Concurrent Shadow Paging: A New Direction for Database Research

Tatu Ylönen

Laboratory of Information Processing Science
Helsinki University of Technology, SF-02150 Espoo, Finland
E-mail: ylo@cs.hut.fi

Abstract

In this paper we present several new ideas about concurrent shadow paging as a crash recovery method in databases. We show how to use shadow paging in a multi-user environment and describe several optimizations and ideas which significantly improve the performance and general usability of shadow paging, making it very competitive to other methods in crash recovery. Shadow paging with our optimizations appears to be much faster than log-based solutions, and it would seem to be very generally applicable. We also present some new, potentially very useful ideas which can be efficiently implemented with shadow paging but not with logging.

1 Introduction

The idea of shadow paging is to never overwrite valid data. A page table is used to map logical page numbers to physical page numbers, and modified pages are always written to unused parts of the database. The page table is updated atomically at transaction commit to reflect the new state of the database. This allows arbitrary atomic transactions.

The original idea of shadow paging was presented in [16]. It was implemented in the System R database manager, which supported concurrent operation using a complicated multi-user shadow paging system with logs on directory pages [6]. Some other approaches to concurrent operation were listed in [28]. However, in general, shadow paging was considered inappropriate for large multi-user systems [6, pp. 239–240].

A simpler multi-user version of shadow paging was presented in [2]. However, in comparison with log-based methods and differential files [2, 3], its performance was found inferior. The primary overhead turned out to be updating and indirection through the page table and the root pointer.

In [9, 10], the overhead due to root pointer updates and page table I/O was reduced by committing several transactions simultaneously as a batch. Their experimental results indicated that shadow paging was better than logging in some environments, but in general logging still performed better. Most of the overhead in shadow paging was caused by page table reads.

Currently, most major database systems use logging [18], and there has been little work on shadow paging for several years. However, memory sizes and correspondingly cache sizes have increased significantly, and it is now quite reasonable to keep the entire page table of even a very large database in main memory. For example, assuming 8 kB pages and 4-byte page table entries, the page table is 50 kB for a 100 MB database and 50 MB for a 100 GB database. The cost of the main memory needed to hold the page table is on the order of one percent of the cost of the disk space for the database.

Even conventional shadow paging performs fairly well when the page table is in main memory [2, 10]. The ideas in this paper speed up writes by a factor of ten, not only compensating for the page table updates but also making shadow paging potentially much faster than update-in-place solutions. We also present several other enhancements and new ideas which significantly improve the usefulness of shadow paging.

2 Concurrent Shadow Paging

The presentation of shadow paging here differs somewhat from that in [9, 10]; however, most of the basic ideas are identical. The physical structure of the database is illustrated in figure 1.

The database has a page table pointer, which contains the address of the page table on disk. The page table pointer points to a valid page table at all times and is updated atomically. We also use the same page to hold other rapidly changing information, and call the whole page the page table pointer.

The page table forms a mapping from logical page numbers to physical page numbers. There is a physical page for each allocated logical page number. The page table on disk is called the shadow page table. We do not have a "current" page table represented explicitly; instead, each transaction has its own incremental page table [2, 13, 17].

The incremental page table is conceptually a list of changes to the page table. Each entry in the incremental page table has the form $\langle L, P_{old}, P_{new} \rangle$, where L is the logical page number, P_{old} is the old physical page number corresponding to the logical page, and P_{new} is the new physical page number.

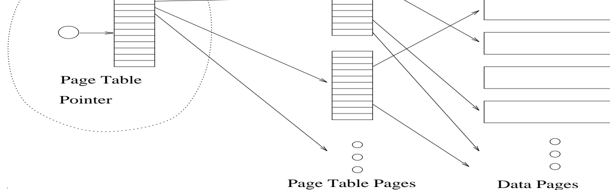

Figure 1: Shadow paging file structure.

To use shadow paging in a multi-user environment, we make all reads lock the accessed logical page in shared mode, and all writes lock the logical page in exclusive mode. Shared locks can be released when the application requests to commit the transaction, and exclusive locks when the transaction has actually committed. (This is effectively two-phase locking.) Since the incremental page table contains entries only for modified pages, and modified pages are exclusively locked, we can guarantee that no page is ever in more than one incremental page table.

