A Multi-User Shadow Paging Transaction Manager on Mach 3.0

Tatu Ylönen <ylo@cs.hut.fi>
Heikki Suonsivu <hsu@cs.hut.fi>

Helsinki University of Technology
Laboratory of Information Processing Science
SF-02150 Espoo, Finland

September 7, 1992

Abstract

This paper discusses the theory of multi-user shadow paging [7, 8] and describes an implementation on Mach 3.0. The implementation makes extensive use of Mach 3.0 features, including multiple threads, Mach RPC, virtual memory, and asynchronous device I/O. The implementation provides a page-level transaction manager for a database system. It allows multiple concurrent transactions, and provides full crash recovery, locking, and cache management features. The cache manager uses innovative, previously unpublished algorithms, and some new innovations are also presented for multi-user shadow paging.

It turns out that multiple threads and other Mach 3 features allow a very elegant and efficient implementation.

1 Introduction

The idea of shadow paging is to never overwrite valid data. Modified pages are always written to unused portions of the database. A page table is used to map logical page numbers to physical page numbers. During transaction processing there are two page tables: a current page table in

memory and a shadow page table on disk. Only the current page table is modified while a transaction is active, and at commit time the current page table is written to disk and made the new shadow page table, thus atomically changing making the new data a part of the database.

The original idea of shadow paging was presented in [11]. It was implemented in the System R database manager. System R supported concurrent operation, and implemented a complicated multi-user shadow paging system using logs on directory pages [4].

Extending shadow paging to concurrent operation was more difficult than expected. Some approaches are listed in [15]; however, none of them is quite satisfactory. The general consensus became that shadow paging is not appropriate for multi-user systems [4, pp. 239–240].

A simpler multi-user version of shadow paging was presented in [10] (also briefly described in [1]). However, in a comparison with log-based methods and differential files [1], its performance was found inferior except in an environment with only large transactions. The primary overhead turned out to be updating the page table.

A much better multi-user shadow paging system was described in [7, 8]. In their system, among other things, the overhead due to page table I/O is greatly reduced by committing several transactions simultaneously. Their experimental results indicate that shadow paging is better than logging if the database contains large records, has high locality within a transaction, or the database is small.

2 Multi-User Shadow Paging

The presentation of shadow paging here differs somewhat from that in [7]; however, most of the basic ideas are identical.[1]

2.1 Definitions

A *transaction* is an atomic operation on the database. The changes made by a transaction become permanent when the transaction *commits*. A

[1]In fact, we independently reinvented most of the ideas in [7], and only later found out that they had been presented before. Several new, previously unpublished ideas will be presented in [16].

transaction may be *aborted* (or *rolled back*) to undo its changes before it has committed. The changes made by a transaction are either entirely made (the transaction committed), or not made at all (the transaction was aborted).

Several transactions can be executing *concurrently*. The database system must ensure *serializability*; that is, it must guarantee that after the transactions have been executed, the database will look as if the transactions had been executed in some (arbitrary) serial order, one at a time. Usually, some sort of *locking* will be used.

A *page* is a data block in the database. Its size is usually a multiple of the size of a disk block. In this paper, all input/output to/from the database is assumed to be done one page at a time.

A *physical page* means an actual disk block (or a group of blocks forming a page) on disk.

A *logical page* is a page of the database as seen by the application (or higher levels of the database system). The higher levels only deal with logical page numbers; physical page numbers are private to the transaction manager.

A *page table* is a mapping from logical page numbers to physical page numbers. There is a physical page for each allocated logical page number. Conceptually, the page table can be thought of as a table which is indexed by the logical page number, and each slot contains the physical page number corresponding to that page. In practice the page table is often stored using two levels.

The *free list* is a list of unused physical page numbers in the database (in some systems, a bitmap is instead of a list). The *shadow free list* is part of the shadow page table on disk. The current free list is in memory, and describes the current database state (possibly including uncommitted transactions). Note that if a crash occurs, the free list in the shadow page table will always be consistent, and will look as if the uncommitted transaction(s) had never been started.

