Controlled Lock Violation

Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, Alistair Veitch
Hewlett-Packard Laboratories

ABSTRACT

In databases with a large buffer pool, a transaction may run in less time than it takes to log the transaction's commit record on stable storage. Such cases motivate a technique called early lock release: immediately after appending its commit record to the log buffer in memory, a transaction may release its locks. Thus, it cuts overall lock duration to a fraction and reduces lock contention accordingly.

Early lock release also has its problems. The initial mention of early lock release was incomplete, the first detailed description and implementation was incorrect with respect to read-only transactions, and the most recent design initially had errors and still does not cover unusual lock modes such as 'increment' locks. Thus, we set out to achieve the same goals as early lock release but with a different, simpler, and more robust approach.

The resulting technique, controlled lock violation, requires no new theory, applies to any lock mode, promises less implementation effort and slightly less run-time effort, and also optimizes distributed transactions, e.g., in systems that rely on multiple replicas for high availability and high reliability. In essence, controlled lock violation retains locks until the transaction is durable but permits other transactions to violate its locks while flushing its commit log record to stable storage.

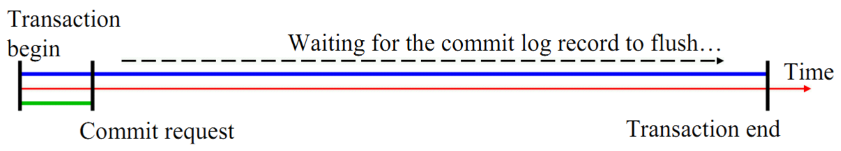

Figure 1. Lock retention times in traditional and optimized commit sequences.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *transaction processing: concurrency*.

Keywords

Transaction processing, DBMS, locking, concurrency.

1 INTRODUCTION

A simple database transaction takes 20,000 to 100,000 instructions [1], depending on the transaction logic, on index structures, on the quality of the implementation, and on compiler optimizations. For example, assuming 40,000 instructions per transaction, 1 instruction per CPU cycle on average, a 4 GHz CPU clock, and no buffer faults, a modern processor core can execute the transaction logic in about 0.01 ms. Committing such a transaction, however, may take much longer. If "stable storage" for the recovery log is a pair of traditional disk drives, the time to commit might approach 10 ms. If stable storage for the recovery log is provided by flash storage, commit time might be faster by two orders of magnitude, i.e., 0.1 ms, but it is still an order of magnitude longer than transaction execution.

If a transaction acquires locks right at its start, e.g., key value locks in a B-tree index, and holds them until transaction commit is complete, it retains the locks for about 0.01 ms while the transaction logic proceeds and for another 0.1 ms (or even 10 ms) during commit processing, i.e., after the transaction logic is complete. Given these relationships, it is not surprising that researchers have sought to reduce lock contention during commit processing. Retaining locks for only 0.01 ms, not 0.11 ms (or even 10.01 ms), should appreciably reduce lock contention, in particular for "hot spot" locks such as appending index entries to an index on transaction time or on an attribute with high correlation to transaction time, e.g., order number or invoice number.

Figure 1 illustrates the relationship between transaction execution time and commit duration. The diagram shows a factor of 10. In the traditional sequence of actions during commit processing, each transaction holds its locks for the entire time shown by the blue line above the time line. In a transaction with early lock release, lock retention is as short as the green line below the time line. The same performance improvement is achieved by a new technique, controlled lock violation, even in its simple form.

Multiple prior research efforts have described early lock release, which lets a transaction release its locks immediately after space for a commit record is allocated in the log buffer in memory. In other words, the locks are released before the commit record is flushed to stable storage and thus before the transaction becomes durable. Soisalon-Soininen and Ylönen [2] proved this technique correct, i.e., recoverable, but their proof does not address concurrency control and transaction isolation. Johnson et al. [3] and Kimura et al. [4] measured dramatic improvements in lock contention and in transaction throughput. Unfortunately, early lock release can also produce wrong results, including incorrect updates, because an implementation may fail to respect commit dependencies among participating transactions, as illustrated in **Figure 2**. In essence, one transaction must not publish or persist another transaction's update until the update is durable.

Recent work [4] describes a new variant of early lock release that avoids these wrong results yet preserves performance and scalability. The principal idea is to remove the locks from the lock manager early but to retain "tags" to convey commit dependencies among transactions. In some cases, commit dependencies are respected that do not exist, i.e., the technique is too conservative. As importantly, the tags do not fully optimize distributed transactions, e.g., in modern database systems that maintain multiple replicas for high availability and high reliability. Nonetheless, lacking other correct implementations, we take this as the prototypical representative of early lock release.

Controlled lock violation is a new and superior alternative to early lock release. It eschews early lock release and the need for tags. Instead, each transaction retains its locks until its commit process is complete. With respect to concurrency, controlled lock violation matches early lock release as it permits subsequent transactions to violate or ignore lock conflicts, but only in very controlled situations. More specifically, a subsequent transaction may acquire a lock that violates an existing lock if the transaction holding the lock is already in its commit process, i.e., if it has allocated a commit record in the log buffer. Thus, controlled lock violation enables concurrency in all situations in which early lock release (corrected with tags) enables concurrency.

Moreover, controlled lock violation can improve the concurrency not only of centralized, single-site transactions but also of distributed transactions. Thus, controlled lock violation can increase concurrency during commit processing in modern system designs that rely on replicas. The same issue – multiple separate recovery logs and two-phase commit – equally applies to many partitioned databases. For example, in a large database partitioned such that individual partitions (and their individual recovery logs) can easily move in order to achieve elastic scaling, transactions over multiple partitions require two-phase commit even when all affected partitions currently reside on the same server. In contrast to early lock release, controlled lock violation optimizes commit processing and lock conflicts in all those settings.

The contributions of our new technique are these:

1. In a single-site setting, controlled lock violation achieves the same transaction processing performance as early lock release, but without new theory, with fewer special cases, and without new data structures.
2. Controlled lock violation covers a broader set of lock modes than early lock release and all granularities of locking, including multi-granularity locking and key-range locking.
3. By retaining the original locks during commit activities, subsequent conflicting transactions can analyze the conflict more precisely than is possible in early lock release.
4. In a distributed setting, controlled lock violation optimizes both phases of two-phase commit. Thus, the conflicts of participant transactions can be reduced tremendously compared to two-phase commit execution. (Early lock release can optimize only the final phase of two-phase commit.)
5. For "canned transactions" coded as stored procedures, static code analysis can enable controlled lock violation even before a transaction's commit request.

The following section reviews related prior work, in particular on early lock release and two-phase commit. Section 3 introduces controlled lock violation and contrasts it to early lock release. Section 4 extends controlled lock violation to two-phase commit and Section 5 applies it to canned transactions. Section 6 compares the performance of controlled lock violation with that of early lock release in those cases when early lock release applies. Section 7 compares controlled lock violation with further related techniques, e.g., speculative execution. The final section contains our summary and conclusions.

2 RELATED PRIOR WORK

DeWitt et al. described early lock release back in 1984, albeit without an implementation [5]. Ailamaki and her research group have described a highly optimized implementation as well as the resulting speed-up in transaction processing [6]. Kimura et al. describe an oversight in the earlier designs as well as a technique to avoid over-eager commit of read-only transactions [4]. These efforts are discussed below, followed by a summary of other but unrelated cases of early lock release as well as some background information on distributed two-phase commit.

We assume concurrent and serializable transactions implemented by commonly used techniques, including write-ahead logging, log sequence numbers, record-level locking, logical "undo" by logged compensating updates rather than rigid physical "undo," an in-memory buffer pool for data and recovery log plus persistent block-access storage such as traditional disks, durability by forcing commit records to the recovery log on "stable storage," two-phase commit in distributed transactions, etc. The intention behind these assumptions is broad applicability of the work.

