Write Optimizations and Clustering in Concurrent Shadow Paging[*]

Tatu Ylönen

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

Abstract

Shadow paging is a method for implementing atomic transactions in databases. In previous studies it has been found to perform rather poorly. However, many of the reasons that originally hampered its performance have become obsolete due to technological development.

Several ideas are presented in this paper which considerably improve the performance of shadow paging. The page table allows all writes to be done sequentially, resulting in very significant speedups, and variable-size pages offer a solution to clustering problems.

A simple performance analysis indicates that in many common applications the performance achieved using these optimizations can be twice or more the performance of log-based approaches.

1 Introduction

1.1 Background

The original idea of shadow paging was presented by Lorie [7]. 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 [3]. The implementors of System R afterwards considered shadow paging inappropriate for large multi-user systems [3, pp. 239–240].

A simpler multi-user version of shadow paging was presented in [1]. However, in comparison with log-based methods and differential files, its performance was found inferior [1, 2]. The primary overhead turned out to be updating and indirection through the page table and the root pointer.

In [4, 5], the overhead due to root pointer updates and page table I/O was reduced by committing several transactions simultaneously as a batch. 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 [9], and there has been little work on shadow paging for several years.

1.2 The Effect of Increased Memory Sizes

In the last few years, main memory sizes and correspondingly cache sizes have increased dramatically, and this development is expected to continue in the near future. Assuming the database contains 4 kB pages and the size of a page table entry is 4 bytes, the size of the page table is 100 kB for a 100 MB database and 10 MB for a 10 GB database. This means that the entire page table can be kept in the cache, and that mapping a page number using the page table, which used to require a disk access, now only requires a few machine instructions.

1.3 Concurrent Shadow Paging

The database consists of a number of disk blocks organized as pages. Each page can hold a fixed number of bytes, and is identified by a number from which its address on disk can be computed. These pages will be called *physical* because they have a direct representation on disk.

The levels of the database system above the transaction manager also see the database as a collection of numbered pages. These will be called *logical* pages to distinguish them from physical pages. High level data structures, such as those used to implement tables, only refer to logical pages. The relation between logical pages and physical pages is hidden from the higher levels, and is implementation dependent. In most log-based databases, there is a fixed one-to-one correspondence between logical

[*]This research was supported in part by TEKES grant 4067/93.

and physical pages. With shadow paging, however, the mapping is not fixed but instead continually changes as the database is modified.

The mapping between logical and physical pages is maintained using a *page table*. Conceptually it is an array of physical page numbers indexed by the logical page number. There is always a valid page table in nonvolatile storage on disk.

Earlier implementations stored the page table in a fixed location in stable storage, and used logs or bitmaps to implement atomic modifications to the page table [3, 7]. This makes the implementation of concurrent transactions difficult and requires the use of logs on page table pages. The implementors of System R concluded that the use of this method for large files was a mistake. This algorithm still is the only one presented in many textbooks on databases.

Another implementation was to use *intention lists* [1, 6, 8]. The idea is to store the changes made by each transaction in a per-transaction *incremental page table*. At commit time, the incremental page table is first written to stable storage as a *precommit record*, and only then are changes made to the actual page table. If the system crashes in the middle of updating the page table, the precommit record can be used to redo and complete the changes. In a detailed study this method was found to perform very poorly for small transactions with a random access pattern [1].

A third alternative is to have the page table itself in shadowed storage. With this method, the database has a *page table pointer* in a fixed location in stable storage. It contains the address of the page table on disk. When a transaction modifies the database, it constructs a new page table (without modifying any of the existing page table pages), writes the new page table to disk, and commits by atomically writing the address of the new page table to the page table pointer. In most cases the page table is implemented as a tree-like structure (figure 1), and thus only modified portions of the page table need to be recreated.