The use of a shadow free list is optional. It is possible (and also quite efficient) to extract the free list from the page table when the database is opened. Maintaining the shadow free list is fairly costly, and thus in server-based systems its use is probably undesirable.

The database has a *page table pointer*, which contains the address of the shadow page table on disk. It is stored in stable storage. The value

of the page table pointer must be valid at all times. Stable storage for the page table pointer can be implemented by writing it to two blocks, each with a checksum.

The term *remap* is used in this paper for the structure (logical_pageno, old_physical_pageno, new_physical_pageno). It is used to describe that the logical page was previously mapped to old_physical_pageno, but a new page has been allocated for it, and it is now mapped to new_physical_-pageno. Either old_physical_pageno or new_physical_pageno can be *null* (an invalid page number), meaning that the logical page number was allocated or freed during this transaction, respectively.

A *remap list* is a list of remaps. (In an actual implementation remap lists might be implemented e.g. as a hash table to avoid linear search times.) All updates made by a transaction are described by its remap list. A transaction can be aborted by returning all "new" pages on its remap list to the current free list (no other action is needed), and committed by creating a new page table where the "new" mappings on the remap list have replaced the corresponding old mappings in the page table. The transaction is committed when the address of the new page table has been written to the page table pointer (the system of course needs to wait for the earlier writes to complete before writing the page table pointer).

2.2 Single-User Shadow Paging

Using this terminology, single-user shadow paging can be described as follows.

When a transaction starts, an empty remap list is created for it.

When a transaction allocates a new logical page, a physical page is allocated, and the structure (logical_pageno, null, physical_pageno) is put on the remap list.

When a transaction frees a logical page, the structure (logical_pageno, old_physical_pageno, null) is put on the remap list.

When a transaction reads a page, its remap list is consulted to find a mapping for the page. If a mapping is found, it is used. Otherwise, the page table of the database will is to find a mapping. The appropriate physical page is then read.

When a transaction writes a page, its remap list is consulted to find a mapping for the page. If a mapping is found, the page has already

been remapped by this transaction, and the new physical page is used. If there is no such mapping, a new physical page is allocated, the structure (logical_pageno, old_physical_pageno, new_physical_pageno) is added to the remap list, and the page is written to the new physical location.

If a transaction is aborted, all "new" pages on its remap list are put back on the current free list. No further action is needed, as the transaction has only modified unused potions of the database.

When an application requests to commit a transaction, a new page table is constructed, containing all the mappings in the old page table, modified by the mappings on the remap list. If a shadow free list is used, a new list is constructed, containing the contents of the current free list, plus the "old" pages on the remap list, plus the pages used by the old page table and free list. The new page table and free list are then written to disk. When all "new" pages on remap list, the new page table, and the new free list have reached non-volatile storage, the page table pointer is updated to point to the new page table. The transaction is now fully committed. (Note that if the page table has two levels, only the first level page and the modified second level pages need to be created. Similarily, if the shadow free list consists of many pages, its tail may be reused).

2.3 Locking and Concurrent Transactions

Concurrent transactions need some control to maintain serializability. This paper only considers two-phase locking [9, 375–377].

With two-phase locking, a transaction consists of two phases: the growing phase and the shrinking phase. During the growing phase the transaction is allowed to lock data items in shared and/or exclusive mode, and to upgrade shared locks to exclusive locks. However, the transaction is not allowed to release any locks during this time. Similarily, during the shrinking phase the transaction can only release locks or downgrade exclusive locks to shared locks.

The two-phase protocol guarantees serializability, and can be used even if the operations that will be performed by the transaction are not known beforehand. It can be easily implemented by locking a data item in shared mode when it is first read, and locking it in exclusive mode when it is first written. Without knowledge about the future operations done by the transaction, it is not possible to release any locks until the transaction has partially committed.

All shared locks can be released when the transaction is partially committed, as it is then known that the transaction will no longer access the data item. Exclusive locks cannot be released until the transaction has been committed. However, if there is only a single thread processing commits as described below, and it is acceptable that a later transaction can read uncommitted data (but not commit before the uncommitted data is committed), exclusive locks can be released as soon as the transaction is partially committed.