When an application requests to commit a transaction, the transaction is marked as partially committed. The database system will periodically (e.g. ten times a second, or when processing of the previous commit batch has completed) take all partially committed transactions, and commit them as a batch. Since no page can be in more than one incremental page table, we know that there are no conflicts between the transactions. The changes made by them are installed in the global page table, allocating new pages for modified page table pages (the page table itself is shadowed). All pages modified by the transaction and the modified page table pages are then flushed to disk. When they have reached the disk, the page table pointer is written to two adjacent physical pages (to implement atomic update). When it has reached the disk, all transactions in the batch have committed and a new batch can begin. The effect of commit batching has been studied for shadow paging in [10] and for logging in [4, 15, 7].

There is no need for garbage collection. If a transaction is aborted, the "new" physical pages in its incremental page table are put back to the free list, and no further action is needed. After a transaction has committed, the "old" physical pages in its incremental page table are put to the free list, as well as old versions of the modified page table pages.

3 Allocating Adjacent Physical Pages in Commit

Allocation of physical page numbers can be delayed until the transaction commits, or the modified page is flushed from the cache. It is possible to allocate all pages needed during a commit batch as a contiguous region (or a two-dimensional array on multiple disks [27]). This means that there is no need to seek between writes, and all pages made by several transactions can be written to disk sequentially. The speedup on writes is on the order of a factor of ten.[1]

This optimization contradicts the usual attempt to keep the logical-to-physical mapping linear, or at least approximately linear [2, 10]. In update-in-place systems, every write eventually results in a seek and a write to the original location (except in hot spots where multiple writes to the same data may be combined). In dynamically mapped systems (shadow paging), it is possible to allocate a contiguous region and avoid almost all of the seeks and rotational latencies. In terms of I/O resources, the cost of a typical transaction is reduced to almost a half (reads are unaffected, but writes become very cheap), almost doubling the rate of transactions per second that can be supported by the same equipment. Moveover, caching can be much more effective if writes are cheap: caching only affects reads, and writes must still eventually go to disk (except in hot-spots). If writes are as expensive as reads and half the operations are writes, we can at most about double the performance by improving caching. If writes only cost a tenth of the cost of reads, performance can be improved by a factor of ten.

Log-based file systems [20, 24] are based on a similar optimization.

4 Clustering

An important problem in databases is clustering related data together. Clustering within a page is easily achieved; however, since the whole idea of shadow paging is to always write a page to a new location, clustering between pages is more difficult. Approaches which attempt to keep the logical-to-physical mapping approximately linear [2, 10] are not compatible with our write optimizations. The problem is at worst with large multi-megabyte objects. In the worst case, every page of a large object could require a separate seek.

Clustering for large objects can be achieved by combining several adjacent physical pages into a multipage. Multipages are only a logical-level concept; they are made of adjacent normal-sized physical pages. Page table entries contain the size (the number of contiguous physical pages) of each logical page in addition to the address of the (first) physical page. Multipages can be fairly large and large objects can be constructed from relatively few multipages. The multipages forming a large object can be stored on different disk drives, allowing parallel reading from multiple drives, analogously to the two-dimensional file system of [27].

The use of multipages requires that there is a sufficient amount of free space available in the database so that contiguous free areas can be found. One general

[1] For a disk with 15 ms seek time (including latency) and 3 MB/sec transfer rate, the speedup is by a factor of 45 for 1 kB pages, by a factor of 12 for 4 kB pages, and by a factor of 6 for 8 kB pages.

thread combine free pages into larger areas by moving used pages adjacent to a larger free area to fill smaller free areas. The moving task is very simple because all modifications are local, affecting only a single logical page at a time. The logical page number of a physical page can be found by a simple reverse mapping table in main memory (the table can be fairly large, on the order of the size of the page table).

Clustering is also important for scanning large tables. Unlike large objects, a table usually consists of a large number of small independently updatable data items. Since updating a part of a multipage requires the entire multipage to be written to a new location, multipages are not appropriate for structures where small data items are updated frequently. There is a tradeoff in multipage size between the efficiency of small updates and the efficiency of sequential scans. It is perfectly feasible to select the multipage size on a per-table or per-object-type basis. Also, since writes are very cheap in the concurrent shadow paging environment, it is possible to maintain more indexes than in update-in-place systems at the same cost, reducing the need for sequential scans.

