Title: Bounded Locking for Optimistic Concurrency Control

Paper# AMERICA119

Shirish Phatak and B. R. Badrinath
(First author is the contact author)

Computer Science, Frelinghuysen Road,
Busch Campus, Rutgers University
Piscataway, NJ-08855
USA

Email: phatak@cs.rutgers.edu

Phone: 732-748-9021

# Bounded Locking for Optimistic Concurrency Control[1]

Shirish Hemant Phatak B. R. Badrinath
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
*e-mail:{phatak, badri}@cs.rutgers.edu*

**Abstract**

Optimistic methods of concurrency control are gaining popularity. This is especially true with the resurgence of mobile and distributed databases, which inherently rely on optimistic techniques to improve availability and performance of the database system. A key problem with optimistic techniques is that they do not perform well in highly conflict prone environments. Pessimistic techniques, especially locking, perform much better under these circumstances.

In this paper we explore a hybrid technique that provides locking for high conflict data items and optimistic access for the rest. While hybrid techniques have been proposed in earlier literature, our technique is unique in that it is self tuning and does not require the transaction manager, the transaction or the user to incorporate any additional knowledge or to specify which data items or transactions are optimistic. Rather, the system uses an LRU data structure called *lock buffer* to maintain an optimal level of locks in the system. This data structure enhances the performance of the basic optimistic model by automatically providing locking for highly conflict prone data. A unique feature of our algorithm is that locks may be evicted from the lock buffer in an LRU fashion, if the number of data items for which locks are requested exceeds the size of the lock buffer. All transactions affected by such an eviction of locks automatically "become" optimistic with respect to the evicted data items.

## 1 Introduction

Optimistic concurrency control was proposed as an alternative to locking by Kung and Robinson in [9]. This approach to concurrency control was designed to overcome the shortcomings of locking based protocols. In particular, for systems that have relatively low rates of conflicts, locking produces an unnecessary overhead, and can lead to loss of throughput. Furthermore, even in systems with relatively high conflict rates, locking can unnecessarily delay read-only transactions. Finally, incremental locking leads to deadlocks, which also causes loss of throughput.

In spite of these problems with locking, problems with implementation and high rollback rates have made optimistic concurrency control an unattractive alternative. In recent years, though, optimistic techniques have seen a resurgence, especially with the increasing popularity of distributed

[1]This research work was supported in part by DARPA under contract numbers DAAH04-95-1-0596 and DAAG55-97-1-0322, NSF grant numbers CCR 95-09620, IRIS 95-09816 and Sponsors of WINLAB.

and mobile databases. These systems rely on optimism to improve overall availability and performance of the database. Unfortunately, the key problem, i.e., the sensitivity of optimistic methods to conflict rates, remains unsolved to a large extent. High rates of conflicts cause a large number of rolled back transactions, which in turn causes loss of throughput and a significant waste of system resources. Locking, on the other hand, behaves more gracefully in the presence of conflicts by blocking transactions which attempt to perform potentially conflicting accesses, rather than by resorting to rollback. Nevertheless, even for locking, rollbacks are still required to resolve deadlocks.

In many systems, only a fraction of data items are highly prone to conflicting accesses. In such systems neither locking nor optimistic techniques are particularly well suited. Nevertheless, most system designers opt for locking over optimism for such systems. In this paper, we present an alternative; a hybrid technique that incorporates both locking and optimistic techniques for concurrency control. In particular, this technique enhances the basic optimistic concurrency control model by using locks for high conflict data items. This technique would be particularly useful in systems that have a few data items that are prone to high rates of conflict. Furthermore, this technique is also likely to be useful when it is easy to bound the number of conflicting data items; but it is not easy to pinpoint exactly which data items are prone to high conflict rates. This could happen, for example, when the set of high conflict data items is highly dynamic, but bounded in size.

## 1.1 Related Work

Optimistic concurrency control was introduced by Kung and Robinson in [9]. In this class of concurrency control protocols, detection of potentially conflicting updates is delayed till the commit of each transaction. The transactions run in three distinct phases:

1. a read phase, during which the transactions may only read data from the database. Any writes by the transaction go to a private data area and are not reflected in the database. The transaction reads its own writes from this private data area.
2. a validation phase, during which the transaction is validated, i.e., any conflicting updates are detected. [9] provides two different techniques for validation. In the first, a transaction is valid if its readset does not intersect with the writeset of any concurrent committed transaction. In the second, the transaction is valid if its readset and its writeset both do not intersect with the writeset of any concurrent committed transaction. The second alternative is more complicated to implement, but allows higher parallelism.
3. a write phase, which takes place iff validation phase determined that the transaction is valid, during which all the writes in the private data area are committed and written to the database.

Optimistic concurrency control is designed to work in environments where read/write conflicts are relatively rare. However, [9] also indicates that optimistic techniques can be enhanced by the use

of some degree of locking. Such hybrid techniques can be used where the conflict sets are relatively small and localized. Other, later work [2, 8, 12, 11] has explored such techniques. Unfortunately, all these enhancements require that the transaction or the transaction manager to make decisions as to which transactions or data items (depending on the granularity at which the technique is hybrid) are pessimistic or optimistic. In one class of hybrid algorithms, entire transactions may be designated as optimistic or pessimistic. In the second class, individual data items can be designated pessimistic or optimistic, and the transactions themselves are hybrid.

Optimistic techniques have also been used to enhance the performance of locking in distributed databases (for example, by Halici and Dogac in [8]). In these techniques, the concurrency control protocol is locally pessimistic and uses locking. However, actual propagation of locks and updates to sites not participating in the execution of the transaction is delayed until the transaction commits. Thus, the commits take place in an optimistic fashion, with intersite read/write conflicts detected only at commit time.