2.1 Main memory databases

An early paper [5] on implementation techniques for main memory database systems described early lock release as follows (original emphasis):

"A scheme that amortizes the log I/O across several transactions is based on the notion of a pre-committed transaction. When a transaction is ready to complete, the transaction management system places its commit record in the log buffer. The transaction releases all locks *without waiting for the commit record to be written to disk*. The transaction is delayed from committing until its commit record actually appears on disk. The 'user' is not notified that the transactions has committed until this event has occurred.

By releasing its locks before it commits, other transactions can read the pre-committed transaction's dirty data. Call these *dependent transactions*. Reading uncommitted data in this way does not lead to an inconsistent state as long as the pre-committed transaction actually commits *before* its dependent transactions. A pre-committed transaction does not commit only if the system crashes, never because of a user or system induced abort. As long as records are sequentially added to the log, and the pages of the log buffer are written to disk in sequence, a pre-committed transaction will have its commit record on disk before dependent transactions."

In this design, locks are released early, without distinction by lock mode, e.g., shared or exclusive. It is well known that read locks can be released early, e.g., during the pre-commit phase of distributed transactions with two-phase commit. Write locks, on the other hand, require longer retention in traditional transaction processing. Similarly, this design does not distinguish between read-only and read-write transactions among the dependent transactions. It turns out that the next design repeats this erroneous omission. (**Figure 2** shows an example error scenario.)

2.2 Early lock release in Shore-MT

Researchers at CMU and EPFL implemented early lock release as part of tuning Shore-MT [6], specifically to remove logging from the critical path of transaction processing by avoiding all lock contention while a transaction becomes durable.

Johnson et al. demonstrate "speedup due to ELR [early lock release] when running the TPC-B benchmark and varying I/O latency and skew in data accesses" while logging on different forms of stable storage, including logging to a slow traditional disk (10 ms for a write to stable storage) and logging to flash (write in 0.1 ms) [3]. Running TPC-B (which produces many lock conflicts) against a traditional logging device, their work shows that early lock release can speed up system throughput 30-fold; for a fast logging device, system throughput improves 3-fold.

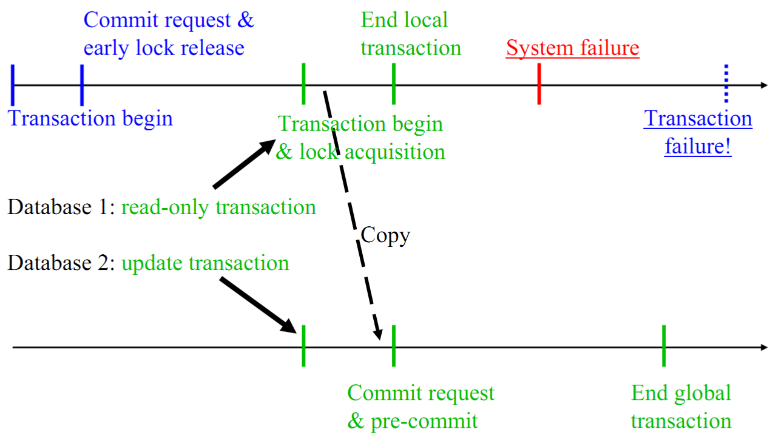

Figure 2. Bad database contents due to early lock release.

While the speed-ups are impressive, early lock release as originally described and implemented in Shore-MT can produce wrong results and even wrong database contents [4]. For example, consider a transaction T_0 acquiring an exclusive lock to update a database record, then releasing the lock after formatting a commit record in the log buffer, whereupon transaction T_1 acquires a lock on the same database record. If both transactions T_0 and T_1 are update transactions, then the sequencing of commit records enforces the commit dependency between them. What happens, however, if the dependent transaction T_1 does not require a commit record because it is a read-only transaction? In this case, it may read and report a value written by transaction T_0 before the commit of transaction T_1 is complete. Transaction T_1 may terminate successfully and thus commit to the user a value that may never exist in the database if the commit record of transaction T_0 is never saved on stable storage, e.g., due to a system crash.

For an example resulting in bad database contents, consider a transaction T_0 that updates a record in the database D_0 from old value 10 to new value 11 and then begins its commit activities. After T_0 has formatted its commit record in the log buffer and released its locks, transaction T_1 reads the value 11, copies it into another database D_1, and then performs its own commit. This commit is a two-phase commit between databases D_0 and D_1. Assume that database D_1 provides commit coordination and logs global commit records. Since transaction T_1 is a read-only transaction in database D_0, the pre-commit phase is sufficient there, the second commit phase is not required, and no log record is written in database D_0 for transaction T_1. Moreover, assume that stable storage for database D_1 is fast (e.g., flash storage) whereas stable storage for database D_0 is slower (e.g., a traditional disk). In this case, it is quite possible that transaction T_1 commits, including saving its final global commit record on stable storage, before transaction T_0 can do the same and become durable. If database D_0 crashes in the meantime, database D_0 rolls the data record back to value 10, whereas database D_1 contains value 11. Clearly, this result of "copying" from database D_0 to database D_1 is not acceptable transactional behavior.

Figure 2 illustrates this example. There are two parallel timelines for the two databases. One transaction (blue) in Database 1 reaches its commit point but never achieves durability due to a system failure (red); thus, this transaction and all its updates are rolled back during system recovery. The distributed transaction (green) attempts a two-phase commit prior to the system failure; the local read-only sub-transaction in Database 1 immediately responds and does not participate in the second commit phase. The coordinator may commit the global transaction even after one of the participating sites fails, because the local sub-transaction on Database 1 terminated successfully. At the end, the value copied from Database 1 to Database 2 will remain in Database 2 but be rolled back in Database 1, such that Database 2 is no longer a faithful copy of Database 1.

2.3 Early lock release in Foster B-trees

As part of an effort to prototype and evaluate a new B-tree variant, early lock release was also implemented in the context of prototyping Foster B-trees [7], although design and implementation of the locking subsystem are orthogonal to its usage in any specific data structure. This is the first implementation of early lock release to avoid the danger of wrong results and wrong database contents. The technique employed relies on tags in the lock manager's hash table [4]. When an update transaction releases a lock as part of early lock release, it leaves a tag containing the log sequence number of its commit record. Any subsequent transaction acquiring a lock on the same object must not commit until the appropriate log page has been written to stable storage.

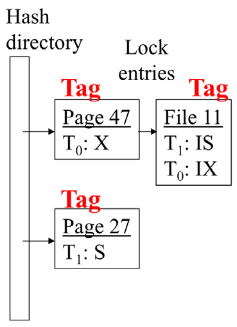

Figure 3. Hash table with tags.

Figure 3 illustrates where tags are attached in the lock manager's hash table. The tag contains a high water mark, i.e., a log sequence number. Any transaction acquiring any lock on the same object cannot commit until this log record is on stable storage.

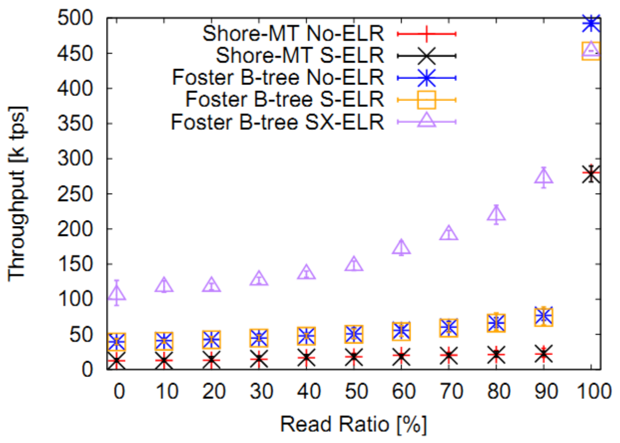

Figure 4. Performance effects of early lock release, © VLDB Endowment.