5 Automatic I/O Load Balancing

Most of the disk I/O in a concurrent shadow paging system is caused by reads. Writes are fairly long sequential bursts followed by a page table pointer write. Reads cannot easily be balanced between disks; however, since we can freely choose where to write a page, we can write to the disk that is the most idle one. Since the page is at the same time removed from its previous location (likely to be a more busy disk), future load of the busy disk is reduced and the load of the more idle disk is increased. With suitable parameterization (to avoid oscillations), automatic load balancing between disks can be achieved.

The page table pointer must reside at a fixed location. However, it is possible to have several page table pointers (e.g., one on each disk) if each pointer contains a timestamp indicating the sequence number of the last commit to that pointer. At startup time, the system selects the pointer with the highest sequence number. At commit time, the pointer on the most idle disk can be selected.

6 Early Releasing of Locks

Normally, shared locks can be released when a transaction is partially committed, and exclusive locks when the transaction is fully committed. However, we have only one commit batch active at a time, and when a new batch begins, all partially committed transactions will be included in the batch. This means that when a transaction partially commits, no other transaction can get fully committed earlier. Thus, we can release exclusive locks as soon as the transaction is partially committed and trust that no transaction that has accessed the uncommitted data can commit its modifications without the earlier transaction committing. This significantly reduces the time exclusive locks are held, allowing more concurrency. If full persistence is not required by the application, successful commit can be reported to the application immediately; otherwise successful commit can be reported after the next commit batch has completed.

A side-effect of this optimization is that later transactions may see uncommitted data. It is guaranteed that the data will not get explicitly aborted, and if it gets implicitly aborted by a system crash, any transactions that have used that data will also get aborted by an implicit cascading rollback. Transactions might display uncommitted data to the user before they die, but since this can only occur in connection with a system failure, this is acceptable in most applications.

A similar optimization has been used in some log-based implementations (e.g. [4, p. 7]).

7 Fine-Granularity Locking

Fine-granularity locking has been used in several database systems [23, 19, 18], although many commercial systems still use page or table level locking. We are not aware of any previous work on fine-granularity locking with shadow paging.

The major problem with fine-granularity locking is that we would like to allow multiple transactions to modify different parts of a page, and it must be possible to individually commit or abort each of these modifications. Thus, the transactions cannot share a copy of the page and install it in the page table when one of them commits. Instead, each transaction must keep its changes separately, and only at commit time store them into the page. The changes can be kept, for example, on a list attached to the transaction, indexed by page number and byte range or an object identifier.

This approach to fine-granularity locking works well for small transactions; however, large transactions may modify many objects (e.g., incrementing the value of a field in every record of a table). Storing all changes in main memory may not be feasible, and commit would become very slow because the changes would need to be installed to their respective pages at commit time. A solution is to use fine-granularity locking for all new transactions, and after a transaction has modified a certain number of objects, begin to use page-level locking for the transaction. A large transaction will often hold a table-level lock, and thus resorting to page-level locking would not reduce concurrency in this case.

Snapshot4 Snapshot5 Snapshot6 Snapshot7 Current state

Figure 2: Snapshots represent consistent database states as of some time in the past.

8 Snapshots

It is possible to take a snapshot [1] of the entire database by saving the address of the page table and preventing freeing of pages that are in use by the snapshot. If the database is accessed using the saved page table address, a fully consistent snapshot of the database can be seen, and that view will not change as long as we can guarantee that no pages referenced by the snapshot are freed.

Taking (and dropping) a snapshot is very efficient (from a few microseconds to a millisecond independent of database size). We can take an arbitrary number of snapshots at different times, and they will all be consistent as long as their pages are not freed. Since a shadow paging system never modifies accessible pages, postponing freeing of updated pages is sufficient to maintain the consistency of all snapshots. Later in this paper we will describe an efficient algorithm for deducing when a page can be freed in the presence of snapshots.

Snapshots can be used to implement consistent dumping of the database, as well as for implementing optimistic multiversion concurrency control. Read-only transactions can be implemented by taking a snapshot and reading from the snapshot.

The idea of snapshots is illustrated in figure 2.

9 Multi-Level Incremental Dumping

Many algorithms have been presented for on-the-fly dumping of log-based databases [25, 22, 11, 26, 8]. Most of the algorithms are based on taking a fuzzy dump of the database and dumping the log records of the changes that have been made during the dump. [22] describes an algorithm which can be used to directly take a consistent dump using locking (and correspondingly, possibly causing some transactions to be aborted).