Our model is different in that we use a hybrid of locking and optimistic techniques on a single site. There is no partitioning of transactions into pessimistic or optimistic classes; rather each transaction is potentially a hybrid transaction. Furthermore, the transactions do not control the set of data items which are optimistic.

# 2 Organization of the Paper

The remainder of this paper is organized as follows: Section 3 describes our hybrid concurrency control model. Section 4 describes a simulation model and our results. Finally, section 5 presents our conclusions and some future work.

# 3 The Model

## 3.1 Concurrency Control Model

We consider a database consisting of data items. These data items could be tuples or pages depending on the granularity of concurrency control. At any given instant the database exists in a state that is completely defined by the values of the data items. A state of the database is consistent iff it conforms to a set of rules. A transaction operates on a state of the database and gives rise to a new consistent state of the database on committing. If a consistent state is not achievable, say due to the effects of other concurrent transactions, the transaction must abort undoing any transient updates performed by it.

All transactions in our model must be well-formed. Thus, before accessing a data item a transaction must acquire a lock on it. Furthermore, a transaction must not release acquired locks until

commit. This is similar to the condition for recoverable 2-phase locking. As long as a transaction has an "active" lock request on a data item, i.e. it holds a granted lock or or is blocked waiting for a lock on the data item, it remains pessimistic with respect to this data item.

As in [9], writes from a transaction do not go through to the database until the transaction commits. Until then writes go to a local area private to the transaction. At commit point the transaction is validated, and on being successfully validated, all writes of the transaction are written to the database, thus modifying the database state. The validate and write must take place as a single atomic operation, though there are enhancements possible to this technique that do not have this constraint (see [9]). We assume that each write to a data item is preceded by a read to that data item. Thus the writeset of a transaction is a subset of its read set. (This requirement can be easily side-stepped. However, it simplifies our description of the validation algorithm.)

The design of the lock manager is the key to our hybrid technique. The lock manager maintains a finite lock buffer. Each slot (or frame) in the lock buffer holds locks and pending lock requests for a single data item. Thus, the number of data items with active lock requests can not exceed the number of slots in the buffer. In a typical system the number of slots in the buffer would be an order of magnitude smaller than the number of data items in the database, but would be larger than the expected number of high conflict data items.

Whenever a lock request for a data item $x$ is received by the lock manager, the lock manager first attempts to locate $x$ in the lock buffer. If it is found the lock manager attempts to post the lock in the corresponding slot. The lock request can either be granted or be blocked in the same fashion as pure locking depending on the status of existing locks on $x$. Thus, a read (shared) lock would be granted iff there is no existing exclusive lock on $x$. A write (exclusive) lock would be granted iff there are no existing locks on $x$. If the lock is granted the transaction now holds or owns a lock on $x$. If $x$ is not located in the lock buffer, a slot must be located for posting the lock request. If a free slot exists, it will be used, otherwise a victim slot must be selected. Note that any slot with no locks or pending lock requests is considered free. The victim slot is selected using an LRU algorithm. The timestamp on the slot is the last "time" a lock was requested on the data item in the slot. The data item $y$ in the victim slot is then evicted, i.e. all locks on $y$ are removed and all transactions blocked on $y$ are unblocked. Such evicted locks are called "rejected" locks. Whenever a lock (request) on a data item gets rejected, the transaction that originally requested the lock becomes optimistic with respect to this data item. The lock for $x$ is now posted to this victim slot.

If the size of the lock buffer is zero, then all the lock requests get rejected and there are never any active locks. In this scenario all transactions become optimistic with respect to all the data items in their read and write sets, and the system becomes purely optimistic. On the other hand, if the number of slots is greater than or equal to the number of data items in the database, then each lock request can be granted without evicting any existing data items and locks. Thus, the system is guaranteed to follow 2PL since each committing transaction will own locks for all the data items it accesses. It

should be noted, however, that setting the size of the lock buffer equal to the number of data items in the database is a sufficient but not a necessary condition for our technique to reduce to 2PL. Rather, the only necessary condition is that data items never be evicted from the lock buffer (i.e. locks never be rejected). As long as no data items are evicted from the lock buffer, the system can be pessimistic even with locks buffers much smaller than the number of data items in the database. We explore the implications of this in our discussion of simulation results in section 4.1.

On commit, the transaction must be validated. For a transaction to be valid, it must be valid with respect to all of the data items in its read and write sets. A transaction $T$ is valid with respect to a data item $x$ if one of the following three conditions hold (a concurrent transaction $T'$ to $T$ is one whose execution overlaps $T$; i.e. either $T'$ started after $T$ but before $T$ entered its validate phase or $T$ started after $T'$ but before $T'$ entered its validate phase):

1. $T$ holds a lock on $x$ (i.e. the lock acquired by $T$ on $x$ was not evicted).
2. $x$ is not in the writeset of $T$, $T$ does not hold a lock on $x$ and both of the following conditions hold:
   (a) if another transaction $T'$ holds a lock on $x$ then it is not an exclusive lock
   (b) $x$ was not overwritten after it was read by $T$ (i.e. $x$ is not in the writeset of any concurrent committed transaction).
3. $x$ is in the writeset of $T$, $T$ does not hold a lock on $x$ and both of the following conditions hold:
   (a) No other transaction holds a lock on $x$ (either shared or exclusive).
   (b) $x$ was not overwritten after it was "read" by $T$ (i.e. $x$ is not in the writeset of any concurrent committed transaction).

After validation succeeds, the transaction's writeset is written to the database and all locks held by the transaction are released.

