A Scalable Lock Manager for Multicores

Hyungsoo Jung
NICTA
Sydney, Australia
hyungsoo.jung@gmail.com

Hyuck Han*
Samsung Electronics
Hwasung, South Korea
hyuck.han@samsung.com

Alan D. Fekete
University of Sydney
Sydney, Australia
alan.fekete@sydney.edu.au

Gernot Heiser
NICTA and UNSW
Sydney, Australia
gernot@nicta.com.au

Heon Y. Yeom
Seoul National University
Seoul, South Korea
yeom@snu.ac.kr

ABSTRACT

Modern implementations of DBMS software are intended to take advantage of high core counts that are becoming common in high-end servers. However, we have observed that several database platforms, including MySQL, Shore-MT, and a commercial system, exhibit throughput collapse as load increases, even for a workload with little or no logical contention for locks. Our analysis of MySQL identifies latch contention within the lock manager as the bottleneck responsible for this collapse.

We design a lock manager with reduced latching, implement it in MySQL, and show that it avoids the collapse and generally improves performance. Our efficient implementation of a lock manager is enabled by a staged allocation and de-allocation of locks. Locks are pre-allocated in bulk, so that the lock manager only has to perform simple list-manipulation operations during the acquire and release phases of a transaction. De-allocation of the lock data-structures is also performed in bulk, which enables the use of fast implementations of lock acquisition and release, as well as concurrent deadlock checking.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*concurrency, transaction processing*; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Design, Performance

Keywords

Multicores, scalable lock manager, concurrent programming

1. INTRODUCTION

Present high-end computing platforms, as they are frequently used for database servers, now feature dozens of processor cores, and will soon exceed 100 cores. It is therefore important that database transaction processing can utilise such a large number of cores efficiently.

This is not trivial, as higher core counts result in increased contention for shared data structures in the DBMS internals. A few years ago, studies [11, 17] showed that many open-source and commercial database engines exhibit limited multi-core scalability, and even performance collapse, due to bottlenecks in their lock, log or transaction manager. This has triggered significant research activity [3, 5, 12, 14–17, 19] aimed at reducing lock or latch contention.

Yet, significant scalability problems still exist for serializable transactions, surprisingly even under read-only workloads, where the locks never conflict with one another (though of course they must still be taken, since the system must work properly if updates are submitted). This is illustrated in Figure 1, which shows throughput of a serializable, read-only workload on MySQL under varying load using 4, 8 or 16 cores of the same 32-core platform. (Detailed specification of the setup is given in Section 5.)

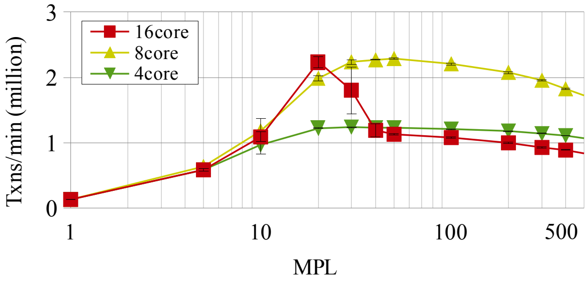
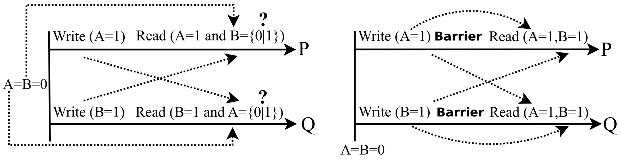


Figure 1: MySQL throughput under varying multiprogramming level (MPL) on 4 (green), 8 (yellow) and 16 cores (red).

The figure clearly shows that on 4 cores the database behaves as expected, with 8 cores throughput degrades at high load to about 75 % of peak throughput. On 16 cores, high-load performance collapses to about 1/3 of peak, and well below what the 4-core configuration can handle. The peak throughput of 16 cores is also below that of 8.

[*] Affiliation: Memory Solutions Lab, Memory Division, Samsung Semiconductor.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2037-5/13/06 ...$15.00.


?
Write (A=1) Read (A=1 and B={0|1})
P
A=B=0
?
Write (B=1) Read (B=1 and A={0|1})
Q
Write (A=1) Barrier Read (A=1,B=1)
P
Write (B=1) Barrier Read (A=1,B=1)
Q
A=B=0

(a) Non-serialisable execution (b) Safe with barriers

Figure 2: Unsafe vs. safe RAW execution.

Our analysis of this undesirable effect shows that it is caused by latch contention in the lock manager. Each time a transaction acquires or releases a lock, it grabs the mutex protecting the global table of locks. Even though the mutex is only held briefly, *cache-line bouncing* [18] makes acquiring or releasing it expensive under high contention [4].

We propose a solution to this problem, based on a (mostly) latch-free implementation of the lock manager. The design is enabled by the observation that the lock acquisition and release operations can be split into two: the allocation (and initialisation) and de-allocation of the lock data structure, and the manipulation of this data structure. The latter can be implemented latch-free, and thus contention-free. We perform allocation and de-allocation of locks in bulk, thus amortising any remaining latching overhead over a large number of locks. Furthermore, de-allocation becomes a garbage-collection exercise which can be performed asynchronously to transaction processing. We refer to this approach as *staged lock acquisition and release.*

This paper makes the following contributions:

- We identify scalability issues leading to performance collapse under high workloads when running on a platform featuring dozens of cores. We show that this collapse occurs on multiple open-source (MySQL, Shore-MT) as well as on a commercial database engine (Section 5.2).
- We propose a design for the lock manager in which much less latching is needed, and we describe its implementation in MySQL (Section 3).
- A key idea that enables our design is bulk allocation and de-allocation of locks, asynchronous to transaction processing (Section 4);
- we evaluate our implementation, demonstrating that staged lock handling avoids the throughput collapse and also leads to improved peak throughput (Section 5.2).