Some of the earlier algorithms can support incremental dumping by having a bit in modified pages which is set whenever the page is modified [8] (this scheme can also be generalized to multi-level dumping). However, most of the existing approaches to incremental dumping require scanning the entire database even if only a few pages have changed. This is very costly for disk-based databases. [26] presents an algorithm where this is not necessary, but their algorithm is too space-consuming for general use.

With shadow paging, it is possible to obtain a transaction-consistent dump of the entire database taking a snapshot of the database and reading the database using the snapshot. No locking is needed for the dump.

Incremental dumping can be implemented by storing in each page table entry the serial number of the last commit batch that modified the page. The serial number can be maintained at almost no CPU overhead, but it increases the size of the page table. Using the serial numbers as timestamps, it is possible to dump only those pages that have changed after a certain point of time. Normally the reference time is the timestamp of the last full dump or the previous incremental dump (the timestamp of a dump is the serial number of the last commit that had been done before the snapshot for the dump was taken).

Incremental dumping with shadow paging is highly efficient because only the cached page table needs to be scanned, and only the pages that are actually dumped need to be read. Only the logical database is dumped, not the unused physical pages.

10 Multiple Database Versions

A snapshot can also be seen as a copy-on-write copy of the database. As long as freeing of pages used by other copies can be prevented, we can run arbitrary transactions against a snapshot, and even take a new snapshot of the modified version. In the following discussion we will call a snapshot a version of the database. Note that database versions differ from the more conventional object versions in that they are global [12]. Object versions can be used together with database versions if desired.

The versions of a database form a tree (figure 3). From the user's point of view, the versions are fully independent of each other. Transactions need only be synchronized against other transactions within the same version, any version can be dropped at any time, and new versions may be forked off any existing version. There is no need to consider some version the "primary" version of the database – it is fully feasible to drop the previous primary version and start using another version of the database as the primary one.

Multiple versions can be made permanent in the database. One possible implementation is to have an atomically updatable master pointer instead of a page table pointer. The master pointer would point to several page table pointer pages (remember, the page table pointer may also contain other data in addition to the actual address of the page table). The page table pointers would reside in normal shadowed database storage.

Multiple permanent versions could be very useful in many applications. Examples include large design databases, where several design directions could be considered simultaneously, poor ones dropped and new forks created as needed. An other application could be asking "what if" questions in a large database without

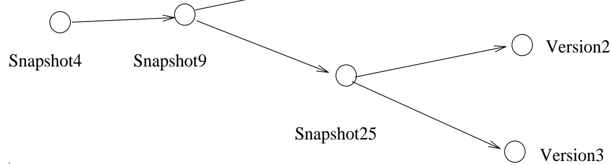

Figure 3: Versions of a database form a tree. Implicit snapshots are retained of all nodes with multiple children.

affecting the original database. In very large knowledge bases, multiple versions could be used to implement interesting searching algorithms. The possibilities for interesting applications are numerous and yet unexplored.

It has not been possible to support multiple independently updatable database versions efficiently with existing databases. With concurrent shadow paging, creating and dropping database versions are very light operations (in the microseconds to milliseconds range).

The algorithm for determining when a physical page can be freed is based on the fact that if a physical page is referenced in one version of the database, it can only occur in the same logical location in other versions. Moreover, if versions with children are never modified but instead a new version is (automatically) created when necessary, only the immediate parent and direct children need to be checked when freeing pages.

Dropping a version is in principle equivalent to freeing all of its pages. However, as long as the version has ancestors, only the modified pages can be without other references, and thus only the modified pages need to be checked. Unmodified snapshots never contain any pages of their own and thus dropping them never causes any pages to be freed unless all ancestors have been dropped.

11 Two-Phase Commit in Distributed Databases

A common method for implementing transactions in distributed databases is two-phase commit. Implementing two-phase commit requires that the changes made by a transaction can be saved so that they survive a possible crash but can still be undone. The algorithm presented here resembles that in [13, 17], but there are also substantial differences.

The basic idea is to store the incremental page table on disk in the first phase of commit, and leave a pointer to the incremental page table in the page table pointer. If a crash occurs, the incremental page table can be recovered from the copy on disk and the coordinator queried about the fate of the transaction. In the second phase of commit, the transaction is committed normally and the incremental page table on disk is freed (atomically, in the same commit batch). rently with processing of local transactions.

