An Almost-Serial Protocol for Transaction Execution in Main-Memory Database Systems

Stephen Blott **Henry F. Korth**

Bell Laboratories, Lucent Technologies
blott,hfk@research.bell-labs.com

Abstract

Disk-based database systems benefit from concurrency among transactions – usually with marginal overhead. For main-memory database systems, however, locking overhead can have a serious impact on performance. This paper proposes SP, a serial protocol for the execution of transactions in main-memory systems, and evaluates its performance against that of strict two-phase locking. The novelty of SP lies in the use of timestamps and mutexes to allow one transaction to begin before its predecessors' commit records have been written to disk, while also ensuring that no committed transactions read uncommitted data. We demonstrate seven-fold and two-fold increases in maximum throughput for read- and update-intensive workloads, respectively. At fixed loads, we demonstrate ten-fold and two-fold improvements in response time for the same transaction mixes. We show that for a wide range of practical workloads, SP on a single processor outperforms locking on a multiprocessor, and then present a modified SP, that exploits multiprocessor systems.

1 Introduction

The rapidly emerging wireless information infrastructure places a premium on the timely delivery of data. The requirement of timeliness recurs at a variety of levels with wireless systems, including the network infrastructure (cell hand-offs, etc.), billing, accounting and monitoring (pre-paid billing, fraud detection and prevention, and financial settlements between cooperating service providers), and high-level applications (instant messaging, personal agents).

Most existing database management systems are designed to maximize overall system throughput, rather than provide short, predictable response times for individual requests. Main-memory database systems [3, 7, 8, 9, 16, 17], however, have been designed to achieve that latter goal. These systems offer most of the features of a traditional disk-based system, while having an architecture tuned to the assumption that the entire database is resident in main memory.

In this paper, we focus on one aspect of main-memory database performance: transaction concurrency control. The architecture of main-memory databases reduces the benefit of supporting full concurrency. The lack of disk I/O for purposes other than logging and checkpointing eliminates most transaction blocking for disk I/O. For this reason, we consider protocols that limit concurrency and reduce task switching, yet exploit the full power of available computing resources. The resulting reduction in complexity of transaction management combined with the reduction in task switching offers substantial improvements in performance, which we document experimentally.

The remainder of this paper is structured as follows. Section 2 discusses the application of low-response-time database management in wireless telecommunication systems. Section 3 covers related work on main-memory concurrency control. Section 4 describes our protocol for serial transaction execution. Section 5 presents an experimental evaluation that illustrates the performance of our method under various types of workload. Section 6 discusses the application of our method to multiprocessors, and Section 7 concludes.

2 Wireless Systems and Response-Time Constraints

A main motivating factor in our work is the tight response-time constraints of telecommunication systems. Unlike most on-line transaction-processing systems, in which throughput is the primary perfor-

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

mance metric, telecommunication systems have stringent response-time requirements both in terms of average response time and variance therein. Although the work we describe applies generally to main-memory database systems regardless of application, we illustrate these needs via the following brief introduction to a wireless telecommunication application.

2.1 Wireless Infrastructure

The infrastructure for wireless telecommunication consists of mobile-switching centers (MSC) that control a set of base stations (BS), each of which manages a changing set of mobile stations (MS). The MSs can be mobile phones or other wireless devices. MSCs must interoperate with the regular phone network (the public, switched telephone network or PSTN), and through it, with MSCs of other wireless networks. Each wireless-service provider keeps a record of its customers in a database called the home-location register (HLR). For a given MS, the HLR record includes such data as phone number, mobile-id number (the unique identifier of the MS), profile, and current location.

When a MS moves outside the range of its home network into that of some other service provider, it must register with the new provider. This registration record is stored in the service provider's visitor-location register (VLR). This service provider must contact the MS's home service provider to obtain HLR data, which it then caches in its VLR. The home service provider updates its record of the current location of the MS, records the registration of the MS with the remote service provider, and may need also to cancel any obsolete registrations the MS may have had with other service providers.

From this, we can see that the simple act of "roaming" to another network can create a flurry of database updates. These updates must be completed before the MS can make or receive a call on the network on which it is roaming.

Suppose someone on the PSTN attempts to call the MS. The telecommunication switches in the PSTN route this call to the home network of the MS, where the HLR is consulted, the location of the MS determined, and the correct VLR identified. This VLR is then accessed to determine the correct MSC to which to route the call. In practice, some caching and optimizations are performed, but regardless, we see that the initial set-up of a call requires multiple database accesses. To provide mobile-phone users with their expected level of response, these database accesses have time budgets of just a few milliseconds.

More critical in terms of the expectations of phone users is fast hand-off of a MS from one BS to another. The complexity of a hand-off increases if the old and new BSs are managed by different MSCs. Total hand-off time in some networks is as low as 20 milliseconds. This time is not just for database access! It also includes time for other computations such as the evaluation of heuristics to minimize thrashing in cases where a user is traveling along a boundary between two BSs' regions, and managing cases where the new BS cannot accommodate the load of an incoming user.

2.2 Databases for Wireless Networks

To meet these technical challenges, network equipment makers use main-memory database management systems. Each aspect of the architecture, algorithms and data structures of these systems has been tuned under the assumption that the entire database is memory resident. By eliminating each source of overhead, unpredictability, and wasted CPU cycles, these systems are able to achieve the types of performance required of wired and wireless network systems.

Locking operations are one substantial source of CPU overhead for these systems. In this paper, we focus on the cost of concurrency, and optimizing transaction execution in light of the fact that the database is memory resident.

3 Related Work

Traditionally, a key motivation for transaction concurrency is to keep the CPU active during disk I/Os. This motivation exists in main-memory database systems as well, but to a lesser extent, as disk I/O within a user transaction's execution comes from only one source — the output of log records prior to transaction commit. The only activity with significant disk I/O is checkpointing, a matter that can be dealt with separately, as in [19], where a technique called *fuzzy checkpointing* is used to avoid disrupting transaction processing when a checkpoint is taken. In our earlier experimental work with real-time telecommunication systems [1, 2], we observed that the overhead of acquiring and releasing locks had a significant impact on performance. This argues for the use of relatively coarser-grain locks. Indeed, the observation that fine-grain locking can incur excessive overhead in certain cases was made in the early days of research in lock protocols [12]. Several authors, including [8, 9, 10, 18, 20], have noted this in the context of main-memory database systems. Salem and Garcia-Molina have proposed an approach where most (short) transactions are scheduled serially, but long transactions and checkpointing operations execute concurrently [20]. Priority is generally given to short, real-time transactions. However, unlike the algorithm proposed here, Salem and Garcia-Molina's algorithm requires locks to be obtained by all classes of transaction, and is also subject to deadlock. Deadlock, though typically a non-issue from a real-world performance standpoint, takes on greater concern in a response-time-constrained system where the impact of deadlock on a single transaction may cause deadlines to be missed.