2. BACKGROUND AND RELATED WORK

This paper is related to an extensive literature on managing concurrency, both in database systems and more general programs. Here we point to a few essential concepts needed for understanding our contribution.

2.1 The Lock Manager

Database management platforms are expected to provide the application programmers with an abstraction of ACID transactions, freeing the programmer from worrying about anomalies that might arise from concurrency or failure. This enables independent development of multiple database clients. Authoritative coverage on the field of transaction management, and its subfield of concurrency control can be found in the textbooks by Bernstein *et al.* [2] and Weikum and Vossen [20]. The internal design of a DBMS is described by Hellerstein *et al.* [9]. The usual approach is for the lower, storage-focused parts of the DBMS to contain a component called a lock manager. Code that fetches or updates a record obtains an appropriate lock in the lock manager; this lock is then held until the transaction completes ("strict two phase locking"). Locks are taken in various modes, such as shared mode for a read, or exclusive mode for a write; other modes are important for preventing phantoms and allowing locking on different granularities of item (such as records, pages and tables). The lock manager arbitrates access by putting a transaction, which attempts to acquire an incompatible lock, into a wait state, and wakes it up when the lock becomes available. Design principles for a lock manager are presented by Gray and Reuter [8].

While the lock manager prevents logical interference between concurrent transactions, there is also a need to prevent interference between DBMS threads as they manipulate the shared physical structures within the DBMS, including within the lock manager. This is typical of the problems of concurrent programming, and is traditionally addressed by what the DB community calls latches (or mutexes) that surround any DBMS internal code that deals with shared data structures. The distinctions between locks and latches are clarified by Graefe [6]. Early open-source DBMS code used a coarse-grained latch on the whole of the kernel internals, but more recent releases (e.g., the InnoDB system we used as the baseline for our evaluation) have moved to a more sophisticated design with separate latches for different parts of the kernel, such as the lock manager, the buffer pool, etc.

Recent research has addressed some bottlenecks in transaction processing on multi-core platforms [11,17], proposing some radical changes to the system architecture or access structures [7, 12, 14, 15, 19]. In contrast, we here consider a localized change to the lock manager, without requiring other parts of the platform to be recoded.

2.2 Synchronising Multithread Programs

Concurrent programs are notoriously hard to code correctly. The simplest approach relies on an ability to make a segment of code execute without interleaving, allowing repeated use of a variable, as in the synchronisation pattern called *atomic write after read* (AWAR) where a thread or process atomically reads and then writes a shared variable. The atomicity of AWAR is typically achieved through latches (such as spin locks) using atomic machine instructions, such as Compare-and-Swap or Test-and-Set. On multiprocessor architectures with a snoop-based cache coherence protocol, this can lead to expensive *cache invalidation storms.*

An alternative coding style is *read after write* (RAW) where a process writes to some shared variable A, followed by the process reading a different shared variable B, without process writing to B in between [1]. Use of the RAW pattern must ensure that concurrent processes only see a compatible (linearizable or sequentially consistent) state of shared variable. The problem is illustrated in Figure 2(a), where concurrent processes P and Q access shared variables A and B according to the RAW pattern. Executing unconstrained, the data race could result in P seeing the value of $B = 0$ **and** Q seeing the value $A = 0$, which violates serialisability. Figure 2(b) shows how correctness can be enforced though the introduction of *memory barriers*, making the data race benign.

While memory barriers can be expensive, the advantage of the RAW pattern over AWAR is that the hardware overhead is paid at most once per thread. This is in contrast to cache lines potentially bouncing repeatedly between processes in AWAR. Hence the RAW pattern tends to scale better with increasing number of cores and processes.

The textbook by Herlihy and Shavit [10] provides extensive coverage on multiprocessor programming, for a wide range of interesting data structures.

3. LOCKING WITH LESS LATCHING

We now describe the approach we take to make the lock manager (mostly) latch-free. As explained earlier, the core idea is to separate the allocation and de-allocation of the lock data structures from lock acquisition and release, and perform the allocation and de-allocation in bulk and asynchronous to transaction processing.

In this section we describe the algorithms for lock acquisition and release, while the bulk allocation and de-allocation of the lock hash table is presented in Section 4.

The LHS of Figure 3 shows high-level pseudo-code of the implementation of the lock acquisition (top) and release (bottom) operations in MySQL; other database engines use a similar approach. Most of the acquisition and all of the release are protected by a mutex on the lock data structure. This ensures the consistency of the lock table and correct synchronisation of transactions.

3.1 Lock Acquisition

The fundamental observation which allows us to reduce latching is that some of the data races in the code are benign: when obtaining a slot in the hash table, it does not matter which slot goes to a particular transaction, as long as all transactions obtain different slots.

