Shadow Paging Is Feasible

Tatu Ylönen <ylo@cs.hut.fi>

Department of Computer Science, Helsinki University of Technology
Otakaari 1, FIN-02150 Espoo, Finland

Abstract

`Shadows` is a high-performance database system that uses *shadow paging* without any logs for recovery. It supports ACID transactions, media recovery, fine-granularity locking, and efficient index management. This paper describes the I/O, recovery, and concurrency control issues involved and their solutions. An overview of the implementation is provided, together with performance results.

1 Introduction

A database system must provide atomic transactions, consistency of data, isolation of concurrent transactions, and durability of committed transactions. Additionally, large systems must provide high performance and features such as on-the-fly dumping and media recovery.

Most database systems use logs to ensure atomicity and durability. If the system crashes in the middle of a transaction, the log can be used to bring the database into a transaction-consistent state. If a disk or other hardware device fails, a backup together with the log can be used to recover the state of any committed transactions. Currently almost all commercial systems and most research prototypes use some form of logging for recovery.

An alternative method for achieving atomicity (*shadow paging*) is to have a mapping from logical to physical database pages (*page table*), and execute transactions by allocating new physical pages for any modified pages and then atomically update the mapping to reflect the new database state. Atomic mapping update can be achieved by using two page tables and a "current" bit in stable storage [7], by using shadow paging together with logging [3], by recording changes in an *intentions list* (kind of temporary redo log) in stable storage before changing the page table [6], or by constructing a new page table which partially shares the old page table and then atomically updating a *page table pointer* in stable storage [5].

Shadow paging has not been used in database implementations because it has had inferior performance and it has not been obvious how to implement fine-granularity locking for concurrent transactions, media recovery, and a number of other desired features.

Shadow paging has recently become more competitive due to development in disk and main memory technologies. By combining several techniques, such

as log-structured file systems [9, 11, 12], striping [12, 15], mirroring [10], shadow updates [1], and group commit [5], the performance of shadow paging can now be made to compete with the best log-based systems. New algorithms have been developed to implement other desired features, such as fast snapshots (transaction-consistent copies of the database) that can be used to do arbitrary read-only transactions or on-the-fly dumping (backups) without any locking [16].

There are a number of application areas where shadow paging may offer important benefits compared to log-based systems. Examples include embedded databases in telecommunications, where fast recovery can be extremely important, and decision support applications with large read-only transactions mixed with small updates (snapshots can be used to execute them without locking).

We have implemented a prototype system using shadow paging without any logs. The prototype is a full database system, with emphasis on the data manager (recovery, concurrency control, and index management issues). Other aspects of database management, such as data models, query languages, and distribution, have only been given cursory consideration to determine the features needed from the database system kernel. The algorithms and the structure of the implementation are described in this paper, together with performance results.

The author is not aware of any serious database system that uses shadow paging without logs. Particularly, the index management and media recovery implementations for shadow paging are novel. Performance results suggest that the prototype is the first shadow paging system that can compete with commercial log-based databases.

This paper is organized as follows. Section 2 describes page-table level shadow paging, and a number of improvements to it that make shadow paging really usable. This level of the system is capable of executing only one transaction at a time. Section 3 introduces concurrent transactions with page-level locking. Concurrent transactions are implemented on top of page-table level transactions, which are used to execute commit batches (multiple transactions are committed together as a single page-table level transaction). Section 4 describes how fine-granularity locking for transactions is implemented, and discusses extended lock modes and index management. Section 5 gives performance results for the TPC-B benchmark [2]. Section 6 concludes the paper. Due to space constraints, snapshots and two-phase commit are not considered in this paper; the interested reader is referred to [16].

2 Page Table Level

The general idea in shadow paging is that existing data is never overwritten, but instead modified pages are always written to new locations on disk, and a mapping is used to keep track of the current location of each page. The mapping is called the page table. All pointers in the database refer to logical page numbers, and must be mapped to physical pages using the page table before reading.

To execute an arbitrary atomic transaction, all modified data pages are written to new locations, and then a new page table is constructed and written to fresh pages on disk, and finally a page table pointer is written to atomically switch to the new permanent database state.

In our implementation, the page table is a tree structure with a fixed number N of slots in each node. On the lowest level (leaf nodes) each slot contains a pointer to a data page. The array of slots in the node is indexed by $L \bmod N$ (where L is the logical page number). On the second lowest level, the array in a node is indexed by $(L/N) \bmod N$, and generally by $(L/N^{level-1}) \bmod N$. The depth of the page table is $\log_N L_{max}$, and grows dynamically whenever needed. There is a page table pointer which records the current depth of the page table and the physical location of the root page. The page table pointer is stored twice to implement stable storage.