Our focus here is on general-purpose transaction management for response-time-constrained systems. True real-time systems have special properties that Graham [11] exploits to achieve serializability via the concept of transaction classes from SDD-1 [5]. Graham formalized the design principles underlying these systems, and defined a broad class of systems for which mutual exclusion of critical sections within transaction classes ensures serializability.

DeWitt et al. [8] have proposed a locking-based approach to concurrency control in main-memory systems whereby locks are released at the point the decision is made to commit a transaction, rather than after the commit record has been written to disk. Under this approach, known as 'precommit,' the lock table is extended to record not only the transactions that are waiting for a lock, and those that hold it, but also those that have released the lock, but are still waiting for their commit record to be written. While the protocol described here is also based on the concept of precommit, our protocol eliminates the costs of maintaining an additional data structure recording this wait-for relationship.

An idea originating within IBM's IMS FastPath, (see [13] and [9], page 511) and pursued within IBM's Starburst System [15] is to attach some lock information directly to objects themselves [17]. These locks are never paged out. Moreover, whenever a lock is required, it is usually to be found already within the CPU cache. In FastPath, lock information is encoded in two bits within object headers. Locking algorithms using these bits are designed in such a way that, if lock contention is low—which is the expected case—then locks can be acquired and released with efficient bit-manipulation operations on objects themselves. In [10], Gottemukkala and Lehman address the overhead of *latches*, short-term low-level locks typically used to control access to system-internal data structures. They found the use of table-level latches in place of page-level latches in the Starburst system's main-memory manager achieved up to a 35% performance gain (although still subject to the risk of deadlock).

One important dependency concerns the interaction between locking and checkpointing. In order to ensure that dirty pages are not written into checkpoints, many systems require the checkpointing process to acquire locks [20]. Since the checkpointing process generally runs frequently, locking should be at a fine granularity to avoid unnecessary blocking (which is in contrast to the argument above for coarse-granular locking). An alternative, non-locking approach is a technique known as fuzzy checkpointing, which was used originally in IMS FastPath [13]. With fuzzy checkpointing, the checkpointing process does not acquire database locks, and thus dirty pages can become part of a checkpoint. However, by also recording a section of the log together with the checkpoint itself, the database can be returned to a transaction-consistent state during recovery. The work described here is based on the DataBlitz^{TM} Main-Memory Database System, which uses a fuzzy checkpointing algorithm [19].

In our application domain, most transactions are extremely short and the probability of lock contention extremely low. Moreover, with short, main-memory transactions, the cost of CPU-cache flushes due to the interleaving of concurrent transactions is significant. For these reasons, we studied alternative approaches to concurrency control in main-memory systems based on extremely-coarse granular locking, at the database level (that is, transactions are executed serially, except for some concurrency during commit). Our work thus takes previous work in this area to its logical extreme.

4 Developing the SP Protocol

This section describes a series of protocols for serial transaction execution, each in turn improving upon the limitations of its predecessor. We begin in Section 4.1 with a naïve protocol that is presented only for illustrative purposes. In Section 4.2, we review and expand upon the precommit concept introduced by [8]. Section 4.3 presents our approach.

Throughout, we assume a process structure in which several processes—or threads, although we use only the term *process* here—access a shared database system. Each transaction is executed by one process only. A single mutex, which we call the *database mutex*, is used to ensure mutual exclusion of database accesses. While one process holds the database mutex, any other process requesting the mutex is blocked, at least until the first process releases it.

4.1 A Naïve Protocol

The simplest approach to concurrency based on a single mutex is the following. Prior to issuing its first operation, each transaction ensures that no other transaction is active by acquiring the database mutex. Since no two transactions are ever active at the same time, database locks need not be acquired, and deadlock cannot occur. The database mutex is released only after a transaction's commit or abort processing has been completed. Clearly, executions governed by this protocol are serializable (indeed, they are serial) [4]. Moreover, the overhead of this protocol is marginal. In contrast, for lock-based protocols, acquiring a single lock can itself involve several mutex operations, in addition to the operations necessary to create and/or update the lock data structures themselves [14].

The major limitation of this approach, however, is that processing and logging operations cannot be overlapped. Since transactions execute serially, the CPU can be idle while logging operations take place. In the worst case, regardless of the multi-programming level, the system alternates between periods of CPU activity, and periods of I/O activity for logging.

```
Read transactions:
   acquire database mutex
   begin transaction
   perform transaction operations
   commit (or abort) transaction
   release database mutex

Update transactions:
   acquire database mutex
   begin transaction
   perform transaction operations
   generate log records in main memory
      (or undo effects)
   release database mutex
   flush log (if committing)
   commit (or abort) transaction
```

Figure 1: Outline of a precommit-based protocol. For correctness, transaction dependencies must be managed (see text).

4.2 A Protocol Using Precommit

A transaction is said to be *precommitted* if its commit record is in the main-memory log buffer waiting for the log to be flushed. True commit does not occur until the commit record is flushed. The precommit operation can be used to overcome the limitations of the naïve approach of Section 4.1. The system must ensure that transactions commit in an order such that commit dependencies among transactions are maintained.

In [8], three sets are associated with each lock: those actually holding the lock, those waiting for the lock, and those that hold the lock but are still in the precommit state. These lock data combined with transaction dependency data are used to ensure a correct ordering of the log-flushing operations.

For our purposes, in which we consider locking at only the database granularity, we may consider an adaptation of the precommit approach based on the outline of Figure 1. An update transaction that is about to commit releases the database mutex not after, but before flushing the log (see the figure). While one transaction is waiting for a log-flush operation to complete, another can begin its processing phase.

As stated in the figure, the protocol is incomplete (and, therefore, actually incorrect). The difficulty is that, because the database mutex is released early, read-only transactions can now commit in the state of having read uncommitted data (since read-only transactions are not required to flush the log). One way to overcome this difficulty, of course, is to force read-only transactions also to flush the log prior to committing. Under this approach, however, read-only transactions always block (regardless of whether they actually read uncommitted data) until all of their predecessors have completed their commit phase. This additional delay and increased variance in response time can be unacceptable for response-time critical applications.

Our goal in this paper is to employ the concept of precommit efficiently to our serial approach to transaction execution, which we do below.

4.3 SP: A Serial Protocol

In this section, we describe our serial protocol, SP. The new protocol is based on two mechanisms: timestamps and a mutex array. Timestamps are used to determine, for each read-only transaction, the identity of the most-recent predecessor from which that transaction has read. The mutex array is then used to ensure that a read-only transaction is blocked at least until that predecessor completes its commit processing.

