Efficient Locking Techniques for Databases on Modern Hardware

Hideaki Kimura*
Brown University
hkimura@cs.brown.edu

Goetz Graefe
Hewlett-Packard Laboratories
goetz.graefe@hp.com

Harumi Kuno
Hewlett-Packard Laboratories
harumi.kuno@hp.com

ABSTRACT

Traditional database systems are driven by the assumption that disk I/O is the primary bottleneck, overshadowing all other costs. However, future database systems will be dominated by many-core processors, large main memory, and low-latency semiconductor mass storage. In the increasingly common case that the working data set fits in memory or low-latency storage, new bottlenecks emerge: locking, latching, logging, and critical sections in the buffer manager. Prior work has addressed two of these – latching and logging. This paper addresses locking and proposes new mechanisms optimized for modern hardware. We devised new algorithms and methods to improve all components of database locking, including key range locking, intent locks, detection and recovery from deadlocks, and early lock release. Most of the techniques are easily applicable to other database systems. Experiments with Shore-MT, the transaction processing engine we used as the implementation basis, show throughput improvement by factors of 5 to 50.

1. INTRODUCTION

We are developing a new transactional storage manager specifically optimized for modern hardware. Traditional databases are optimized to balance CPU operations against the bottleneck of disk I/O. However, databases on modern hardware face a future dominated by many-core processors, large main memory, and low-latency semiconductor mass storage, and thus face different bottlenecks. As illustrated in Figure 1 with data from [8], when all data in a database fits into main memory, about 80% of the CPU cycles are accounted for by B-tree lookup and latching, logging, locking, and buffer management. Furthermore, the numbers on the figure reflect single-threaded execution. The rapidly increasing number of CPU cores on modern hardware causes physical and logical contention in each of these modules, making bottlenecks even more severe.

The right-hand side of Figure 1 sketches our goal – to reduce each of these four costs by an order of magnitude. In our prior work [6], which uses Shore-MT [10] as an implementation base, we developed a novel B-tree variant that addresses the latching bottleneck, *Foster B-trees*. Foster B-trees achieved a 3× to 6× higher throughput than Shore-MT for a highly concurrent workload, reducing the cost of latching from a dominant bottleneck to a negligible overhead (for more details, we refer readers to [6]). We also leverage the logging optimizations devised by the Shore-MT team [11]. In particular, their "flush-pipeline" and "consolidation array" eliminate the bottlenecks of log flush and contention in the log manager, largely reducing the overhead of logging.

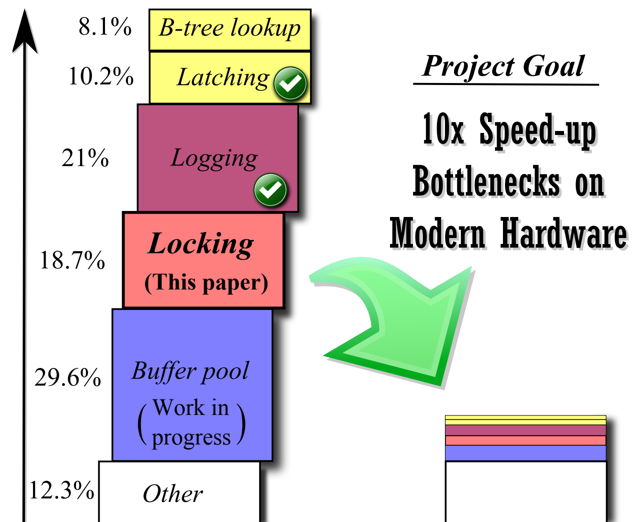


Figure 1: OLTP overhead breakdown [8] and our goal.

In this paper, we focus on the locking bottleneck. We address all aspects of locking in databases with four techniques presented in Sections 2 through 5: (1) efficient key range locking, (2) lightweight intent locks, (3) a deadlock detection and recovery scheme that enhances an existing adaptation of the Dreadlocks technique [12] to databases, and (4) an improvement on early lock release algorithm [11] to ensure correct treatment of read-only transactions, respectively. Each section discusses prior work relevant to the technique described in that section. Two of these techniques (intent locks and deadlock detection) focus on shortening code paths, while the other two (key range locking and early lock release) focus on increasing concurrency. In a different dimension, two of the techniques (deadlock detection and early lock release) focus on quickly identifying and resolving instances of contention, whereas the other two (intent locks and key range locking) take a more global perspective, reducing the potential for contention in the first place.

Section 6 gives an empirical evaluation, demonstrating that our storage manager performs up to 50× faster than the base Shore-MT. Finally, Section 7 presents our summary, conclusions, and our ongoing work for further speeding up databases on modern hardware.

*This work was done while the author was at HP Labs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. This article was presented at:
The Third International Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures (ADMS'12).
Copyright 2012.

2. KEY RANGE LOCKING

Locking is a common mechanism to separate concurrent transactions. Most locking schemes distinguish *share* (S) mode from *exclusive* (X) mode (N is *no-lock*). As shown in Table 1, S-locks are compatible with each other while X-locks are exclusive.

Table 1: Basic Locks.

	N	S	X
N	✓	✓	✓
S	✓	✓	×
X	✓	×	×

Serializable transaction isolation protects not only existing records and key values but also non-existing ones. For example, after a query such as "Select count(*) From T Where T.a = 15" has returned a count of 0, the same query within the same transaction must return the same count. In other words, the absence of key value 15 must be locked for the duration of the transaction. Key range locking achieves this with a lock on a neighboring existing key value in a mode that protects not only the existing record but also the gap between two key values.

For our implementation, we extended previous descriptions of the locking protocol with specific locking instructions for cursors, in particular the end points of inclusive and exclusive, ascending and descending cursors.

2.1 Prior Techniques

ARIES/KVL and ARIES/IM: Mohan et al. developed ARIES/KVL [14], a locking protocol to ensure serializability by locking neighboring keys. In addition to the newly inserted key, it locks the next key until the new key is inserted and locked. ARIES/IM [16] reduces the number of locks for tables with multiple secondary indexes. However, in some cases, these designs unnecessarily reduce concurrency, because they do not differentiate locks on keys from locks on ranges between keys.

Key range lock modes: Lomet [13] defined a set of key range lock modes that were later implemented in Microsoft SQL Server. The design began to separate between *key* and *range*. In the design, a lock mode can consist of two parts, range mode and key mode.

The key mode protects an existing key value while the range mode protects the range down to the previous key, aka 'next-key locking.' For example, the 'RangeX-S' lock protects a range in exclusive mode and a key with share mode. Two locks on the same key value are compatible if the first components (protecting the range) are compatible and the second components (protecting the key value) are compatible.

Central to Lomet's design are 'insert' locks on key ranges, which conflict with shared and exclusive locks but not with each other. In that sense, they are similar to 'increment' locks on key values. Like Mohan's design, Lomet's lock protocol also requires locks with 'instant' duration, i.e., locks that are held only for an instant. When a user transaction inserts a new key value into a B-tree index, it acquires a 'range insert' lock for instant duration.

Since the design treats key and range not completely orthogonally, however, it is sometimes too conservative. For example, it lacks a 'RangeS-N' mode (N stands for 'not locked'), which would be the ideal lock to protect the absence of a key value. For example, suppose an index on column T.a has keys 10, 20, and 30. One transaction issues 'Select * From T Where T.a = 15', which leaves a 'RangeS-S' lock on key value 20. When another transaction issues 'Update T Set b = 1 Where T.a = 20', its lock request conflicts with the previous lock although these transactions really lock different database contents and actually do not violate serializability. Similarly, the design lacks 'RangeS-X' and 'RangeX-N' modes.

Protecting a non-existing key value beyond the highest existing key value in a B-tree leaf requires locking a key value in the next leaf node. Thus, traditional key range locking sometimes adds I/O operations merely for the purpose of locking.

Data-oriented execution: The Data-Oriented execution (DORA) approach suggested by Pandis et al. [19] eliminates physical lock contentions by assigning threads for logical partition of data. However, the tie between execution model and the locking protocol has some assumptions and limitations analogous to PLP [20] and PALM [21] for latching. Also, the work is orthogonal to concurrency of lock modes because they eliminate only *physical* lock contentions, not *logical* contentions (logical concurrency).

2.2 Locking protocol

Graefe [4] defined a comprehensive and orthogonal set of key range lock modes to improve simplicity as well as concurrency. Our new transactional storage manager is the first to implement these lock modes and thus permits a first empirical evaluation and comparison of the design.