The page table can be used to execute one transaction at a time by storing uncommitted changes in a *remap* in main memory. The remap is a mapping of $\langle L, P_{new} \rangle$, where L is a logical page number (used as a key) and P_{new} is the new physical location of the page on disk (or *nil* if the page has been freed by this transaction). A read is performed by first checking if there is a new version in the remap, using that version if found, and by mapping through the page table otherwise. A write is performed by allocating a copy of the page and adding it to the remap (unless we already have a new copy of it), and modifying the copy. Note that actual copying of pages can be avoided by renaming the page in cache to match the new address.

The transaction can be aborted by simply freeing all new copies in the remap. It can be committed by constructing a new page table. The new page table is constructed by allocating new copies of modified page table pages but sharing unmodified parts. Both the modified data pages and modified page table pages are then written to disk, and when the writes have completed, the page table pointer is written (at which point the transaction commits permanently). Old physical pages are freed once the page table pointer has been written. The algorithm is outlined in Figure 1.

Note that new copies of page table pages can be managed easily by adding a page table level field to the remap, and using the combination of L and *level* as a key. For data pages, L is the logical page and *level* is 0; for page table pages, L is the number of the first page mapped by the page table page, and *level* is the level of the page in the page table (1 for leaf-level page table pages).

2.1 Making Writes Sequential

Shadow paging implies the *force policy*; that is, all modified pages must be written to non-volatile storage before a transaction can commit. It is thus very important to make these writes efficient.

The new physical locations of pages are not used for anything until the transaction requests to commit (except, on the implementation level, to access the pages from the cache). It is possible to use arbitrary identifiers until that point, and only when committing do the actual allocation of physical pages. At that point, it is known which pages are going to be written. Space for the

for L in each logical page in the remap **do**
  for $level = 1$ **to** *pagetable_depth* **do**
    if no entry in remap for $L - L \bmod N^{level}$ at $level$
      Make a copy of page table page $L - L \bmod N^{level}$ at $level$
      Add entry $(L - L \bmod N^{level}, level, P_{new})$ to remap
    Update mapping for L at $level - 1$ in $L - L \bmod N^{level}$ at $level$
    Save old physical position of the page in a list
Write all modified pages to disk
Write page table pointer
Free old physical pages

Figure 1: Algorithm for page table level commit.

new pages can be allocated from a contiguous region of disk (or several disks to implement striping), and all of the writes can be combined to a sequential write. Modified page table pages can be written the same way.

The commit algorithm is divided into several steps: first new copies are made of all affected page table pages, then space is allocated for the data and page level pages, then the new addresses are updated into the new copies of page table pages, and finally all modified data is written to disk and the page table pointer is updated. The new algorithm is outlined in Figure 2.

Late allocation of physical page numbers is implemented by allocating *virtual page identifiers* instead of physical page numbers for new copies. Old pages in the cache are renamed whenever possible to avoid copying. At commit, a physical page number is associated with each virtual page identifier, and the physical page number is stored in the page table.

The cache can associate physical pages with virtual identifiers before commit to cope with very large update transactions. It will still need to maintain a mapping from the virtual identifiers to the physical page numbers, but is free to flush dirty pages before commit. When the cache does this, it has enough data to again make long sequential writes.

Sequential writes are only possible if there is sufficient contiguous free space on disk. Without any countermeasures, free space will quickly get fragmented, and no large contiguous regions will exist. We do space management much in the same way as log-structured file systems [9, 11]: we divide the disk space into fixed-size segments, and have a separate thread actively cleaning up segments. Any pages that the cleanup thread wishes to move are made part of the next page table level transaction (i.e., copied to new locations without actually changing them), and they will be freed after the next commit. Writes are done to empty segments only, which implies that they can be made sequentially. The relative cleanup cost depends on the fraction of free space available in the segments being cleaned up.

The system collects statistics about the modification rate of each logical page. Pages which have short life expectancies are written to the same segment, making it likely that the segment will quickly become empty even without cleanup. Pages which are long-lived are also put together, making it likely that

for L in each logical page in the remap (note: all at $level = 0$) **do**
  for $level = 1$ **to** *pagetable_depth* **do**
    if entry in remap for $L - L \bmod N^{level}$ at $level$
      then break /* do next iteration of outer loop */
    Make a copy of page table page $L - L \bmod N^{level}$ at $level$
    Add entry $(L - L \bmod N^{level}, level, I_{new})$ to remap
Allocate physical storage for all virtual identifiers in the remap
for each mapping in the remap (L and $level$; all $level$s) **do**
  Save old physical position of the page L at $level$ in a list
  Update mapping for L at $level$ in $L - L \bmod N^{level+1}$ at $level + 1$
Write all modified pages to disk (sorted by position)
Write page table pointer
Free old physical pages

Figure 2: Algorithm for page table level commit with sequential writes.

the segment is going to be almost full for a long time, reducing the need for cleanup.