4.3.1 Timestamps in SP

Key to SP is the use of timestamps. Unlike the traditional timestamp-based concurrency-control schemes [4], SP uses transaction timestamps for the sole purpose of ensuring that commit dependencies among transactions are satisfied. Actual concurrency is managed by the database mutex (thus, the name *serial protocol*).

As in prior timestamp-based schemes, each transaction T is assigned a timestamp $TS(T)$ (which is initialized to 0) when it begins, and each data item d is assigned a write timestamp $WTS(d)$. Unlike traditional timestamp-based schemes, no read timestamp is needed. A further distinction of our scheme is that transaction timestamps are not fixed. Transaction timestamps are maintained as follows:

- Immediately before transaction T issues its first write operation, T is issued a new $TS(T)$ greater than any issued previously. (This may be implemented by incrementing a global counter. Note also that transactions need *not* be pre-declared as "read-only" or not under SP.)

- Write timestamps are maintained in the usual way: when a transaction T writes data item d, $WTS(d)$ is set to be $TS(T)$. (Note that, because transactions execute serially, $WTS(d)$ is non-decreasing.)

- When transaction T reads data item d, $TS(T)$ is set to be the maximum of $TS(T)$ and $WTS(d)$.

Thus, upon completion of an update transaction T, $TS(T)$ is a new timestamp, higher than any other within the system. Upon completion of a read-only transaction, $TS(T)$ is the timestamp of the most recent transaction from which T reads. For each data item d, $WTS(d)$ is the timestamp of the most recent transaction that wrote into d. Notice that there are no aborts or waits induced by this timestamp scheme. Moreover, the database mutex and the monotonicity of timestamps ensures that, prior to an update, $WTS(d) \leq TS(T)$.

```
Read transaction($T$):
   acquire database mutex
   begin transaction
   perform transaction operations
   release database mutex
   if aborting then abort
   else
      acquire $A[TS(T)]$ /* instantaneous */
      release $A[TS(T)]$
      commit

Update transaction($T$):
   acquire database mutex
   begin transaction
   perform transaction operations
      (including timestamp updates)
   generate log records in main memory
   if aborting
      perform transaction-undo steps
      release database mutex
   else /* $T$ is precommitted */
      acquire $A[TS(T)]$
      release database mutex
      flush log
      /* $T$ is now committed */
      release $A[TS(T)]$
```

Figure 2: SP: A serial protocol that uses timestamps and an array of mutexes to ensure that read-only transactions are blocked until the predecessors from which they read commit.

The basic scheme we have outlined above can be tuned for efficient timestamp maintenance and for space efficiency in the representation of timestamps. We discuss this later in Section 4.3.4.

4.3.2 The Mutex Array

The commit process in SP is governed by an array that enforces all transaction dependencies. Let A denote an infinite array of mutexes (which might be implemented by a finite array that is indexed modulo the array size). Consider an update transaction T. When T intends to commit, it acquires $A[TS(T)]$—that is, the mutex in position $TS(T)$ in array A—prior to releasing the database mutex. $A[TS(T)]$ is then held until T receives the acknowledgment that its commit record has been written to disk, at which point $A[TS(T)]$ is released. Now consider a read-only transaction T. When T intends to commit, it first releases the database mutex and then acquires $A[TS(T)]$—but only instantaneously. T may then commit. By holding the mutex briefly, T ensures any transaction on which it has a commit dependency has itself committed.

This protocol is sketched in Figure 2. Note that the flush-log operation is performed sequentially on log records (or pages) so that log records are flushed in the order in which they were created.

4.3.3 Correctness of SP

The proof that SP ensures serializability and atomicity is relatively straightforward. We present the details in this section.

In the absence of failures (and therefore, ignoring the issue of commit), SP's use of a single lock over the entire database guarantees serial (and thus serializable) execution. Under this assumption, atomicity is guaranteed automatically, so the timestamps do not matter.

Relaxing our assumption about failures, we must ensure that (1) no transaction commits in a state where it has read uncommitted data and (2) the commit order of transactions corresponds to the serialization order. For example, a read-only transaction must not read a value written by an update transaction that commits after the read-only transaction commits; as the update transaction may then subsequently abort. We show that SP in fact meets these requirements.

Let the *final timestamp* of T, $TS_f(T)$, be the final value of $TS(T)$, that is, the value of the timestamp of T at the time T finishes its execution and attempts to commit.

Since log records are usually flushed in units of pages, transactions whose commit records share a log page may commit "simultaneously." Thus, in our discussion below, we may use phrases like "no later than" rather than the word "before" in discussing commit.

Proposition 1 *Update transactions commit in timestamp order. That is, if T_1 and T_2 are update transactions that commit, then if $TS_f(T_1) < TS_f(T_2)$, then T_1 commits no later than T_2.*

Proof: Since $TS_f(T_1) < TS_f(T_2)$, T_1 performs its first update before T_2 did; so T_1 acquires the database mutex before T_2. As specified in Figure 2, T_1 creates its log records in main memory before T_2 creates any log records. Thus, the flush operation will output the log records of T_1 before or concurrent with those of T_2. □

Proposition 2 *Read-only transactions that commit do not do so before any committed update transactions from which they read.*

Proof: Let T_2 be a read-only transaction that commits. Let T_1 be the transaction from which T_2 reads that has the highest final timestamp. By Proposition 1, T_1 commits last among all transactions from which T_2 reads. By the definition of timestamp assignment to read-only transactions, $TS_f(T_1) = TS_f(T_2)$. Since T_2 reads from T_1, it cannot begin its execution until after T_1 releases the database mutex. But, prior to doing so, T_1 acquires $A[TS_f(T_1)]$, which it holds until its commit record has been flushed. T_2 cannot commit without first acquiring $A[TS_f(T_2)] = A[TS_f(T_1)]$. □

The next proposition is based on a standard assumption about the failure model of computing systems – that a failure causing loss of main memory is a system crash.

Proposition 3 *Any transaction that reads data written by an aborted transaction itself aborts.*

Proof: Let T_1 be an update transaction that aborts. We consider two cases. First, assume T_1 aborts without having precommitted. Any transaction T_2 that reads data written by T_1 must acquire the database mutex after T_1 releases it. But since T_1 aborts prior to precommitting, it aborts prior to releasing the database mutex, and its updates are undone. Next, consider the second case in which T_1 aborts after having precommitted. Then the log in main memory (which contains T_1's commit record) must have been lost before it could be flushed. By assumption, a failure that causes a loss of main memory is a system crash in which all active transactions abort. By Proposition 1 (for update transactions) or Proposition 2 (for read-only transactions), any transaction that read from T_1 cannot have committed prior to the time of the crash and therefore aborts. □

Next, we address the issue of blocking on the database mutex. We ignore the case where an operating system process dies while holding the database mutex. This issue is treated outside the scope of the transaction manager. Indeed DataBlitz, the system we used in our experiments, detects and properly recovers from such scenarios.