Figure 4, copied from [4], shows performance improvements due to early lock release in Shore-MT [6], including the version with Foster B-trees [7] instead of the original Shore-MT indexes. The relative performance of Foster B-trees and the original Shore-MT B-tree implementation is not relevant here as it is partially due to differences in representation and compression of records in pages. A comparison of fully implemented early lock release (purple triangles) with the same code minus early lock release (blue stars) demonstrates performance advantages across almost the entire range from an insert-only workload (read ratio 0%) to read-only queries (read ratio 100%). The performance advantage reverses for a read-only workload "because ELR checks the transaction's lock requests twice at commit time" [4]. This experiment logs on flash storage; the effects are similar or stronger with the recovery log on traditional disks.

The performance of early lock release in **Figure 4** is also indicative of the performance of controlled lock violation. This is because they are equivalent in their effect, at least in single-site cases (they are not in the context of two-phase commit). After all, one transaction releasing its locks is effectively the same as other transactions ignoring those same locks (in the same cases).

Unfortunately, after the introduction of tags had "repaired" early lock release, it needed yet another repair. Early lock release for intention locks requires tags just like absolute locks (i.e., non-intention locks), but they require a different kind of tag. For example, two transactions may both acquire intention locks on the same index but may touch different index entries. Thus, tags for intention locks should introduce commit dependencies only in conflicts with absolute locks but not with intention locks [4].

Of course, one wonders how many additional kinds of tags might be required. For example, Microsoft SQL Server uses multiple special locks such as 'insert' locks, 'bulk insertion' locks, 'schema stability' and 'schema modify' locks, etc. In other words, what seems needed is either a theory for deriving appropriate tags from lock modes (as in [8]) or a mechanism (including an appropriate theory and policy) that avoids tags altogether.

One may also wonder how early lock release may optimize distributed transactions with two-phase commit. It turns out, unfortunately, that the optimizations of early lock release apply to the final commit phase only but not to the pre-commit phase. In other words, a participant transaction may release its read-only locks during the pre-commit phase (as is well known) but it must retain its update locks until the local participant transaction has appended its final commit record to the log buffer. Lock retention time is reduced only by the time to write the final commit record to stable storage but not by the time for writing the pre-commit record during the initial phase or for communication and coordination during a two-phase commit. This is discussed further in Section 4.

2.4 Other cases of early lock release

In addition to the instances outlined above, other instances of early lock release have been mentioned in the literature or are used in real systems.

First, when a transaction employs "save points," e.g., between individual SQL statements, and if a transaction rolls back to an earlier save point, then it may release the locks it acquired after that save point. For example, if a transaction's first statement touched only index I_1 and the second statement touched index I_2, but the second statement failed and the transaction rolls back to the save point between the two statements, then all locks on index I_2 may be released during this partial rollback.

Second, when a transaction must abort, it may release all its read-only locks immediately. Only the write locks are required to protect the rollback actions. As the rollback logic proceeds, it may release locks as it completes the rollback ("undo", compensation) actions, i.e., in reverse order of lock acquisition.

Third, when the global coordinator in a two-phase commit requests that local participant transactions prepare for transaction commit, read-only participants may terminate immediately [9]. This includes releasing all their locks; obviously, those are all read-only locks. In addition, a read-only participant has no need to participate in the second phase of the two-phase commit. This is perhaps the most well known example of early lock release.

Fourth, read-write participants in a two-phase commit may release their read-only locks immediately upon receiving the request to prepare for a transaction commit. In other words, early lock release in two-phase commit is not dependent on the overall behavior of the local participant but on the nature of the locks – a transaction may release its read-only locks immediately after the commit request.

Fifth, even in the absence of distribution and two-phase commit, a transaction may release its read-only locks as soon as it has allocated space for a commit record in the log buffer. This technique requires that each transaction scans its set of locks twice, once to find (and release) its read-only locks and once again (at the very end) to release the remaining locks. Early lock release requires only one such scan, which releases all locks and inserts tags into the lock manager's hash table. These tags will be cleared out by subsequent transactions finding tags that expired. Controlled lock violation requires only the final scan after all commit activities; it may, however, use an earlier scan over all its locks in order to identify all transactions already waiting for those locks and able to resume with controlled lock violation.

	IS	*IX*	*S*	*X*	*SIX*
IS	ok	ok	ok		ok
IX	ok	ok			
S	ok		ok		
X					
SIX	ok				

Figure 5. Compatibility of hierarchical locks.

Sixth, existing locks may be modified in their lock modes in order to strip out the read-only aspects. For example, an SIX lock [10] may be reduced to a lock in IX mode. (An SIX lock combines a shared S lock and an intent-exclusive IX lock on the same object – the compatibility matrix for intention locks is shown in **Figure 5**.) Locks may also be reduced in their coverage or granularity of locking. For example, an exclusive X lock on an entire index may be reduced to an IX lock on the index combined with appropriate X locks on individual pages or index entries. Note that lock de-escalation can proceed without checking for lock conflicts and cannot create a deadlock. Thus, the X lock on the index is not released but instead it is much reduced in its strength or coverage.

Finally, key range locking raises special cases for the last two points. Some key range locks guarantee the absence of possible future database contents, specifically locks on the open interval between two existing key values in an ordered index such as a B-tree. Such locks are required to prevent phantoms and to ensure true serializability [11]. As special forms of read-only locks, they can be released prior to the transaction's commit activities. Other key range locks combine read-only and update parts, with only the update aspect required during commit activities. For example, a XS lock ("key exclusive, gap shared") [12] can be reduced to an XN lock ("key exclusive, gap free") as part of releasing all read-only locks. **Figure 6** shows a B-tree node with a few key values as well as some of these lock modes and their scope.

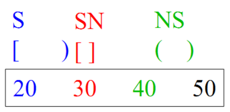

Figure 6. Lock scopes in key range locking.

2.5 Distributed commit

The traditional standard for committing a transaction over multiple nodes (or multiple recovery logs) has been two-phase commit, with all locks retained until all commit activities are complete. Thomson et al. [13] observe that "The problem with holding locks during the agreement protocol is that two-phase commit requires multiple network round-trips between all participating machines, and therefore the time required to run the protocol can often be considerably greater than the time required to execute all local transaction logic. If a few popularly-accessed records are frequently involved in distributed transactions, the resulting extra time that locks are held on these records can have an extremely deleterious effect on overall transactional throughput." Therefore, their design sequences transactions deterministically in order to "eliminate distributed commit protocols, the largest scalability impediment of modern distributed systems." In contrast, controlled lock violation reduces the "contention footprint" [13] of two-phase commit by judiciously violating locks held by committing transactions.

Kraska et al. [14] similarly aim to reduce lock conflicts by reducing commit communication. They "describe a new optimistic commit protocol for the wide-area network. In contrast to pessimistic commit protocols such as two-phase commit, the protocol does not require a prepare phase and commits updates in a single message round-trip across data centers if no conflicts are detected" [14]. An essential part of this protocol seems to be that each request for work includes a request to prepare for commit, quite comparable to the first phase of a traditional two-phase commit. In either case, a local transaction guarantees that it will abide by a subsequent global decision to commit or abort.

With this guarantee in place, a single message carrying the global decision can achieve the final commit, comparable to the second phase of a traditional two-phase commit. As each work request includes a request for commit preparation, each work step with local database updates requires a prepared-to-commit log record in each local recovery log. Moreover, a prepared local transaction is protected, i.e., it cannot be terminated, e.g., if it becomes part of a deadlock. Note that a local participant may subsequently receive additional work requests for the same transaction. If the additional work request includes database updates, it invalidates the prior commit preparation and requires a new one, including another suitable log record written to stable storage.

Kraska et al. measured round-trip response times between various regions on Amazon's EC2 cluster over a number of days [14]. Even communication links with response times around 0.1-0.2 seconds most of the time can spike to about 1 second and even to about 4 minutes. Note that these are message delays, not failures. If such delays affect communication during two-phase commit, and if transactions retain locks during commit coordination, spikes in communication latency likely produce spikes in lock contention.

Kraska et al. [14] propose to address the problem by imposing the cost to prepare a two-phase commit on every remote invocation within every transaction, whether or not network latency is currently spiking. In contrast, controlled lock violation addresses the problem by permitting subsequent transactions to violate locks and proceed while earlier transactions run their commit activities.