2.2 Media Recovery

Since there is no log, backups plus log cannot be used for media recovery. Instead, a variant of mirroring is used. The idea is to have two copies of the page table pointer (on two failure-independent disks), two copies of each page table page, and two copies of each data page. There are no fixed pairs of disks which mirror each other; there is just the constraint that the two copies of a page must be on failure-independent disks. Again, the actual allocation of physical page numbers is postponed until the commit batch, and the pages will then be allocated as two sequential regions on independent disks. There is a location for the page table pointer on each disk, and it is written to the two disks which are most idle (a sequence number is used to identify which disk contains the newest pointer at recovery time).

This implementation of mirroring has some interesting consequences. First, a page can be read from the more idle disk (as in normal mirroring). Second, since both writes are sequential, and are done to different disks in parallel (and the seek time is small compared to the transfer time for long writes), there is little write penalty for doing mirroring. This gives most of the benefits of doubly distorted mirrors [8], but no special hardware of any kind is needed. If the system crashes in the middle of the write, neither copy will be part of the committed database, and thus will never be read.

Recovery from a corrupted disk block is simple: just read the other copy (corruption can be detected either by the disk controller, or by the database system when it fails to find a proper page header) and schedule the page for relocation (as in cleanup). If a whole disk crashes, data can be recovered by traversing the page table, and relocating all those pages which have one copy on the failed disk.

2.3 Other Optimization Issues

Shadow paging, especially the write optimization described above, destroys any clustering between different logical pages. Thus, clustering is only possible within a logical page. However, it is easy to have logical pages of different sizes by recording the size of the page in the page table. Small pages can be used for frequently updated data with random accesses, whereas large pages are good for data which is typically accessed sequentially, such as blobs and multimedia data.

In a system with multiple disks, one issue is balancing the load among the disks. One of the heuristics used to select on which disk to write next is to write to the disk which is most idle. This tends to move more traffic to the disk, essentially balancing load. Other factors that affect the allocation of segments include independency of copies (absolute constraint), available disk space, and recent allocation patterns.

One small but important optimization is caching logical to physical mappings. There is a small translation cache ("translation lookaside buffer" [13]) that is used to cache frequently used mappings. Benchmarks indicate that a small 4096-entry 2-way set-associative cache [13] handles over 90 % of all translations in TPC-B.

2.4 Structure of a Page Table Entry

The size of a page table entry is 16 bytes. This allows 40 bits for each physical page number, 40 bits for timestamp (used for snapshots and on-the-fly incremental dumping), and one byte for page size, flags and free space information. With 512 byte blocks, 40 bits for the address allows 512 terabyte maximum database size. The timestamp is a commit batch sequence number. At ten batches per second it wraps around after 3 500 years. Page size is stored as $\log_2 S - 9$ in 4 bits. 4 bits are available for flags and allocation information.

Typical page sizes are a few kilobytes for frequently updated relational data, and up to 1 MB for blobs and multimedia data. The size of the page table is about 0.01 % to 1 % of the size of the database. In many applications it fits entirely in main memory cache.

2.5 Example

To clarify what has been explained so far, let us consider a simple example with just basic shadow paging, mirroring, and sequential writes.

Suppose we have a transaction which executes the following sequence of actions (page numbers refer to logical pages): Read[1], Read[10], Allocate[8], Write[8], Write[1], Write[10]. Assume that logical page 1 is originally stored at 1:55 and 2:78 (disk-number:block) and 10 at 1:98 and 3:12. Assume there is only one page table page, stored at 2:17 and 3:103. All pages are assumed to be eight physical blocks (4 kB) in size.

The remap after executing this transaction will contain {(8, 0, I_0), (1, 0, I_1), (10, 0, I_2)}, where each entry is of the form (logical, page-table-level, new-identifier). The transaction then requests to commit. First, all affected page

table pages are identified and added to the remap: (0, 1, I_3) in this case. Then, disk space is allocated for all of the new pages. Assume there is enough space available at 2:50 and 3:200. Space is allocated: I_0 at 2:50 and 3:200; I_1 at 2:58 and 3:208; I_2 at 2:66 and 3:216; I_3 at 2:74 and 3:224. Old addresses of modified pages are saved in a list, resulting in (1:55, 2:78, 1:98, 3:12, 2:17, 3:103). New addresses for data pages are stored in the page table page. All writes are then sorted and performed as a sequential write in parallel to both disks. (If there had not been enough contiguous free space in one location, the allocation would have been made as a few large regions.) Finally, when all writes have completed, the new page table address (2:74, 3:224) is written to page table pointers on two disks (say, 1 and 3). Then the old copies of the pages are freed.