11.1 First Phase

When a node is requested to do the first phase of commit, all shared locks are released immediately. The incremental page table of the transaction and any modified pages are written to disk. The saved information must be sufficient to reconstruct the incremental page table and locks after a crash, and to identify the coordinator of the transaction and the transaction within the coordinator. During processing of the next commit batch, a pointer to the incremental page table on disk will be added to the page table pointer. When the page table pointer has been stored on disk, the system reports "ready" to the coordinator.

The algorithm can be extended to handle fine-granularity locking by storing also the modified versions of objects.

11.2 Second Phase

The second phase is executed like any transaction commit. The pointer to the incremental page table on disk is removed from the page table pointer during the commit and the disk space is freed.

If the coordinator decided to abort the transaction, the pointer to the incremental page table on disk is removed from the page table pointer during the next commit batch and the disk space is freed. Otherwise the abort is done normally.

11.3 Crash Recovery

During crash recovery, the system must read the incremental page tables of partially processed global transactions from disk. The pages mentioned in the incremental page tables must be counted as used when extracting the free list from the page table. All exclusive locks on logical pages modified by the transactions are restored, and the system must query the coordinator of each global transaction about the fate of the transaction. Normal transaction processing can begin when the locks held by recovered global transactions have been restored.

The coordinator of a global transaction must keep enough information to answer queries about the fate of the transaction, e.g. by keeping a list of global transactions that have been partially (phase one) committed but that have not yet been reported fully committed by all other machines. The list of active global transactions can be kept in the page table pointer, with extension to disk.

ging

Concurrent shadow paging does not use logging for crash recovery. However, this makes media recovery more important than in log-based systems where the log can be used to bring the system up to date from the last dump.

Shadow paging is very well suited for high concurrency environments with many relatively small disks. A very natural solution to media recovery is to use a RAID disk system [21, 27].

Shadow paging supports consistent multi-level incremental dumping without disturbing normal transaction processing. This allows dumping even a very large database quite frequently – for example, a full dump once a week, an incremental against that once a day, an incremental against the daily dump once an hour, and an incremental against the hourly dump every ten minutes.

Logs are sometimes used for other purposes besides recovery. Maintaining logs is possible also in a shadow paging system. The log can, for example, be implemented as a large object to which transactions append a log record before they change the value of a data item. The log resides in normal logical pages and can be manipulated by transactions like any other object. Appending of log records probably needs built-in support to avoid concurrency bottlenecks and to append commit and abort records. The tail of the log is flushed to disk as part of processing a commit batch, and the log is always in a consistent state with the database. Logging would cause very little overhead to transaction processing, as it only adds a little more data to the sequential write.

13 Main-Memory Databases

Concurrent shadow paging may also be interesting in main memory databases. Most current main memory databases use fuzzy dumps with some form of disk-based logging [8, 5, 14, 26]. With concurrent shadow paging, a main memory database can be seen as a database with a 100 percent cache hit rate. Since disk I/O in such an environment consists mostly of sequential writes, fairly high performance can be expected (as illustrated in table 3). Locks never need to be held during disk I/O, and if full persistence is not required, transactions can commit without waiting for any disk I/O [4] (section 6).

An advantage in using shadow paging instead of log-based approaches is faster recovery time. Since the main memory database is only a large cache, the system is operational as soon as the software has been started and the free list extracted. It is not necessary to wait for the database to be loaded into memory and log entries to be processed; instead, transaction processing can be started immediately (although at reduced performance), reaching full performance levels when the entire database is in memory. Similarily, the system would not be significantly affected if the database was a little larger than the memory. This would make the system more robust in the face of a growing database or part of the memory being temporarily offline due to a hardware failure. Interesting performance tradeoffs might also be seen in applications where a fairly large part of the database is accessed very frequently, but then there is a vast body of bulk data (images, text etc.) that is accessed less frequently. In such an application, transactions to the frequently accessed part would be done at main memory speeds, still allowing easy access to the parts of the database that cannot be kept in main memory.