Kent [5] used this method in his thesis. In his system, every transaction has an incremental page table which is used to store the changes made by the transaction. The incremental page table is implemented as a list of tuples $\langle L, P_{old}, P_{new} \rangle$, where L is the logical page number, P_{old} is the old physical page number, and P_{new} is the new physical page number. P_{old} corresponds to the global version of the page, and P_{new} corresponds to the local version of the page.

Two-phase locking on logical pages is used for concurrency control. To read a page, the transaction first acquires a shared lock on the logical page. It then looks for a corresponding entry in its incremental page table, and if not found, maps the page using the global page table. To write a page, a transaction must obtain an exclusive lock on the logical page. It then checks if it already has an entry for the page in its incremental page table. If so, it uses the already-existing local version of the page. If not, it creates a copy of the page, and adds a corresponding entry to its incremental page table. Locking can be optimized by checking the incremental page table first, as an entry can exist there only if the corresponding page is already exclusively locked.

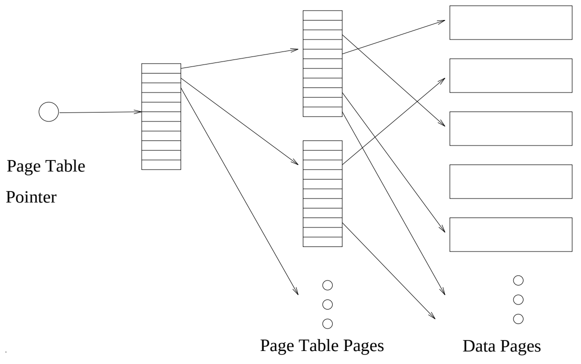

Figure 1: The shadow paging file structure.

Since a transaction must have an exclusive lock before it can modify a page, only one transaction can have a local version of any given logical page and there can be no conflicts between the incremental page tables of different transactions. This means that it is possible to install the modifications of several transactions into the global page table at once, reducing the overhead due to constructing new page tables. This can be implemented by putting all partially committed transactions in a queue, and having a separate process periodically (a few times a second) take all transactions in the queue and install their modifications (local versions) to the page table. This is called *commit batching*.

Incremental page tables and commit batching significantly decrease the cost of updating the page table. The cost of each page table update is divided among several transactions, and the per-transaction overhead decreases as the multiprocessing level increases. Also, since the page table is implemented as a tree of pages, only those parts of the page table that are actually modified (and their parent nodes) need to be rewritten; other parts of the page table can be shared with previous versions.

No garbage collection is needed with this method. After a transaction has committed, it can free the “old” versions of the pages (including page table pages) that it has modified. If a transaction needs to be aborted, all that has to be done is to free the “new” versions in its incremental page table.

No modifications need to be done to the global database in this case.

The free list (or bitmap) of physical pages can either be extracted from the page table on disk at startup time, or be stored on disk at every commit. It is possible to compute the permanent free list and write it to disk at every commit batch. Because of the write optimizations of section 2, it would not cause significant I/O overhead. However, the CPU overhead usually makes it undesirable to compute and store the physical free list at every commit. In most applications extracting the free list from the page table at startup time is preferable [5].

2 Write Optimizations

In most database systems, the mapping between logical and physical pages is fixed. A fixed mapping has the advantage that maintaining a page table is not required, and it allows efficient clustering of related logical pages. On the other hand, in systems with a fixed mapping, each modified data page must eventually be written back to its original location. Since modifications are usually scattered throughout the entire database, each modification will typically require a seek to the desired location on disk. This need is not removed by delaying or batching writes (except in hot spots), since the database is usually very large compared to the number of pages modified, and thus the density of writes is still low.

Shadow paging allows the database system to choose where to write a modified page. The system can choose to write all modified pages in a contiguous area on disk, or just near each other. This allows many pages to be written with a single seek.

A similar idea is used in log-based file systems [10, 12, 13], where long writes are used to improve write performance.