3 Concurrent Transactions with Page-Level Updates

Everything that has been described so far deals with a single transaction at a time. From now on, we will be having multiple concurrent transactions, and what has been the single transaction so far will be called a commit batch.

The idea is that each transaction has its own remap, and two-phase locking on logical page numbers is used for concurrency control. If a transaction requests to abort, any "new" copies in its remap are freed and its locks are released. If it requests to commit, it is put in a queue of transactions waiting to be committed, and a separate thread will periodically take all transactions from the commit queue, combine their remaps together, and commit the combined single transaction (commit batch) exactly as described above. This is possible because a transaction must have an exclusive lock for any logical page it has on its remap, so there can be no conflicts between the remaps of individual transactions.

3.1 Early Releasing of Locks

We have shown in [14] that all locks of a transaction can be released when the commit request is received, provided that actual commits are performed in the same order as the commit requests are received (implied by queuing committing transactions), and transactions only read and write data that has been either committed or at least commit-requested. This was called *partially strict two-phase locking*, and it was shown that such locking maintains strictness as long as an abort does not follow the commit request (an abort caused by deadlock or requested by the transaction cannot follow the commit request, so such aborts can only happen due to lack of disk space or other exceptional errors). If an abort follows the commit request, we suggested that all active transactions be aborted.

This optimization reduces the time exclusive locks are held, and it is particularly important with main-memory databases and shadow paging, where the cost of commit processing is relatively high. The benefit is greatest when the application makes non-commutative updates to hotspot data.

A requirement for this optimization to work is that when locks are released, other transactions will see commit-requested data. What we actually do is that

Prevent transactions from merging remaps
Build new page tables (allocate and update pages in cache)
Update in-memory copy of page table pointer
Clear global remap
Permit transactions to merge remaps
Write modified pages to disk (sorted by position)
Write page table pointer to disk
Free old physical pages

Figure 3: Page table level commit when concurrent transactions are considered.

there is a separate global remap in addition to the per-transaction remaps, and when a transaction requests to commit, it merges its own remap to the global remap, then releases its locks, and enters a wait until it has been committed or aborted by the commit thread. Note that it is possible that a logical page be on both the global remap and a local remap; however, the transaction that has it on local remap has the page exclusively locked and the page must have been on the global remap before this transaction locked it. The older version of such entries is freed during the merge. This frees the intermediate version of the page but not the original version on disk (which cannot be freed until the transactions have really committed).

Reads will first check the transaction's own remap, then the global remap, and if these fail read the page using the page table. Writes will first check if there already is a copy on own remap, then check the global remap (and make a copy to the local remap), and if these fail read it using the page table and make a copy to the local remap.

The commit batch works by taking the global remap, building the new page tables in cache, and then it updates an in-memory copy of the page table pointer (before actually writing anything) and clears the global remap. During this time no additions may be made to the global remap (thus, commit requests must wait while the page tables are built). Since these are main-memory operations, they are fast. The algorithm is outlined in Figure 3.

The commit batch executes in a loop, starting a new batch as soon as the previous one has completed, provided there is work to do.

3.2 Example

As an example, suppose there are three transactions: T_1: Read[1], Read[4], Write[1], Write[4], Commit; T_2: Read[1], Write[1], Commit; T_3: Read[1], Write[5], Read[4], Commit. For clarity, suppose the transactions execute in this order, and T_1 and T_2 end up in one batch, and T_3 in another. Physical page numbers, mirroring and other such issues will not be considered on this level, as they are handled as described earlier for the single transaction.

T_1 will have the following remap when it requests to commit: $\{(1,\ 0,\ I_0),\ (4,\ 0,\ I_1)\}$. This remap will be merged with the global remap, and all locks are released. T_2 then executes. Page 1 is read from I_0 as indicated by the global

remap; when it is written, T_2's own remap becomes $\{(1, 0, I_2)\}$. This is then merged with the global remap, freeing I_0, and the global remap becomes $\{(1, 0, I_2), (4, 0, I_1)\}$. T_2's locks are then freed.