3 CONTROLLED LOCK VIOLATION

While early lock release removes locks from the lock manager before writing the commit record to stable storage, controlled lock violation retains the locks until transaction end. In order to match the concurrency and performance of early lock release, however, controlled lock violation permits new transactions to acquire conflicting locks in a controlled way. Specifically, in the simplest form of controlled lock violation, lock acquisition is permitted if the transaction holding the lock has appended its commit record to the log buffer. The advanced forms of controlled lock violation have more relaxed rules, e.g., for two-phase commit (Section 4) or for canned transactions (Section 5).

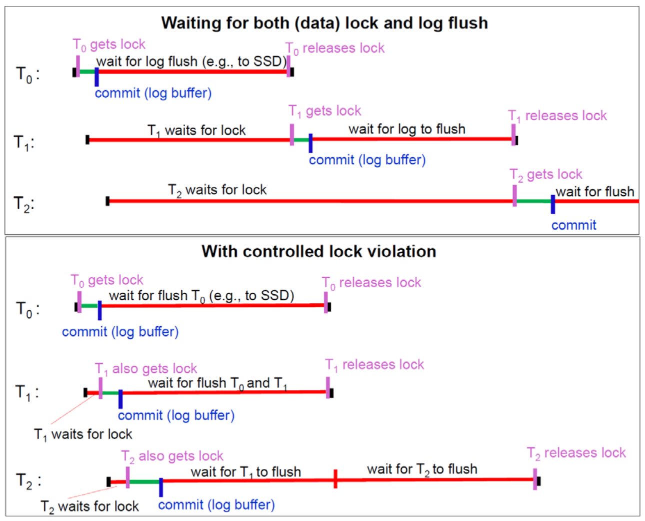

Figure 7. Sequences of three transactions (T_0, T_1, and T_2) without (top) and with (bottom) controlled lock violation.

Figure 7 illustrates a sequence of three transactions that all require a lock on the same database object. The green horizontal lines represent the time needed to execute transaction logic; the red horizontal lines represent time spent waiting for locks or for log records to flush to stable storage. The top diagram shows subsequent transactions acquiring the contested lock only after each preceding transaction has finished all its commit activities. The bottom diagram shows how subsequent transactions may acquire the contested lock and run the transaction logic immediately after the preceding transaction has started its commit activities, specifically after adding its commit log record to the in-memory log buffer. Note that in the lower figure, the commit log records of transactions T_0 and T_1 happen to go to stable storage in the same I/O operation. Obviously, controlled lock violation enable transactions to follow each other in faster succession. The "price" for this performance gain is that multiple transactions require rollback during recovery in case of a system failure; recall that a transaction with a commit log record in the log buffer can fail only if the entire system crashes.

3.1 Principles

It is well known that if a transaction T_1 reads an update by a preceding yet uncommitted transaction T_0, also known as a dirty read, then T_1 incurs a commit dependency on T_0. In other words, in a serializable transaction execution schedule, transaction T_1 may commit only after transaction T_0 commits. More generally, if a transaction T_1 ignores or violates a concurrency control conflict, e.g., conflicting locks, with another transaction T_0, a commit dependency results.

The only exception is a read-only lock held by the earlier transaction. In this case, a commit dependency is not required. On the other hand, a read-only lock acquired in conflict with an earlier lock causes a commit dependency. In fact, this is the case overlooked in early designs for early lock release.

Of course, should transaction T_0 subsequently read or update a data item after transaction T_1 has updated it, then a circular dependency results. In such cases, serializability becomes impossible and transaction T_1 must abort.

One can consider a rollback of transaction T_0 as such a case: modern recovery techniques essentially "update back" the changes of failed transactions rather than reverse byte for byte. Logical transaction compensation is required whenever the granularity of locking, e.g., key value locking, lets multiple concurrent transactions modify the same physical data structure, e.g., a B-tree node. If transaction T_0 fails and rolls back after transaction T_1 has updated a data item already updated by transaction T_0, then the "update back" action completes a circular dependency that can be resolved only by aborting transaction T_1. Thus, thinking of rollback as "updating back" nicely explains the commit dependency mentioned above.

3.2 Approach

In a traditional database system without early lock release, imagine a transaction T_0 holding a lock and another transaction T_1 requesting a conflicting lock. In this case, T_1 might wait for T_0's commit (and the implied lock release) or T_1 might abort T_0 (and thus force immediate lock release). For a decision between these alternatives, it compares the priorities of T_0 and T_1. It also checks the state of T_0. If T_0 has already added a commit record to the log buffer, or if T_0 is part of a distributed transaction in pre-commit state, or if T_0 is already aborting and rolling back, then T_0 is protected and T_1 cannot force T_0 to abort. Importantly, before transaction T_1 decides on a course of action, it must acquire information about transaction T_0 and its transactional state.

Controlled lock violation aims to achieve the performance advantages of early lock release (corrected with tags) but without releasing locks until all commit activities are complete. In controlled lock violation, transaction T_0 retains all its locks until its commit record indeed is on stable storage. However, if transaction T_1 comes along and looks at transaction T_0 (in order to decide its course of action, e.g., whether to abort T_0), and if T_0 already has a commit record in the log buffer (even if not yet on stable storage), then T_1 may proceed in spite of any locks T_0 still holds. Specifically, transaction T_1 may acquire its desired lock in spite of the conflict but it must take a commit dependency on T_0. Transaction T_0 retains its locks, so the lock manager may grant and hold conflicting locks in this case.

Actually, the commit dependency is required only if the lock held by transaction T_0 is an update lock, e.g., an exclusive lock, 'intent exclusive' lock, 'increment' lock, etc. In other words, violation of anything but a read-only lock creates a commit dependency. A write lock acquired in conflict with an earlier lock must be registered as a commit dependency, even if two update transactions and their commit records usually ensure a correct commit ordering. This is necessary because the write lock might be used only for reading: if the entire transaction remains a read-only transaction without any log records, then no commit record is required and the error case may occur unless the commit dependency is explicitly enforced.

3.3 Implementation techniques

In the above examples, the crucial task missing from transaction T_0's commit is writing the commit record to stable storage; the other tasks are acknowledgement of the commit to the user (or application) and releasing locks and other resources such as threads and memory. As soon as the log write is complete and thus the transaction durable, the commit dependency is resolved.

Thus, the commit dependency is really equivalent to a high water mark in the recovery log. When the recovery log has been written to stable storage up to and including this high water mark, transaction T_1 is free to commit. In other words, when transaction T_1 acquires a lock that conflicts with a non-read-only lock held by transaction T_0, which is permitted because transaction T_0 has already written its commit record to the log buffer, then the LSN (log sequence number) of T_0's commit record is registered as the high water mark governing the commit of T_1. Transaction T_1 cannot commit until the recovery log has been written to stable storage up to and including this log record.

If transaction T_1 is an update transaction, then it will eventually append its own commit record to the log buffer and the log on stable storage. In this case, the sequencing of commit records in the recovery log will ensure correct enforcement of the commit dependency. In other words, the commit dependency is almost meaningless and enforced as a matter of course.

If, on the other hand, transaction T_1 is a read-only transaction without any log records and thus without a commit record, and if transaction T_1 has incurred a commit dependency by violating a lock of a committing transaction T_0, then T_1 must wait until the commit record of T_0 is saved on stable storage. This is precisely what the high water mark enforces. Thus, this technique delays a committing read-only transaction only if a lock violation has indeed occurred and only as much as absolutely necessary.

If transaction execution is faster than writing a commit record to stable storage, perhaps even orders of magnitude faster, then transactions may form long chains of commit dependencies. In the extreme example of 0.01 ms execution time and 10 ms commit time, a chain of 1,000 transactions seems possible. If only one of those transactions were to abort, all subsequent ones must abort, too. In other words, this seems to be a bad case of "abort amplification" or "cascading abort."