Looking at the acquire code in the LHS column of Figure 3, we can see that the transaction first creates a lock on the data item and inserts it into its list of held locks.[1] It then checks whether other transactions hold incompatible locks on the same data, in which case it marks the new lock as being in waiting state, followed by checking for deadlocks. The complete sequence is protected by a latch on the hash table.

The second part makes the transaction wait until all preceding incompatible locks are released, as required for the usual database locking protocols. It does this by calling the operating system to suspend the process, until the holder of the incompatible lock signals its release. This code is protected by the transaction latch.[2]

Re-writing this operation in the RAW pattern allows us to eliminate the hash-table latch, as shown on the RHS of Figure 3. Inserting the lock into the hash table is now performed by a latch-free implementation of the insertion function (**S1**), discussed below. Each occurrence of writing a shared variable followed by reading a different one (**S2**, **S3**) is protected by a barrier (represented by the `atomic_synchronize()` function), in line with the safe RAW pattern of Figure 2(b) . The new lock state `OBSOLETE` is a result of the staged de-allocation, and indicates that the lock is no longer in use. The deadlock-checking function (line **A12**) must be replaced by a race-tolerant implementation `new_deadlock()`.

Traditional Lock Manager | **Our Lock Manager**

Lock Acquire in Growing Phase

Traditional Lock Manager:

```
mutex_enter(lock_table->mutex);
n_lock = lock_create();
n_lock->state = ACTIVE;
lock_insert(n_lock);
for all locks (lock) in hash_bucket
  if (lock is incompatible with n_ lock)
      n_lock->state = WAIT;
      if (deadlock_check() ==TRUE)
          abort Tx;
       break;
  else
      continue;
  end if
end for
mutex_exit(lock_table->mutex);
if (n_lock->state==WAIT)
    mutex_enter(Tx->mutex);
    Tx->state = WAIT;
    os_cond_wait(Tx->mutex);
    mutex_exit(Tx->mutex);
end if
```

Our Lock Manager:

```
A1:  n_lock = lock_create();
A2:  n_lock->state = ACTIVE;                          S1
A3:  atomic_lock_insert(n_lock);
A4:  for all locks (lock) in hash_bucket
A5:    if (lock is incompatible with n_lock)
A6:        n_lock->state = WAIT;                      S2
A7:        atomic_synchronize();
A8:        if (lock->state==OBSOLETE)
A9:            n_lock->state=ACTIVE;                  S3
A10:           atomic_synchronize();
A11:           continue;
A12:       if (new_deadlock()==TRUE)
A13:           abort Tx;
A14:       break;
A15:   end if
A16: end for
A17: if (n_lock->state == WAIT)
A18:   mutex_enter(Tx->mutex);                        S4
A19:   atomic_synchronize();
A20:   if ( n_lock has to wait )
A21:       Tx->state = WAIT;
A22:       os_cond_wait(Tx->mutex);
A23:   else
A24:       n_lock->state = ACTIVE;                    S5
A25:       atomic_synchronize();
A26:   end if
A27:   mutex_exit(Tx->mutex);
A28: end if
```

Lock Release in Shrinking Phase

Traditional Lock Manager:

```
mutex_enter(lock_table->mutex);
for all locks (lock1) in Tx
  lock_release(lock1);
  for all locks (lock2) following lock1
    if ( lock2 doesn't have to wait )
       lock_grant( lock2 );
       lock2->state=ACTIVE;
    end if
  end for
end for
mutex_exit(lock_table->mutex);
```

Our Lock Manager:

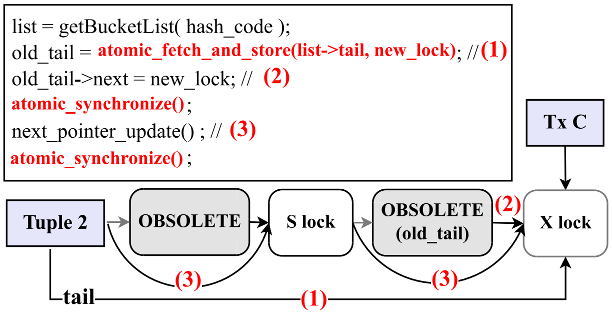
```
R1:  for all locks (lock1) in Tx
R2:    lock1->state = OBSOLETE;                       S6
R3:    atomic_synchronize();
R4:    for all locks (lock2) that follow lock1
R5:      mutex_enter(lock2->Tx->mutex);
R6:      if ( lock2->Tx->state==WAIT &&
R7:           lock2 does not have to wait )
R8:            lock2->Tx->state=ACTIVE;
R9:            lock2->state=ACTIVE;                   S7
R10:           atomic_synchronize();
R11:           os_cond_signal(lock2->Tx);
R12:     end if
R13:     mutex_exit(lock2->Tx->mutex);
R14:   end for
R15: end for
```

Figure 3: Implementation of lock acquisition and release in MySQL. The left column shows the existing lock manager code, while the right column shows our reduced-latch implementation. S1, . . . , S7 represent *RAW synchronisation points.*

Operation of `atomic_lock_insert()` is shown in Figure 4. The function inserts a new lock into the hash table by using `atomic_fetch_and_store()` on the tail pointer of the hash chain, returning the old tail pointer. The old tail is then made to point to the new lock. The function `next_pointer_update()` implements a *compare-and-swap-*

[1] Since InnoDB uses a small bitmap in a lock to represent other record locks of the same transaction in the same page, and the use of the bitmap can only be decided by seeing all locks in the hash list, `lock_create()` occurs while the hash table mutex is held.