Strictly speaking the requirement in conditions 2 and 3 that the $T$ not hold a lock on $x$ is unnecessary since condition 1 precludes any of the subconditions of condition 2 and condition 3. To see this, note that if $x$ is not in the writeset of $T$ then $T$'s holding of a (shared) lock on $x$ implies that no other transaction can hold an exclusive lock on $x$. Furthermore, since $T$'s lock on $x$ lasts the lifetime of $T$, no other transaction can write $x$. Thus, neither condition 2a nor condition 2b can hold. Similarly, if $x$ is in the writeset of $T$ and $T$ holds a (exclusive) lock on $x$, then no other transaction can hold a lock on $x$ (since all such locks would conflict) and our conditions preclude any other transaction writing $x$. Thus, neither condition 3a nor 3b can hold.

These conditions are incorporated into algorithm 1, which is the validation algorithm. For a transaction $T$, *READSET*$(T)$ is the set of data items read by $T$, *WRITESET*$(T)$ is the set of

**Algorithm 1** Validation Algorithm

**Ensure:** *// Inputs: Transaction* $\mathsf{T}$
1: *// Main Loop:*
2: **for all** $x \in$ *READSET*($\mathsf{T}$) **do**
3: **if** $x \notin$ *LOCKSET*($\mathsf{T}$) **then**
4: **if** $x \notin$ *WRITESET*($\mathsf{T}$) $\wedge \exists \mathsf{T}' : x \in$ *WLOCKSET*($\mathsf{T}'$) **then**
5: *Abort* $\mathsf{T}$
6: return
7: **end if**
8: **if** $x \in$ *WRITESET*($\mathsf{T}$) $\wedge \exists \mathsf{T}' : x \in$ *LOCKSET*($\mathsf{T}'$) **then**
9: *Abort* $\mathsf{T}$
10: return
11: **end if**
12: **if** $\exists$*committed* $\mathsf{T}''$ concurrent to $\mathsf{T} : x \in$ *WRITESET*($\mathsf{T}''$) **then**
13: *Abort* $\mathsf{T}$
14: return
15: **end if**
16: **end if**
17: **end for**
18: *Commit* $\mathsf{T}$
19: **END**

data items written by $\mathsf{T}$ and *LOCKSET*($\mathsf{T}$) is the set of all data items that $\mathsf{T}$ owns locks for and *WLOCKSET*($\mathsf{T}$) $\subseteq$ *LOCKSET*($\mathsf{T}$) is the set of all data items that $\mathsf{T}$ owns exclusive locks on at the instant of validation. Note that both the locksets do not contain data items with rejected locks. Also, for our model *WRITESET*($\mathsf{T}$) $\subseteq$ *READSET*($\mathsf{T}$). A transaction $\mathsf{T}'$ concurrent to $\mathsf{T}$ is any transaction which was active concurrently with $\mathsf{T}$ (i.e. their lifetimes overlap). Any transaction that owns locks when $\mathsf{T}$ is validated is automatically concurrent to $\mathsf{T}$ (since it must be active to hold locks).

These validation conditions manage read/write and write/write conflicts amongst various transactions in an intricate fashion. For any data item $x$, if all transactions hold locks on $x$ or are blocked waiting for locks on $x$, then pure 2PL prevents conflicts via condition 1, since the locking is always performed in a 2PL fashion. On the other hand, if all transactions are purely optimistic with respect to $x$ then conditions 2b and 3b (which are identical) prevent conflicts, using the strictly optimistic model. The interesting case is when some of the transactions hold locks on $x$ and other transactions are optimistic. By condition 1 and conditions 2a and 3a, a transaction that has an existing lock on $x$ is always given "priority" over a transaction that does not. Thus, our algorithm works by prioritizing pessimistic data accesses over optimistic data accesses. The validation conditions prevent any transaction that is optimistic with respect to $x$ from writing $x$ when another transaction holds a (shared or exclusive) lock on $x$. Similarly, if a transaction owns a exclusive (write) lock on $x$, then no other transaction that has read $x$ can commit until this lock is released (obviously no other un-

blocked transaction can own a lock on $x$ while it is exclusively locked by this transaction). Thus the "sanctity" of a lock is always preserved (standard lock management prevents multiple transactions from holding conflicting locks). Purely optimistic conflicts are handled using conditions 2b and 3b, which also happen to be the only conditions relevant when the lock buffer is of size zero and the system is purely optimistic.

A key detail of our algorithm is that since locks and lock requests might be evicted, the transaction may only request a lock on a data item once. Otherwise, the following scenario can happen: transaction $T_1$ locks $x$ and $y$, reads $x$, writes $y$. $x$ and $y$ are then evicted from the lock buffer. Another transaction $T_2$ locks $x$ and $y$, writes $x$, reads $y$ and commits. $T_1$ again locks $x$ and $y$, writes $x$ and $y$ and commits. $T_1$ will be allowed to commit since it obviously holds locks on both $x$ and $y$. However, the above schedule is not serializable.

Though a transaction can not request the same lock for the same data item twice, it can upgrade a shared lock to an exclusive lock. The upgrade request is treated much the same as a lock request, expect that in case the data item is not in the lock buffer, the request is "rejected" without selecting a victim slot, and the transaction must continue be optimistic with respect to this data item (note that if the upgrade failed, then the lock was previously rejected and the transaction is already optimistic with respect to this data item).

Note that the well-formedness constraint along with the single posting constraint given in the last paragraph guarantee that if a transaction is holding a lock on a data item, it will be of correct type. Thus, if the transaction is writing the data item, it owns a exclusive write lock, otherwise if it read the data item, it owns a shared (read) lock.

## 3.2 Correctness

We sketch a proof of correctness of our algorithm in this section. The full proof requires a consideration of all lock/operation conflicts and will appear in an extended version of this paper. We use the notation used by Bernstein et. al. in [3].