The design applies the theory of multi-granularity and hierarchical locking to key values, gaps (open intervals) between two neighboring key values, and the half-open intervals comprising a key and a gap. The locks modes simply are combinations of no-lock, shared, and exclusive modes applied to a key value and gap. There is no need for 'insert' or 'instant' locks. While Mohan's and Lomet's designs employ 'next-key locking,' i.e., a gap between keys is locked using the next-higher key value, this design uses 'prior-key locking' instead.

Table 2: Key range lock modes.

	N	S	X	NS	NX	SN	SX	XN	XS
N	✓	✓	✓	✓	✓	✓	✓	✓	✓
S	✓	✓	×	✓	×	✓	×	×	×
X	✓	×	×	×	×	×	×	×	×
NS	✓	✓	×	✓	×	✓	×	✓	✓
NX	✓	×	×	×	×	✓	×	✓	×
SN	✓	✓	×	✓	✓	✓	✓	×	×
SX	✓	×	×	×	×	✓	×	×	×
XN	✓	×	×	✓	✓	×	×	×	×
XS	✓	×	×	✓	×	×	×	×	×

Table 2, copied from [4], shows our key range lock modes. These protect half-open intervals [A,B). For example, 'SX' mode (pronounced 'key shared, gap exclusive') protects the key A in shared mode and the open interval (A,B) in exclusive mode. S is a synonym for SS, X for XX.

Using these locks, locks on key values and gaps are truly orthogonal. In the example above, the first transaction and its query 'Select * From T Where T.a = 15' can lock key value 10 (using prior-key locking) in 'NS'-mode (key free, gap shared). Another transaction's concurrent 'Update T Set b = 1 Where T.a = 10' can lock the same key value 10 in 'XN'-mode (key exclusive, gap free). Lomet's design would take a lock in RangeS-S mode and thus have lower concurrency than our NS-lock, which allows concurrent updates on neighboring keys because NS and XN are compatible.

Like many other B-tree implementations [5], ours uses 'ghost records' (aka 'pseudo-deleted' records) to simplify key deletion and transaction rollback of user transactions. A logical deletion,

Algorithm 1: INSERT locking protocol.

```
Data: B: B-tree index, L: Lock table
Input: key: Inserted key
leaf_page = B.Traverse(key); // hold latch(*)
slot = leaf_page.Find(key);
if slot.key == key then //Exact match
    L.Request-Lock(key, XN);
    if slot is not ghost then
        return (Error: DUPLICATE);
    leaf_page.Replace-Ghost(key);
else //Non-existent key. In this case, slot is the previous key
    if slot < 0 then //hits left boundary of the page
        L.Check-Lock(leaf_page.low_fence_key, NX);
    else
        L.Check-Lock(slot.key, NX);
    begin System-Transaction
        leaf_page.Create-Ghost(key);
    L.Request-Lock(key, XN);//lock the ghost
    leaf_page.Replace-Ghost(key);
```

(*) To reduce the time latches are held, all lock requests are *conditional*. If denied, we immediately give up and release latches, then lock unconditionally followed by a page LSN check.

Algorithm 2: SELECT locking protocol.

```
Data: B: B-tree index, L: Lock table
Input: key: Searched key
leaf_page = B.Traverse(key); // hold S latch
slot = leaf_page.Find(key);
if slot.key == key then //Exact match
    L.Request-Lock(key, SN);
    if slot is not ghost then
        return (slot.data);
    else
        return (Error: NOT-FOUND);
else //Non-existent key
    if slot < 0 then //hits left boundary of the page
        L.Request-Lock(leaf_page.low_fence_key, NS);
    else
        L.Request-Lock(slot.key, NS);
    return (Error: NOT-FOUND);
```

instead of actually erasing a record, merely marks it invalid by flipping a 'ghost bit' in the record header. Ghost records do not contribute to query results, but the key of a ghost record does participate in key range locking just like the key of a valid record. Ghost records are removed after the user transaction is committed, e.g., when the space is needed for an insertion.

As described in [3], each B-tree node in our design contains two fence keys that define the lowest and highest permissible key value in that node. One fence key is inclusive, the other is exclusive. When required, fence keys are ghost records that cannot be removed. The fence keys of the root node have values $-\infty$ and $+\infty$. The fence keys in all other nodes are copies of separator keys in ancestor nodes. Fence keys enable efficient key range locking as well as inexpensive and continuous, yet comprehensive, verification of the B-tree structure and all its invariants [6].

When a query searches for a non-existent key that sorts below the lowest valid key value in a leaf page but above the separator key in the parent page, a 'NS'-lock on the low fence key in the leaf is used. Since the low fence key in a leaf page is exactly equal to the high fence key in the next leaf page to the left [3], key range locking works efficiently across leaf page boundaries.

Point queries: Algorithms 1 and 2 show the pseudo code for INSERT and SELECT queries (UPDATE and DELETE are omitted due to lack of space). We first check if the corresponding leaf page has the key we are searching for. If so, a key-only lock mode such as SN and XN suffices. This is true even if the existing record is a ghost record. Furthermore, the existing ghost record speeds up insertion, which only has to turn it into a non-ghost record (toggling the record's ghost bit and overwriting non-key data).

The design uses system transactions for creating new ghost records as well as all other physical creation and removal operations. As a system transaction does not modify the database's logical contents, only the database's physical representation, it takes no locks (e.g., there is no need for 'insert' locks) but only checks for existence of conflicting locks (e.g., an existing shared lock on the key range). Moreover, a system transaction runs in the same thread as the invoking user transaction, commits without flushing the log to stable storage and remains committed even if the invoking user transaction rolls back. This separation of user transactions and system transactions greatly simplifies and speeds up internal code paths.

Table 3: Lock modes for cursors.

Cursor Type	Ascending		Descending	
Boundary Type	Incl.	Excl.	Incl.	Excl.
Initial (Exact Match)	S*	NS	SN*	N
Initial (Non-Existent)	NS	NS	S	S
Initial (Non-Existent; Fence-Low)	NS	NS	NS	NS
Next / Page Move	S (SN if last)		S (NS if last)	

Range queries: Range queries such as 'Select * From T Where T.a Between 15 And 25' need cursors protected by lock modes as shown in Table 3. The lock mode to take depends on the type of cursors (ascending or descending) and on the inclusion or exclusion of of boundary values in the query predicate (e.g., $key > 15$ or $key \geq 15$). When a cursor initially locates its starting position, it either takes a lock on the existing key (exact match), or the previous key (non-existent) or the low fence key of the page). Then, as it moves to next key or next page, it also takes a lock on the next key (including fence keys).

Because a cursor takes a lock for each key, the overhead to access the lock table is relatively high. This is the reason why the locks marked with (*) in Table 3 are more conservative than necessary. For example, an ascending cursor starting from exact-match on A could take only an 'SN' lock on A and then convert to an 'S' lock on the same key when moving on to the next key. However, this doubles the overhead to access the lock table. Instead, the storage manager takes the two locks at the same time to reduce the overhead at the cost of slightly lower concurrency, which is the same trade-off as the coarse-grained lock discussed in next section. In Section 6, we evaluate the performance of this scheme.

3. INTENT LOCKS

The previous section discussed the design of record-level locking. Although such granular locks guarantee correctness with maximal concurrency, they might cause an unacceptable overhead for a transaction that reads or writes a large number of records. Hence, most DBMSs also provide coarse-grained *intent locks* in order to support both coarse and fine-grained locks on the same data.

However, intent locks may become a source of *physical* contention as a large number of concurrent threads simultaneously acquire and release them. In this section, we present our novel intent locking algorithm, *Lightweight Intent Locks*, which is a substantially simpler, faster, and more scalable implementation of intent locks for modern hardware.

3.1 Prior Techniques

Intent locks allow scanning and bulk-modification transactions to protect their accesses with only a single lock, dramatically reducing overhead compared to taking potentially millions of record-level locks. With the exception of absolute locks, intent locks are compatible and cause no *logical* contention.

Granularity of locks: Table 4, below, shows coarse locks invented by Gray et al. [7]. A transaction takes an IS or IX lock on a high-level object (e.g., Index) in addition to record-level locks. These *intent* locks are compatible each other. On the other hand, *absolute* locks such as S, X and SIX (S+IX) on higher levels are taken by table scan or lock escalation, which conflict with all the other transactions.