One must recognize, however, that a traditional transaction processing system would have let none of the subsequent transactions acquire conflicting locks; thus, transactions aborted in this design would never even have started or made progress past the conflicting lock request. More importantly, all transactions whose locks may be violated have reached their commit point and have appended a commit record to the log buffer; thus, practically the only cause for a transaction failure is a system failure, in which all subsequent transactions would fail even without the chain of commit dependencies.

3.4 Comparison with early lock release

Controlled lock violation is similar to early lock release, at least in some aspects. First, a transaction T_1 can proceed as soon as an earlier, conflicting transaction T_0 has added its commit record to the log buffer in memory (or at least allocated space and an address for it). Second, most read-only transactions can commit instantly, except those that encounter a lock conflict and incur a commit dependency. (In this exception case, controlled lock violation is "slower" than early lock release because controlled lock violation behaves correctly whereas early lock release does not – see **Figure 2** and the corresponding discussion).

Controlled lock violation is also quite different from early lock release. First, transaction T_0 retains its locks in controlled lock violation as long as in traditional lock release, so it does not release any locks early. Second, locks can be violated while transaction T_0 commits, but only after T_0 has added its commit record to the log buffer. Third, hardly any new mechanism is required – for example, neither the allocation nor the clean up of tags are required. Fourth, if there is no conflict between committing and active transactions, controlled lock violation does not impose any restrictions or overheads, whereas early lock release forces committing transactions to install tags in the lock manager's hash table just in case a subsequent transaction might create a lock conflict. Finally, and perhaps most importantly, controlled lock violation requires very little if any new theory – violation of lock conflicts and resulting commit dependencies are already part of the traditional theory of concurrency control and recovery.

3.5 Combined locks

Retaining the original locks during commit activities is more precise than early lock release. This is because controlled lock violation preserves all lock information during commit activities whereas early lock release reduces that information to a tag. The detailed information about locks is particularly useful in cases with combined lock modes such as SIX (see **Figure 5**).

A conflict involving a combined lock mode does not always induce a commit dependency, because a combined lock may include read-only and update parts. Thus, controlled lock violation is more precise than early lock release, because early lock release reduces the conflict analysis to tags whereas controlled lock violation retains the precise lock. Early lock release would introduce commit dependencies in all of the following examples.

Consider, for example, transaction T_0 holding a combination lock, e.g., an SIX lock on a file (to read the entire file and update selected pages). A commit dependency is required only if a lock acquired by a later transaction conflicts with the update part. For example, if transaction T_1 requests an IS lock (in preparation of locking individual pages in S mode), there is no conflict at all because lock modes SIX and IS are compatible. (There is no conflict over this lock for the entire file; there may be a conflict if T_0 and T_1 attempt to lock the same page.) Further, if transaction T_2 acquires an IX lock on the file (in preparation of acquiring X locks on individual pages), which conflicts only with the read-only S part of the SIX lock of transaction T_0, no commit dependency is required. On the other hand, if another transaction T_3 acquires an S lock on the file, which conflicts with the IX part of the SIX lock of transaction T_0, then T_3 incurs a commit dependency on T_0. (In this particular case, transaction T_3 also incurs a commit dependency on T_2 due to the conflict between IX and S locks.)

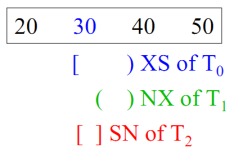

Figure 8. Conflicts and dependencies in key range locking.

Another example is key range locking in B-tree indexes. Consider the specific case illustrated in **Figure 8**. Transaction T_0 has locked key value 30 in XS mode ("key exclusive, gap shared") [12] and is in its commit phase with the commit record in the log buffer. If transaction T_1 acquires a violating NX lock ("key free, gap exclusive" – N stands for no lock), should that imply a commit dependency? The correct answer is 'no'. If transaction T_2 acquires a violating SN lock ("key shared, gap free"), should that imply a commit dependency? The correct answer is 'yes'. With early lock release and a tag in the lock manager's hash table, it is impossible to derive these correct answers. Controlled lock violation makes the correct decision easy: As the NX lock of transaction T_1 conflicts only with the read-only part of the XS lock held by transaction T_0, no commit dependency is required. On the other hand, since the SN lock of transaction T_2 conflicts with the update part of the XS lock held by transaction T_0, T_2 incurs a commit dependency on T_0. (Locks of modes SN and NX do not conflict; therefore, T_2 does not incur a commit dependency on T_1.)

3.6 Weak transaction isolation levels

Many databases and their applications run with a transaction isolation level weaker than serializability, e.g., "read committed." This isolation level permits releasing read-only locks immediately after use, e.g., when a scan advances to the next record. Write locks, on the other hand, are retained until the transaction ends.

If a transaction finds a desired data item locked, it waits. Only the "dirty read" transaction isolation level ignores existing locks (and consequently provides no meaningful transaction isolation). With controlled lock violation, this wait may not be required – a subsequent transaction may violate locks held by a transaction already in its commit activities. If a read-only lock of transaction T_1 violates a write lock of an earlier transaction T_0, it incurs a commit dependency. This is true even if the transaction T_1 releases this read-only lock soon thereafter, i.e., long before it attempts to commit.

A special case is an earlier transaction T_0 that deletes a record. Most implementations employ ghost records, also known as invalid records or pseudo-deleted records. These records still exist in the data structure but are marked as logically deleted. Their value is that a locked data item remains in the data structure until the transaction commits, thus avoiding special cases in the concurrency control code and simplifying rollback code. Any scan or query must ignore ghost records. If a transaction T_1 finds a locked ghost record, and if the lock-holding transaction T_0 is in its commit activities, then T_1 may violate the lock but it incurs a commit dependency. Transaction T_1 may commit only if transaction T_0 indeed commits the ghost record. If transaction T_0 fails and rolls back, transaction T_1 erroneously ignored a valid record and it therefore must abort as well.

3.7 Summary

In summary, controlled lock violation matches early lock release in the principal goal: when and where early lock release applies, controlled lock violation permits the same concurrency. However, controlled lock violation is simpler, e.g., with respect to data structures, yet it is more general, e.g., with respect to lock modes, and more accurate, e.g., with respect to combined locks and key range locking.

Early lock release and controlled lock violation can complement each other, for example in the following way. Once a transaction has determined its commit LSN, it may release its read-only locks. All remaining locks remain active but subsequent lock requests may violate them.

The early lock release part of this hybrid design can also weaken combined locks by removing the read-only component, e.g., from SIX to IX in hierarchical locking or from XS to XN in key-range locking. Moreover, it may notify waiting threads and transactions of released or weakened locks. Weakening SIX to IX locks seems particularly valuable in systems that let threads retain intention lock (e.g., IX) from one transaction to another, a technique known as speculative lock inheritance [15]. Incidentally, both early lock release and controlled lock violation can treat a U (upgrade [8]) lock as if it were an S lock.

Hybrid models are readily possible because in some sense, controlled lock violation differs from early lock release only in the information they retain: while tags in corrected early lock release retain a synopsis of the released lock, controlled lock violation retains the entire lock with all its detailed information. Retaining all lock information permits precision and functionality impossible with early lock release, unless each retained tag effectively replicates the released lock.

4 DISTRIBUTED TRANSACTIONS

As mentioned earlier, early lock release applies only to the final commit phase of two-phase commit, not to the pre-commit phase. In other words, early lock release cuts the lock retention time by the time for writing the final commit record but not by the time for communication during the two-phase commit and for writing the pre-commit record to stable storage.

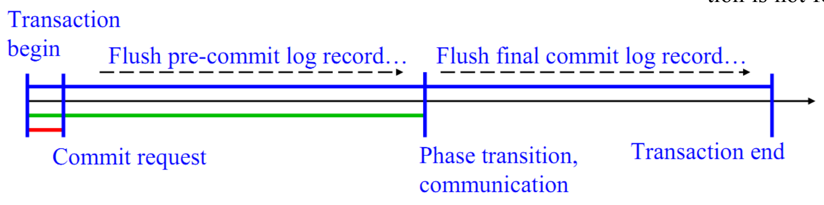
Transaction begin
Flush pre-commit log record…
Flush final commit log record…
Commit request
Phase transition, communication
Transaction end

Figure 9. Lock retention times in two-phase commit.