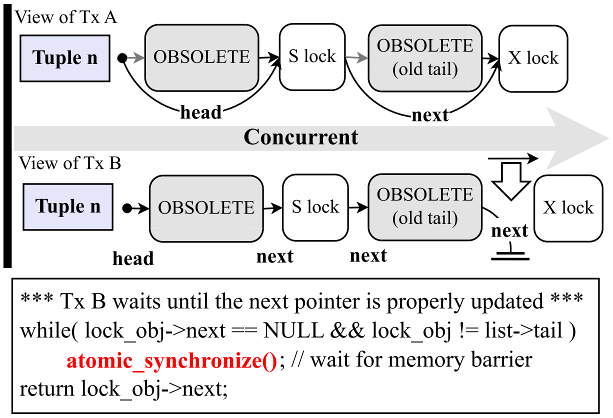
[2] In InnoDB, when a MySQL thread is suspended, it should release the mutex it holds, and after being woken up, the thread must regain the mutex and do the post-processing.



```
list = getBucketList( hash_code );
old_tail = atomic_fetch_and_store(list->tail, new_lock); //(1)
old_tail->next = new_lock; //  (2)
atomic_synchronize();
next_pointer_update() ; // (3)
atomic_synchronize() ;
```

Figure 4: Operation of `atomic_lock_insert()` on a lock hash chain.

based concurrent update [10] of the next pointers of all active locks in the list. Barriers are again used to ensure safe RAW execution. Since the next pointer update follows the atomic update of the tail pointer, we have to guarantee that other concurrent transactions accessing this list see a consistent view of this list.



```
*** Tx B waits until the next pointer is properly updated ***
while( lock_obj->next == NULL && lock_obj != list->tail )
      atomic_synchronize(); // wait for memory barrier
return lock_obj->next;
```

Figure 5: Pseudo-code of latch-free list iteration.

The traversal of the lock list (line **A4** of Figure 3) must preserve the invariant that all locks inserted before the present one are examined. The implementation of the iterator is explained in Figure 5. The original iterator in LHS of Figure 3 simply traverses the list by using the next pointer of a lock until the end of the list. Our iterator is augmented with the `while` loop, and this ensures that any locks previously inserted into the list become visible by waiting until the next pointer of a lock points to a newly inserted lock. As shown in Figure 4, since the next pointer update is done after the atomic update of a tail pointer, concurrent list traversal should see either the new lock or the old tail. This is why we need the while loop.

The second part of the acquire function retains the transaction latch, which is fine, as this latch is not contented by multiple processes. Barriers must again be inserted to ensure safe RAW execution.

3.2 Lock Release

The release code is shown in the bottom part of Figure 3. Our implementation again removes the lock-table latch, and inserts barriers according to the RAW pattern. Note that, rather than de-allocating the lock, its status is set to `OBSOLETE`, so that *no dangling pointers are generated.* The actual de-allocation is performed asynchronously and in bulk.

Importantly, the release code must examine locks which were allocated after the present lock, to detect any sleeping transactions trying to acquire an incompatible lock on the same data. These are signaled via a system call which wakes up the sleeping process.

3.3 Deadlock Detection

The deadlock checking function is to find any cycle in the wait-for graph by traversing a lock hash table. The traditional approach does this traversal while holding the lock-table latch. We avoid this by keeping any released lock and transaction data structures until they are guaranteed to be no longer required, thus avoiding dangling-pointer dereferences in the deadlock checker. By making any other data structures required for deadlock detection transaction-local, we enable a latch-free implementation.

When a deadlock is detected in the traditional, latch-based implementation, the lock manager can use a number of different policies for selecting a victim transaction to abort. For example, it can choose the transaction which (1) holds the smallest number of (write) locks, (2) generates the fewest log records, or (3) actually causes the deadlock (joining lock).

In our latch-free implementation, safety prevents a thread from changing the transaction state while other threads may access the state. Hence, the thread must abort its own transaction when detecting a deadlock, i.e., we can only implement the above policy (3).

3.4 Correctness

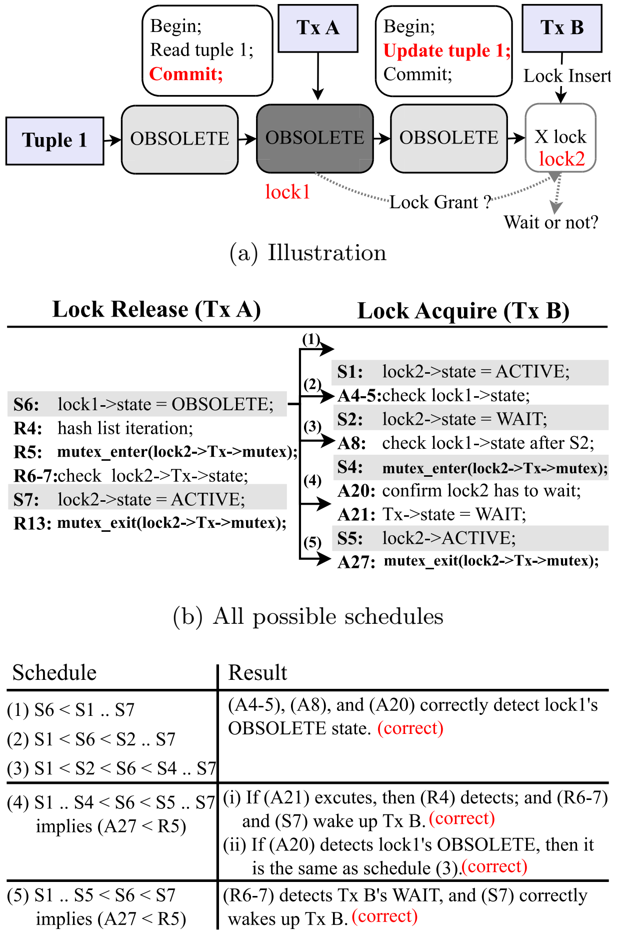
The main correctness concern is that concurrent acquisition and release operations operate correctly, and that deadlock detection is correct.