Suppose that a commit batch now starts, and takes the global remap, makes allocations and builds a new page table as indicated by it, and clears the global remap. It then begins to write. T_3 will be executing in parallel; the only restriction is that if it commits while the page table is being built, it must wait until the build is complete. T_3 will read page 1 from I_2 if the read is performed before the page table is built, and from the new physical location (in cache) using the new page table (in cache) if the read happens after the page table is built. Similarly for page 4. T_3's remap will be $\{(5, 0, I_3)\}$. This will be merged with the global remap (now empty) when T_3 requests to commit. Transactions T_1 and T_2 will be awakened by the commit thread when its writes have completed. This happens asynchronously to the execution of T_3. The next batch begins then, taking care of T_3 and any later transactions.

4 Fine-Granularity Locking

Fine-granularity locking [4] has been difficult with shadow paging because changes cannot be made directly to data pages. The reason is that if two transactions modify different parts of the same page, they cannot commit independently because one might want to abort after the other has committed and already written its changes to disk, and without a log there would be no way to undo.

What we do is that we keep uncommitted changes made by transactions in a separate data structure in main memory. For example, we record that a record with with a certain identifier was modified and keep enough information to actually do the modification later.

4.1 Concurrency Control

Two-phase locking on logical identifiers (record identifiers, key values, etc.) is used for concurrency control. It is quite possible that the actual data moves to a different page while the transaction is active. When the transaction requests to commit, its changes are made to actual data pages (see below).

An important distinction to the page level algorithm is that all locks are on logical identifiers. The identifier may identify a record, be a key value, refer to an entire table, or something else. There is a hierarchy of lock identifiers; hierarchical locking is used [4]. A lock on a logical identifier is not bound to any particular page; for example, a lock on a single key value covers no pages at all since the key might move to a different page, whereas a lock on a whole table covers all pages of that table.

4.2 Large Transactions

If the transaction holds a lock which is known to cover a page (such as a lock on the whole table), it can use the page-level update scheme described earlier. However, if its lock does not cover the page containing the modified data, it has

to store the change separately until it commits, and modify any reads it makes so that they reflect the change. When the transaction requests to commit, any of its page-level updates are merged with the global remap, and any changes it has stored separately are made to actual data pages in cache, creating new entries on the global remap as pages are modified.

This is implemented so that a transaction initially uses fine-granularity locking (record locking, key-value locking, or something similar) and stores any changes in main-memory data structures. If the transaction makes very many updates to a table, it takes a coarse granularity lock or table-level lock that covers the pages it is changing ("lock escalation"). After that, it is free to use page-level updates for those pages covered by the lock. It immediately makes the updates in main memory to the cache, and makes any further updates to the pages covered by the lock using page level updates. Since the cache can flush modified pages before commit, transactions sizes are not limited by the size of the main memory cache.

4.3 Committing

Actually committing a transaction is handled by a separate commit thread which executes batches, one at a time. There is a global remap for pages which have been modified and requested to be committed; this remap will form the next commit batch. The modifications of each transaction must be merged to this remap atomically with respect to commit batches. Commit batch processing is essentially identical to that described in Figure 3.

Then a transaction requests to commit, it merges its local remap to the global remap, and performs any stored fine-granularity modifications to the actual data pages, using page level updates and putting the new copies of pages on the global remap. We call this *patching*. No new transaction-level locks are acquired at this time; instead, low-level latches are used to protect consistency of individual updates. When the transaction has patched all its changes, it frees its local data structures, releases all locks (see early releasing of locks, above), and waits for the next commit batch to commit it.

If there is an error during patching, the whole commit batch and all later transactions will need to be aborted. Thus, transactions must check for common errors, such as the requested record being nonexistent, while the transaction is still active. When consistency checks are performed before actual commit, a transaction should never need to abort after commit request, except in situations such as a system crash or lack of disk space.

4.4 Commutative Operations

This scheme supports commutative operations (extended lock modes). Several transactions can have an update mode lock on the same data item as long as the operation is stored in the per-transaction data structure in a commutative fashion. The data structure must support multiple operations by different transactions for a single object.

4.5 Index Management

The same scheme is used for index locking. The operations include insertions, deletions, updates, and range queries. To support locking for range queries, the uncommitted changes of all transactions for an index are stored in a global data structure ordered by key value (such as a skip list or a tree). Next-key or previous-key locking can be used, and both the data structure in memory and the index on disk are consulted to find the next/previous key value.

Index management is very clean: the actual index is built above page-level operations, and does not need to worry about crash recovery because it is handled by the underlying shadow paging mechanism, or about transaction-level recovery (abortions) because they are handled by the fine-granularity layer above the index. The index is just the basic algorithm (e.g., B-tree) with latches used for what is called locks in the literature. The fine-granularity layer above the index provides arbitrary transactions and transaction-level locking. Implementation is simplified and made more reliable by not mixing the different aspects, and many other index data structures can be substituted with only minor changes in the fine-granularity layer. For a detailed description, see [16, pp. 75–92].