Table 4: Intent locks.

	N	S	X	IS	IX	SIX
N	✓	✓	✓	✓	✓	✓
S	✓	✓	×	✓	×	×
X	✓	×	×	×	×	×
IS	✓	✓	×	✓	✓	✓
IX	✓	×	×	✓	✓	×
SIX	✓	×	×	✓	×	×

Physical contention on intent locks: However, each transaction must create a lock request for intent locks in the lock table and then remove it when it commits. Further, because intent locks are coarse locks, a large number of transactions will take intent locks on the same object (e.g., disk volume intent lock). This causes physical contention on the lock bucket because all operations in a lock bucket are synchronized by mutexes.

Johnson et al. [9] observed that the physical contention on intent locks causes a significant bottleneck on many-core architectures where tens or hundreds of concurrent threads might be racing on the same intent lock. They proposed Speculative Lock Inheritance (SLI) to eliminate the contention. SLI allows a transaction to inherit intent locks from the previous transaction on the same thread, bypassing both the acquisition and release of intent locks.

Even in this scheme, all transactions must release intent locks upon absolute lock requests because otherwise absolute locks would never be granted. In other words, a single lock escalation flushes out all inherited intent locks. All concurrent threads then must re-acquire intent locks, again causing physical contention.

The root problem here is the inefficiency and low scalability of intent locks. Instead of working around it by inheriting locks, our goal is to improve the performance of intent lock itself.

3.2 Lightweight intent lock

In order to address the problem, we devised Lightweight Intent Lock (LIL), a dramatically simpler and faster intent lock scheme designed for modern hardware. The core idea of LIL is to maintain, instead of lock queues, a set of lightweight counters that is separate from the main lock table. Almost all code paths in LIL are extremely short and only use lightweight spinlocks. Mutexes are

Algorithm 3: Lightweight intent lock: Request-lock.

```
Data: G: Global lock table, P: Private lock table
Input: i: Index to lock, m: lock mode (IS/IX/S/X)
if P[i].granted[m] is already true then
    return;
while Until timeout do
    begin Critical-Section{G[i].spinlock}
        if m can be granted(*) in G[i] then
            ++G[i].granted_counts[m];
            P[i].granted[m] = true;
            return;
        if m ∈ {S, X} then
            Leave a flag to announce absolute locks(*);
        base_version = G[i].version;
    cur_version = base_version;
    while cur_version == base_version do
        Conditional-Wait(G[i].mutex, 1 millisec);
        cur_version = G[i].version;
```

(*) To not starve absolute locks, we also maintain the count of waiting locks for each lock mode and give absolute locks higher priority. For example, we do not grant IX locks while S lock request is waiting.

Algorithm 4: Lightweight intent lock: Release-lock.

```
Data: G: Global lock table, P: Private lock table
Input: i: Index to release
if P[i].granted[m] are all false then
    return;
begin Critical-Section{G[i].spinlock}
    ++G[i].version;
    foreach m do
        if P[i].granted[m] == true then
            --G[i].granted_counts[m];
if Released any lock that was blocking other thread then
    Broadcast(G[i].mutex);
```

used only when an absolute lock is requested.

The design of LIL is based on the observation that intent locks have a limited number of lock modes and infrequent logical contention. Therefore, a simpler method is more appropriate than the heavyweight mutexes, lock queues and point-to-point communications used in the main lock table for non-intent locks.

Global and private lock table: LIL maintains a private lock table (PLT) for each transaction in addition to a single global lock table (GLT) shared by all transactions. The PLT records intent locks obtained by the transaction. As the PLT has per-transaction data, the transaction can efficiently access its own PLT without synchronization. The GLT records the count of granted lock requests for each lock mode (S/X/IS/IX). The GLT has no lock queues, thus the only inter-thread communication is a broadcast.

Lock acquisition and release: Algorithm 3 and 4 show the pseudo code for lock acquisition and release in LIL.

When a transaction requests an intent lock, it first checks its own PLT. If it already has a granted lock that satisfies the need, it does nothing. Otherwise, it atomically checks the GLT and increments the counter for the desired lock mode. Whether the lock request is immediately granted or not, the critical section for this check is

extremely short and a spinlock suffices, avoiding mutex overheads.

If the request is not immediately granted, we wait for the release of locks preventing this request from being granted. We use mutex lock for this situation to avoid wasting CPU cycles, but this happens only when there is an absolute lock request or this transaction is requesting an absolute lock.

Upon lock release, we do the reverse, atomically decrementing the counter. If other requests on the lock were waiting on the current transaction, we broadcast a message to all waiting threads.

As a mutex broadcast after the critical section might cause a race condition, each waiting thread wakes up after a short interval (e.g., 1ms) and repeatedly checks the *version* of the lock and tries again if some transaction released a lock.

Deadlock prevention in LIL: Regarding deadlocks, we employ a simple timeout policy to *prevent* deadlocks in LIL. Waits on intent locks happen much less often than non-intent locks. In addition, the latency of scanning and bulk-modification transactions, which are the only types of transactions that could cause waits in LIL, is much higher than that of other types of transactions. Thus, delayed deadlock detection due to the timeout policy does not have a significant impact on overall performance. Hence, a transaction is simply aborted when its wait time exceeds a certain threshold. To avoid repeatedly aborting a scanning transaction, we assign a longer timeout for absolute lock requests.

LIL can cause neither deadlocks nor long waits; therefore there is no chance of deadlocks between locks in LIL and locks in the main lock table (we discuss how we handle deadlocks in the main lock table in the next section). In other words, the main lock table does not need to be aware of intent locks at all. Thus, LIL simplifies not only intent locks but non-intent locks and shortens their critical sections. We evaluate the performance of LIL in Section 6.

4. DEADLOCKS

Deadlocks can cause major bottlenecks in databases when two or more competing transactions permanently block each other from acquiring locks that they both need in order to succeed. For example, concurrent transactions may acquire locks in an order that causes a cycle in wait-for relationships. Deadlock resolution requires at least one of the transactions causing the deadlock to release locks. This involves either a partial rollback, a lock de-escalation, or most commonly a transaction termination. The throughput of the entire system depends on the efficiency and accuracy of the deadlock detection and resolution algorithms.

In this section, we describe our deadlock detection algorithm, which extends the native Shore-MT adaptation of the Dreadlocks [12] technique for the database context.

4.1 Prior Techniques

Traditional methods: Deadlock handling methods in databases are grouped into two categories [22]. The deadlock *prevention* approach ensures that the database never enters into a deadlock — for example our timeout policy for intent locks described in Section 3.2. Another approach is deadlock *detection*, which detects deadlocks when they happen and resolves the situation by rolling back some transactions.

The downside of the prevention approach is that prevention algorithms such as wound-wait and wait-die proactively catch suspicious situations and rollback transactions, which may result in false positives. A timeout algorithm with long waits causes fewer false deadlocks, but delays resolution of the situation.

The main drawback of the detection approach is its high computational overhead. Constructing a wait-for graph and detecting a cycle in it requires checking all transactions' status and probing the lock queues they are waiting on. This is especially problematic on many-cores due to synchronization between threads. Atomically constructing or maintaining such a global data structure requires either a long blocking or a large number of mutex calls for synchronization, both unacceptable overheads in a many-core architecture. Thus, a common practice is to run detection only periodically (e.g., once a minute), but, again, this delays deadlock detection.

Agrawal et al. [1] evaluated the performance of each approach by simulation. One of the conclusions was that there is no one-size-fits-all solution among them. The best algorithm and its parameters highly depend on characteristics of transactions that are usually unknown a priori.

Dreadlocks Algorithm: Shore-MT [10] implemented the Dreadlocks technique (notice the "r") to address the issues. To the best of our knowledge, Shore-MT is the first and to this date the only DBMS that employs Dreadlocks. Dreadlocks is a spinlock-based algorithm invented by Koskinen et al. to efficiently detect deadlock when running on many-core hardware [12]. The basic idea is that each core (thread) recursively collects the identity of cores it depends on (dependency). If the core finds itself in the dependencies, there must be a cycle in the wait-for relationships. A similar idea has been explored in deadlock detection in distributed databases [18]. In order to efficiently collect dependencies on many-core hardware, the Dreadlocks algorithm maintains only local information in each core, called a *digest*, which is asynchronously propagated to the other cores waiting for that core.