2.4 Extending Shadow Paging to Concurrent Transactions

2.4.1 Overview

In this section the concepts developed earlier are used to extend shadow paging to handle concurrent transactions. The main ideas are that each transaction has its own remap list, all locking is done on logical pages, and the locking system ensures that no logical page can be on two remap lists simultaneously. When a transaction partially commits, its remap list is updated to the current page table, and the transaction is committed very similarily to the non-concurrent case. The remap lists of multiple partially committed transactions can be merged, and multiple transactions committed simultaneously.

2.4.2 Two-Phase Locking

A page is locked in shared mode when it is first read, and in exclusive mode when it is first written. As a result, any page on a remap list is exclusively locked. Since only one transaction can have an exclusive lock on a page, a page can never be on more than one remap list.

Any number of transactions can be active simultaneously. The locking protocol will ensure serializability. Occasionally, two or more transactions may deadlock; in such a case, one of the transactions has to be aborted. Aborting a transaction is very cheap, as all that is needed is to put the "new" pages on its remap list back to the current free list.

2.4.3 Commit Processing

Whenever an application requests the database system to commit a transaction, the transaction is put in partially committed state. All shared locks held by the transaction can be released at this point.

To actually commit transactions, the database system will periodically find all transactions that are partially committed.

Commit processing begins by finding the set T_{pc} of partially committed transactions. A new page table is constructed by applying the remap lists of all T_{pc} to the current page table, and the new page table is written to disk. There is no need to reconstruct the entire page table; instead, since the page table itself is shadowed, only the modified pages need to be written. If a shadow free list is used, it is constructed (see below) and written to disk.

The commit process will have to wait until all "new" pages on remap lists of T_{pc}, as well as pages used by the new page table and the new free list, have reached non-volatile storage. When they are safely on disk, the page table pointer can be written to stable storage.

When the page table pointer has reached stable storage, all T_{pc} are fully committed. The "old" pages on their remap lists are put on the current free list, and the applications are notified of a successful commit.

2.4.4 Shadow Free List

If a shadow free list is used, it can be constructed at commit time to hold the following pages numbers.

- All pages on the current free list
- All "old" pages on remap lists of T_{pc}
- All "new" pages on remap lists of all transactions other than T_{pc}
- All "old" pages of the old page table and the old free list

The shadow free list is only used when the database is opened (possibly after a crash).

An alternative to using a shadow free list is extracting the current free list from the shadow page table when the database is opened. This avoids

the cost of computing and writing the shadow free list at every commit, and incurs only minor costs at startup time. In large systems where the database is managed by a multithreaded server, it is probably best not to maintain a shadow free list. In small applications without a centralized database server, the use of a shadow free list somewhat speeds up opening the database.

Assuming database pages are 8kB, and the size of the database is 1GB, there are about 130.000 pages in the database, and about 64 pages in the page table. Reading and processing the page table will take at most a few seconds.

2.5 Evaluation

An interesting feature of multi-user shadow paging is that due to commit merging its I/O overhead approaches zero as the level of concurrency increases. Thus, if locking and I/O bandwidth do not limit transaction throughput, very high TPS (transactions per second) rates can be achieved.

Assuming that the total I/O bandwidth does not become a problem (it can be solved by adding more disks), response time depends on how often commits are done. The time needed for a commit is in the worst case twice the maximum duration of a commit. Further speedup is possible if the shadow page table and the page table pointer are stored on a separate disk.

Except for the page table address, all data is stored as normal non-volatile database pages. If multiple disks are used, I/O can be divided quite evenly between disks. Provided that locking does not become a bottleneck, the achievable TPS rate should improve linearily as a function of the number of disks. Also, CPU time overhead per transaction is very small, so CPU time should not be a bottleneck. Since the algorithm is most easily implemented as a multithreaded server, it adapts nicely to multiprocessors.

The biggest performance problem with shadow paging is the cost of reading the page table if it is not in the cache. However, if the disk cache is sufficiently large, the page table can be held in the cache. With 8 kB pages and 4 byte page table entries, the size of the page table is 1/2000 of the size of the database. For a 1 GB database this means 0.5 MB of page tables; for a 1000 GB database the size of page tables is 500 MB.