4.6 Example

Suppose a transaction wants to make a search in a B-tree, then insert a record, and then insert a record into another B-tree. Suppose next-key locking [4] is used for B-tree concurrency control, and records are stored directly in the tree. It is assumed that the implementation has three layers: fine-granularity locking code for B-trees (manipulating main-memory data), normal B-tree code, and page-level shadow paging to implement recovery. The B-tree code uses page-level operations. All transaction-level locking is handled by the fine-granularity layer. Low-level locks are not discussed here, but are needed in the implementation on all levels.

First, the transaction looks up the first record with higher or equal key value to that specified in the query. The record is looked up both in the disk-based B-tree and in the main memory data structure for that index. A lock is obtained on the smaller key value found (as required by next-key locking). The found record (from either the B-tree, main memory, or combined from both) is returned. This process may be repeated several times if the search is a range query; if the query is very big, the system may at some point decide to obtain a shared table-level lock and stop taking key value locks.

Then, the transaction performs the insertion. This is done by first performing a search to verify that the record does not already exist, and to locate the next higher key value as needed by the next-key locking protocol. Both the next key and the inserted key value are locked. Finally, the new record is added into the main-memory data structures, and the insertion is complete.

The insertion for the other B-tree is performed identically, except that each tree has its own main-memory data structures and locks.

If the transaction had made very many updates, at some point it would

have obtained a table-level exclusive lock, made any updates in main memory to the B-tree, and used direct B-tree updates without further locking for the rest of its updates.

When the transaction requests to commit, the system prevents a new commit batch from starting (and waits if the current batch is building page tables). The transaction then merges its remap to the global remap and makes all its modifications in main memory to the B-tree on disk (adding changed pages to the global remap). This operation is fast because all relevant pages should be in the cache. Note that unless the transaction was very large, this is the first time that page-level updates are performed. All locks of the transaction are released when the changes have been made. From this point on, the changes will be visible to other transactions through the global remap.

Finally, the transaction is inserted into a queue of transactions waiting to be committed, and a new commit batch is allowed to start. The transaction waits until the commit batch has either committed it or aborted it (in which case all other transactions will be aborted as well).

5 Performance Results

We have implemented a prototype system using the algorithms in this paper ([16] has some more details). It uses shadow paging for crash recovery and has no logs. Mirroring is used for media recovery. Fine-granularity locking with the early releasing of locks optimization as described above is used for concurrency control. B-trees are used for indices, with recovery and arbitrary transactions as outlined above. A simple relational database has been implemented on top of the core system. The prototype totals about 50 000 lines of C++ source code (it has some features not described here, and the source tree contains a fair amount of experimental or debugging code).

To get some idea about the performance of the system, we have run the TPC-B benchmark [2]. The benchmark simulates a large bank, and all transactions are simple debit-credit updates. Full serializability, durability, and recovery from any single-point hardware failure are required by the benchmark. The size of the database scales with the benchmark results.

The benchmarks were run on a Sun SparcStation 10/402 with 10 1–2 GB SCSI disks from various manufacturers (IBM, HP, Quantum) for the database. The machine had 96 MB main memory, of which 30 megabytes were used for database cache (including page table cache). The results were obtained by running TPC-B; most tests were run three times, and the best result was used (to reduce adverse effects caused by other users of the same system during test). The benchmark is believed to match the official benchmark, except for the following: it was only run for 5 minutes in each case (to reduce the real time needed for the tests), and the cleanup thread was not fully active, because it would have taken substantial amount of time and effort to ensure that it has reached steady state. Thus, we chose to use a fresh database to keep the results for each set of parameters comparable. Figure 6 depicts the performance degradation due to cleanup as a function of time in our configuration.