Figure 9 illustrates a distributed transaction and its two commit phases. The execution time is short but each commit phase requires communication and at least one write to stable storage. Traditional lock release retains locks until all commit activities are complete, shown by a blue line above the time line. Early lock release retains locks throughout the first phase of a two-phase commit, shown by a green line below the time line. In contrast, controlled lock violation enforces locks only during transaction execution, shown by a short red line at the bottom left.

Using the times for transaction execution and for flushing commit records to stable storage from the introduction, and assuming for simplicity negligible communication times, execution takes 0.01 ms and each commit phase takes 10 ms (logging on a pair of traditional disks) or 0.1 ms (logging on flash memory). Traditional commit processing holds all locks for 20.01 ms (log on disks) or 0.21 ms (log on flash); early lock release holds all locks for 10.01 ms or 0.11 ms; and controlled lock violation enforces locks for only 0.01 ms (independent of the log device). In other words, with the log on a pair of traditional disks, effective lock retention times are 2,000 times shorter than with traditional commit processing and 1,000 times shorter than with early lock release; with the log on flash memory, effective lock retention times are 20 or 10 times shorter, respectively.

Specifically, imagine an update transaction T_1 that is a local participant in a distributed transaction coordinated by remote transaction T_0, and another transaction T_2 that requests a lock conflicting with a lock held by T_1. With early lock release, T_2 must wait until T_1 has added its final commit record to the log buffer. With controlled lock violation, once the local transaction T_1 has received the request for the first commit phase, T_2 may acquire a conflicting lock. Of course, T_2 incurs a commit dependency on T_1 and thus on T_0. If those fail, T_2 must roll back as well.

4.1 Implementation of commit dependencies

Implementation of commit dependencies in distributed settings are somewhat complex. Fortunately, the commit dependencies required here are always within a single site (node). Thus, fairly simple mechanisms suffice.

A recent detailed description of such mechanisms was published, for example, by Larson et al. [16]. They call their design a "register-and-report approach" in which "T1 registers its dependency with T2 and T2 informs T1 when it has committed or aborted." Essential to the implementation is a counter (similar to a reference counter) and a set data structure capturing dependent transactions. They explain further:

"To take a commit dependency on a transaction T2, T1 increments its CommitDepCounter and adds its transaction ID to T2's CommitDepSet. When T2 has committed, it locates each transaction in its CommitDepSet and decrements their CommitDepCounter. If T2 aborted, it tells the dependent transactions to also abort by setting their AbortNow flags. If a dependent transaction is not found, this means that it has already aborted. Note that a transaction with commit dependencies may not have to wait at all - the dependencies may have been resolved before it is ready to commit. Commit dependencies consolidate all waits into a single wait and postpone the wait to just before commit." [16]

4.2 Performance effects

Perhaps a concrete example is best to illustrate the advantages of eliminating lock conflicts during both phases of a two-phase commit. If the transaction logic runs 0.01 ms (e.g., 40,000 instruction cycles on a core running at 4 GHz) and each commit phase runs 0.1 ms (to force a commit record to flash storage), then early lock release improves the contention footprint by almost a factor of 2 (0.21 ms ÷ 0.11 ms) whereas controlled lock violation improves the time with lock conflicts by more than a factor of 20 (0.21 ms ÷ 0.01 ms). If stable storage is realized with traditional disks, and if the time to force a log record to stable storage is 10 ms, then the factor for early lock release remains at about 2 (20.01 ms ÷ 10.01 ms) whereas the factor is about 2,000 (20.01 ms ÷ 0.01 ms) for controlled lock violation.

If locks can be violated immediately after a distributed transaction begins its commit sequence, lock contention during the two-phase commit sequence may cease to be a concern for performance and scalability. In other words, the major concern about or argument against two-phase commit loses weight and credibility. By removing lock conflicts during communication and coordination of two-phase commit, controlled lock violation may substantially contribute to increased use of two-phase commit with distributed transactions and thus to the consistency and reliability of distributed systems.

4.3 Summary

Both early lock release and controlled lock violation reduce the effective lock retention times during two-phase commit. Neither improves the elapsed time, communication time, or logging effort during commit processing. However, since early lock release pertains only to the final phase of the two-phase commit, its improvement of effective lock retention times is very small in comparison to that of controlled lock violation.

5 CANNED TRANSACTIONS

In some cases, controlled lock violation may be advantageous even before the user (or application) requests a transaction commit for the transaction holding the lock. In general, such lock violation is a bad idea. For example, if transaction T_0 needs and acquires a lock, transaction T_1 violates this lock, and then T_0 performs another action that requires the same lock again, then transaction T_0 needs to violate the lock held by T_1 and transactions T_0 and T_1 have mutual, i.e., circular, commit dependencies on each other. Only aborting transaction T_1 can resolve this situation.

If, however, it is certain that transaction T_0 will not require again a specific lock that it holds, then another transaction T_1 may violate this lock. For example, a "canned" transaction T_0 may run a stored procedure, that stored procedure may consist of multiple statements, and each statement may touch its own set of tables, i.e., disjoint from the tables in other statements. All of these are not unreasonable assumptions as many stored procedures satisfy them. When they apply, then another transaction T_1 may violate any lock from an earlier statement. The precise condition is that locks may be violated if neither the current nor any future statement might need them.

```
Begin transaction
Update accounts set balance += …
Update accounts set balance −= …
Insert activities values (…)
Commit transaction
```

Figure 10. A canned transaction.

Figure 10 shows source code for a very simple stored procedure. It moves money from one account to another and then inserts a record of it in a table of activities. After the first two statements, the table of accounts will not be touched again, except perhaps to roll back the updates in case of a transaction failure, e.g., due to a deadlock. Thus, while the third statement is still executing, a later transaction may violate the locks on the accounts table still held by an active transaction. Even in this simple example, controlled lock violation during one of three statements reduces the time with lock conflicts for the accounts table by one third (not including lock retention after the commit request). In other words, in addition to eliminating lock contention while a commit record is written to stable storage, controlled lock violation can reduce lock contention even further.

Note that controlled lock violation of read-only locks does not incur a commit dependency. In other words, controlled lock violation of a read-only lock has no negative effect at all. Thus, for tables touched only by a single statement of a stored procedure, controlled lock violation gives the semantics and consistency of full serializability but with the contention footprint and with the lock conflicts of "read committed" transaction isolation.

The tables involved in each statement can easily be extracted from the source code of the stored procedure. If disambiguation of table names requires a binding based on the user invoking the stored procedure, such static analysis might not be possible, in particular if tables and views may have multiple names or aliases. Cases requiring user-specific name resolution are discouraged in practice, because these cases also prevent pre-compilation, cached query execution plans, and compile-time query optimization. Therefore, static analysis is usually possible. It might focus on tables and materialized views (i.e., objects of the logical database design) or on indexes and partitions (i.e., objects of the physical database design). In the former case, it is sufficient to analyze the request syntax; in the latter case, query execution plans must also be considered.

Gawlick and Kinkade [17] wrote: "Consider, for example, an ultra-hot spot: a counter updated by every transaction. To achieve a high transaction rate, we want a transaction to be able to access the counter without waiting for any other transaction. We also want to guarantee the integrity of the counter, consistency with other parts of the data base, etc." Their solution introduced a new "change" verb to the language, i.e., a restriction to increment and decrement operations instead of general read and write operations. Note that an 'increment' lock is perfectly compatible with serializable transactions but it does not imply a read lock, i.e., it does not bestow the right to expect repeatable reads.