N_{psz}	page size
N_{ptesz}	page table entry size
N_{ptp}	$= \frac{N_p N_{ptesz}}{N_{psz}}$ number of page table pages
N_d	number of disks
U_{io}	I/O time utilization factor
t_{aio}	$= N_d U_{io}$ available I/O time
t_s	average seek time (including latency)
N_{xps}	disk transfer rate bytes/sec
t_x	$= \frac{N_{psz}}{N_{xps}}$ page transfer time (read or write)
p_{ch}	cache hit probability in reads
N_{tps}	transactions per second
N_{cps}	commit batches per second
N_{tpc}	$= \frac{N_{tps}}{N_{cps}}$ transactions per commit batch
N_{ur}	user read requests per transaction
N_{uw}	user write requests per transaction
N_{ptw}	page table writes per commit batch
t_c	average I/O time per commit batch
t_t	average I/O time per transaction

Table 1: Notations used in the performance analysis.

14 Performance

Using the notations in table 1 and assuming uniform distribution of writes in the logical database, the number of page table pages written in a commit batch is

$$N_{ptw} = N_{ptp} - N_{ptp}(1 - \frac{1}{N_{ptp}})^{N_{tpc}N_{uw}}$$

where the latter term corresponds to the probability that a user write already had its page table page written earlier in the same batch.

In addition to page table writes, a commit batch writes the modified data pages and the root pointer. We assume that all user data writes access separate pages. If a sufficiently large contiguous region of free space can be found, only two seeks will be needed (one for the data and page tables, and one for the page table pointer which is written twice to implement atomic write). Thus, the I/O time needed to execute a com-

$$t_c = t_s + (N_{tpc} N_{uw} + N_{ptw}) t_x + t_s + 2t_x$$

In addition to the commit, a transaction also uses I/O resources for reading. Some user reads are satisfied from the cache and some result in physical reads. The cost of the commit batch is divided among all transactions participating in the batch. Thus, the average I/O time per transaction is

$$t_t = (1 - p_{ch}) N_{ur} (t_s + t_x) + \frac{t_c}{N_{tpc}}$$

Assuming the transactions per second rate is only limited by I/O time,

$$N_{tps} = \frac{t_{aio}}{t_t} \quad (1)$$

Denoting N_{tps} with x and using the constants

$$\begin{aligned} a_1 &= N_{cps} N_{ptp} t_x + 2N_{cps}(t_s + t_x) - N_d U_{io} \\ a_2 &= (1 - p_{ch}) N_{ur} (t_s + t_x) + N_{uw} t_x \\ a_3 &= -N_{cps} N_{ptp} t_x \\ a_4 &= (1 - \frac{1}{N_{ptp}})^{\frac{N_{uw}}{N_{cps}}} \end{aligned}$$

we can write the equation as

$$a_1 + a_2 x + a_3 a_4^x = 0 \quad (2)$$

This can be solved by elementary numerical methods. Since $0 < a_4 < 1$ and $|a_3| \ll |a_1|$ in all practical cases, the equation is close to linear, and the performance of the system scales quite linearly when more disks are added.

Table 2 displays N_{tps} in a disk-based database application for different numbers of disks and various cache hit rates. Small transactions (two reads, two writes) were analyzed ($U_{io} = 0.9$, $t_s = 0.015$, $N_{psz} = 4096$, $N_{xps} = 3000000$, $N_{cps} = 5$, $N_{ur} = 2$, $N_{uw} = 2$).

The behaviour at high cache hit rates is very interesting. This is illustrated in table 3 which shows N_{tps} for various page sizes and numbers of disks ($U_{io} = 0.9$, $t_s = 0.015$, $N_{xps} = 3000000$, $N_{cps} = 5$, $N_{ur} = 2$, $N_{uw} = 2$, $p_{ch} = 1.0$). The figures for higher numbers of disks are mostly theoretical, as performance would be limited by available CPU time, which is not considered in our current model. However, concurrent shadow paging does scale very nicely to shared memory multiprocessors, so some of the figures may not be as unrealistic as they first appear.

p_{ch}	$N_d = 1$	5	10	100
0.00	19	114	234	2488
0.25	25	146	300	3234
0.50	34	202	420	4619
0.60	40	240	500	5575
0.70	49	295	620	7029
0.80	62	384	821	9509
0.90	86	557	1235	14692
0.95	107	730	1673	20198
1.00	143	1088	2647	32303

Table 2: TPS rate for different numbers of disks and cache hit rates.

N_{psz}	$N_d = 1$	5	10	100
512.00	1104	6643	14121	225185
1024.00	555	3463	7683	122074
2048.00	280	1880	4460	63419
4096.00	143	1088	2647	32303
8192.00	74	644	1466	16298

Table 3: TPS rate for different numbers of disks and page sizes with 100 percent cache hit rate (a main-memory database).