Figure 2 and Table 5 illustrate how the Dreadlocks technique works. Each core starts with only itself in the digest. At the second step, each core checks another core it is waiting for and adds its digest to its own digest, for example A adds B to its digest. At the third step, C, D, and E find more digests in the cores they are waiting for because of the previous propagation. As a consequence, C, D, and E all contain each other in their digests. Hence, at the last step, C finds itself in D's digest, detecting a deadlock. D and E detect deadlocks accordingly. As for A, no deadlock is raised because B's digest does not contain A.

The Dreadlocks technique spins on each waiting thread or each waiting lock request. In the case of per-lock spins on each lock, the technique works well only when the number of locks is smaller than the number of cores. However, in databases, there are usually many more locks than threads and cores. Hence, per-thread spinning is the more practical choice.

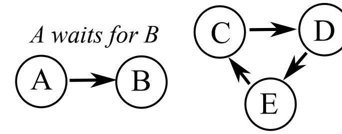


Figure 2: Examples of Livelock and Deadlock.

Table 5: How the Dreadlocks technique works.

Step	Digests				
	A	B	C	D	E
1	{A}	{B}	{C}	{D}	{E}
2	{A,B}	{B}	{C,D}	{D,E}	{E,C}
3	{A,B}	{B}	{C,D,E}	{D,E,C}	{E,C,D}
4	{A,B}	{B}	Deadlock!	Deadlock!	Deadlock!

The Dreadlocks can either fully store the identity or use a Bloom filter to probabilistically (but without false negatives) detect deadlocks. As the maximum number of concurrent transactions is not known a priori, Bloom filters are more appropriate. They are also much more efficient to read and compute than other forms of full dependency information such as the 'String' transmitted in [18].

4.2 Dreadlocks technique for databases

The Dreadlocks approach is highly scalable, simple, and applicable to many situations. It finds deadlocks very accurately and quickly with low overhead because of its simplicity and local-only spin accesses. However, Dreadlocks have a few issues to be adapted for use in database systems.

In this section, we describe our implementation of Dreadlocks, which is based on the Shore-MT's implementation of Dreadlocks with the following additional improvements: (1) capture deadlocks caused by lock conversions (e.g., from shared to exclusive), (2) avoid pure spinning without causing false deadlock detection, (3) resolve deadlocks more efficiently in the existence of flush pipelining.

Lock modes, queues, and conversions: First, the original Dreadlocks algorithm assumes that each lock has a single "owner". Each waiter takes the union of its digest and that of the owner of the lock. In databases, locks have various lock modes such as S, X, and NX. Furthermore, a threads may *convert* an already-granted lock. Suppose a thread A took an SN lock on some key. Another thread B then took an SN lock on the key. The thread A then tries to convert the lock to XN mode, becoming a waiter due to B's SN lock (SN and XN are incompatible). *Even a granted lock might be also a waiter*, thus database locks do not have a good notion of "owner".

Also, in order to achieve fair scheduling, database lock requests are placed in lock queues which grants locks in the request order. In the above example, if another thread C comes with a request for an SN lock, it must *not* be granted because of the waiting conversion request by A. If the lock manager were to grant an SN lock to C (and other subsequent requests), A might starve. Thus, C should wait until B and then A finish and release their locks. Hence, a database lock might have to wait even though all of the granted locks in the queue are compatible with the request.

Algorithm 5 shows a Dreadlocks-based deadlock detection algorithm adapted to databases, initially implemented in Shore-MT. The difference from the Shore-MT implementation is that it can handle deadlocks caused by lock conversions.

Each thread repeatedly collects the digest and computes the union of its own *fingerprint*. The fingerprint of a thread is a randomly and uniquely chosen n bits out of m bits, the size of Bloom filters. For example, let $n = 3, m = 512$. The fingerprint of thread A [1] might be (12,43,213) while that of B might be (43,481,500). The initial digest of Transaction A is an array of 512 bits. All bits are OFF except 12th, 43rd, and 213th bits which are ON. When we take the union of the other digests, we simply compute bitwise OR.

Consider two threads A and B that both have lock requests on the same queue. If B precedes A in the lock queue, B has higher priority and A can be granted only when its requested lock mode is compatible with B's **requested** (not only granted) lock mode. On the other hand, if A precedes B in the queue, A has priority and A can be granted as far as its requested lock mode is compatible with B's **granted** lock mode. In either case, if A's lock request cannot be granted because of B, B is said to be A's *dependency* and B's digest is added to A's digest. Further, if B's digest contains A's fingerprint, it implies a deadlock. Hence, either of the transactions is aborted, depending on the deadlock recovery policy.

Algorithm 5: Request-Lock with Dreadlocks

```
Data: L: Lock table
Input: xct: Transaction, key: Locked Key, m: Lock mode
struct request {x: transaction, gm: granted mode, rm:
requested mode, st: status};
queue = L.find_or_create_queue (key);
myreq = (x: xct, gm: N, rm: m, st: waiting);
Add myreq to queue or convert existing request;
while true do
    digest = xct.fingerprint;
    foreach req ∈ queue do
        if req == myreq then
            continue; // ignore myself
        if req precedes myreq then
            if compatible(req.rm, m) then
                continue; // ignore compatible predecessor
        else
            if compatible(req.gm, m) then
                continue; // ignore compatible successor
        if req.x.digest ⊇ xct.fingerprint then
            if myself should be rolled back then
                return (Error: DEADLOCK);
        digest = digest ∪ req.x.digest;
    if digest == xct.fingerprint then
        myreq.st = granted;
        myreq.gm = m;
        return (SUCCESS);
    xct.digest = digest;
```

Sleep on spin and backoff: Second, the original Dreadlocks approach used spinlocks. However, databases might have to process more concurrent threads than the number of cores. Suppose an ad hoc query arrives when there are already as many running threads as the number of cores. If the new query simply waits, its query latency could be severely affected — especially when the query is short and read-only (as often is the case with ad hoc queries). On the other hand, if we were to immediately run the query, a purely spin-based Dreadlocks would severely damage the overall throughput, greedily wasting CPU resources. This is an even more significant issue because databases have various background threads such as buffer pool cleaners and log flushers. Keeping all CPU cores busy might affect such critical operations.

A simple solution for this problem is to have each thread sleep after each spin, and this was in fact the approach implemented in Shore-MT. However, this caused frequent false deadlock detections in some cases.

For example, suppose thread A, B and C update the same resource. Let A currently hold an X lock on the resource. First B and then C request locks on the resource and start waiting; thus their digests contain A. To avoid wasting CPU cycles, B and C fall into a sleep. When A commits and releases locks, A wakes up B who will be granted the lock next. However, C is still asleep. Then, thread A starts another transaction and accesses the same resource. Because C has not yet refreshed its digest, A finds itself in the digest of C and aborts itself as deadlock. This repeats until C wakes from sleeping, wasting CPU cycles and lowering system throughput.

[1] We assign fingerprints per-thread instead of per-transaction because a transaction might be carried out by multiple parallel threads.

In the experimental section, we find that frequent false deadlocks rapidly reduce throughput as the number of concurrent threads increases, defeating the purpose of the sleep.

The problem here is that the digest of threads waiting on some lock becomes outdated when its dependency is released. In the pure spinning algorithm, such a digest is quickly refreshed and never causes false deadlocks. However, pure spinning wastes too many CPU cycles.

Our solution for this problem is to add *backoff* at lock release. Whenever we release a lock, we leave a flag on all threads waiting for the lock which tells the digest of the thread is outdated. Upon the next spin, such a thread is tentatively ignored from the digest computation to avoid false deadlocks. The flag is turn off by the marked thread itself when it wakes up next time and refreshes its digest. Rather than actually waking up all the waiting threads to make them immediately update the digest, this approach minimizes the overhead of lock release (which is the critical path of the highly contended resource). In Section 6, we observe that this sleep-and-backoff method prevents extreme performance degeneration against the number of concurrent threads.

4.3 Deadlock resolution for flush-pipeline

When we detect a deadlock, we must roll back a transaction to release its locks. The deadlock resolution policy affects the entire throughput because an inefficient policy keeps voiding the work each transaction made and might prevent the entire workload from proceeding. Shore-MT's policy is, like traditional approaches, to select the transaction that started most recently, hopefully thus rolling back the least amount of work while also avoiding starvation.