At current cache memory prices, considering the database size, it is quite realistic to use this much memory for caching the page table.

3 Implementation

3.1 Overview

We have implemented a multithreaded Mach server called **ioserv** which implements concurrent shadow paging. Clients talk to the server using Mach 3.0 RPC. The transaction server talks to the device server to access devices. (Actually, we have a separate small server called **opendev**, which opens a device and passes its device port to **ioserv**. The **opendev** server and the actual device may reside on a different machine than the transaction server.)

The **ioserv** transaction server provides the following functions (mig-generated RPC interfaces):

open_database(server, host, device, db_port_return, pagesize_return)
close_database(db_port)
start_transaction(db_port, transaction_id_return)
commit_transaction(db_port, transaction_id)
abort_transaction(db_port, transaction_id)
alloc_page(db_port, transaction_id, pageno_return)
free_page(db_port, transaction_id, pageno)
read_page(db_port, transaction_id, pageno, data_return)
write_page(db_port, transaction_id, pageno, data)
read_page_hint(db_port, transaction_id, pageno)

All I/O is done with full pages. There is no way to bypass the transaction system. Transactions are identified by integers (actually just a sequence number of the transaction). Inside the transaction server, there is a data structure for each active transaction.

The server has a pool of threads waiting for user requests. When a request arrives, a thread begins to serve it. The thread may block (e.g. waiting for a lock) while processing the request.

Many threads may be executing normal database operations concurrently. Access to critical data structures is protected by locks. For this purpose, Mach cthreads style mutexes are used. Sometimes, a thread needs to block on a lock, I/O wait, or pending commit processing. For

this purpose, cthreads style conditions are used. A single thread is used to do all the commits. It will sleep if it has no work. It can commit many transactions simultaneously.

We have chosen not to store the free page list in the database. Instead, we extract the free page list from the page table when the database is opened. In memory, we have a free list for physical page numbers, and a free list for logical page numbers.

3.2 Transaction Manager

The transaction manager implements concurrent shadow paging. It uses the cache manager to do disk caching and I/O and the lock manager to maintain locks on pages.

3.2.1 Processing of a Transaction

A transaction is started when an application calls the **start_transaction** function. A data structure is created for the transaction, an identifier is allocated, and the structure is stored in a per-database hash table of transactions.

Most operations begin by looking up the transaction from the hash table. To read a page, the logical page is first locked in shared mode by calling the lock manager (this operation may block). If the lock manager reports deadlock, the transaction is aborted. Otherwise, the logical page number is mapped to a physical page number by first looking for a remap on the transaction's remap list, and if not found, by mapping the page number using the page table. The cache manager is then called to read and return the data. Write operations are done similarily, except that if the page was not already on the remap list, a new page will be allocated, added to the remap list, and the page will be written to it. Allocating and freeing pages only affects the remap list.

If the transaction needs to be aborted, either due to a request from the application, or due to an error (such as deadlock), all "new" pages on the transactions remap list are put back to the system's free list the data structure of the transaction is removed from the hash table and freed.

When an application requests to commit a transaction, the transaction's data structure is put on a list of transactions waiting to be com-

mitted, and the commit thread is awakened if it was waiting for work. The current thread is suspended until commit processing is complete.

When the commit thread begins executing (or has finished processing the previous batch of commits), it takes all partially committed transactions (these are on the commit list) and empties the list. It will then install the mappings on their remap lists into the page table, allocating new pages for changed page table pages. All pages modified by the transactions, and the modified mapping pages, are flushed to disk. When all pages have reached disk, the root pointer is written to two adjacent physical pages. When the root pointer pages have reached disk, the transactions are irrevocably committed. Their "old" pages are put to the systems free list, and the threads processing the application commit are awakened.

3.2.2 Atomic storage of the root pointer

The root pointer is stored twice, in physical pages 0 and 1. Both root pointers are normally identical, and contain a checksum (actually a CRC). When opening the database, both root pointers are read. If they are identical, no special actions are needed. If they differ, their checksums are verified, and the pointer with the correct checksum is used. If both checksums are valid but the pointers are different, both root pointers point to a valid page table. We arbitrarily pick pointer 0.