2.1 Analysis

A typical low-cost disk currently has about 15 ms average access time (t_a) and 4 MB/s sustained data transfer rate (R). The time required to transfer a page of data $t_x = \frac{S}{R}$, where S is the size of the page in bytes. If a seek is required, the time to read or write S bytes is $t_a + \frac{S}{R}$. If many adjacent pages are read or written, no physical seek is needed except for the first page. Similarly, if one page is read or written, the next is skipped, and the next written, no seek is needed for the latter page, but extra transfer (rotational latency) time must be counted for the skipped page.

When a seek is required for each page, the time required to transfer N pages of size S is $N(t_a + \frac{S}{R})$. When a seek is required for the first page only, the time is $t_a + N\frac{S}{R}$. The maximum speedup factor F that can be achieved by doing the writes sequentially is

$$F = \frac{N(t_a + \frac{S}{R})}{t_a + N\frac{S}{R}}. \tag{1}$$

As $N \to \infty$, the limit on speedup

$$F_\infty = \lim_{N\to\infty} F = 1 + t_a\frac{R}{S}. \tag{2}$$

Figure 2 shows the speedup factor F for various page sizes and write lengths for a typical low-cost disk ($t_a = 15\,\text{ms}$ and $R = 4\,\text{MB/s}$). Table 1 shows the limit of potential speedup when $N \to \infty$.

The maximum speedup can be achieved if a sufficiently large contiguous area can be found in the database. This is often not the case in practice, and the part of the database with the highest density of free pages should be used for writing. The pages which must be skipped between writes must be counted in the transfer time. This can be represented by a skip factor s, which is the amount by which the transfer time must be multiplied. If the area is contiguous, the skip factor is 1. If 25 percent of disk space is free and it is evenly distributed (the worst case), $s = 4$. Even now, the maximum speedup would be by a factor of 31 for 512 byte pages and by a factor of 4 for 4096 byte pages.

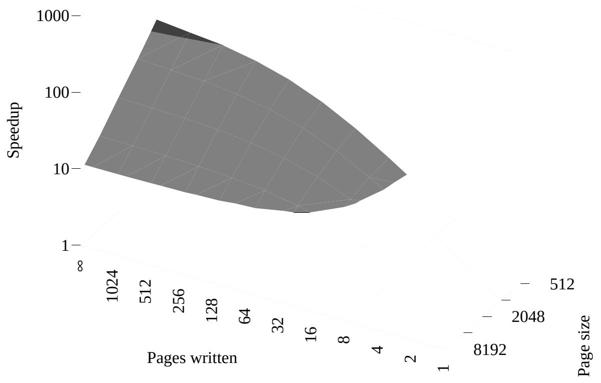

Figure 2: Speedup factor F for different page sizes S and transfer lengths N.

S	512	1024	2048	4096	8192
F_∞	124	62	32	16	9

Table 1: Limit of potential speedup F as $N \to \infty$ for different page sizes S.

2.2 Implementation

The idea in shadow paging is to allocate a new physical page whenever a page is modified. In concurrent shadow paging, the new page number is kept in the incremental page table until the transaction is committed. The new page number is internal to the transaction until the transaction commits and its changes are made visible to other transactions.

Allocation of physical page numbers can be delayed by introducing the concept of *virtual page numbers* (note that these have nothing to do with logical page numbers). A virtual page number is an identifier for a page known by the transaction and the cache manager. The transaction may access a page using the virtual page number, and the cache will return the appropriate page. Virtual pages do not usually have a physical page associated with them; they only exist in the cache.

A virtual page number is allocated whenever a page is first modified. Typically this is done by specifying an existing page number to the cache manager. The cache manager will then allocate a unique virtual page number and rename the page in its memory to use the virtual number. The old physical number is effectively removed from the cache. Shadow paging guarantees that the old page in the cache will not be dirty, and unless snapshots [16] are used, no-one will ever access the old page except in the case that the current transaction is aborted.