Wolfson [18] described a static analysis of stored procedures and an algorithm to identify points when explicit early lock release may be permitted without the danger of deadlocks. The analysis algorithm is, unfortunately, NP-complete and needs to run whenever the text of a stored procedure is altered. Moreover, the work is limited to single-site deployments and to shared and exclusive locks.

The present design offers an alternative to both prior approaches: a transaction may acquire a traditional write (exclusive) lock on the counter, increment its value, and then hold the lock until all commit activities are complete. Because such a transaction touches the counter only once, subsequent transactions may violate this lock immediately after the increment operation. In other words, controlled lock violation can offer practically the same concurrency but without the need for 'increment' locks.

In summary, controlled lock violation can happen even before the commit request in some cases that may be expected common in practice. It might for many applications combine the advantages of "read committed" and serializable transaction isolation levels. In many cases, it also enables traditional exclusive locks with the concurrency of special 'increment' locks.

6 PERFORMANCE EVALUATION

We implemented and measured controlled lock violation in the context of Shore-MT [6], an experimental database system.

6.1 Implementation

We implemented controlled lock violation in Shore-MT by adding a binary flag to each transaction descriptor that indicates whether other transactions may violate the transaction's locks. Lock acquisition ignores conflicts with locks held by transactions with this flag set when determining if locks can be granted. The flag starts unset and is set at the same point early lock release would release locks, namely once the commit record has been allocated in the log buffer.

We also modified the lock acquisition code so that when a transaction violates another transaction's lock, the acquiring transaction advances its high-water mark to equal or exceed the holding transaction's commit LSN. A transaction's high water mark (part of the preexisting implementation of early lock release) is the highest LSN that must be flushed to stable storage before a read-only transaction is allowed to commit and return data to a client. The preexisting code to implement this delay takes less than 60 lines of C++ including comments and debugging code but not tests. Such delays affect only read-only transactions.

The present implementation omits many possible optimizations. For example, it maintains a transaction's high-water mark even when a lock acquisition violates a read-only lock and it fails to wake up waiting transactions when lock violation becomes possible. (We expect to have this remedied very soon.) We also have not yet implemented controlled lock violation in the context of two-phase commit or of canned transactions.

6.2 *Results*

In the following, we report the performance of controlled lock violation versus two variants of early lock release (releasing S locks only and releasing both S and X locks) for the industry standard TPC-B benchmark, which models simple database transactions. To cover a range of logging delays, we logged to a RAM disk but added an extra delay ranging from 0.1 ms to 10 ms to simulate a range of stable storage devices. The database data itself was stored on disk and our experimental machine is a 4-socket Intel Xeon X7542 machine running at 2.7 GHz with 24 cores.

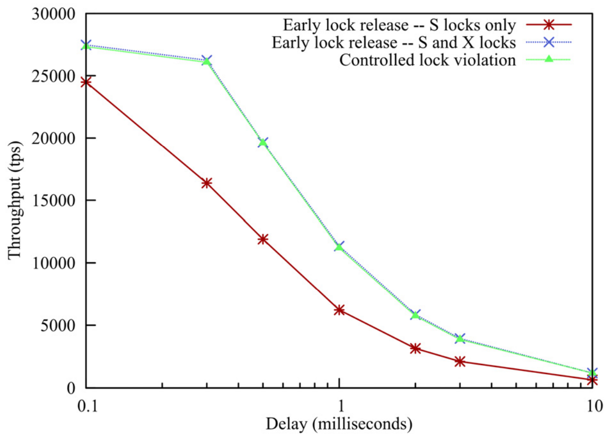

Figure 11. 24 cores, 24 threads.

Figure 11 and **Figure 12** show the transaction throughput (transactions per second) for competing commit processing techniques as the extra delay is varied. **Figure 11** is for 24 threads (one per core) while **Figure 12** is for 48 threads (two per core). Extra threads can perform useful work when other threads are blocked, waiting for locks or, more likely in the case of controlled lock violation, for their commit record to be flushed to stable storage.

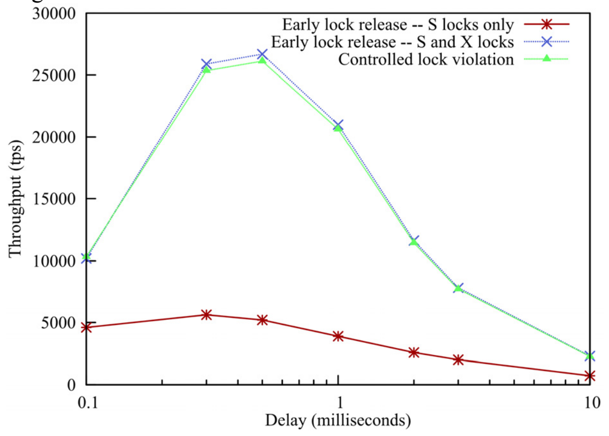

Figure 12. 24 cores, 48 threads.

Note that the relatively low performance for 48 cores at small delay is expected, since the threads block only for a very short time and hence the additional threads merely add overhead (e.g., due to true contention, context switching overhead, etc.).

As can be seen, both controlled lock violation and early lock release for S and X locks outperform the traditional approach of releasing read-only locks early by large factors for sizable delays: up to 5× at 1 ms commit delay and 2× at 10 ms commit delay. The improvement is smaller at small delays: up to 2.2× at 0.1 ms commit delay and 4.5× at 0.3 ms commit delay. The performance of controlled lock violation and early lock release for S and X locks are not significantly different; a T-test fails to reject the null hypothesis at p=0.05. However, as our implementation of controlled lock violation is missing several possible optimizations (see Section 6.1) it may be possible for controlled lock violation to exceed the performance of early lock release.

Since controlled lock violation sometimes delays read-only transactions, we also experimented with a variant of TPC-B with 70% of the transactions made read-only by omitting their writes. The results, shown in **Figure 13** for 48 threads, again show sizable improvements for controlled lock violation.

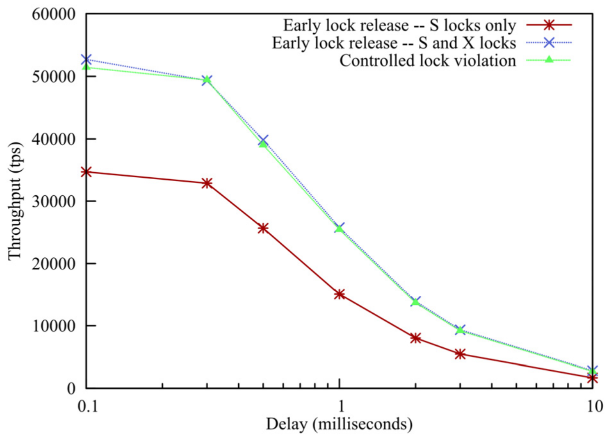

Figure 13. 24 cores, 48 threads, 70% read-only.

7 DISCUSSION

Both early lock release and controlled lock violation are specific forms of speculative execution. The speculation risk, however, is very small, as both techniques require that the earlier transaction reaches its commit point and formats a commit record in the log buffer before speculative execution begins. Nonetheless, with all other things equal, a system should schedule (process) a transaction without commit dependency ahead of one with, or one with fewer commit dependencies ahead of one with more. In other words, a transaction with commit dependencies should progress only if there no is work pending that is less speculative, or when the processing resources would remain idle except for speculative work. This is particularly true in the advanced forms of controlled lock violation discussed in later sections.

Outside the research area of database management, Nightingale et al. [19] investigated scenarios similar to ours. The issue was the same – guaranteeing durability yet hiding the latency until information had been written to disk. Their context was a file system and its "sync" operations. Their solution acknowledges the write operation immediately and lets the invoking process continue while preventing the process from communication. In other words, speculative execution of the process enabled further local progress but in the unlikely case that the sync operation failed, the process can be rolled back and the speculative execution is wasted. Thus, by relying on inexpensive checkpoints and speculative execution, their system achieved performance similar to asynchronous writes to disks but the semantics of synchronous writes.