Figure 6(a) illustrates the scenario where a transaction Tx A executes `COMMIT` and is about to release lock1. Concurrently, transaction Tx B tries to insert an exclusive lock (lock2) into the same lock hash list (and is sent to sleep). We must guarantee that the commit of Tx A will wake up Tx B.

Figure 6(b) shows possible schedules resulting from the interleaving of the two transactions. The critical point is the change of the state of the lock by Tx A (**S6**), we must ensure that Tx B will be woken, no matter how the execution of **S6** is interleaved with its own execution. Depending on where **S6** occurs during Tx B's execution, we have five different interesting cases, as indicated in Figure 6(b).

The correctness argument is summarised in Figure 6(c). In the first three schedules, (1)–(3), **S6** happens before Tx B examines the state of `lock1`, and the barriers ensure that Tx B sees the correct lock state. For schedule (4), Tx B may or may not see the updated lock state. If it does, (4)(ii) then this is the same as schedules (1)–(3), and thus executes correctly. Else the checking routine **R6-7** of Tx A, which follows Tx B's wait execution due to Tx B's transaction mutex, correctly captures Tx B's "WAIT" state and wakes up Tx B correctly. In schedule (5), Tx B cannot detect `S6` and waits until Tx A correctly finds Tx B's state, as in schedule (4)(i).

To reason about the correctness of concurrent deadlock checking, suppose we have multiple transactions, all executing **S2**, changing their lock state to `WAIT`. Assuming that


Begin;
Read tuple 1;
Commit;
Tx A
Begin;
Update tuple 1;
Commit;
Tx B
Lock Insert
Tuple 1
OBSOLETE
OBSOLETE
OBSOLETE
X lock
lock2
lock1
Lock Grant ?
Wait or not?
Lock Release (Tx A)
Lock Acquire (Tx B)
S6: lock1->state = OBSOLETE;
R4: hash list iteration;
R5: mutex_enter(lock2->Tx->mutex);
R6-7: check lock2->Tx->state;
S7: lock2->state = ACTIVE;
R13: mutex_exit(lock2->Tx->mutex);
(1)
(2)
(3)
(4)
(5)
S1: lock2->state = ACTIVE;
A4-5: check lock1->state;
S2: lock2->state = WAIT;
A8: check lock1->state after S2;
S4: mutex_enter(lock2->Tx->mutex);
A20: confirm lock2 has to wait;
A21: Tx->state = WAIT;
S5: lock2->ACTIVE;
A27: mutex_exit(lock2->Tx->mutex);

(a) Illustration

(b) All possible schedules

Schedule	Result
(1) S6 < S1 .. S7 (2) S1 < S6 < S2 .. S7 (3) S1 < S2 < S6 < S4 .. S7	(A4-5), (A8), and (A20) correctly detect lock1's OBSOLETE state. (correct)
(4) S1 .. S4 < S6 < S5 .. S7 implies (A27 < R5)	(i) If (A21) excutes, then (R4) detects; and (R6-7) and (S7) wake up Tx B. (correct) (ii) If (A20) detects lock1's OBSOLETE, then it is the same as schedule (3).(correct)
(5) S1 .. S5 < S6 < S7 implies (A27 < R5)	(R6-7) detects Tx B's WAIT, and (S7) correctly wakes up Tx B. (correct)

(c) Correctness

Figure 6: Concurrent execution of `Lock Acquire` and `Lock Release`.

these transactions create a real deadlock, then all interleaving schedules of concurrent deadlock checking must guarantee that at least one of the checks must see the `WAIT` state of all other locks. This is ensured by the RAW pattern. Note that, unlike the traditional approach, latch-free deadlock checking may abort more than one transaction, if multiple deadlock checking invocations find the same deadlock simultaneously. This is correctness-neutral (but may result in performance degradation).

4. LOCK MANAGEMENT

The RAW-style algorithms for acquisition and release of locks described in the previous section can only work because we do not de-allocate a lock as soon as it has been released—this is the essential idea of staged lock release. It means that we can tolerate dangling references to a released lock, as long as its obsolete status is visible to any transaction still holding such a reference.

4.1 Safe Lock De-allocation

Locks must be de-allocated eventually, else we would be creating a memory leak. Given that de-allocation is delayed, the time when it happens is flexible, as long as we ensure that no dangling references are left, and that de-allocation happens eventually—a classical garbage-collection problem.