Virtual page numbers cannot be stored in the page table, as they are only temporary identifiers in volatile storage. Before a transaction can commit, actual physical pages must be allocated for its virtual pages (this is called the *realization* of the virtual pages). Since many transactions are usually committed as a batch, a large number of modifications can be combined, and all virtual pages of the committing transactions can be realized at the same time. The system can now allocate contiguous disk pages and write them sequentially. This means that close to the maximum speedup can be achieved in practice, especially in high-concurrency environments where a large number of transactions end up in the same batch.

Modification of page table pages is also done using virtual pages. When a commit batch is being processed, virtual pages are created for all page table pages that are going to be modified. The virtual page table pages are then realized together with modified data pages, and the actual physical page numbers are stored in the page table pages. Finally, both modified data and page table pages are flushed to disk, and when they have all been written, the page table pointer is updated atomically to point to the new page table.

It is also possible to allocate the new physical pages on multiple disks, analogously to the two-dimensional file system of [14]. Space allocation problems are easier with shadow paging, as it is possible to skip used pages if a sufficiently large contiguous area cannot be found.

The cache is allowed to allocate physical pages for virtual pages even before realization. This is desirable if there are large update transactions which create very many virtual pages. In such cases the cache can make long enough writes to achieve most of the maximum speedup. Transactions will still access those pages with the virtual page number, and the new physical page numbers will be internal to the cache until the virtual pages are realized (for such pages, no pages are allocated during realization as they already have a physical page). The virtual page number can be released after the transaction holding the number has terminated.

It should be noted that this optimization makes conventional clustering between logical pages impossible, as no attempt is made to keep the logical-to-physical mapping linear. An alternate approach to clustering is presented in section 4.

3 Performance

In this section a model is developed for the performance of concurrent shadow paging with the write optimizations of this paper. The model is based on analyzing the I/O time (that is, disk seek/transfer time) required for each operation. A steady state solution is computed using the model. It is assumed that performance is only limited by I/O resources. Table 2 summarizes the notations used.

The first write processed in a commit batch will always result in a page table modification. Assuming uniform distribution of writes, the second write will cause a page table modification unless it is on the same page as the first one (on the average, this results in $(1 - \frac{1}{N_{ptp}})$ writes), and so on. The total number of data pages written per commit batch is $N_{tpc}N_{uw}$. This leads to the geometric series

$$N_{ptw} = \sum_{i=0}^{N_{tpc}N_{uw}-1} (1 - \frac{1}{N_{ptp}})^i$$

for the number of page table pages written. The sum of this series is

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

In addition to leaf-level page table pages, a commit batch writes a second-level page table page, the modified data pages, and the root pointer. All user data writes are assumed to access separate pages

N_p	number of pages in the database
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_a	average access time (including latency)
R	disk transfer rate bytes/sec
t_x	$= \frac{N_{psz}}{R}$ page transfer time (read/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_r	average I/O time for reads per txn
t_t	average I/O time per transaction
s	skip factor in writes

Table 2: Notations used in the performance analysis.

(the worst case). Only two physical seeks will be needed for the whole commit batch (one for the data and page tables, and one for the page table pointer which is written twice to implement atomic write). The number of pages written in the batch is $N_{tpc}N_{uw} + N_{ptw} + 1$, plus the page table pointer write. The skip factor multiplies the normal writes. The following formula gives the I/O time used for processing a commit batch:

$$t_c = 2t_a + (N_{tpc}N_{uw} + N_{ptw} + 1)st_x + 2t_x.$$

Besides commit processing and writes, a transaction also uses I/O resources for reading. Some user reads are satisfied from the cache and some result in physical reads. The number of physical reads per transaction is $(1-p_{ch})N_{ur}$, and each read typically requires a separate seek. Thus,

$$t_r = (1 - p_{ch})N_{ur}(t_a + t_x).$$

The cost of the commit batch is divided among all transactions participating in the batch. Thus, the average I/O time per transaction is

$$t_t = t_r + \frac{t_c}{N_{tpc}}.$$

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

$$N_{tps} = \frac{t_{aio}}{t_t}. \qquad (3)$$

p_{ch}	$N_d = 1$	3	10	30
0.00	23	74	254	781
0.25	30	96	331	1021
0.50	42	136	476	1476
0.75	72	236	853	2662
0.90	125	436	1645	5141
0.95	168	620	2385	7454
1.00	266	1112	4336	13552

Table 3: TPS rate for different numbers of disks and cache hit rates for transactions making two reads and two writes (shadow paging).