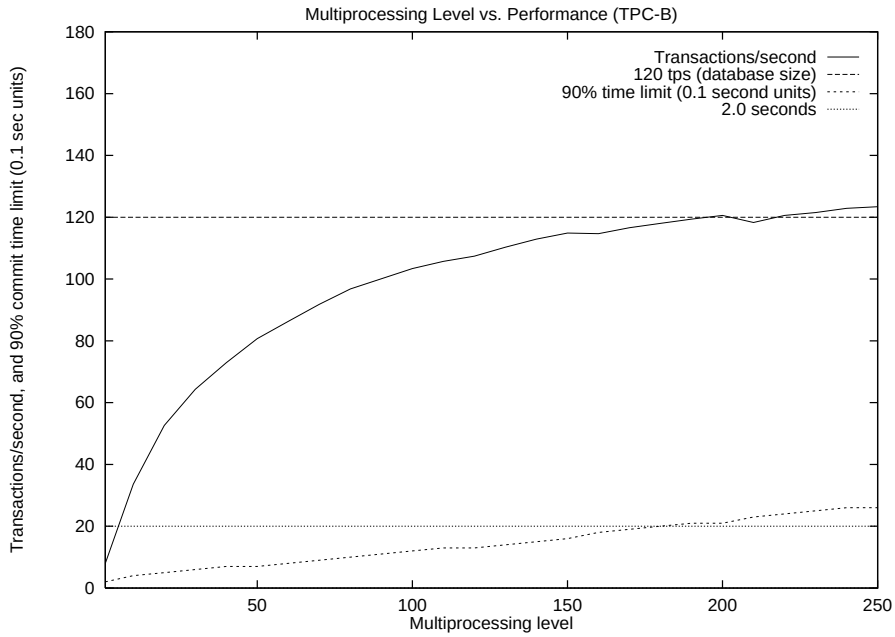

Figure 4: Performance of `Shadows`: transactions per second (upper curve) and the limit under which 90 percent of all committed transactions complete (lower curve, 0.1 second units).

Current benchmark results (as of May 1995) are given in Figure 4. It gives the transactions per second rate as a function of the multiprocessing level, and the time limit under which 90 percent of committed transactions complete. The distribution of completion times for a number of multiprocessing levels is shown in Figure 5. The number of aborted transactions is not shown, but was less than 0.1 percent of started transactions in all tests.

The performance is currently in the same range as good commercial databases for the same machine (though commercial bechmarks often use a larger number of disks). Considering that the prototype has not undergone the same research, design, implementation, and tuning effort as commercial log-based databases, the results strongly indicate that it is feasible to build competitive high-performance databases using shadow paging without logging.

6 Conclusion

An efficient implementation of shadow paging has been described. Due to space limitations, the description has necessarily been too concise to thoroughly cover all the details. The interested reader is referred to [16] for more information.

A prototype relational database system with full serializability and recovery has been built. The system uses shadow paging for recovery and uses no logs at all. The TPC-B benchmark has been used to measure the performance of the prototype.

Benchmark results indicate performance in the same class as good commercial databases on the same platform. The system can offer the necessary

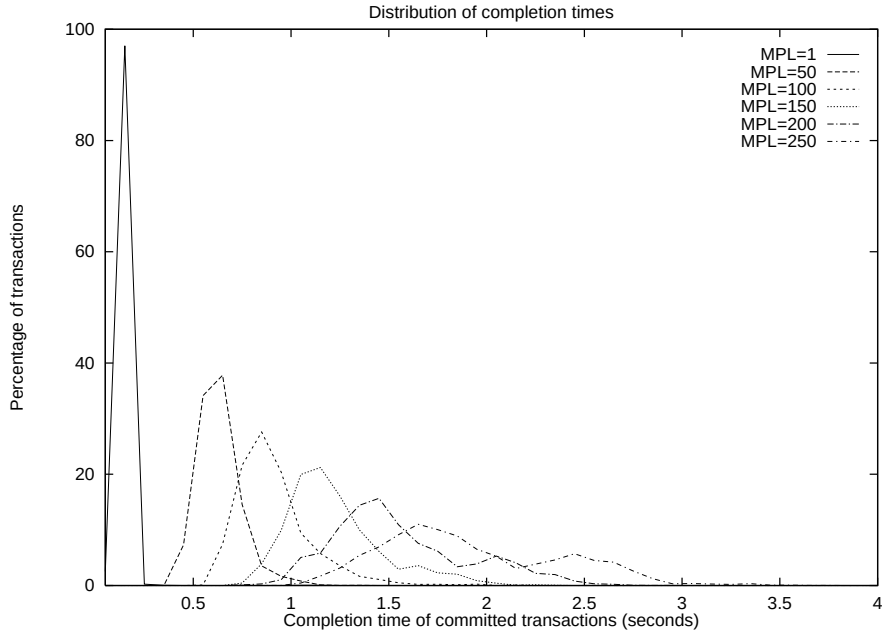

Figure 5: Completion time distribution for committed transactions (seconds).