To prove correctness of our algorithm, we need to show that all the all histories produced by the algorithm are serializable. Suppose $H$ is a history produced by the algorithm, and $SG(H)$ is the corresponding serializability graph. By the Serializability Theorem (for a proof see [3]), $H$ is serializable iff $SG(H)$ is acyclic.

Suppose $T_i$ and $T_j$ are transactions in $H$ and $T_i \rightarrow T_j$ is an edge in $SG(H)$. Thus there must exist a pair of conflicting operations on some data item $x$, $p_i(x)$ in $T_i$ and $q_j(x)$ in $T_j$ such that $p_i(x) < q_j(x)$ (i.e. $p_i(x)$ occurred before $q_j(x)$). For the operations to be conflicting, either $p_i(x)$ or $q_j(x)$ or both must be writes. We claim that $T_i$ must have committed before $T_j$ (note that commits are atomic). Suppose instead that $T_j$ committed before $T_i$. There are two (non mutually exclusive) cases:

1. $q_j(x)$ is a write. For $T_j$ to commit, $T_i$ may not own a lock on $x$. If it does then by applying rule

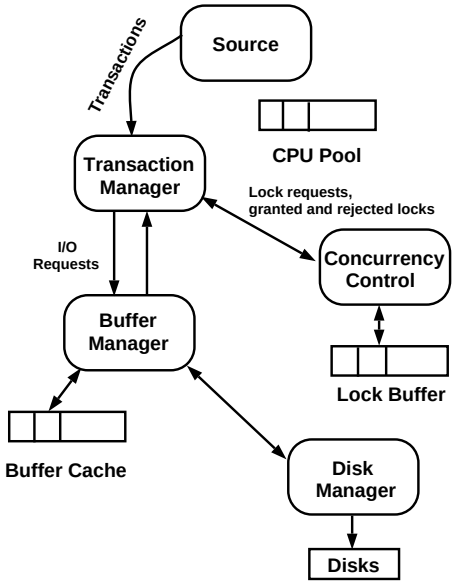


Figure 1: Simulation Block Diagram

2a or 3a, we must abort $T_j$, contradicting our assumption that $T_j$ committed. Now suppose $T_j$ commits writing $x$. Then we must abort $T_i$ by either one of rule 2b or 3b since the data item $x$ was overwritten since it was accessed by $T_i$ (note that writes are always treated as reads followed by writes in our model). This is also a contradiction.

2. $p_i(x)$ is a write. The only interesting case here is if $T_i$ owns an exclusive lock on $x$. If not, the write to $x$ is not scheduled until $T_i$ commits, which implies that $q_j(x) < p_i(x)$ (since $T_j$ committed before $T_i$) contradicting our assumption. However, if $T_i$ owns an exclusive lock on $x$ (which in turn implies that $T_j$ does not own a lock on $x$), then by rule 2a or 3a, $T_j$ must be aborted, contradicting our assumptions.

Thus, $T_i$ must commit before $T_j$. Then, if there is a cycle in $SG(H)$ each transaction in the cycle committed before itself, which is a contradiction. Thus, $SG(H)$ is acyclic and $H$ is serializable, as required.

# 4 Simulation Model and Results

In this section, we describe our simulation model and some of our results and observations. A block diagram of our simulation model is given in figure 1. The basic blocks of our simulation model are as follows:

1. Source: Creates all the transactions in the system, along with a list of tuples (data items) these transactions will access.

2. Transaction Manager: This module provides all the functionality of a transaction manager for each transaction, including handling of lock requests, page requests, deadlock management, and the read, validation and write phases of transactions. These functions are executed by the CPUs. We manage deadlocks using deadlock avoidance.
3. CPU pool: Each CPU is capable of multitasking multiple transactions. The CPUs merely execute functions provided by the transaction manager.
4. Concurrency Control: This module manages the *Lock Buffer*, handles lock and unlock requests along with functionality to evict data items from the lock buffer, and performs validation phase of transactions.
5. Buffer Manager: Manages an LRU buffer cache. Each slot in the cache holds a page. All i/o requests from the transactions go though the buffer cache.
6. Disk Manager: Manages the disk pool. The pages of the relation are assumed to be striped across the disks. Only the buffer manager can access the disks directly.

The database is a single relation consisting of *NumTuples* tuples. The relation is stored on disk as a set of pages, each page consisting of *TuplesPerPage* tuples. The pages in turn are striped across *NumDisks* disks. All page accesses go through the buffer manager which manages an LRU, write-back buffer cache consisting of *BufferSize* slots. Each slot can hold a single page.

The source generates transactions at maximum rate as follows: a "pending" queue of transactions waiting to be scheduled is maintained by the source, and every time the number of transactions in this queue drops below a threshold *QueueLen*, the source creates a new transaction. Thus the system receives transactions at the maximum possible rate. Additionally our simulation setup is also capable of generating transactions in a Markovian fashion at various arrival rates. However, we do not report those results in this paper.

A set of tuples in the form of i/o requests is generated for each transaction. A transaction is read-write (i.e. an update transaction) with the probability *ProbWrite*, otherwise it is read-only (i.e. it is a query transaction). The number of requests is uniformly distributed with an average of *TransactionSize*. For a read-write, transaction a request is a write with probability *ProbReqWrite*.

The transactions are sent to the transaction manager after being created. The transaction manager attempts to schedule the transaction on one of the *NumCPUs* CPUs in the system. Each CPU can multi-task at most *DegMulti* transactions (including blocked transactions). Transactions that can not be scheduled are held in a pending queue awaiting a free CPU.