Denoting N_{tps} with x and using the constants

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

this equation can be written as

$$a_1 + a_2x + a_3a_4^x = 0. \qquad (4)$$

This can be solved using 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 3 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 for a 1 GB database ($U_{io} = 0.9$, $t_a = 0.015$, $N_{psz} = 4096$, $R = 4\,\mathrm{MB/s}$, $N_{cps} = 2$, $N_{ur} = 2$, $N_{uw} = 2$). It was assumed that a sufficiently large contiguous area can always be found ($s = 1$).

The behaviour at high cache hit rates is interesting. This is illustrated in table 4 which shows N_{tps} for various page sizes and numbers of disks in a main-memory environment ($U_{io} = 0.9$, $t_a = 0.015$, $R = 4\,\mathrm{MB/s}$, $N_{cps} = 2$, $N_{ur} = 2$, $N_{uw} = 2$, $p_{ch} = 1.0$, $s = 1$). The figures for higher numbers of disks are mostly theoretical, as performance would be limited by the available CPU time, which is not considered the current model.

Table 5 illustrates how the performance changes when the skip factor is greater than one ($N_d = 1$, $N_{psz} = 512$, $U_{io} = 0.9$, $t_a = 0.015$, $R = 4\,\mathrm{MB/s}$, $N_{cps} = 2$, $N_{ur} = 2$, $N_{uw} = 2$).

N_{psz}	$N_d = 1$	3	10	30
512.00	1768	5910	24433	95114
1024.00	909	3226	14575	51355
2048.00	480	1879	8198	26630
4096.00	266	1112	4336	13552
8192.00	157	613	2226	6834

Table 4: TPS rate for different numbers of disks and page sizes with 100 percent cache hit rate (a main-memory database) for transactions making two reads and two writes.

p_{ch}	$s = 1$	2	4	8
0.00	27	27	26	25
0.25	36	35	34	32
0.50	54	52	49	44
0.75	104	98	88	73
1.00	1768	871	432	215

Table 5: The effect of the skip factor on TPS rate (one disk, 512 byte pages).

3.1 Comparison with Log-Based Systems

A similar model can be constructed for a log-based system with a fixed logical-to-physical mapping. It is assumed that maintaining the log will not require any I/O resources (in practice there is a certain overhead associated with logging). Furthermore, it is assumed that a significant portion of transactions access random locations in the database; in practice some writes could be avoided in hot spots (for reads, the effect of caching is the same for both shadow paging and logs).

Using the same notations as before, the I/O time required by a transaction can be computed as follows. A read is satisfied from the cache with the probability p_{ch}, and thus the average number of physical reads per transaction is $(1-p_{ch})N_{ur}$. Since each read typically refers to a random location, a seek is required for each read. The I/O time required for reads per transaction is

$$t_r = (1 - p_{ch})N_{ur}(t_a + t_x).$$

All writes must eventually be done to their original location on disk, and this will usually require a seek since the density of writes compared to the size of the database is low. Thus, ignoring the effects of multiple writes to a hot spot page, the I/O time required for doing the writes t_w is

$$t_w = N_{uw}(t_a + t_x).$$

p_{ch}	$N_d = 1$	3	10	30
0.00	14	42	141	422
0.25	16	48	161	483
0.50	19	56	188	563
0.75	23	68	225	676
1.00	28	84	282	845

Table 6: TPS rate for different numbers of disks and cache hit rates for transactions making two reads and two writes (fixed mapping).