15 Conclusion and Further Research

We have presented several new ideas which significantly improve both performance and applicability of shadow paging in general-purpose databases. Based on our preliminary research, shadow paging seems to offer everything that log-based approaches offer. It also appears to be potentially much faster than any update-in-place method. In many applications, shadow paging is much more flexible than log-based approaches. Multiple database versions and other new ideas may find interesting applications on many fields. We feel that concurrent shadow paging definitely deserves more research, and has a good chance of improving the performance of many database applications by a factor of two or more.

The work on concurrent shadow paging is just beginning, and all results in this paper are preliminary. Much theoretical and experimental work is still needed to substantiate many of the expectations in this paper. We have not yet fully analyzed how the different ideas interact with each other. The ideas must be refined, analyzed and implemented, and no doubt many new ideas will come up.

We are in the process of implementing concurrent shadow paging. We have implemented a very simple prototype, and are in the process of building a more complete system. We are implementing the new ideas in practice, and will evaluate their performance and compare them to log-based approaches both experimentally and analytically.

Acknowledgements

Many of the ideas presented in this paper have arisen as a result of discussions with Heikki Suon-

sanen, Heikki Saikkonen, and Eljas Soisalon-Soininen. I thank them for many fruitful discussions, suggestions, and comments.

References

[1] M. E. Adima and B. G. Lindsay. Database snapshots. In *Very Large Data Bases*, pages 86–91, 1980.

[2] R. Agrawal and D. J. DeWitt. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Transactions on Database Systems*, 10(4):529–564, 1985.

[3] R. Agrawal and D. J. DeWitt. Recovery architectures for multiprocessor database machines. In *ACM PODS*, pages 131–145, 1985.

[4] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory databases. In *ACM SIGMOD*, pages 1–8, 1984.

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

[6] 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. *ACM Computing Surveys*, 13(2):223–242, 1981.

[7] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. 11th ACM Symposium on Operating System Principles*, pages 155–162, 1987. Published as ACM Operating Systems Review, Vol. 21, No. 5, 1987.

[8] R. B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computers*, C-35(9):839–843, 1986.

[9] J. Kent, H. Garcia-Molina, and J. Chung. An experimental evaluation of crash recovery mechanisms. In *ACM PODS*, pages 113–121, 1985.

[10] J. M. Kent. *Performance and Implementation Issues in Database Crash Recovery*. PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, 1985.

[11] R. P. King, N. Halim, H. Garcia-Molina, and C. A. Polyzois. Management of a remote backup copy for disaster maintenance. *ACM Transactions on Database Systems*, 16(2):338–368, 1991.

[12] P. Klahold, G. Schlageter, and W. Wilkes. A general model for version management in databases. In *Very Large Data Bases*, pages 319–327, 1986.

[13] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Computer Science Lab., Xerox Palo Alto Research Center, Apr. 1979.

gorithm for a high-performance memory-resident database system. In *ACM SIGMOD*, pages 104–117, 1987.

[15] C.-C. Liu and T. Minoura. Effect of update merging on reliable storage performance. In *IEEE Data Engineering*, pages 208–213, 1986.

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

[17] D. A. Menasće and O. E. Landes. On the design of a reliable storage component for distributed database management systems. In *Very Large Databases*, pages 365–375, 1980.

[18] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[19] J. E. B. Moss, B. Leban, and P. K. Chrysanthis. Finer grained concurrency for the database cache. In *IEEE Data Engineering*, pages 96–103, 1987.

[20] 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.

[21] 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.

[22] C. Pu. On-the-fly, incremental, consistent reading of entire databases. In *Very Large Databases*, pages 369–375, 1985.

[23] D. R. Ries and M. Stonebraker. Effects of locking granularity in a database management system. *ACM Transactions on Database Systems*, 2(3):233–246, 1977.

[24] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proc. 13th ACM Symposium on Operating System Principles*, pages 1–15, 1991. Published as ACM Operating Systems Review, Vol. 25, No. 5, 1991.

[25] D. J. Rosenkrantz. Dynamic database dumping. In *ACM SIGMOD*, pages 3–8, 1978.

[26] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *IEEE Data Engineering*, pages 452–462, 1989.

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

[28] J. S. M. Verhofstad. Recovery techniques for database systems. *ACM Computing Surveys*, 10(2):167–195, 1978.