On being scheduled, a transaction starts its read phase. In this phase, locks are acquired for each i/o request. However, only the read requests are actually passed to the buffer manager, while all write requests are held pending the write phase. Locking is performed in a recoverable 2-phase manner. Thus, all locks are acquired right before the requests are scheduled, but are not released till the end of the write phase.

| Parameter | Description | Value |
|---|---|---|
| *NumTuples* | Number of tuples in the relation | 100000 |
| *TuplesPerPage* | Number of tuples in a page | 10 |
| *NumDisks* | Number of disks | 10 |
| *BufferSize* | Number of frames in the buffer pool | 1000 |
| *QueueLen* | Minimum length of the pending queue | 0.1 - 1.0 |
| *ProbWrite* | Probability of a transaction being read-write | 0.1 - 1.0 |
| *TransactionSize* | Average number of tuples accessed by a transaction | $1000 - 10000$ |
| *ProbReqWrite* | Probability of a tuple access by a read-write transaction being a write | 0.1-1.0 |
| *NumCPUs* | Number of CPUs | 10 |
| *DegMulti* | Maximum degree of multitasking on each CPU | 10 |
| *LockBufferSize* | Size of the lock buffer | $0 - 100000$ |

Table 1: Parameters used in the simulation model and their values

At the end of the read phase, the transaction enters an atomic validation phase. For each tuple in the readset and writeset of the transaction the concurrency control manager checks if one of the conditions in section 3.1 hold. If the transaction is valid with respect to all the tuples it accessed, the transaction enters a write phase where all pending writes are written to disk.

All the locks are managed by the concurrency control manager using an LRU lock buffer of *LockBufferSize* tuples. The concurrency control manager is also responsible for blocking transactions whenever locks can not be granted, and for unblocking them whenever either the lock is granted or the corresponding tuple is evicted from the lock buffer.

## 4.1 Results and Discussion

The values of all the parameters that were used for our simulation are given in table 1. We conducted two broad classes of runs. For the first class, *TransactionSize* was set to 1000 (i.e. 1% of the database size) and *ProbWrite* was set to 0.1, 0.5 and 1.0 (i.e. 10%, 50% and 100% of the transactions were read-write), and *ProbReqWrite* was set to 0.1, 0.5 and 1.0 (i.e. 10%, 50% and 100% of the tuples accessed by the read write transactions were written to). This gives us 9 sets of results. For the second class of runs, we fixed both *ProbWrite* and *ProbReqWrite* to 0.1 and varied *TransactionSize* from 1000 to 10000 (i.e. between 1% and 10% of the database size). This gives us another 5 sets of results (one each for 1000, 2500, 5000, 7500 and 10000). For the results presented here the transactions were generated at maximum rate as described earlier. We also conducted experiments with different transaction inter-arrival rates which have not been presented here.

The experiment was conducted using a series of runs with different lock buffer sizes. For each