Each transaction requires $t_r + t_w$ time. Thus,

$$t_t = (1 - p_{ch})N_{ur}(t_a + t_x) + N_{uw}(t_a + t_x).$$

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

$$N_{tps} = \frac{t_{aio}}{t_t}. \tag{5}$$

For comparison, results computed using this formula and the same parameters as table 3 are shown in table 6. It is easy to see that, with these parameters and a low cache hit rate, shadow paging provides almost twice the performance of the system with a fixed mapping, and with a high cache hit rate almost ten times the performance.

If hot spots were considered, they would reduce the number of writes in a log-based system. If 20 percent of all writes were to hot spots, the total I/O time would be reduced by somewhat less than 20 percent. This is greatly outweighted by the overall speedup. If almost all writes go to hot spots, this analysis is not valid.

The effect of seek length was not considered in this analysis. However, it would not change the results very much, since the rotational latency already forms a significant portion of the average access time.

4 Clustering

The optimizations in section 2 prevent conventional clustering of related logical pages. Because it is desirable to be able to write a page to a new location without constraints, earlier clustering algorithms for shadow paging which attempted to keep the logical-to-physical mapping approximately linear [5, 7, 11] are not applicable. The problems are at worst when reading large multi-megabyte objects or scanning a large table.

The time required to read an object (or table) of S bytes in N_{parts} parts is

$$t(S, N_{parts}) = \lceil \frac{S}{N_{psz}} \rceil t_x + N_{parts} t_a.$$

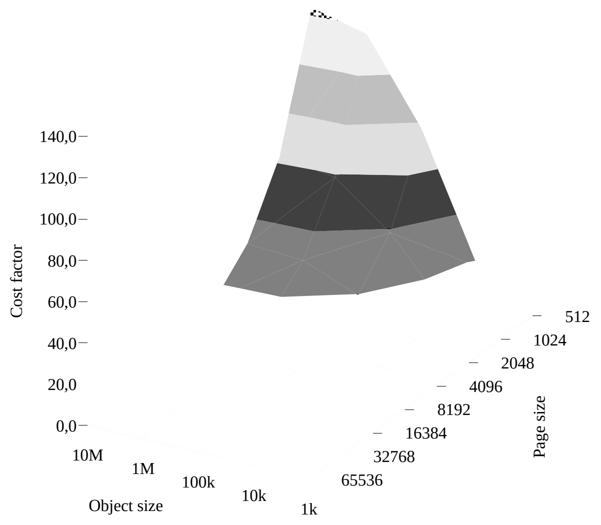

Figure 3: Worst-case fragmentation cost factor C for different page sizes N_{psz} and object sizes S.

It is now assumed that in systems with a fixed mapping the object can always be stored as a contiguous region ($N_{parts} = 1$), although this optimal case is not always possible in practice due to free space fragmentation. It is also assumed that each page of the object in a shadow paging system is stored in a separate location ($N_{parts} = \lceil \frac{S}{N_{psz}} \rceil$), which is the worst case.

The worst-case fragmentation cost factor C can be defined as

$$C = \frac{t(S, \lceil \frac{S}{N_{psz}} \rceil)}{t(S,1)}. \qquad (6)$$

This expresses the worst-case overhead due to fragmentation, with the value 1 meaning no overhead, value 2 meaning that I/O time is doubled, etc. Values for the fragmentation cost factor are shown in figure 3. In practice the cost factors are smaller, partly because it becomes impossible due to free space fragmentation to find sufficiently large contiguous free areas to store the object in one piece, partly because there might be locality in a shadow paging system due to correlation in update times, and partly because it is often more efficient to store a large object divided on several disks instead of just one contiguous region on one disk.

It can be seen that fragmentation can be helped by increasing the page size. On the other hand, as seen from table 1, the write optimizations become less effective when the page size increases. Also, since the entire page needs to be written to a new location if any part of it changes, transfer time increases with page size.