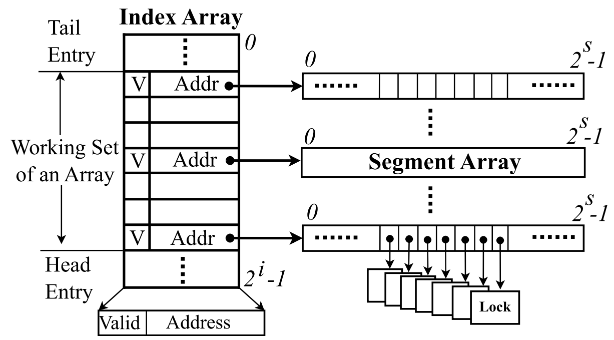


Figure 7: Lock index data structure.

The timing flexibility means that we can reduce overheads by cleaning up locks in bulk. The same is true for allocation: by pre-allocating locks, we can also reduce overheads through bulk operations.

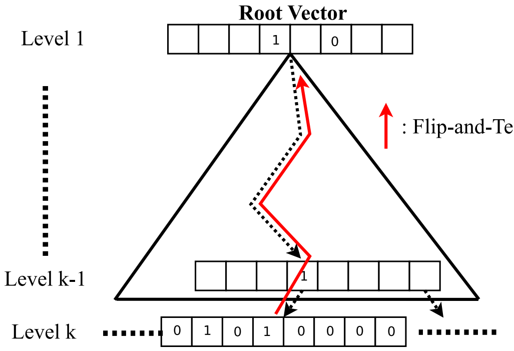
When is it safe to de-allocate a lock? As each lock belongs to a unique transaction which created it, and transactions do not pass to other transactions references to locks, a sufficient condition is the following: *A lock L can be safely de-allocated if the oldest transaction in the system has started after L's transaction terminated.*

For bulk de-allocation of a group of locks, this means that we need to keep track of the highest transaction commit ID for the group, and check that this ID is less than the oldest active transaction ID in the transaction table. A simple way to achieve this is to record the highest commit ID, T_M, for each group whenever one of the group's transaction terminates, and maintain the ID, $T_{\min}$ of the oldest still active transaction. The group can then be cleaned up when (i) all of its transactions have terminated and (ii) $T_M < T_{\min}$ for the group.

4.2 Lock Pool Implementation

To support bulk allocation and de-allocation of locks, we use a data structure which can grow and shrink on demand and naturally groups locks of similar age, and allows us to locate locks quickly. Specifically, we use a page-table-like index structure to locate a lock from its ID, as shown in Figure 7.

The data structure consists of a top-level *index array* of 2^i entries. Each entry has a valid flag and a pointer to a second-level *segment array* of size 2^s, and both fields are packed into a single 8-byte word together. A segment array contains pointers to lock objects, as well as other meta data, such as an index number that is assigned to a lock when it is allocated to a transaction. The index array has two associated pointers, the *head entry* and the *tail entry*. Index-array entries between the head and the tail may point to segment arrays (if their valid flag is set), entries outside do not point to segment arrays (irrespective of the valid flag). The array is accessed in a circular fashion, meaning that if head<tail then the potentially valid entries are the ones ≤head as well as those ≥tail. A third pointer (not shown) points to the highest (newest) used segment, all segments between that pointer and the head are unused.



(a) Byte-vector forest

```
/*  C-like pseudo-code for Flip-and-Test with a byte-vector of level k.
 *  BVector         : char *              /* a bytevector array */
 *  index           : int                 /* index for s-array */
 *  BVector_size  : int                   /* the size of byte vector */
 */
int j     =    k-1;
int i     =    BVector_size - (0x8 << (3*j));
int pos  =    index;
for(    ;   i >= 0;  i  -= (0x8 << (3*(- - j))) ) {
      BVector[i+pos] = 0x0 ;
      __sync_synchronize() ;            /* memory barrier */
      if( * (long *) ( BVector + i + (pos&0xFFFFFFF8 ) ) !=  0 )
           return;
      pos>>=3;                             /*position for a next level vector*/
}
```

(b) Latch-free pseudo code for Flip-and-Test

Figure 8: Hierarchical free-list maintenance.

4.3 Pre-allocation of Locks

A background pre-allocator will allocate additional segment arrays if the number of unused segments falls below a low-water mark (i.e. the highest used segment is close to the head). This is done by finding an index-array entry whose valid flag is FALSE, starting from the tail pointer. This array is then moved to the head, its valid flag set to TRUE, and the head and possibly tail pointers are incremented (modulo array size). If no invalid segment is found, a new one must be allocated and initialised. As long as lock consumption does not outrun the pre-allocator, pre-allocation and lock consumption never overlaps, avoiding contention. However, a dangerous race condition in this producer-consumer loop occurs if `lock_create()` finds no free locks. We use the CAS instruction to make `lock_create()` (consumer) wait until the pre-allocator (producer) creates (or recycles) and initializes a new free segment array.

4.4 Lazy De-allocation of Locks

Bulk cleanup of locks happens at the granularity of segment arrays, once the whole array is invalidated (and the safety condition is satisfied). Simple approaches to detecting the invalidated state of the whole segment array would be (1) scanning the entire segment array frequently, or (2) keeping a single reference counter, which should be atomically incremented/decremented by concurrent transactions. The first approach needs a dedicated scanning process (not efficient), and the second one has a shortcoming of serializing atomic instructions on a single shared variable, which we aim to avoid. For efficient detection of invalid arrays, we associate each segment array with a hierarchical free list, as shown in Figure 8.