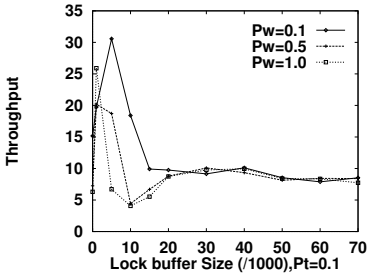


Figure 2: Throughput (Completed Transactions/Sec), $P_t = 0.1$, $|T| = 1000$

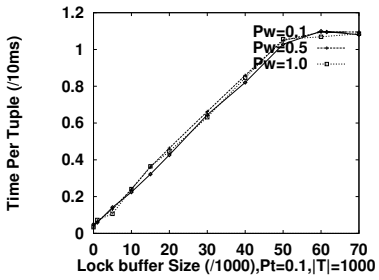


Figure 3: Time per Tuple (/10ms), $P_t = 0.1$, $|T| = 1000$

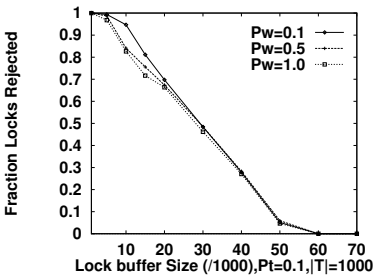


Figure 4: Fraction Locks Rejected, $P_t = 0.1$, $|T| = 1000$

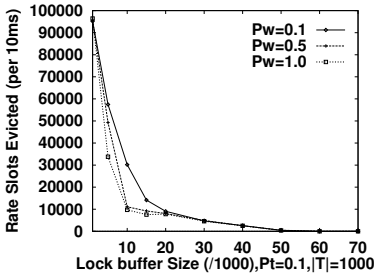


Figure 5: Rate Slots Evicted (per 10ms), $P_t = 0.1$, $|T| = 1000$

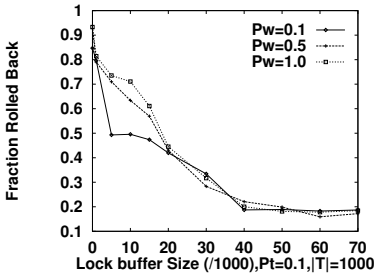


Figure 6: Fraction Rolled Back, $P_t = 0.1$, $|T| = 1000$

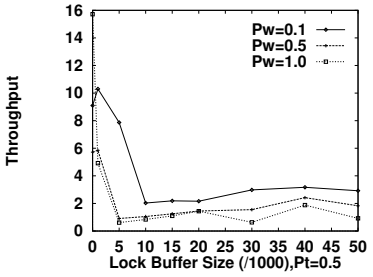


Figure 7: Throughput (Completed Transactions/Sec), $P_t = 0.5$, $|T| = 1000$

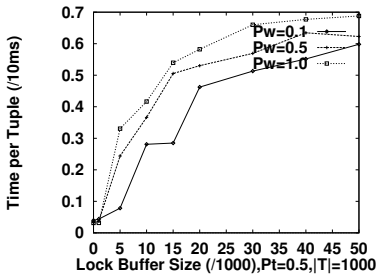


Figure 8: Time per Tuple (/10ms), $P_t = 0.5$, $|T| = 1000$

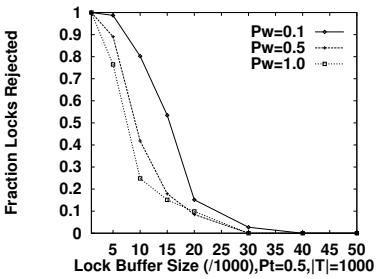


Figure 9: Fraction Locks Rejected, $P_t = 0.5$, $|T| = 1000$

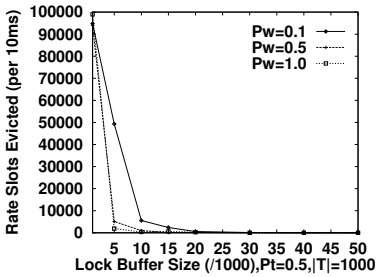


Figure 10: Rate Slots Evicted (per 10ms), $P_t = 0.5$, $|T| = 1000$

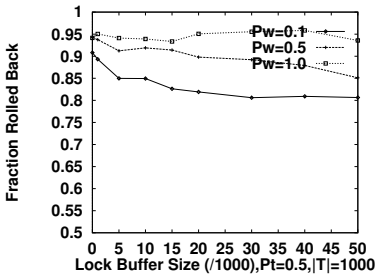


Figure 11: Fraction Rolled Back, $P_t = 0.5$, $|T| = 1000$

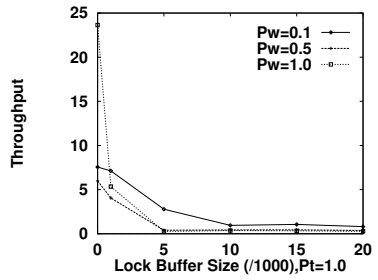


Figure 12: Throughput (Completed Transactions/Sec), $P_t = 1.0$, $|T| = 1000$

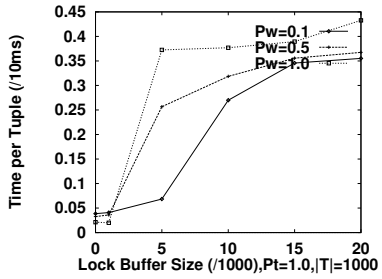


Figure 13: Time per Tuple (/10ms), $P_t = 1.0$, $|T| = 1000$

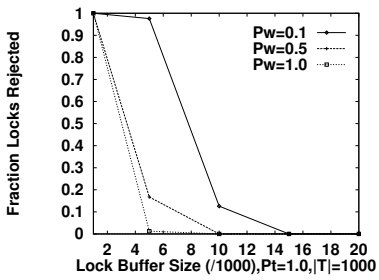


Figure 14: Fraction Locks Rejected, $P_t = 1.0$, $|T| = 1000$

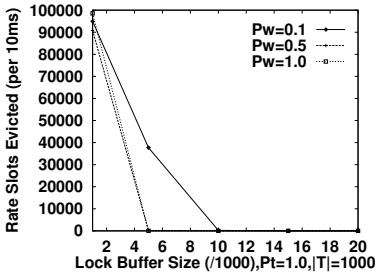


Figure 15: Rate Slots Evicted (per 10ms), $P_t = 1.0$, $|T| = 1000$

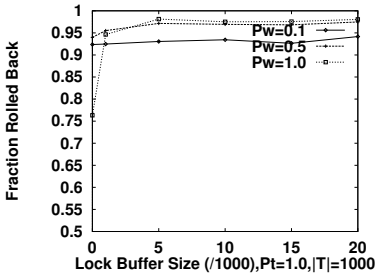


Figure 16: Fraction Rolled Back, $P_t = 1.0$, $|T| = 1000$

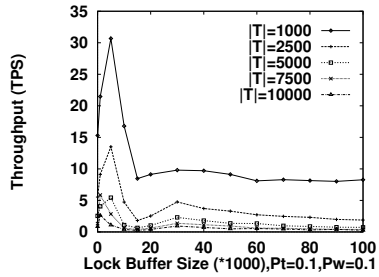


Figure 17: Throughput (Completed Transactions/Sec), $P_t = 0.1$, $P_w = 0.1$

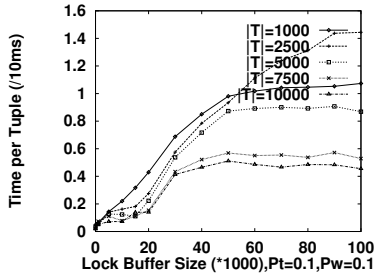


Figure 18: Time per Tuple (/10ms), $P_t = 0.1$, $P_w = 0.1$

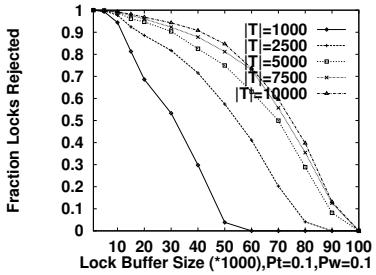


Figure 19: Fraction Locks Rejected, $P_t = 0.1$, $P_w = 0.1$

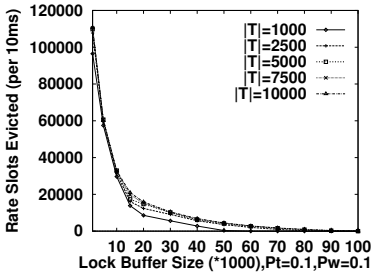


Figure 20: Rate Slots Evicted (per 10ms), $P_t = 0.1$, $P_w = 0.1$

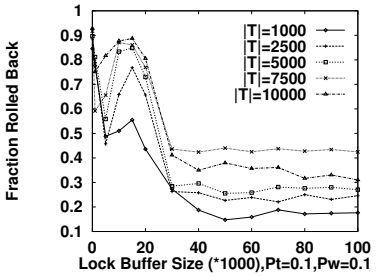


Figure 21: Fraction Rolled Back, $P_t = 0.1$, $P_w = 0.1$

data point 10 runs of 4 simulated hours each were performed. Significant results are summarized in figures 2 through 21. In each case the independent parameter is the lock buffer size scaled down by a factor of 1000. Our results are subject to a maximum variation of less than $\pm 2.3\%$. For the purpose of graphing, we denote *ProbWrite* by $P_t$, *ProbReqWrite* by $P_w$ and *TransactionSize* by $|T|$.

Based on these graphs we make the following observations:

1. Figures 4, 5, 9, 10, 14, 15, 19 and 20 are crucial for determining the degree of pessimism in the system. The slot eviction rate as well as the fraction of locks rejected are essentially measuring the same aspect of the system: how many data items "became" optimistic. The slot eviction rate measures the rate at which data items become optimistic, whereas the fraction of locks rejected measures the absolute magnitude of locks rejected. Note that eviction of a slot may resulting in eviction of one or more locks. This could happen either if there are blocked lock requests or multiple granted locks. Thus the slot eviction rate is not a good measure of the absolute magnitude of the locks rejected.

   As the number of locks rejected approaches one, the system becomes more and more optimistic. At unity, the system is fully optimistic. Furthermore, as the fraction of locks rejected approaches zero, the system becomes more and more pessimistic. When this number becomes zero, the system is fully pessimistic and follows 2PL. It might appear in the graphs that lock eviction rate reaches zero before the lock rejection rate. This is merely an artifact of the precision of the graphs.