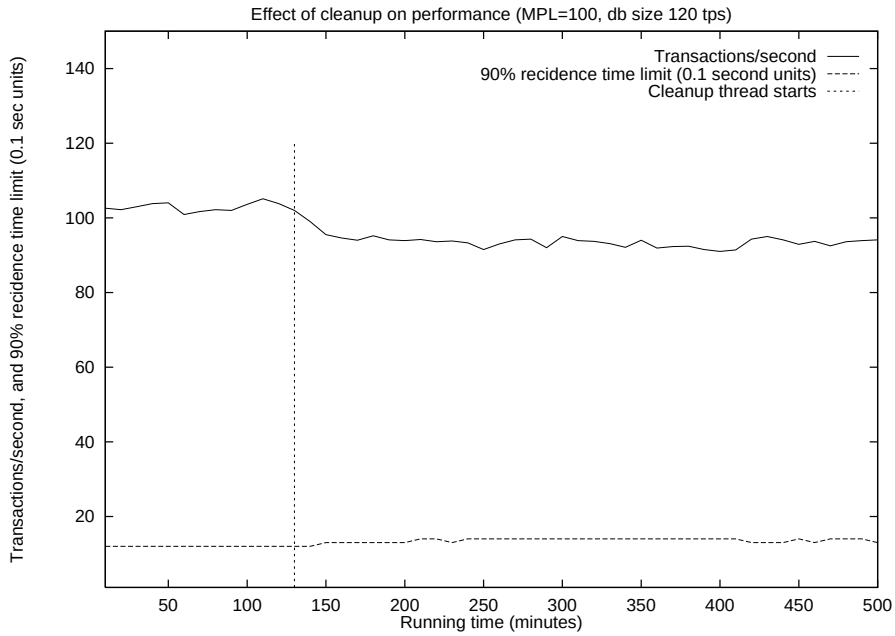

Figure 6: Performance as a function of the time the system has been running after the database was created. Performance degrades slightly when the cleanup thread starts.

features for industrial-strength database systems (although the prototype is not up to that level yet). Balancing this with the effort expended on research and development of log-based databases compared to the almost nonexistent research on shadow paging, shadow paging offers an interesting alternative for future databases.

Shadow paging sometimes behaves differently from log-based systems. Not all benchmark and research results from log-based systems are applicable to shadow paging. Media recovery, index management, dumping, and two-phase commit are all distinctively different.

There is no single shadow paging algorithm, just as there is no single log-based algorithm. Four published variants were mentioned in the introduction, and these only refer to the single-transaction-at-a-time layer of the system. Many more alternatives exist on the higher levels of the system. One must be careful not to condemn all alternatives (with strikingly different performance profiles) on the basis of just one or a few alternatives.

Acknowledgements

Tero Kivinen, Heikki Suonsivu, and the author built the first prototype using these ideas, and several other people (Johannes Helander, Sampo Kellomäki, Kenneth Oksanen, Kengatharan Sivalingam, and Eljas Soisalon-Soininen) have participated in the development of the current prototype and the algorithms in it.

The project was supported by TEKES (Finnish Technology Development Centre), Academy of Finland, Nokia Telecommunications, Sun Microsystems, Solid Information Systems, and New Generation Software (NGS).

References

[1] M. H. Eich. A classification and comparison of main memory database recovery techniques. In *Data Engineering*, pages 332–339, 1987.

[2] J. Gray. *The Benchmark Handbook.* Morgan Kaufmann, second edition, 1993.

[3] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the System R database manager. *Computing Surveys*, 13(2):223–243, 1981.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[5] J. M. Kent. *Performance and Implementation Issues in Database Crash Recovery.* PhD thesis, Princeton University, 1985.

[6] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, April 1979.

[7] R. A. Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems*, 2(1):91–104, 1977.

[8] C. U. Orji and J. A. Solworth. Doubly distorted mirrors. In *ACM SIGMOD*, pages 307–316, 1993.

[9] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *ACM Operating Systems Review*, 23(1):11–28, 1989.

[10] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD*, pages 109–116, 1988.

[11] M. Rosenblum. *The Design and Implementation of a Log-Structured File System*. Kluwer, 1995.

[12] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Symposium on Operating Systems Principles*, pages 1–15, 1991.

[13] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, 1982.

[14] E. Soisalon-Soininen and T. Ylönen. Partial strictness in two-phase locking. In *Proc. International Conference on Database Theory*, number 893 in Lecture Notes in Computer Science, pages 139–147. Springer-Verlag, 1995.

[15] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The design of XPRS. In *Very Large Data Bases*, pages 318–330, 1988.

[16] T. Ylönen. *Shadow Paging Is Feasible*. Licentiate's thesis, Department of Computer Science, Helsinki University of Technology, 1994. Available on WWW as http://www.cs.hut.fi/~ylo/techreports/lisuri.ps.