.

[KGC85] Kent, J., Garcia-Molina, H. & Chung, J. “An experimental evaluation of crash recovery mechanisms”. In Proc. 4th ACM Symposium on Principles of Database Systems (1985) pp 113-122.

[Kol87] Kolodner, E.K. “Recovery Using Virtual Memory”. M.Sc. Thesis, MIT (1987).

[Kra87] Krablin, G.L. "Building Flexible Multilevel Transactions in a Distributed Persistent Environment". 2nd International Workshop on Persistent Object Systems, Appin, (August 1987) pp 213-234.

[Lor77] Lorie, A.L. Physical Integrity in a Large Segmented Database, ACM Transactions on Database Systems, 2,1 (1977) pp 91-104.

[Mos81] Moss, J.E.B. “Nested Transactions: An Approach to Reliable Distributed Computing”. Ph.D. Thesis, MIT (1981).

[Mun93] Munro, D.S. “On the Integration of Concurrency, Distribution and Persistence”. Ph.D. Thesis, University of St Andrews (1993).

[NZ92] Nodine, M.H. & Zdonik, S. B. “Co-operative Transaction Hierarchies: Transaction Support for Design Applications”. VLDB Journal 1,1 (1992) pp 41-80.

[OLS85] Oki, B., Liskov, B. & Scheifler, R. “Reliable Object Storage to Support Atomic Actions”. In Proc 10th Symposium on Operating Systems Principles, 1985 pp 147-159.

[OS94] O'Toole, J. & Shrira, L. "Opportunistic Log: Efficient Installation Reads in a Reliable Object Server". Technical Report MIT/LCS-TM-506, March 1994. To appear in 1st International Symposium on Operating Systems Design and Implementation, Monterey, CA (1994).

[PS87] “The PS-algol Reference Manual fourth edition”. Technical Report PPRR-12 (1987), Universities of Glasgow and St Andrews.

[Ree78] Reed, D.P. “Naming and Synchronisation in a Decentralised Computer”. Ph.D. Thesis, M.I.T. (1978).

[SM92] Stemple, D. & Morrison, R. "Specifying Flexible Concurrency Control Schemes: An Abstract Operational Approach". Australian Computer Science Conference 15, Tasmania (1992) pp 873-891.

[Sto86] Stonebraker, M. (Editor) “The Ingres Papers”. Addison-Wesley, Reading, MA (1986).

[Sut91] Sutton, S. “A Flexible Consistency Model for Persistent Data in Software-Process Programming”. In Dearle, Shaw, Zdonik (eds.), Implementing Persistent Object Bases, Principles and Practice, Morgan Kaufmann, 1991 pp 305-319.

[VDD+91] Velez, F., Darnis, V., DeWitt, D., Futtersack, P., Harrus, G., Maier, D., and Raoux, M. “Implementing the O_2 object manager: some lessons”. In Dearle, Shaw, Zdonik (eds.), Implementing Persistent Object Bases, Principles and Practice, Morgan Kaufmann, 1991 pp 131-138.