However, we observed that this policy is inefficient in the existence of flush pipelining. When the database is pipelining transactions, the cost of aborting one transaction is not only wasting its own work. To release locks after commit, a transaction has to make sure its log is flushed. Therefore, the aborted pipeline has to flush its logs before releasing its locks. This causes a substantial wait in the pipeline which would be otherwise free from flush waits. If we frequently and randomly keep aborting each pipeline, we lose the benefit of using flush pipelines.

Our solution for the issue is to consider the length of the pipelines, not the current transaction. When two transactions are in deadlock, we check their pipelines and compare the number of completed transactions in each pipeline. In Section 6, we observe that *kill-short* policy, which aborts the pipeline with fewer completed transactions, avoids repeated deadlocks and achieves up to 4× faster throughput than other deadlock resolution policies.

5. EARLY LOCK RELEASE

This section presents a novel variant of *Early Lock Release* (ELR), the technique that allows a transaction to release its locks as soon as it has formatted a commit record in the log buffer, i.e., before it flushes its commit record to a stable storage. The basic idea of ELR has been previously proposed and refined [2, 11, 23]. However, a straightforward implementation violates serializability for read-only transactions as detailed in Section 5.2. Obvious remedies have critical shortcomings, e.g., limiting ELR to S (shared) locks or adding commit delays to all read-only transactions.

We propose a simple, efficient and safe (serializable) implementation of ELR that addresses all of the issues above. In our experiments, we verified that such a complete ELR (which we call "SX-ELR") achieves 3×-10× better throughput compared to ELR of S-locks only (S-ELR) and without ELR. To the best of our knowledge, our work is the first to repair the serializability anomaly of a straightforward ELR scheme without holding all read-only query results, to verify it in an implementation, and thus to realize a serializable ELR of both S and X locks.

5.1 Prior Techniques

In the context of distributed databases, it has been known that S locks can be released right after the commit request. This also means that a read-only transaction can release all locks and immediately finish without interacting with logs at all [17], which is essential to ensuring low query latency for read-only queries. Such S-lock only ELR (S-ELR) is indeed always serializable because a transaction will never read any resources after its commit request.

DeWitt et al. [2] briefly mentioned the possibility of ELR for all kinds of locks by considering dependency between transactions. However, the paper does not mention the case where read-only transactions bypass logging. Although the obvious alternative is letting all read-only transactions wait for log flushing, it has the problem we discuss later in this section.

Johnson et al. [11] implemented ELR for all kinds of locks by releasing all locks as soon as the location of the commit log in the log buffer (LSN) is finalized, observing a significant performance improvement. This straightforward ELR guarantees serializable results for read-write transactions because a transaction log manager by its nature prevents them from returning results to users until their dependent transactions exit [11]. However, it is not serializable when there exists a read-only transaction which bypasses logging.

5.2 Anomaly of straightforward ELR

We begin by giving an example case where a straightforward ELR causes a non-serializable result, and then describe the principles to make it serializable.

Table 6 demonstrates such an anomaly. Consider a read-only transaction, A, and a read-write transaction, B. When B updates the tuple D3, the uncommitted data is protected by an X lock until B's commit. Then, ELR releases the X lock right after B requests commit. A proceeds to read D3 and immediately commits without log flush because it is a read-only transaction. Because the commit log of B has not yet been flushed, if a crash happens at this point, B is rolled back during recovery. However, the user already received D3 updated by B, which is not a serializable result.

The anomaly can cascade arbitrarily if the user does subsequent operations based on transaction A's result, e.g., inserting the value into another database. The root problem is that a read-only transaction never interacts with logs, thus simply doing SX-ELR allows them to publish uncommitted data that might yet roll back during recovery after a system failure.

Table 6: Non-serializable result with Straightforward SX-ELR.

Step	Read-Only Xct A	Read-Write Xct B	Log Buffer LSN Latest	Log Buffer LSN Durable
1 Locks		Write D3 D3 (X-granted)	130	100
2 Locks	(Read D3) D3 (S-wait)	Write J5 D3,J5 (X-granted)	150	100
3 Locks	(Read D3) D3 (S-wait)	Commit Request D3,J5 (X-granted)	200	100
4 Locks	Read D3 D3 (S-granted)	(Flush wait) ELR	230	100
5	Commit	(Flush wait)	250	120
6	User sees D3.	(Flush wait)	270	140
7	**Crash!**			140

One naive fix for the problem is to make all read-only transactions wait for the log buffer before returning results. For example, transaction A could check the latest LSN of log buffer as of its own commit time (250), then wait until the log buffer makes all logs up to the LSN durable. However, this essentially means that all read-only transactions have to do log flushes even when they have not touched any uncommitted data. This substantially slows down *all* read-only queries because a typical read-only query finishes within micro-seconds while a log flush takes at least several milli-seconds on hard disks. Instead, if another concurrent and ad hoc (not in flush-pipeline) read-only transaction C reads only committed data, C should immediately return the results without log flush.

5.3 Safe SX-ELR

The principle rule here is that a read-only transaction must wait until the log flush of other transactions it depends on. In the above case, transaction A should wait until the log buffer's durable LSN becomes 200, which is the LSN of B's commit log. Notice that A must wait until 200, not 130 (LSN of D3). We initially considered another naive fix, which checks the maximum page LSN each read-only transaction touched (130) and waits until the LSN becomes durable. However, even if the particular update operation log becomes durable, the dependent transaction (B) might be later rolled back if its commit log is not yet durable.

Based on the observations above, our solution for SX-ELR is described in Algorithm 6 and 7. The key idea is to leave a *tag* on each lock queue in the lock table. The tag annotates when the latest durable modification happened to the data protected by the lock. Every transaction checks such tags whenever it acquires a lock and stores the maximum value of the tags it observed. When the transaction turns out to be read-only at commit time, it compares the maximum tag with the durable LSN and immediately exits if the maximum tag is already durable. Otherwise, it wakes up the log flusher and waits until the LSN becomes durable. In other words, the maximum tag is the serialization point of the read-only transaction. If the thread is pipelining a next transaction, we inherit the maximum tag (commit LSN if the current transaction is read-write) to the next transaction, anticipating the case where the next transaction is also read-only.

Read-write transactions, on the other hand, update the tags with their commit log's LSN when they release X locks during SX-ELR. In the above example, transaction B updates the tag of D3 and J5 with the value 200, its commit LSN. This is required only for X locks which imply logical data update done by the transaction. The same rule applies to coarse locks (e.g., volume-lock) with an additional *descendant tag*. The descendant tag is updated when early-releasing SIX or IX locks while the other tag (*self tag*) is updated only when early-releasing absolute X locks. Transactions that take intent locks (e.g., IS) check the self-tag while those that take absolute locks (e.g., S) check both the self and descendant tags.

The above scheme is very simple and has negligible overhead because it adds just one integer comparison during lock acquisition and release. One assumption here is that the deletion of a row must leave a ghost record, and system transactions must not eliminate this ghost record until the modification has been committed and made durable. Because we rely on the existence of the lock queue which holds the tag, eliminating the ghost record or corresponding lock queue while running the user transaction does not ensure serializable results. Therefore, system transactions for maintenance database operations (e.g., defragmentation) must ensure the pages they are cleaning do not have any uncommitted data. This can be done by tracking the starting LSN of the oldest active transaction in the system, and by comparing it with the PageLSN [15].

Algorithm 6: Safe SX-ELR: Request-Lock

```
(based on Algorithm 5)
...
if myreq is granted then
    myreq.st = granted;
    myreq.gm = m;
    xct.max_tag = max(xct.max_tag, queue.tag);
    return (SUCCESS);
...
```

Algorithm 7: Safe SX-ELR: Commit Protocol

```
Data: L: Lock table, M: Log manager
Input: xct: Transaction to commit
if xct did not make any writes then //read-only xct
    Release all locks. (S-ELR: Always safe);
    M.check_durable(xct.max_tag);
else//read-write xct
    commitLSN = M.append_commit_log();
    // SX-ELR with the commit LSN
    foreach req ∈ xct.locks do //in reverse acquire-order
        queue = L.find_queue (req.key);
        queue.release(req);
        if req.gm ∈ {X, XN, XS ...} then //update tag
            queue.tag = max(queue.tag, commitLSN);
    M.flush_until(commitLSN);
```

6. PERFORMANCE EVALUATION

We have developed a prototype database engine with all of our techniques proposed in this paper by modifying Shore-MT 6.0.1. Shore-MT is specifically designed to achieve high performance and concurrency on many cores. The original Shore-MT team demonstrated performance and scalability with comparisons to several commercial and open-source database systems [10]. To separate the effects of other improvements we made in B-trees [6], our experiments mainly compare performance of the modified storage engine (***Modified***) with and without individual techniques. However, we additionally compare with the original Shore-MT (***Original***) in the last ELR experiment to show the end-to-end performance improvements with all of our techniques combined. In other experiments, we observe that *Modified* performs substantially faster than *Original*, too, although the improvements come from both the techniques in this paper and the other techniques in [6].