   In each of these graphs except the last two, the system becomes fully pessimistic even though the size of the lock buffer is less than the size of the database. After this point, increasing the size of the lock buffer does not affect any of the observations. Thus, the x-axis of each graph is truncated at this point. Also note that for purely optimism (i.e. lock buffer size of 0), we do not report either lock rejection or slot eviction rate. Thus, for these graphs the x-axis starts at 1 and not 0. This again is an artifact of our simulation, which for efficiency reasons bypasses the lock buffer subsystem for purely optimistic accesses. Since the lock buffer subsystem maintains the statistics for rejection rates, we are currently unable to gather these statistics for a purely optimistic system.

2. Figures 2, 7 and 17 each show a spike in performance between lock buffer sizes of 1000 and 10000 (i.e. 1% to 10% of the database size). These spikes show a throughput increase of upto 4 to 5 times for low conflict rates. As expected spikes become less pronounced with increasing conflict rates (i.e. increasing probability of a write). Thus, we conclude lock buffers enhance performance as long as conflict rates are low.

   On the other hand figures 3, 8, 13 and 18, show that as the lock buffer size increases the time per tuple access increases. We have chosen to scale down the access time by the size of the

transactions to accommodate our last set of graphs where the transaction sizes are different. We can see that as the system becomes more pessimistic the response time increases. The initial improvement in performance is thus obtained at the cost of increased response time.

For our choice of parameters, pure optimism wins over pure pessimism. However, other experiments we have conducted indicate that for larger transactions sizes and larger write probabilities this is not true. Moreover, under those circumstances bounded lock buffers usually end up performing worse than either pure pessimism or optimism. A hint of this behavior can be seen in figures 2 and 17 where dips can be seen in the throughput figures after optimal lock buffer size is reached. This can be explained as follows: our system forces each transaction to first be subject to the overheads of locking and then to the overheads of being optimistically rolled back. If the lock buffer is too large more transactions get blocked waiting for locks and are then optimistically rejected. The time spent blocked increases the collision cross section of the transaction subjecting it to more optimistic invalidations. However, after a point when the lock buffer is large enough, the number of optimistic invalidations drops, hence restoring throughput. This shows that the lock buffer must be carefully tuned to achieve optimal performance. However, the dip is only seen when the lock buffer size is roughly twice the optimal size, giving a system designer enough leeway in designing the system.

3. Figures 6, 11, 16 and 21 measure the fractions of transaction rolled back. As expected from throughput results, figures 6 and 21 show "humps" wherever there is a dip in the throughput graph. This confirms our assertion that loss of throughput is due to increased number of rollbacks.

4. Figures 17 through 21 show our results for varying transaction sizes with both *ProbWrite* and *ProbReqWrite* set to 0.1. The results mimic the first set of graphs with *ProbWrite* set to 0.1 and *ProbReqWrite* set to 0.1.

   As expected, the performance is best for small transaction sizes. Note that smaller transactions have smaller collision cross sections. This is reflected figure 19 where the number of locks rejected per slot buffer size drops more rapidly for smaller transaction sizes. Also note that for transactions of size greater than 5000, the system becomes pessimistic only when the look buffer size is equal to database size. Also note that for transaction size of 10000, purely pessimistic and purely optimistic settings have almost identical throughput.

   For transaction sizes of 1000, 2500 and 5000 the system reaches optimal throughput when the lock buffer size is 5000 (i.e. 5% of the database size). For larger transaction sizes of 7500 and 10000 the optimal lock buffer size is 1000 (i.e. 1% of the database size).