Proposition 4 *A transaction holding the database mutex cannot block for disk I/O nor for another mutex.*

Proof: The only disk I/O, the log flush, does not occur while the database mutex is held. The only case where a transaction acquires a mutex while holding the database mutex is the case of an update transaction T acquiring $A[TS_f(T)]$. But since it holds the database mutex at this point, no other transaction could have read from T and no other update transaction could have performed an update after T began. Therefore, $TS_f(T)$ is strictly the largest timestamp of any transaction and $A[TS_f(T)]$ cannot have been acquired previously by any other transaction. □

4.3.4 Implementation of SP

We now describe several implementation details that, for simplicity, we have ignored thus far.

Bounding the Size of a Timestamp: As presented above, timestamps grow ever larger. In practice, we can restart issuance of timestamps to update transactions at 1 whenever the systems restarts, provided that we also reset all data timestamps to zero. At system restart, resetting data timestamps is easy to do, as the database is reloaded into memory. At the price of slightly greater complexity, timestamps can be reset any time the system is quiescent.

Timestamp Granularity: A timestamp is associated with each data item, however, we have not specified so far exactly how large a "data item" is. The relative advantages of coarse and fine granularity are well known. Fine granularity allows greater potential concurrency at the price of higher overhead to store and maintain timestamps. In SP, timestamps control only write-read synchronization and relatively few update transactions are likely to be awaiting a log flush at any point in time. Thus, only a low degree of concurrency is usually needed and coarse timestamp granularity should suffice in general.

For our telecommunication applications, the application itself usually indicates the "natural" granularity, e.g., a customer record or an HLR record.

5 Performance Experiments

In order to evaluate SP, we carried out a series of performance experiments using DataBlitz. DataBlitz uses strict 2PL for concurrency control. However, DataBlitz also supports the ability to turn off locking, offers control over the point at which logging operations are performed, and provides an efficient, user-level semaphore implementation [6]. Using these features, we implemented the SP protocol described in Section 4.3 atop the existing, lower-level primitives of DataBlitz. This provided a single platform whose behavior could be toggled between strict 2PL and SP.

Our experimental database consisted of a single hash table, with 20,000 64-byte records. We performed experiments on a uni-processor system varying the multi-programming level from 1 to 35. Before beginning, each transaction determined randomly whether it was a read-only or an update transaction (according to a probability that was also varied). Each transaction probed 20 entries in the table, either reading or updating each entry (as determined above). Our experiment was designed to simulate the workload generated by telecommunication applications such as those discussed in Section 2. In order to avoid the possibility of deadlock biasing our results against the strict 2PL case, table entries were probed sequentially.

We discuss both throughput and the more important metric for our applications, response time.

5.1 Throughput as a Function of Updates and Multiprogramming

In our first set of experiments, we measured maximum throughput for DataBlitz using SP, and for DataBlitz using strict 2PL, as a function of both the multiprogramming level and the percentage of transactions that are update transactions. The results are illustrated in Figure 3. In the figure, we plot throughput (transactions per second) as a function of the fraction of transactions that are update transactions and do this for 4 distinct levels of multiprogramming. Each graph has 2 curves: one for SP, and one for 2PL. We see that SP always performs better, especially when fewer than 30% of transactions are update transactions. In practice, we can say without qualification