3.2.3 Crash recovery

Crash recovery must be done whenever the database is opened. In our case, the only thing to do is extracting the free page list from the page table.

To extract the list, we create a bitmap containing a bit for each physical page of the database. Physical pages 0 and 1 are marked as used, as they contain the root pointers.

Then, for each page of the page table, for every pointer on the page, if the pointer is valid, we mark the corresponding page as used in the bitmap. If the pointer is invalid (null), we put the logical page number corresponding to the pointer on the logical page free list.

We then go through the bitmap, putting all those page numbers which have not been marked as used on the physical free page list.

3.3 Cache Manager

The cache manager is an independent subsystem providing efficient cache management on top of Mach 3.0. The cache manager deals with physical pages.

3.3.1 ELRU Page Replacement Method

The cache manager uses ELRU page replacement strategy [13] to add LFU (Least Frequently Used) behaviour to a LRU (Least Recently Used) cache. ELRU behaves very similarly to GCLOCK [3, 12], but does not suffer the bad worst case of GCLOCK. ELRU is simple to implement and provides almost constant and quick page replacement time.

The ELRU method (figure 1) is based on having several LRU lists for different priorities similar to previously presented methods [2, 6]. As a general cache manager it is not acceptable to require application intervention to set the priorities, so ELRU uses page requests by applications to quantify the priority each page should have. This is done by moving the page to a higher priority LRU list each time the page is requested. The pages are inserted in the ELRU structure, not at the bottom list, but to the the *insertion list* to give the pages reasonable initial priority. To find a page to replace, simply take the least recently used page in the bottom priority list. If the list became empty, all lists are "rolled" a step down, thus a page once frequently used but now unused will wander from the high priority list towards the bottom list. For more ideas and detail, see [13].

ELRU is less trivial to implement than GCLOCK would be, more so in hardware. For our application, however, it seems quite reasonable solution.

3.3.2 The Cache Structure

The cache structure is presented in figure 2. The pages are linked in the ELRU structure by their priority and to a hash table by the page number.

To avoid copying, page descriptor and data are kept separate. If the data was copied by the application, the pointer in the descriptor is replaced and original data freed.

To avoid copying the cache system keeps the page buffer and page

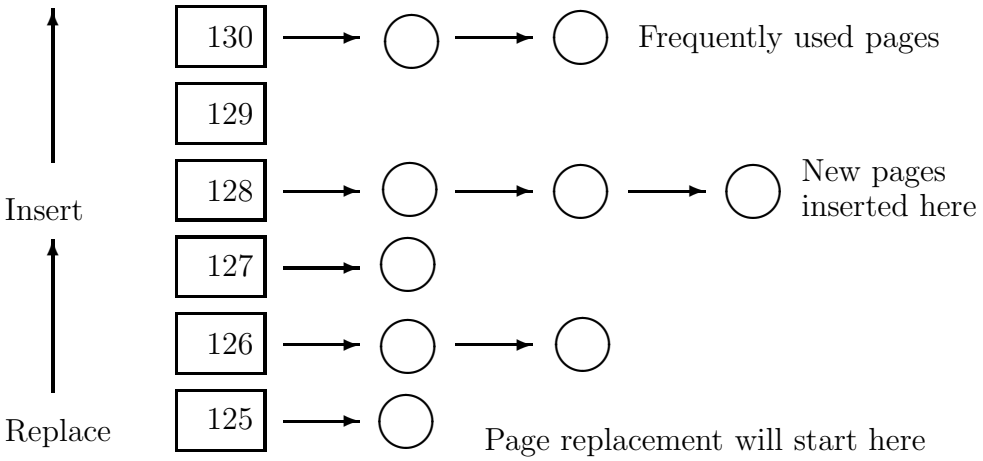

Figure 1: ELRU structure.