5. Some anomalies seem to be present in figures 12 and 18. It is odd that throughput when both *ProbWrite* and *ProbReqWrite* are both set to 1.0, we should see very high throughput for a

purely optimistic setting (i.e. lock buffer size equal to zero) in figure 12. After some analysis and traces of our simulation, we realized that this is due our requirement that the validate and commit be performed atomically. Since at these probability settings, every single access is a write, all the transactions' i/o is performed in a non-concurrent fashion. In essence all transaction execute serially. For such a high level of conflicts this ends up being the best mode of operation. Similar, but less pronounced behavior is seen in figure 7 with $P_w = 1.0$.

In figure 18, the graph for transaction size 2500 shows extremely high access time per tuple. This is the only graph that shows such a significant increase. We were able to spot this behavior only after the access times were scaled down by the transaction sizes. We have as yet been unable to explain this behavior. However, our other experiments indicate that this behavior is consistent even when we increase the probability of a read-write transaction and probability of a write request.

# 5 Conclusions and Future Work

As our simulation results demonstrate, the performance of optimistic concurrency control techniques can be significantly improved by using a relatively small lock buffer. Thus our technique achieves our goals of improving the performance of optimistic concurrency control. Additionally bonus, in many cases it also performs better than 2PL.

This technique is particularly effective when the size of the set of high conflict data items is relatively small at any given instant of time. In particular, the technique should work well with the examples give by Kung and Robinson [9]. In addition, this technique introduces some new alternatives and research problems:

1. **Pinned Locks and Unlocked Data Items**: In this paper we have assumed that transactions have no control over how and when the locks can be evicted. However, it is possible to enhance this technique by allowing transactions to pin some locks. A pinned lock cannot be removed from the lock buffer until the transaction unlocks it. This causes the transaction to be purely pessimistic with respect to this data item. On the other hand, a transaction can choose to be purely optimistic with respect to a data item. This can be done by not requesting any locks on the data item.

2. **Buffer Replacement Strategies**: In this paper we have assumed an LRU strategy for selecting a victim slot in the lock buffer. However, other strategies are possible. For example, LRU-k [10] might be used instead of simple LRU. Also the number of pending/granted locks on a tuple can be used as parameters in the victim selection process. Note that high conflict data items will usually have larger number of granted locks and/or larger number of blocked lock requests.

3. **Mobile Applications**: Hybrid techniques can be used to enhance the performance of sporadically connected systems. In particular, a mobile transaction can be pessimistic whenever the mobile client it is executing on is connected, and can be optimistic whenever the client is disconnected. Furthermore, lock buffers along with *lock leases* can be used to phase out locks belonging to stale or dead mobile transactions. A lock lease provides a lock for a limited period of time. If the lease is not renewed the lock is (permanently) rejected, and the owner must now be optimistic with respect to the previously locked data item.

4. **Lock Buffer Size Tuning**: In this paper we have assumed a fixed size lock buffer. Thus the system designer must have a priori information about the size of the high conflict data set.

   However, it might be possible to use some of the metrics provided by the simulation to choose an optimal size for the lock buffer. For example, the average number of requests evicted per slot and the number of slots evicted per second could be used to dynamically tune the size of the lock buffer. Another useful parameter is the optimistic rollback rate. The lock buffer should be large enough to keep this metric low. Our simulation results can be used as a rough benchmark for this purpose.

   Lock buffer size tuning is also crucial because our results indicate that if the lock buffer is too large then performance might degrade rather than improve.

# References


[1] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil and P. E. O'Neil, A Critique of ANSI SQL Isolation Levels, *Proceedings of ACM SIGMOD Conference*, 1995, pages 1–10

[2] P. A. Bernstein and N. Goodman, Concurrency Control in Distributed Database Systems, *ACM Computing Surveys*, 13(2), Jun. 1981, pages 185–221.

[3] P. A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[4] B. R. Badrinath and S. H. Phatak, An Architecture for Mobile Databases, *Department of Computer Science Technical Report DCS-TR-351*, Rutgers University, New Jersey.

[5] B. R. Badrinath and K. Ramamritham, Semantics-Based Concurrency Control: Beyond Commutativity, *ACM TODS* 17(1), pages 163–199.

[6] S. B. Davidson, Optimism and Consistency in Partitioned Distributed Database Systems *ACM Transactions on Database Systems*, 9(3), Sep. 1984, pages 456–481.

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

[8] U. Halici and A. Dogac, An Optimistic Locking Technique For Concurrency Control in Distributed Databases, *TSE*, 17(7), 1991, pages 712–724.

[9] H. T. Kung and J. T. Robinson, On Optimistic Methods of Concurrency Control, *ACM Transactions on Database Systems*, 6(2), Jun. 1981, pages 213–226.

[10] E. J. O'Neil, P. E. O'Neil and G. Weikum, The LRU-K Page-Replacement Algorithm for Database Disk Buffering, *Proceedings of ACM SIGMOD Conference*, May 1993, pages 296–306.

[11] U. Prädel, G. Schlageter and R. Unland, Redesign of Optimistic Methods: Improving Performance and Applicability, *Proceedings of IEEE ICDE Conference*, Feb. 1986, pages 466–473.

[12] A. P. Sheth and M. T. Liu, Integrating Locking and Optimistic Concurrency Control in Distributed Database systems, *Proceedings of ICDCS Conference*, May 1986, pages 89–99.

[13] A. Silberschatz, H. Korth and S. Sudarshan, *Database System Concepts*, McGraw-Hill, 1997.

[14] A. Thomasian, Distributed Optimistic Concurrency Control Methods for High-Performance Transaction Processing, *TKDE*, 10, 1998, pages 173–189.

[15] R. Unland, U. Prad̈el, and G. Schlageter, Design Alternative for Optimistic Concurrency Control Schemes, *Proceedings of ICOD*, Sep. 1983, pages 288–297.