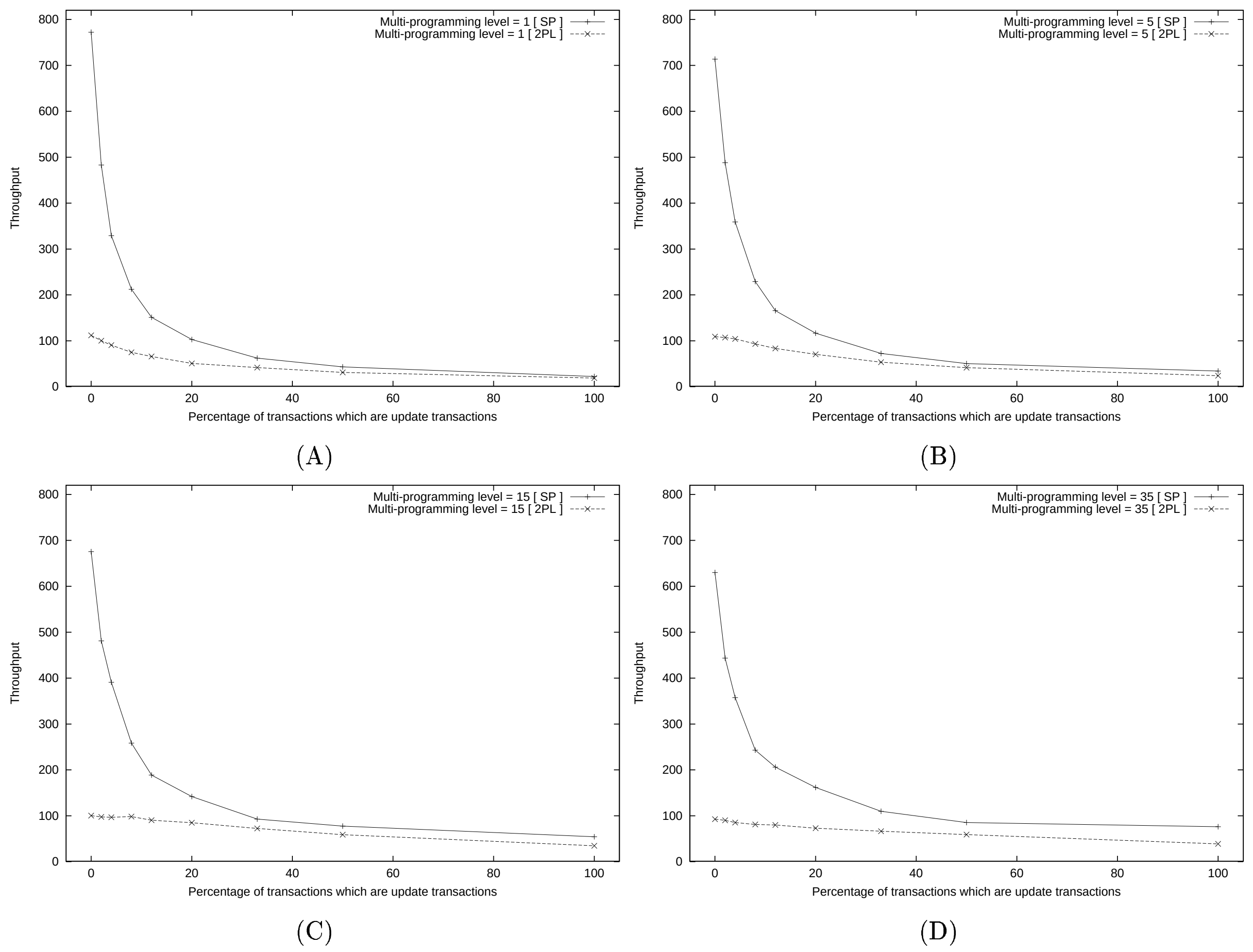


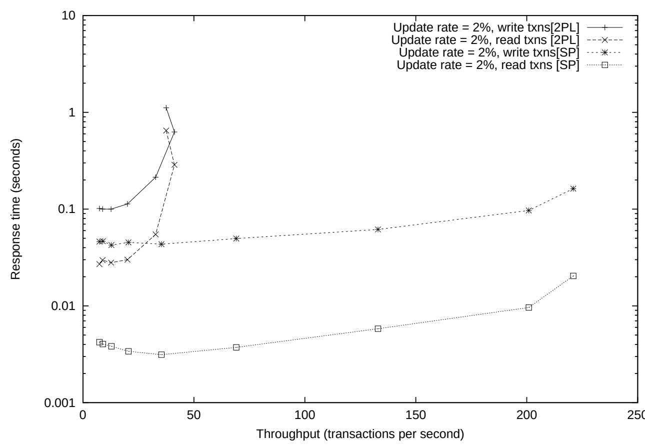
Figure 3: Maximum throughput in transactions per second as a function of the fraction of transactions that are update transactions, for multi-programming levels of (A) 1, (B) 5, (C) 15 and (D) 35.

that SP performs better since real-world percentages of update transactions in our motivating applications are much smaller than 30%.

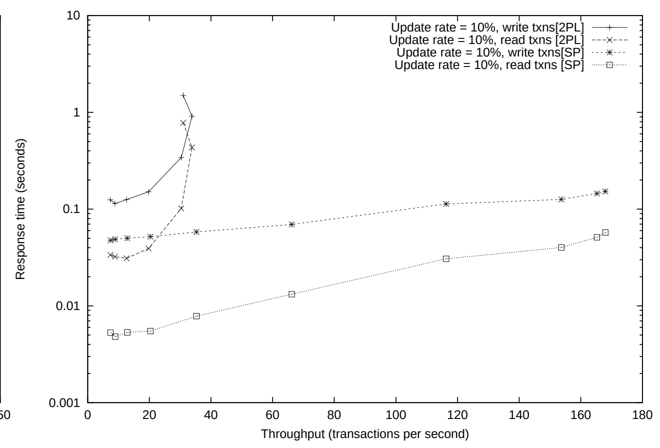
First, consider Figure 3(A), for which the multi-programming level is 1. Let p denote the time to execute the processing steps of one transaction, and l denote the time taken by the locking steps of the transaction. From the graph, we observe that approximately 110 transactions run per second under 2PL at multiprogramming level 1 with 0% update transactions. Since there are no I/O operations, $110p + 110l = 1sec$. Under SP, 780 transactions run per second with no locking, so $780p = 1sec$. From this, we derive that the overhead of locking is approximately 6 times the cost of performing the database operations themselves.

In Figures 3(A) through 3(D), the multi-programming level is raised from 1 process to 35 processes[1] (all on the one processor of our test system). For low update percentages, increased multi-programming has a marginal effect on throughput (due mainly to the additional overhead of context switching). For high update percentages, however, the effect on throughput is significant. For both SP and strict 2PL, throughput increases with the multi-programming level. This is because, as more processes are added, the probability that all of them are blocked on log flushing operations decreases. However, as throughput increases, the effect of the locking overhead also becomes increasingly apparent, and, at a multi-programming level of 35, the throughput of SP is double that of strict 2PL, even for 100% update transactions. SP's throughput continues to improve at higher levels of multiprogramming until we reach a point at which it is never the case that processes are waiting for log flushing to complete. The performance of 2PL eventually degrades significantly as the multi-programming level rises due to lock contention (and deadlock as well, had we not designed our experiment specifically to remove the possibility of deadlock under the 2PL scenario).

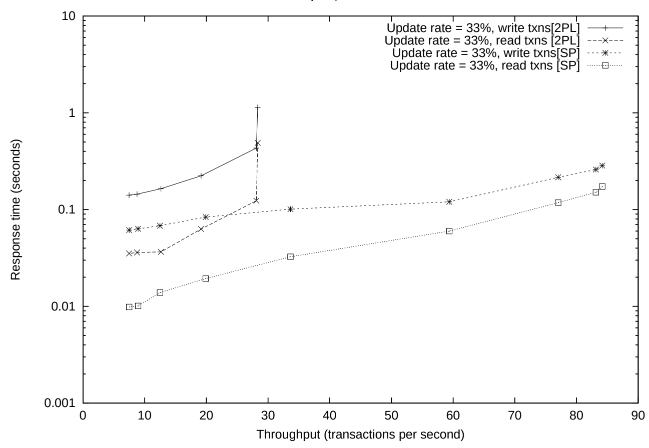
[1]a level near the saturation point for our experimental environment



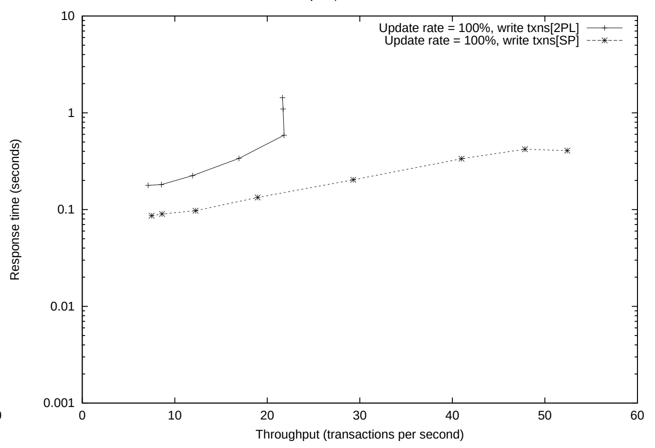
(A)



(B)



(C)



(D)

Figure 4: Average response time as a function of throughput, for percentages of update transactions of (A) 2%, (B) 10%, (C) 33% and (D) 100%; multi-programming level is 35.

5.2 Response Time as a Function of Load

In a second series of experiments, we varied the offered load, and measured average response time. These experiments allow us to determine how response time degrades in our wireless telecommunication application as the network reaches peak utilization. In practice, this is of greater importance than our measurements in the first series of experiments, since we are concerned with being able to guarantee with high probability that the response time required by the application is met. We report average response time, rather than worst case, since the variance in response time in our main-memory environment is low.

Our experiments were done with a multiprogramming level of 35 with varying arrival rates. We set the percentage of transactions that are update transactions to be 2%, 10%, 33%, and 100%, and show the results for each in Figure 4 (A) through (D), respectively. Response time is plotted separately for read-only and for update transactions (there are, of course, no read-only transactions in Figure 4(D)). The scales on the y-axes are logarithmic. The x-axes actually show throughput rather than load. This allows us more clearly to explain the behavior of 2PL under heavy load.