The list uses hierarchical byte-vector compression. At the leaf level, each byte represents a valid flag for an individual lock. Each byte of a non-leaf level compresses 8 bytes of the level below, and maintains the invariant that if the byte is zero, all the lower-order bytes mapped by it are zero as well. The top-level byte is the valid flag for the whole segment array.

The free list is updated latch-free (using hierarchical byte flipping implemented in the RAW pattern) whenever the state of a lock is changed to "OBSOLETE" (**R2** in Figure 3). The transaction first clears the valid flag of the lock, and invokes a memory barrier. Then it reads the complete 8-byte word containing the valid flag just cleared. If the word is zero, the transaction clears the next byte up the hierarchy. This is repeated until either a non-zero word is found, or the top-level valid flag is cleared (red line in Figure 8(a)). Flags are encoded as bytes (rather than bits) as the x86 architecture supports atomic updates of bytes but not of individual bits. Figure 8(b) is the C-like pseudo-code for the implementation of latch-free Flip-and-Test. This pseudo code depends on the assumption that, before executing Flip-and-Test, `next_pointer_update()` is performed to correctly adjust pointer values (Figure 4). This ensures that transactions started after the release of a lock can never see the obsolete lock.

Once the top-level valid flag is cleared, the complete segment array and the locks it references can be cleaned up (recycled) *as soon as the safety condition holds*. As indicated in Section 4.1, we do this by keeping for each segment track of the highest commit ID, T_M, whenever a lock is freed. The nontrivial part is to keep a system-wide oldest active transaction ID, $T_{\min}$. This bookkeeping can be done rather easily by using an existing global transaction table (or list) in database systems. For example, MySQL maintains a global shared transaction list, and it adds a newly started transaction to the end of the list, while removal can happen anywhere. This global list meets two important invariants; (1) the youngest live transaction is placed at the end of the list and (2) the oldest live transaction is always at the beginning of the list. The ID of the first transaction in the list is the one that we look for. By comparing that oldest live transaction ID with T_M for a given segment array, we can decide whether or not the segment array can be safely recycled.

4.5 Lock Manager Architecture

The resulting lock manager architecture is shown in Figure 9. The traditional hash-based lock manager architecture [8], prevalent in contemporary database engines, is augmented by the lock index data structure.

A new lock is obtained by using an atomic `fetch_and_add` on a shared pointer into the segment array; something like a *sloppy counter* [3], could be used should this atomic increment turn into a bottleneck. Once a transaction has obtained a unique lock ID, it can access the lock data structure directly. Locks released by committed transactions are not safe to be cleaned, since some old active transactions can still hold the references to these locks. Advancing the head

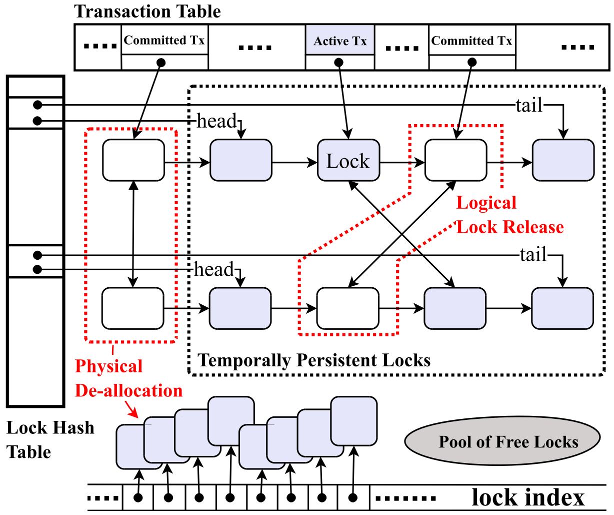


Figure 9: High-level view of the lock manager data structures.

pointer of a lock list eventually makes obsolete locks ready for clean-up, and we then physically de-allocate those locks.

MySQL recycles TRANSACTION objects. We therefore have locks point not at the TRANSACTION object of its owner, but a small shadow transaction data structure, which is also indexed by transaction ID. These shadow objects can be cleaned up safely when the segment array is cleaned, as the safety criterion on locks ensures that the transactions are no longer active.

5. EVALUATION

We measured multicore scalability on two open-source and one commercial database: MySQL-5.6.10-GA with InnoDB storage engine-1.2.10, Shore-MT-6.0 (the official Wisconsin branch), and a commercial DBMS X.[3] We also measured an implementation of our design (Section 3 and Section 4) for a reduced-latching lock manager using staged allocation and de-allocation; this was coded as modifications to the InnoDB storage engine as back-end for the same MySQL code, and we refer to it below as "Our System."

5.1 Evaluation Setup

5.1.1 System Setup

All databases are running on a 32-core Dell PowerEdge server whose hardware characteristics are listed in Table 1. The operating system is Linux 3.1.5, which incorporates the scalability improvement patch by Boyd-Wickizer *et al.* [3].