Except where explicitly noted, all runs ensure full serializability. Lock escalation is disabled for higher concurrency. For serializability with flush-pipelines, our implementation flushes the log before releasing locks when the current transaction of the pipeline is selected for rollback and restart.

6.1 Environment

Except where explicitly noted, all experiments were performed using a HP Z600 system (6-core 2.67 GHz Intel Xeon CPU with 12 MB cache, 6 GB of RAM, running x86_64 Fedora Core 14 Linux). The size of the buffer pool always exceed the size of the working data set. For transactional logging, we used either a hard disk (random seek time: 10ms) or an SSD (random seek time: 20-50us) formatted in ext4.

We also performed a small number of experiments on Sun Niagara hardware with 64 hardware contexts (running Solaris 10). On the Niagara system, the transactional log is in a RAM-resident *tmpfs*. Experiments that require a large number of CPU cores were

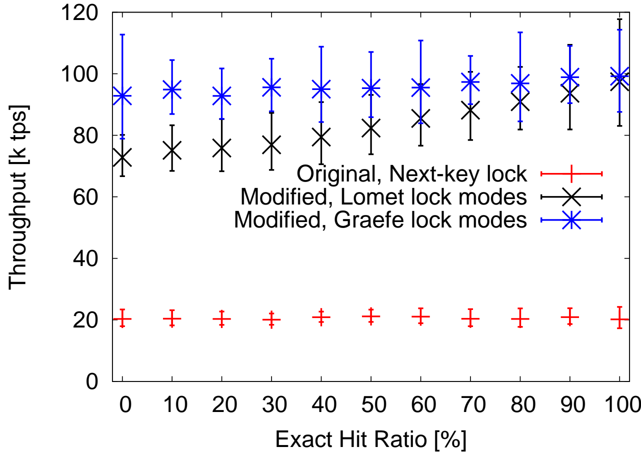


Figure 3: Key range lock concurrency.

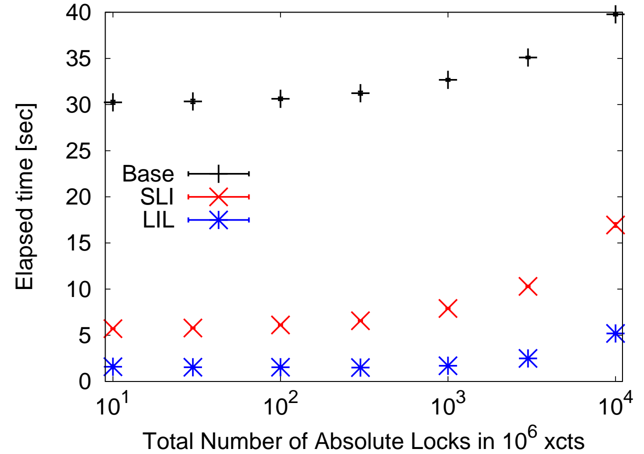


Figure 4: Intent lock overhead. **(on 64-core Niagara)**

run on Niagara while experiments that are sensitive to logging performance were performed on the Z600 because the Niagara system has no SSD drive.

6.2 Dataset and query workload

In all experiments, we use the TPC-B [2] benchmark datasets and query workloads, which consists of four tables (branch, teller, account and history) and one transaction that updates all four tables. We chose a relatively simple benchmark to study the basic behavior of fundamental choices regarding transaction processing in databases. Also, the closest prior work (Shore-MT) used TPC-B in their experiments [11]. Repeating the same setting is another reason we use TPC-B for our experiments.

The database initially contains 20 branches, 200 tellers, 2 million accounts, which constitute a database of about 250 MB. All experiments are hot-start, meaning we load all data into the buffer pool before measuring the performance.

In some experiments, we modify the update queries in the TPC-B transaction to SELECT queries on the same record so that we also have read accesses to the tables. In such experiments, we vary the fraction (Read Ratio) of update queries converted to read queries. When the Read Ratio is zero, the workload is the original TPC-B.

6.3 Results

All results are based on 20 runs and we show the arithmetic mean and the 95% confidence interval.

Key range locks: First, we evaluate the concurrency of the key range lock modes from Section 2, focusing on the differences between Lomet's and Graefe's design of lock modes. In this experiment, each transaction issues 4 range searches and then 1 update on the TPC-B teller table. The update operation happens always on an existing key. The range search, on the other hand, starts and ends at existing keys for P% of the range queries, with P varying from 0 to 100. Otherwise the cursor scans a range between keys. The range is chosen with random uniform. The table size is chosen to be small in order to force lock contention and bring out the differences between the locking designs. A similar situation would happen on much larger tables when the access is skewed, for example when each transaction focuses recent data.

[2] http://www.tpc.org/tpcb/default.asp

We measured the elapsed time to process 100K such transactions on 6 threads in the Z600 machine. Figure 3 shows the performance with Lomet's and Graefe's lock modes. We simulated Lomet's lock modes by taking slightly conservative lock modes e.g., SS instead of NS, when Lomet's set of lock modes is missing a more precise mode, e.g., when a search does not hit an existing key. As the result shows, Graefe's lock modes are more precise and concurrent, achieving up to 15% higher throughput.

We also compared the original Shore-MT locking system, which uses only S or X locks on the next key as suggested in ARIES/KVL. It resulted in 4× to 5× lower throughput. However, a good part of this difference is due to an optimized page layout [6].

Intent locks: Next, we verify the efficiency and scalability of light-weight intent locks (LIL) from Section 3. We implemented LIL and compared it to the performance of Shore-MT's Speculative Lock Inheritance (SLI) [9]. In order to isolate the overheads of intent locks, this experiment only acquires and releases intent locks. Each transaction takes intent locks on the 4 tables in TPC-B (branch, teller, account and history).

Half of the transactions take IX locks on the tables and the volume. The other transactions take IS locks on the tables and the volume. We randomly take absolute locks on the tables and the volume, taking X locks on the tables and IX on the volume. 1 in 4 transactions takes an X lock on the volume.

Because the physical contention of intent locks is particularly significant with many cores [9], this experiment runs 1 million transactions with 60 concurrent threads on the Niagara machine.

Figure 4 plots the total elapsed time for varying numbers of absolute locks taken during the experiment. If it takes 10 seconds to process 1 million empty transactions, the maximum possible throughput is only 100k even if we have no other bottlenecks.

As the figure shows, SLI performs about 5× to 6× faster than the base performance. LIL performs even faster than SLI for another factor of 3 to 5. The speed up is due to two reasons. First, LIL is much less affected by absolute locks while SLI relies on inherited locks, which are flushed out by absolute locks. Second, our simpler and lightweight intent locking protocol has much shorter code paths, fewer shared objects and less mutex usage. Lock acquisition and release on LIL is significantly faster than that of SLI. Also, LIL simplifies the code paths for non-intent locks because it completely

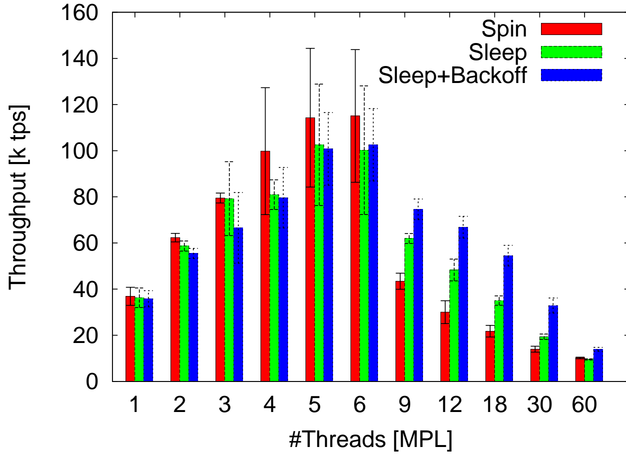


Figure 5: Dreadlocks: spin, sleep, and backoff. (on 6-core Z600)

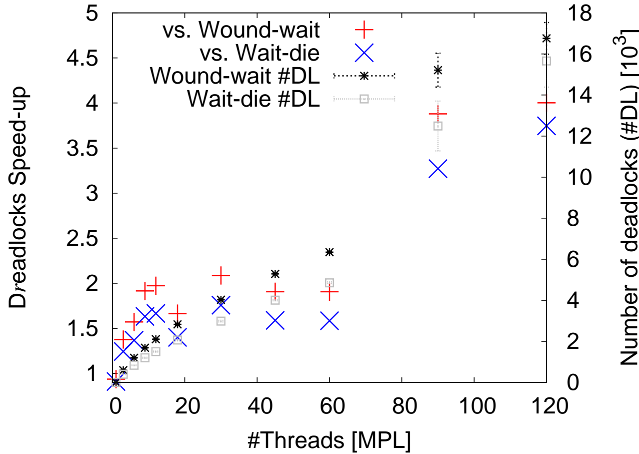


Figure 6: Speed-up by Dreadlocks compared to traditional Wound-Wait and Wait-Die (on 64-core Niagara).

separates intent locks and non-intent locks.