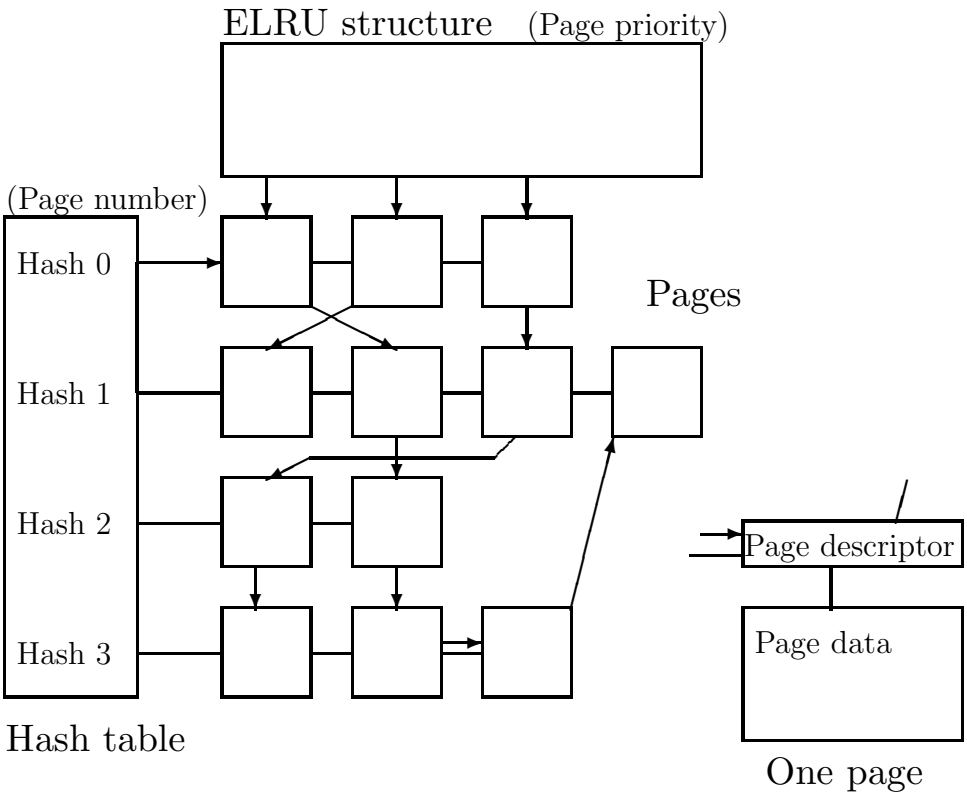

Figure 2: The cache structure.

descriptor (bucket) separate, thus when the application takes a page, copies it to a new buffer and requests a write for that buffer, no more copying needs to be done, but instead the pointer to the new buffer is replaced and the old buffer can be freed.

The cache manager provides the following functions, used by the **ioserv**:

cache_init(cache, device, cache_size, device_size)
cache_free(cache)
cache_read_hint(cache, pageno)
cache_read(cache, pageno, data_return)
cache_release(cache, pageno, data_return)
cache_flush_wait(cache, pageno)
cache_flush_write(cache, pageno)
cache_write(cache, pageno, data_return)

The pages are identified by the physical page number, unlike the **ioserv**, which uses logical page numbers. All I/O goes through the cache manager to the **opendev** server.

3.3.3 Page replacement

The cache manager uses Mach 3.0 asynchronous writes to allow the device driver (or the device itself) to optimize seeks.

The page replacement algorithm is presented in figures 3.3.3, 4 and 5.

3.4 Lock Manager

The lock manager provides two-phase locking [9, 375-377] and deadlock detection. Our multi-user shadow paging implementation uses logical page numbers to identify locks.

Locks are stored as structures in a hash table. The deadlock detection algorithm uses wait-for graph search.