From the figure, we see immediately that, as we saw in the first series of experiments, SP's performance degrades gracefully under heavy load, unlike 2PL. For heavy loads, only SP is feasible.[2]

In the range of loads where response time comparisons can be made, SP offered significant response time improvements over strict 2PL. For example, in Figure 4(B), where 10% of transactions are update trans-

[2]The graphs in Figure 4 for 2PL show a doubling back of the curves. This occurred at the point of maximum throughput for 2PL. Offered loads beyond the level of maximum throughput cause contention and thrashing that actually reduce throughput as load rises. Thus, we see two response-time values for certain throughput levels. The lower value comes at a point where offered load is less than maximum throughput; the higher value comes at a point where offered load is greater than maximum throughput (a region certainly to be avoided in practice).

actions, at 20 transactions per second, the response time for read-only transactions is 6 ms in the case of SP, as opposed to 30 ms in the case of 2PL. The response time for update transactions is 50 ms in the case of SP, as opposed to almost 200 ms in the case of strict 2PL.

These response time improvements—of a factor of 4 to 5—are significant for our telecommunication applications. This difference is enough to allow a general-purpose database solution to be employed in place of a custom-coded, special-purpose system. Similar performance gains cannot be obtained simply by upgrading the hardware to a faster processor and using 2PL, since we would still face 2PL's inability to scale to high loads. SP's lack of lock contention not only offered better response time, but also better scalability.

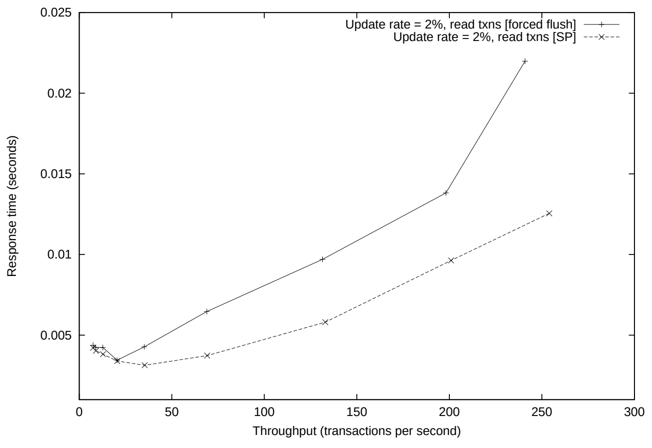
5.3 The Effect of the Commit Mechanism

Up to this point, our focus has been on comparing SP with 2PL. Now we consider whether the incremental complexity of the mutex array A is justified relative to the simpler approach described in Section 4.2, in which read-only transactions must flush the log to disk prior to committing. Once again, we measured average response time as a function of offered load (using the same techniques as in Section 5.2). We present in Figure 5 (A), (B), and (C), the average response-time graphs for read-only transactions when the percentage of transactions that are update transactions is 2%, 10%, and 33%, respectively. We focus on read-only transactions here, since the commit protocols differ only in their treatment of read-only transactions.

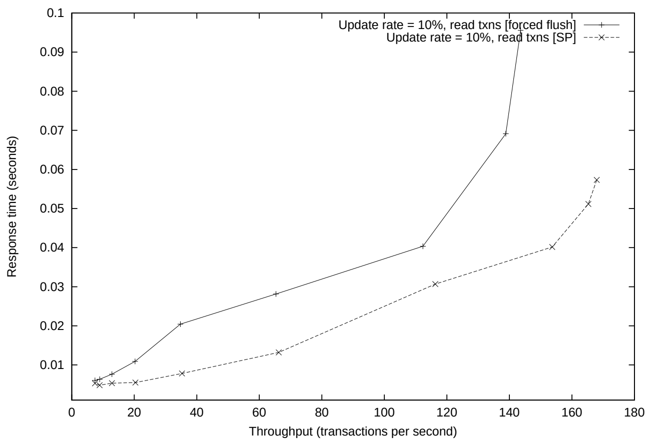
These graphs suggest that the effect of the choice of commit mechanism is dependent first upon the offered load, and then upon the fraction of transactions that are update transactions. At low load, the probability of a read-only transaction encountering a situation where the log buffer is non-empty is small, and the two mechanisms perform nearly identically. Conversely, at higher load, the probability of a read-only transaction encountering a non-empty log buffer is increased, and our mechanism using the mutex array can reduce response time significantly. On the other hand, this effect diminishes as a larger fraction of the transactions are update transactions. This is because as update transactions become more frequent, so too do the number of read-only transactions whose commit is delayed necessarily under both mechanisms, due to conflicts. For low-to-moderate ratios of update transactions to read-only transactions, the benefit of our commit mechanism appears worth the slightly added complexity.

6 SP on Multiprocessor Systems

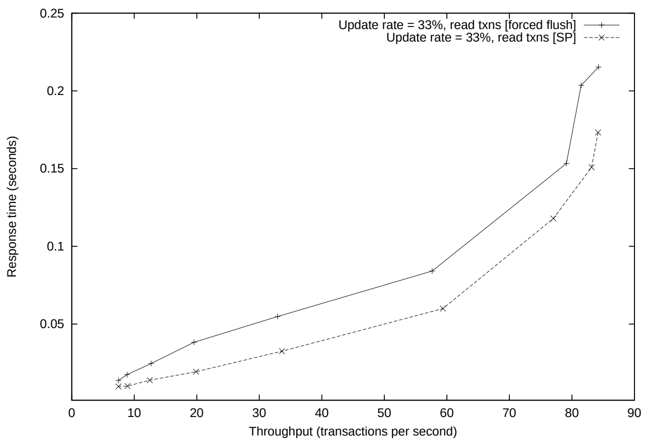
One apparent limitation of SP is that, prima facie, it cannot exploit the parallelism of multiprocessor systems. If transactions execute serially, then at most



(A)



(B)



(C)

Figure 5: Average response time of read-only transactions as a function of throughput, update rates of (A) 2%, (B) 10% and (C) 33%; multi-programming level is 35.

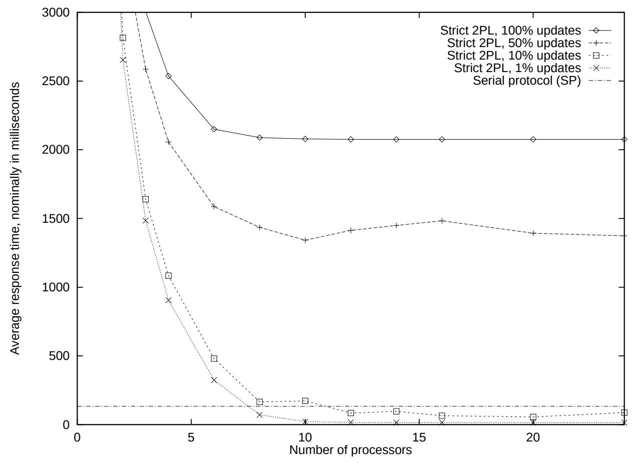


Figure 6: Average response time as a function of the number of processors: 2PL and unmodified SP.