Dreadlocks: As described in Section 4, Dreadlocks is an efficient and scalable algorithm to detect deadlocks in many-core settings, but when there are many more concurrent threads than the number of CPU cores, Dreadlocks may greedily waste CPU cycles.

In this experiment, we evaluate the algorithm to alleviate the bottleneck. Figure 5 compares the three policies described in the earlier section; simple spinning, sleep (10 ms) after each spin and sleep with backoff to suppress false deadlock detections. The figure shows the throughput of original TPC-B queries with varying numbers of concurrent transactions. In order to isolate the CPU-overhead related to deadlocks, this experiment uses lazy commits. Hence, almost all bottlenecks are on CPUs.

We observe that the spinning policy performs best (up to 5-20% better than others) until the number of streams reaches the number of cores (6 MPL), after which performance quickly deteriorates due to CPU cycles wasted by waiting threads. The sleep method offers better scalability but still suffers at larger MPL because of false deadlocks (without flush pipelining, original TPC-B transactions can have no true deadlocks). The sleep with backoff method scales best, causing no false deadlocks.

In summary, if the number of concurrent transactions is less than the number of cores, Dreadlocks with simple spinning minimizes the overheads caused by sleep and mutex synchronization. However, if there are more concurrent transactions than cores, then the sleep-backoff method ensures higher scalability with reasonable overheads. The best choice seems a hybrid scheme, monitoring the number of active threads (including background threads) and automatically switching between pure-spinning and sleep-backoff methods. Note that the switch does not have to be precise nor even atomic because both additional sleeping and backoff have no effect on correctness.

Comparison to traditional methods: We then evaluate the scalability of the modified Dreadlocks algorithm with traditional deadlock detection methods, wound-wait and wait-die. Figure 6 shows the speed-ups achieved by Dreadlocks (its throughput divided by that of Wound-Wait and Wait-Die) and the number of deadlocks (unit is 10k) on Niagara. The tested workload is similar to the no-true-deadlock experiment in [12]. We run 100k transactions which update 5 resources (tellers) in canonical order. Thus, all detected deadlocks are false positives which slow down the workload.

Our Dreadlocks algorithm reported almost no deadlocks except occasional false positives caused by Bloom filters. On the other hand, both wound-wait and wait-die cause thousands of deadlocks as the number of concurrent threads increases. As a consequence, their throughput quickly drops about half that of Dreadlocks as soon as the number of concurrent threads exceeds 10.

For a comparison between the Dreadlocks and timeout algorithms, we refer readers to [12]. Generally speaking, it is hard to choose the right timeout parameter, as this is highly workload-dependent. An incorrect timeout parameter can result in not only degraded performance but also starvation.

Deadlock resolution: Next, we evaluate algorithms to resolve deadlocks in the context of flush pipelines. As flush pipelining was proposed relatively recently, this is the first study to empirically compare deadlock resolution policies for flush-pipelines.

We ran 1M modified TPC-B transactions with flush pipelining, doing flushes after each deadlock-abort or upon completion of 100k transactions. We did not use ELR in this experiment and used hard-disks for logging.

Figure 7 compares the kill-short (rolls back shortest pipeline) and kill-long (chooses longer pipeline instead) policies as well as the policy in the original Shore-MT which rolls back the youngest transaction (kill-young), ignoring the length of the pipeline. These and all subsequent experiments were run on the Z600 at MPL 6.

We observe that kill-short performs better than kill-long for a factor of 2 to 4 on hard-disk logging. Because of slow disk latency, single-threaded execution reduces the deadlocks and the log flushes caused by them. All policies transition to such single-threaded execution after one pipeline dominates the lock table and prevents other pipelines to proceed and cause deadlocks. However, the way the two policies transition to single-threaded is very different.

After each 100k transactions (the maximum length of the pipeline), there is a pause. This is because threads are competing with each other to "dominate" the lock table. Kill-long takes longer than kill-short before one thread dominates the lock table without deadlock; kill-short quickly reaches a state of domination by a single thread because a longer pipeline is never rolled back and easily reaches its maximum pipeline length. The conventional kill-young behaved similarly to kill-long because it ignores the pipeline length, causing the same problem as kill-long.

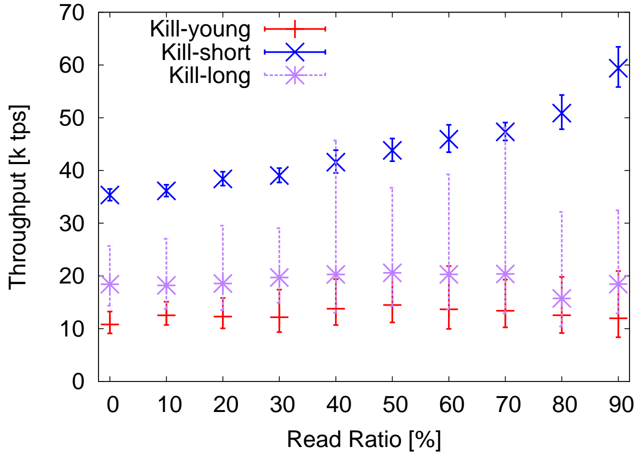


Figure 7: Deadlock resolution policy.

In summary, when we use flush pipelining under high lock contention, kill-short causes fewer log flushes and performs substantially faster if commit-flush is the bottleneck. We observed that the difference between policies is much less noticeable when we use low-latency SSD for logging.

However, kill-short policy poses another problem: lower concurrency. Hence, in the next experiment we evaluate early lock release (ELR) methods to improve concurrent execution.

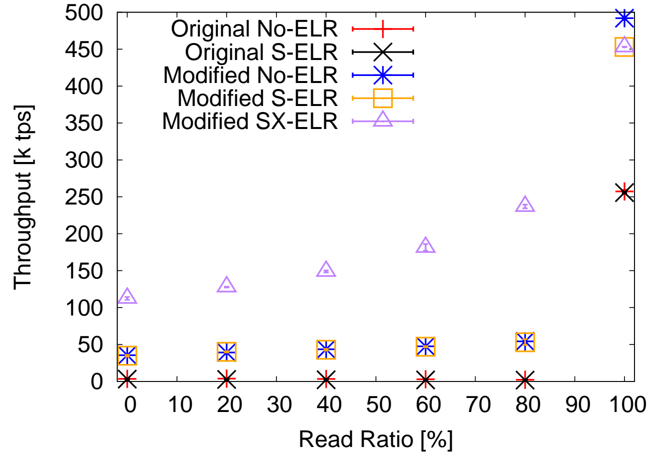
Early lock release: Finally, we verify the overheads and benefits of our early lock release (ELR) algorithms. We ran the same workload as the previous "kill-short" experiment, enabling ELR.

Our ELR implementation in Section 5 supports both modes of ELR; S-ELR (releases only S locks) and SX-ELR (releases both S and X locks) without violating serializability. Shore-MT's implementation of SX-ELR might cause dirty reads if read-only transactions do not wait for log flush. Hence, we tested only S-ELR (called "Quarks") on Shore-MT, which does not violate serializability but cannot release X locks.

Figure 8 shows the results on MPL 6 with HDD and SSD for logging respectively. HDD results have fewer data points and each point represents only 3 runs because those runs take a long time to complete.