The lock manager provides the following functions, used by the **ioserv**.

locks_db_init(locks_db, size)
locks_transaction_init(locks_db, locks_transaction)
locks_transaction_free(locks_db, locks_transaction)
locks_db_free(locks_db)

```
replace_one_page(page)
  setup Mach reply port
  asynchronous_write(page)
```

Figure 3: Replace one page.

```
write_reply(port)
  map the reply port to a page
  if the page has become dirty while the write was active
    no need to do anything
  else
    remove the page from the cache
  endif
```

Figure 4: Mach write reply processing.

```
replace()
  while more free space needed
    take a page from the bottom LRU list
    if the page is dirty
      replace_one_page(the page)
    else
      remove the page from the memory
    endif
  end while
```

Figure 5: Page replecement request.

locks_lock_shared(locks_db, locks_transaction, pageno)
locks_lock_exclusive(locks_db, locks_transaction, pageno)
locks_unlock_shared(locks_db, locks_transaction, pageno)
locks_unlock_all_shared(locks_db, locks_transaction)
locks_unlock_all(locks_db, locks_transaction)

4 Conclusions and Future Work

The Mach 3.0 microkernel system seems to be a good, production level operating system. The kernel interface is quite clean and straightforward to program with, and has no kludgy feel that the traditional Unix has. Mach proved to be a stable environment.

The usefullness and efficiency of asynchronous writes did create a lot of discussion. We did not do any comparision between synchronous and asynchronous writes.

The system could be implemented in about 5000 lines of code, which we think is quite reasonable for a system which can be a base for many applications, like file systems, database systems and any applications which require transaction processing.

We also have several completely new ideas on shadow paging [16, 14], that warrant further research. These papers will address problems like delaying allocation of physical pages until commit, releasing exclusive locks as soon as a transaction has partially committed, using several root pointers to avoid bottlenecks, dumps of a live database, two-phase commit in a distributed system, concurrent access to the parts of a page, management of large objects and shadow paging for variable-sized objects.

References

[1] Agrawal, R., Dewitt, D. J. *Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation*, ACM Transactions on Database Systems, Vol. 10, No. 4, 1985, pp. 529–564.

[2] Michael J. Carey, Rajiv Jauhari, Miron Livny, *Priority in DBMS Resource Scheduling*, Proceedings of the Fifteenth International Conference on Very Large Data Bases, Amsterdam, 1989, pp. 397-410.

[3] Wolfgang Effelsberg, Theo Haerder, *Principles of Database Buffer Management*, ACM Transactions on Database Systems, Vol. 9, No. 4, December 1984, pp. 560-595.

[4] 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, 1981, pp. 223–242.

[5] Haerder, T., Reuter, A. *Principles of Transaction-Oriented Database Recovery*, Computing Surveys, Vol. 15, No. 4, 1983, pp. 287–317.

[6] Rajiv Jauhari, Michael J. Carey, Miron Livny, *Priority-Hints: An Algorithm for Priority-Based Buffer Management*. Proceedings of the Sixteenth International Conference on Very Large Data Bases, Brisbane, Australia, 1990, pp. 708-721.

[7] Kent, J. M. *Performance and Implementation Issues in Database Crash Recovery*, Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, Princeton University, 1985.

[8] Kent, J., Garcia-Molina, H., Chung, J. *An Experimental Evaluation of Crash Recovery Mechanisms*, Proc. ACM Principles of Database Systems, 1985, pp. 113–121.

[9] Korth, H. F., Silberschatz, A. *Database System Concepts*, McGraw-Hill, New York, 1986.

[10] Lampson, B., Sturgis, H. *Crash Recovery in a Distributed Data Storage System*, Computer Science Lab., Xerox PARC, 1979.

[11] Lorie, R. A. *Physical Integrity in a Large Segmented Database*, ACM Transactions on Database Systems, Vol. 2, No. 1, 1977, pp. 91–104.

[12] Alan Jay Smith, *Sequentiality and prefetching in database systems*, ACM Transactions on Database Systems, Vol. 3, No. 3, September 1978, pp. 223-247.

[13] Heikki Suonsivu, *Extended Least Recently Used Algorithm*, article/tech report, in preparation, 1992.

[14] Heikki Suonsivu, *Shadowing Variable-sized Objects*, article/tech report, in preparation, 1992.

[15] Verhofstad, J. S. M. *Recovery Techniques for Database Systems*, Computing Surveys, Vol. 10, No. 2, 1978, pp. 167–195.

[16] Ylönen, T. *Concurrent Shadow Paging: An Alternative to Logging?*, Technical Report, Laboratory of Information Processing Science, Helsinki University of Technology, 1992.