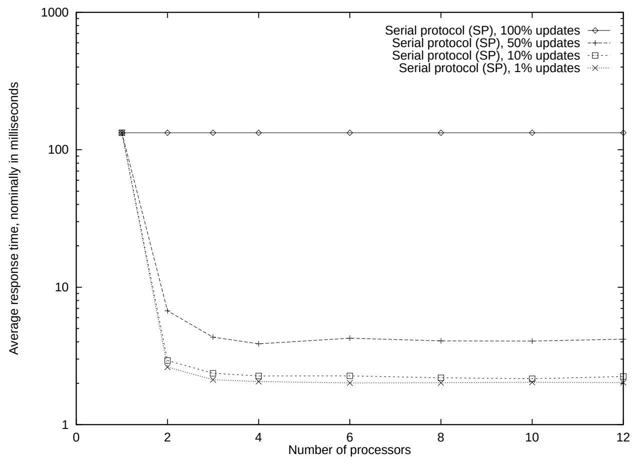


Figure 7: Average response time as a function of the number of processors: modified SP.

one processor can be busy at any time, leaving additional processors idle. In this section, we present a series of experiments that help quantify and explain the trade-offs of concurrency and serial execution in multiprocessor systems. Unlike the experiments reported in Section 5, those reported here are based on an analytical model, driven by real call-data traces. In general, each transaction goes through three phases: waiting, processing, and logging. In order to avoid the added variables of disk simulations, we model only the waiting and processing phases. Thus, "response time" here refers to the time until a transaction decides to commit (the actual response time may be longer because of logging).

Our model sets the ratio of the service time for 2PL transactions to that of SP transactions based on our earlier experimental results in Section 5.1, where we found locking overhead to be roughly 6 times the execution time for a transaction, resulting in a 7 : 1 service-time ratio. The workload determined the interarrival times. There was no a-priori bound on the degree of multiprogramming.

We conducted three experiments. The first uses SP as presented earlier in its non-parallelizable form. This experiment allows us to gain insight into the degree of parallelism 2PL needs to outperform non-parallel SP. The second experiment uses a modified form of SP that allows parallelism during those times when the only active transactions are read-only. Here, SP can take some advantage of the multiprocessor system; the experiment tells us how much. Finally, we considered a partitioned-database architecture in which the workload and data can be partitioned cleanly. Although this third experiment appears designed to make SP look good (and indeed it does exactly that), we note that the scenario of this experiment in fact conforms closely to several telecommunication applications [1, 2].

6.1 Unmodified SP versus 2PL

The first experiment we performed was to measure the effect on average response time of using additional processors. This we did for a workload generated from 1 day's worth of call data. The percentage of update transactions we used were 100% update transactions, 50%, 10%, and 1%.

The results of this first experiment are presented in Figure 6. First, note that the average response time using unmodified SP is unaffected by changes in the number of processors, or by changes in the degree of contention between transactions; it is 133 ms in all cases. This is not surprising given that unmodified SP processes transactions serially and cannot exploit additional processors. Second, we observe that the key factor affecting average response time in the case of locking is the degree of data contention among transactions. If contention is low, then the potential for parallelism is high, and strict 2PL can exploit that parallelism. If, on the other hand, contention is high, then the potential for parallelism is low, and many transactions must be executed serially (even under strict 2PL). This effect can be observed in Figure 6. Locking delivered better response times than SP only when the number of processors exceeded 7 or 10, for update-transaction rates of 1% and 10%, respectively. We observe that at low contention, the 2PL and SP curves cross when the number of processors corresponds to the service-time ratio and conjecture that this holds in general. At high-contention, we conjecture that the crossover point would occur at a larger number of processors as the probability of lock waits and deadlock adversely impact 2PL.

This experiment, though clearly designed to show SP at its worst, demonstrates that for small multiprocessors, it is better to run SP and leave all but one processor idle, rather than to use locking.

6.2 Modified SP: Concurrent Readers

For our second set of experiments, we consider an extended version of SP in which update transactions execute strictly serially as before, but read-only transactions execute in parallel. To achieve this extension, the SP algorithm described in Section 4 is adapted such that database access is controlled not by a mutex, but by a single-writer, multiple-readers semaphore (effectively, there is one shared/exclusive lock for the entire database). All other aspects of this set of experiments are the same as in Section 6.1.

The effect of this change is illustrated, for the same workload, in Figure 7 (the scale on the y-axis is now logarithmic). With 100% update transactions, average response time of SP is unaffected and remains 133 ms (there are no readers to benefit from additional concurrency). However, with just a few readers able to execute in parallel (for 50% and fewer update transactions), wait times are starkly reduced, and average response times improve accordingly, even with just a few processors, as shown in the figure. At low update rates, the average response time approaches the service time (of 2 ms). Once this level is reached, adding more processors does not help.

This series of experiments shows that with our minor modification, SP can take full advantage of multiprocessors when the percentage of update transactions is low to moderate, as is the case in most applications.[3] In our experimental scenario, a higher load could easily have been accommodated in a system offering 3 or more processors. Still better results may be possible if we had implemented an intelligent transaction-queueing scheme that attempts to rearrange the workload to run in parallel as many read-only transactions as there are processors.

6.3 A Partitioned Scenario

A limiting factor for SP in the series of experiments in Section 6.2 is the need to lock the entire database in order to run an update transaction. We have found this requirement to be excessive in many of the telecommunication applications that we have considered. Often, the database is partitioned, usually based on a key such as a customer identifier, mobile-id number, or a telephone number. This partitioning allows multiple sub-databases to coexist on a single multiprocessor machine. Each individual sub-database uses SP, but parallelism is possible among sub-databases.

Assuming that a database and its workload can be partitioned in this way, we next consider the case of running one database instance (which we call a "site") for every processor. Since each site has a separate database mutex, this partitioning introduces the possibility of parallelism not just among readers, but also among all transactions. We again repeated our earlier experiment, but with this extension. The results are illustrated in Figure 8 (the scale on the y-axis is again logarithmic). This figure shares the same y-axis as the previous two figures. We show results here only for SP as there is no change in how 2PL is modeled in the scenario for this experiment. Because of the additional possibility of parallelism involving update transactions, this technique demonstrates significant improvement in average response time for all workloads. We note that, with more than around three processors (and for the workload assumed here), the average response time using SP is lower, in all cases, than the service time using 2PL (14 ms).