Early lock release and controlled lock violation similarly rely on speculative execution with an extremely low risk of failure and of wasted work.

Both early lock release and controlled lock violation also seem related to optimistic concurrency control, in the sense that new transactions may proceed ignoring existing transactions and their concurrency footprint. Both techniques, however, are forms of pessimistic concurrency control, i.e., locking. Both techniques employ traditional locking techniques for synchronization atomicity or concurrency control – only during the phase that ensures durability of transaction, i.e., flushing the commit record to stable storage, are locks released or violated.

This reliance on pessimistic concurrency control is very deliberate. Carey's extensive simulation studies (e.g., [20] and multiple subsequent studies) point out that the mechanism of concurrency control matters little in systems with few conflicts. In systems with many conflicts, avoiding wasted work by early detection of conflicts is the most important determinant of performance (other than a fine granularity of locking). Controlled lock violation wastes work (due to "cascading aborts" or "abort amplification") only if a transaction fails after reaching its commit point.

In a single-site or single-log system, a transaction starts cascading aborts only if a system failure (crash) occurs in the time between adding a commit record to the log buffer and completing the write to stable storage, i.e., the time required for a single write operation. Moreover, in a traditional system that retains locks until a transaction is durable, the transactions failed due to cascading abort would not have started. In other words, controlled lock violation is pessimistic with respect to synchronization atomicity but it is optimistic with respect to durability once a transaction has started its commit activities. The risk and extent of wasted work are miniscule compared to the performance advantage of early lock release and controlled lock violation.

In a system with multiple logs and thus with two-phase commit, frequent failures during the first commit phase would suggest delaying controlled lock violation to the second phase. If, however, most transactions that start their commit activities also finish them successfully, the risk of cascading aborts is low. Similar considerations apply to controlled lock violation prior to the commit point of canned transactions – if failures are frequent, controlled lock violation should be restricted to the commit phase.

Early lock release and controlled lock violation avoid multi-version concurrency control and its complexities by delaying any conflicting transaction until the lock-holding transaction has finished its updates. Thus, there is no need for multiple versions of the same record. However, should a rollback be required, e.g., because a distributed transaction fails in the second phase of its two-phase commit, it is possible that multiple transactions need to roll back, which could take a single record back in time through multiple states. Nonetheless, at any one time, there is only a single version of each record in the database.

8 SUMMARY AND CONCLUSIONS

In summary, the simple form of controlled lock violation is comparable to early lock release. Early lock release can boost the performance of transaction processing by a small factor or even an order of magnitude as shown in **Figure 4**. In those cases in which early lock release applies, controlled lock violation enables the same amount of additional concurrency compared to traditional commit processing. However, there are multiple reasons to prefer controlled lock violation over early lock release.

First, controlled lock violation is simpler and more robust because it has fewer special cases. It applies to all lock types – any lock may be violated and violation of any but a read-only lock induces a commit dependency. Even after multiple rounds of correction and improvement, early lock release still does not cover 'increment' locks. The same is true for more specialized locks that are used in real database systems, e.g., 'bulk insertion' or 'schema stability' or 'schema modify' locks in SQL Server, as well as the various special designs for key range locking and their lock modes. Controlled lock violation is a simple, consistent solution for all of these lock types, easy enough for implementation, quality assurance, and maintenance by most software engineers working on data management code.

Second, controlled lock violation is more precise than early lock release with tags. For key range locking, a precise separation of concurrency and conflict is required, in particular for hot spots known in many databases and their indexes. Controlled lock violation carries that precision to the commit dependency, whereas early lock release may introduce a commit dependency where none is required.

Third, controlled lock violation works well with two-phase commit. With early lock release, a transaction can avoid lock conflicts if an earlier transaction is in the final phase of the two-phase commit. With controlled lock violation, a transaction can avoid lock conflicts during both phases, i.e., already during the initial phase. Thus, the opportunity for lock conflicts during two-phase commit is much smaller with controlled lock violation than with early lock release. It could be argued that this reduction in lock conflicts takes most of the performance costs out of two-phase commit. For example, it might enable immediate (as opposed to eventually consistent) maintenance of all copies in a system relying on replicas for high reliability and high availability. These effects and their implications require further research.

Fourth, controlled lock violation applies even before the user (or application) requests a commit. In a "canned" transaction with a fixed sequence of statements, locks can be violated prior to the commit request if neither the current nor any future statement might need the locks, and this can be based on static analysis of the stored procedure and its source code.

In conclusion, we believe that controlled lock violation matches the promise of early lock release but it is simpler, more accurate, and more general. In other words, we believe it is superior in multiple dimensions.

ACKNOWLEDGEMENTS

Hideaki Kimura implemented the tag-based design for early lock release with read-only transactions. Eric Anderson pointed out the possible relationship between controlled lock violation and optimistic concurrency control. Haris Volos helped with the performance evaluation. Martin Scholl alerted us to the recent work on MDCC. Gary Smith suggested some concerns regarding dispatching waiting transactions and transaction isolation levels weaker than serializability. Barb Peters suggested some improvements in style and grammar.

We thank all developers and researchers of the Shore-MT team at EPFL, CMU and UW-Madison for making the Shore-MT code-base available. We especially thank Anastassia Ailamaki and her students for their thoughtful work on early lock release.

REFERENCES

[1] Anon et al.: A measure of transaction processing power. Datamation, 1 April 1985. Also: http://research.microsoft.com/~gray/papers/AMeasureOfTransactionProcessingPower.doc; retrieved 11/10/2012.

[2] Eljas Soisalon-Soininen, Tatu Ylönen: Partial strictness in two-phase locking. ICDT 1995: 139-147.

[3] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, Anastasia Ailamaki: Aether: a scalable approach to logging. PVLDB 3(1): 681-692 (2010).

[4] Hideaki Kimura, Goetz Graefe, Harumi Kuno: Efficient locking for databases on modern hardware. ADMS workshop, Istanbul August 2012.

[5] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, David A. Wood: Implementation techniques for main memory database systems. ACM SIGMOD 1984: 1-8.

[6] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, Babak Falsafi: Shore-MT: a scalable storage manager for the multicore era. EDBT 2009: 24-35.

[7] Goetz Graefe, Hideaki Kimura, Harumi Kuno: Foster B-trees. ACM TODS 37(3) (2012).

[8] Henry F. Korth: Locking primitives in a database system. JACM 30(1): 55-79 (1983).

[9] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, Bruce G. Lindsay: Transactions and consistency in distributed database systems. ACM TODS 7(3): 323-342 (1982).

[10] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger: Granularity of locks in a large shared data base. VLDB 1975: 428-451.

[11] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger: Granularity of locks and degrees of consistency in a shared data base. IFIP Working Conf. on Modeling in Data Base Management Systems 1976: 365-394.

[12] Goetz Graefe: A survey of B-tree locking techniques. ACM TODS 35(3) (2010).

[13] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, Daniel J. Abadi: Calvin: fast distributed transactions for partitioned database systems. ACM SIGMOD 2012: 1-12.

[14] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden: MDCC: multi-data center consistency. Submitted for publication, available at http://mdcc.cs.berkeley.edu.

[15] Ryan Johnson, Ippokratis Pandis, Anastasia Ailamaki: Improving OLTP scalability using speculative lock inheritance. PVLDB 2(1): 479-489 (2009)

[16] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, Mike Zwilling: High-performance concurrency control mechanisms for main-memory databases. PVLDB 5(4): 298-309 (2011).

[17] Dieter Gawlick, David Kinkade: Varieties of concurrency control in IMS/VS Fast Path. IEEE Database Eng. Bull. 8(2): 3-10 (1985).

[18] Ouri Wolfson: An algorithm for early unlocking of entities in database transactions. J. Algorithms 7(1): 146-156 (1986).

[19] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn: Rethink the sync. ACM TOCS 26(3): (2008).

[20] Michael J. Carey, Michael Stonebraker: The performance of concurrency control algorithms for database management systems. VLDB 1984: 107-118.