SX-ELR performed significantly faster than No-ELR and S-ELR by a factor of 3 to 5 (compared between *Modified*). It is worth mentioning that because lock waits and deadlocks are only caused by X-locks, S-only ELR does not improve throughput either in *Modified* or *Original*. As long as a pipeline leaves X locks, lock contention is inevitable.

This result verifies that SX-ELR is essential to eliminate lock contention. Also, the result is highly consistent with the prior observation made by Johnson et al [11] shown in Figure 9, which reports speed-ups achieved by ELR with varying skew and latency of the logging device. Our experiment used the standard Zipfian (S=1). Hence, the speed-up of 5$\times$ in HDD and 3$\times$ in SSD we observed exactly matches with their result. The difference is that our SX-ELR is fully serializable even in the case of read-only transactions. While our SX-ELR protocol ensures full serializability, it does not force an ad hoc read-only transaction to wait unless it has unluckily touched uncommitted data. Almost all read-only transactions can go through without waits for log flush.



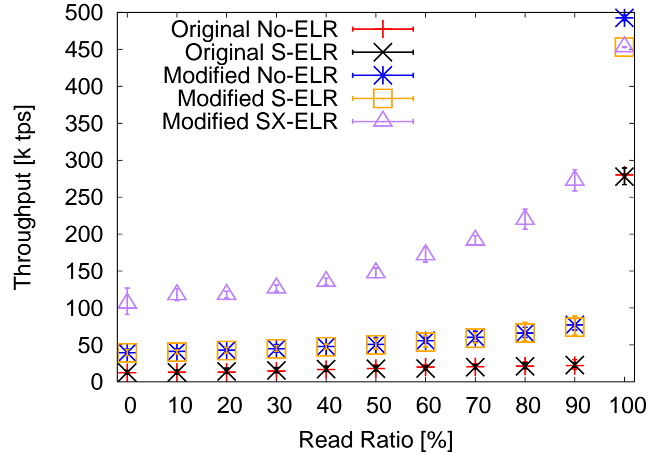


Figure 8: Early Lock Release: logging with HDD (above) and SSD (below). Only the modified storage engine has serializable SX-ELR. It also uses LIL and Foster B-trees.

If we compare the performance of *Modified* with *Original*, the difference sometimes exceeds a factor of 50. The speed-up is especially large when contention is high, for example 0% read ratio (original TPC-B) on HDD. This is because the modified storage engine's other improvements such as LIL and Foster B-trees further reduce the overhead and contention in locking and latching.

When all of the transactions are read-only, No-ELR performs slightly faster because ELR checks the transaction's lock requests twice at commit time. However, this overhead causes only 5% difference even in such an extreme case. On the other hand, the overhead to collect and update the tags in lock queues was negligible.

7. SUMMARY AND CONCLUSIONS

In recent years, large main memory sizes and low-latency storage have become increasingly common. This often completely eliminates the traditional database bottleneck of disk I/O. Four new bottlenecks arise in this context: logging, latching, locking, and the buffer pool. These bottlenecks are exacerbated by the rapidly increasing counts of CPU cores in modern hardware.

This paper focuses on the locking bottleneck, re-defining the

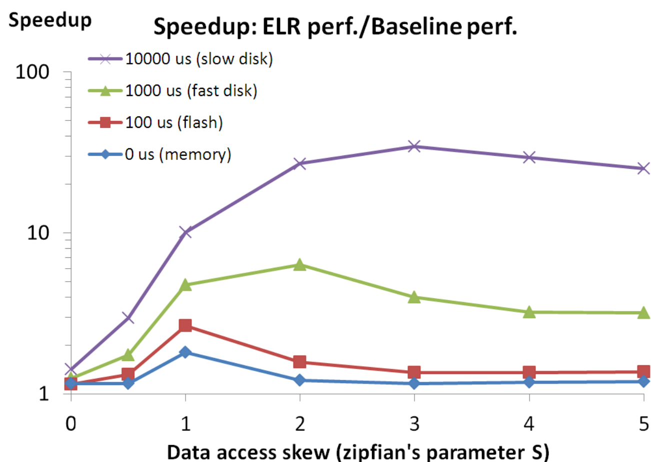


Figure 9: cf. Prior Study of ELR Performance by Johnson et al. Adopted from [11] with permission by the author.

locking module in order to exploit the performance opportunities of modern hardware. First, we implemented and evaluated a new design for key range locking that combines orthogonal lock modes and fence keys to maximize concurrency among updates in B-tree leaves. Second, our novel Lightweight Intent Locks dramatically simplify intent-locks and non-intent-locks to improve the performance and scalability of critical sections in the lock table. Third, we improve the Dreadlocks algorithm for deadlock detection earlier implemented in Shore-MT, adapting it to work with flush pipelining and to simplify deadlock resolution. Last, we improve the Early Lock Release algorithm [11] with our tagging mechanism in the lock manager, allowing read-only transactions to safely exit without unnecessary commit delays.

Our extensive experiments demonstrate that each technique substantially improves transaction throughput. Together, our storage manager achieves up to $50\times$ better throughput than the original Shore-MT. CPU profiling after these changes indicates that more than half of CPU cycles are spent in the buffer pool and binary searches during B-tree lookups. We are currently working on the remaining bottlenecks. In the buffer pool, we plan to evaluate optimizations that reduce searching and latching in the buffer pool's hash table. For searching in B-trees, we will evaluate cache-conscious page layouts, interpolation search, and large pages.

Our project aims to eliminate all four bottlenecks (see Figure 1) on modern hardware. Our new variant of B-trees, the Foster B-tree, eliminates the latching bottleneck. We have inherited logging optimizations from the Shore-MT code base and have overhauled the locking module in this paper. We are now in the process of addressing the last bottlenecks, the buffer pool and index search.

Acknowledgments

We are very grateful to the Shore-MT team for making their code base publically available, and in addition thank them for kindly helping us work on Shore-MT, for letting us use their Sun Niagara, and for giving us insightful comments on the manuscript. We are particularly grateful to Anastasia Ailamaki, Ryan Johnson, Nancy Hall, and Pinar Tozun, as well as Ippokratis Pandis, Danica Porobic, and Manos Athanassoulis.

8. REFERENCES

[1] R. Agrawal, M. Carey, and L. McVoy. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE TSE*, (12):1348–1363, 1987.

[2] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD*, pages 1–8, 1984.

[3] G. Graefe. Write-optimized B-trees. In *VLDB*, pages 672–683, 2004.

[4] G. Graefe. Hierarchical locking in B-tree indexes. In *BTW*, pages 18–42, 2007.

[5] G. Graefe. A survey of B-tree locking techniques. *ACM TODS*, 35(3), 2010.

[6] G. Graefe, H. Kimura, and H. Kuno. Foster B-trees. In *ACM TODS (to appear)*, 2012.

[7] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[8] S. Harizopoulos, D. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[9] R. Johnson, I. Pandis, and A. Ailamaki. Improving OLTP scalability using speculative lock inheritance. *PVLDB*, pages 479–489, 2009.

[10] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009.

[11] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: A scalable approach to logging. *PVLDB*, 3(1):681–692, 2010.

[12] E. Koskinen and M. Herlihy. Dreadlocks: efficient deadlock detection. In *Symposium on Parallelism in algorithms and architectures*, pages 297–303, 2008.

[13] D. B. Lomet. Key range locking strategies for improved concurrency. In *VLDB*, pages 655–664, 1993.

[14] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *VLDB*, pages 392–405, 1990.

[15] C. Mohan. Commit_lsn: A novel and simple method for reducing locking and latching in transaction processing systems. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *VLDB*, pages 406–418. Morgan Kaufmann, 1990.

[16] C. Mohan and F. Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In *SIGMOD*, pages 371–380, 1992.

[17] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM TODS*, 11(4):378–396, 1986.

[18] R. Obermarck. Distributed deadlock detection algorithm. *ACM TODS*, 7(2):187–208, 1982.

[19] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[20] I. Pandis, P. Tozun, R. Johnson, and A. Ailamaki. PLP: Page latch-free shared-everything OLTP. *PVLDB*, 2011.

[21] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *PVLDB*, 4(11), 2011.

[22] A. Silberschatz, H. Korth, and S. Sudarshan. *Database system concepts*. McGraw-Hill, 2005.

[23] E. Soisalon-Soininen and T. Ylönen. Partial strictness in two-phase locking. *ICDT'95*, pages 139–147, 1995.