[3]For high percentages of update transactions, even *unmodified* SP is superior to 2PL, as shown in Section 6.

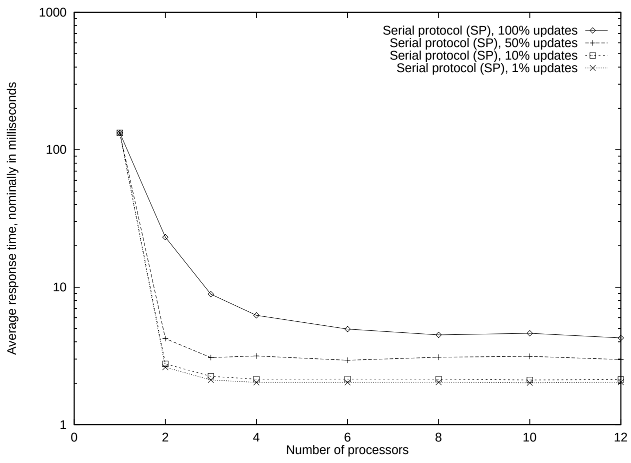


Figure 8: Average response time as a function of the number of processors: modified SP, partitioned database.

The scenario of this experiment corresponds closely to the model used in the Sunrise system described in [1, 2]. The Sunrise system processes "events" such as a phone call and maintains aggregate summary data in a DataBlitz database. When a single machine is insufficient for an application, the database is partitioned over several machines and a front-end "mapper" directs each event to the appropriate machine. Sunrise has provisions for managing queries across partitions and similar provisions can be made for SP in our current experimental environment. We did not incorporate this into our experiment since the frequency of such queries is extremely low and they appear as regular queries to each site.

7 Conclusion

This paper has investigated the issue of concurrency in main-memory databases. Focusing primarily on response-time requirements in telecommunication applications, we have proposed a serial protocol ("SP") for transaction execution in main-memory systems. The advantage of SP is that the overhead locking is all but eliminated, as too is deadlock, and the unpredictable response time that deadlock can cause. The novelty of SP lies in the use of timestamps and mutexes to allow parallelism between transaction execution and log flushing. It allows transactions to begin processing before their predecessors have completed flushing the log, while still ensuring that no transaction commits in the state of having read uncommitted data. By implementing SP atop the DataBlitz Main-Memory Database System, we were able to compare the performance of SP against that of strict 2PL. Our results indicate the potential for a significant performance gain—up to a factor of ten—in terms of both response time and throughput. These gains are in addition to those of a main-memory system over disk-based database systems.

We have described two extensions for multiprocessors, and presented a performance evaluation of these extensions. Perhaps surprisingly, our results suggest that—for certain workloads—the reduced overhead of a serial execution can even outweigh the advantages of moderate levels of parallelism in a multiprocessor. If contention is low, then 2PL can offer better performance since it can exploit the parallelism of the platform. For such situations, we proposed a modified version of SP that allows parallelism among read-only transactions and showed that it outperformed 2PL for levels of contention that generally occur in practice.

Our work was motivated by telecommunication applications, especially those of wireless systems. In Section 2, we described scenarios where operations, such as a hand-off of a mobile station between base stations, have time budgets in the tens of milliseconds. The performance numbers shown here indicate that SP can be the deciding factor in whether a general-purpose system like DataBlitz is suitable or whether a custom special-purpose system must be built.

Acknowledgments: We would like to acknowledge the help of Rajeev Rastogi, Phil Bohannon, Yuri Breitbart and the DataBlitz team for making this work possible, and for their many helpful insights.

References

[1] J. Baulier, S. Blott, H. F. Korth, and A. Silberschatz. A database system for real-time event aggregation in telecommunication. In *Proc. of the Int'l Conf. on Very Large Databases*, Aug. 1998. Industrial track paper.

[2] J. Baulier, S. Blott, H. F. Korth, and A. Silberschatz. Sunrise: A real-time event-processing system. *The Bell Labs Technical Journal*, 3(1), Jan-Mar 1998.

[3] J. Baulier, P. Bohannon, S. Gogate, C. Gupta, S. Haldar, S. Joshi, A. Khivesera, H. F. Korth, P. McIlroy, J. Miller, P. P. S. Narayan, M. Nemeth, R. Rastogi, S. Seshadri, A. Silberschatz, S. Sudarshan, M. Wilder, and C. Wei. Datablitz storage manager: Main memory database performance for critical applications. In *Proc. of the ACM SIGMOD Int'l Conf. on the Management of Data*, 1999. Industrial track paper.

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

[5] P. A. Bernstein, D. W. Shipman, and J. B. Rothnie. Concurrency control in a system for distributed databases (SDD- 1). *ACM Trans. on Database Systems*, 5(1):18–51, Mar. 1980.

[6] P. Bohannon, D. Lieuwen, A. Silberschatz, S. Sudarshan, and J. Gava. Recoverable user-level mutual exclusion. In *Proc. of the IEEE Symposium on Parallel and Distributed Processing*, 1995.

[7] P. L. Bohannon, R. R. Rastogi, A. Silberschatz, and S. Sudarshan. The architecture of the Dalí main memory storage manager. *The Bell Labs Technical Journal*, 2(1):36–47, Winter 1997.

[8] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, 1984.

[9] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowledge and Data Engineering*, 1992.

[10] V. Gottemukkala and T. Lehman. Locking and latching in a memory-resident database system. In *Proc. of the Int'l Conf. on Very Large Databases*, 1992.

[11] M. H. Graham. How to get serializability for real-time transactions without having to pay for it. In *Proc. of the IEEE Real-Time Systems Symposium*, 1993.

[12] J. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course: Lecture Notes in Computer Science 60.* Springer-Verlag, 1978.

[13] J. Gray. IMS FastPath. Lecture notes, Oct. 1980.

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

[15] L. Haas, W. Schang, G. Lohman, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):143–160, Mar. 1990.

[16] H. V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dalí: A high performance main memory storage manager. In *Proc. of the Int'l Conf. on Very Large Databases*, 1994.

[17] T. Lehman, E. Shekita, and L. Cabera. An evaluation of Starburt's memory-resident storage component. *IEEE Trans. on Knowledge and Data Engineering*, 4(6):555–566, Dec. 1992.

[18] T. J. Lehman and M. J. Carey. A recovery algorithm for high-performance memory-resident database systems. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 104–107, May 1987.

[19] R. Rastogi, P. Bohannon, J. Parker, A. Silberschatz, S. Seshadri, and S. Sudarshan. Distributed multi-level recovery in main-memory databases. *Distributed and Parallel Databases*, 6(1):41–71, 1998.

[20] K. Salem and H. García-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):161–172, Mar. 1990.