The solution is to use variable-size pages (multiples of the basic block size). The size of each logical page can be stored in the page table where it is easily and efficiently accessible whenever needed. The basic page size should be chosen as small as possible to make small updates as fast as possible where clustering is not needed, and larger pages should be used for storing large objects or tables which are often accessed sequentially. Variable-size pages allow tuning the sequential read versus small update performance tradeoff on a per-object basis to best suit the needs of each application.

Page size is optimal when the average cost of an access is minimal. To express this formally, let $p_r(S)$ be the probability that a particular access is a read of size S and $p_w(S)$ be the probability that the access is a write of size S. The probability that the access is a read is $p_r = \sum_{S=0}^{\infty} p_r(S)$, and the probability that it is a write is $p_w = \sum_{S=0}^{\infty} p_w(S)$. By definition, $p_r + p_w = 1$.

The cost of a read in terms of I/O time is

$$c_r(S, N_{psz}) = \lceil \frac{S}{N_{psz}} \rceil (t_a + \frac{N_{psz}}{R}).$$

Writes are cheaper due to batching of writes. Using the skip factor s, the cost of a write (ignoring the initial seek as it is divided between many writes) is

$$c_w(S, N_{psz}) = \lceil \frac{S}{N_{psz}} \rceil s \frac{N_{psz}}{R}.$$

The average cost of an arbitrary I/O operation is

$$\begin{aligned} c_{io}(N_{psz}) &= \sum_{S=0}^{\infty} p_r(S) c_r(S, N_{psz}) + \\ & \sum_{S=0}^{\infty} p_w(S) c_w(S, N_{psz}). \qquad (7) \end{aligned}$$

The optimal value for the page size is the value at which $c_{io}(N_{psz})$ is minimum.

5 Conclusion and Further Research

This paper began with a review of earlier results on concurrent shadow paging, and pointed out that technological development has removed the major obstacles that have caused shadow paging to perform poorly. It was then shown how writes can be speeded up by an order of magnitude or even two orders of magnitude by using the flexibility offered by the page table to make all writes sequential, and described a clustering method using variable-size pages which is compatible with the write optimizations.

The performance comparison in section 3.1, although simplified, indicates that at low cache hit rates shadow paging could have twice the performance of current log-based systems, and at high cache hit rates as much as ten times the performance of current systems. This definitely deserves further research.

Recovery is trivial in the version of shadow paging used in this paper, and this may be even more important than performance. Recovery code currently forms a large and hard-to-debug part of log-based database systems. It is much easier to implement and debug a recovery system based on shadow paging.

Topics for current and future research include snapshots and on-the-fly multi-level incremental dumping [15, 16], fine-granularity locking [15], early releasing of locks [15], two-phase commit in distributed databases [15], automatic I/O load balancing between disks [15], use in main-memory databases [15], implementation of savepoints and nested transactions, algorithms for efficient variable-length object allocation, and index management. In addition, work is underway on formal proofs of some aspects of the system and on prototype implementation and performance evaluation.

Acknowledgements

Many of the ideas presented in this paper have arisen as a result of discussions with Heikki Suonsivu, Tero Kivinen, Johannes Helander, Kenneth Oksanen, Heikki Saikkonen, and Eljas Soisalon-Soininen. I thank them for many fruitful discussions, suggestions, and comments.

References

[1] 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.

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

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] A. Reuter. A fast transaction-oriented logging scheme for UNDO recovery. *IEEE Transactions on Software Engineering*, SE-6(4):348–356, 1980.

[12] 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.

[13] M. Seltzer and M. Stonebraker. Transaction support in read optimized and write optimized file systems. In *Very Large Data Bases*, pages 174–185, 1990.

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

[15] T. Ylönen. Concurrent shadow paging: A new direction for database research. Technical Report 1992/TKO-B86, Helsinki University of Technology, Finland, 1992.

[16] T. Ylönen. Snapshots, read-only transactions and on-the-fly multi-level incremental dumping with concurrent shadow paging. Technical report, Helsinki University of Technology, 1993.