We run MySQL with the default configuration, except for the following parameters:

`max-connections`	1000
`innodb_buffer_pool_size`	20 GiB
`innodb_log_file_size`	256 MiB
`innodb_log_buffer_size`	16 MiB
`innodb_flush_method`	fsync
`innodb_flush_log_at_trx_commit`	2

The last setting (2) means the log flushing policy of "writing at commit, flushing once per second." Shore-MT is also configured with 20 GiB of buffer pool (`sm_bufpoolsize`), 256 MiB of log file (`sm_logsize`), and 16 MiB of log buffer (`sm_logbufsize`). All transactions in Shore-MT experiments are committed enabling the "`lazy`" option to enable the lazy log-flushing policy. Shore-MT provides the "group commit" function that enables multiple transactions to commit together with a single log record, and this leads to modification of the benchmark program. Thus, we do not use the "group commit" function in all Shore-MT experiments. DBMS X is configured with similar parameters: the same size of buffer pool, log file, and log buffer, and the group commit policy.

Component	Specification
Processors	8-Core Intel Xeon CPU E7-8837
Processor Sockets	4 Sockets
Hardware Threads	32 (No HyperThreading Support)
Clock Speed	2.66 GHz
L1 D-Cache	32 KiB (per core)
L1 I-Cache	32 KiB (per core)
L2 Cache	256 KiB (per core)
L3 Cache	24 MiB (per socket)
Memory	128 GiB DDR3 1066 MHz
Network	Ethernet 1 Gbps

Table 1: Dell PowerEdge R810 hardware specifications.

We use the isolation level `SERIALIZABLE` in all databases, and in addition we have run MySQL with `REPEATABLE READ` (RR) isolation for comparison. For Shore-MT, we use key-value locking for the B-tree index and the record-level locking iterator in accessing data records.

All experiments have a single database instance running on the server. A varying number of clients is emulated on a separate client computer (also multi-core, 128 GiB of RAM, running Linux 3.0.0). To expose performance bottlenecks in the lock manager, we configure all databases to have plenty of buffer pool space (i.e., 20 GiB). We store all data to *tmpfs*, an in-memory file system, to avoid disk bottlenecks.

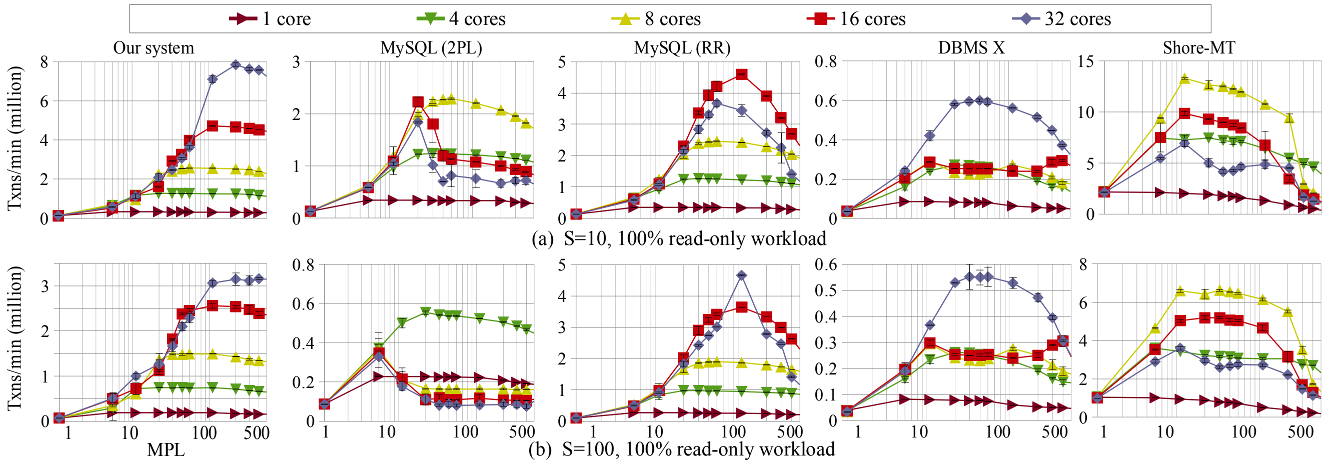
Client and server machines are connected with a 1 Gbps Ethernet network. The benchmark client runs the OpenJDK 64-Bit Server Java VM (build 19.0-b09, mixed mode) with the MySQL connector JDBC driver version 5.1.1. The only exception is that in the Shore-MT experiments, all clients are running on the server machine and directly call API functions of Shore-MT to access data, eliminating communication overhead. This is the setup used by Johnson *et al.* [11], which we retain to make our results comparable to theirs. This gives Shore-MT a potential performance advantage over the other systems, but the purpose of our evaluation is to examine scalability, not to compare performance across databases.

5.1.2 Microbenchmark

Our microbenchmark schema uses three tables called `txbench-{1, 2, 3}` with two non-null integer and ten variable sized character columns (`b_value-{1, 2, ..., 10}`); one of the integer columns (`b_int_key`) is a primary key. Each table is populated with 100 k randomly chosen items.

We use two types of queries: query transactions (read-only) and update-after-read transactions (read-update). A transaction reads S data items, either $S = 10$ or $S = 100$.

[3] The commercial system's license prohibits us from identifying it.



(a) S=10, 100% read-only workload

(b) S=100, 100% read-only workload

Figure 10: Throughput as a function of load (MPL) for varying number of cores.

To create *rw*-conflicts, we configure the read-update transactions to use the following access rules: a transaction reading data items from `txbench-i` updates rows from `txbench-j`, where `j=i + 1 mod 3`.

The read-only transaction consists of a single Select-From-Where query:

```
SELECT sum(b_int_value)*rand_number FROM txbench-i
 WHERE b_int_key > :id and b_int_key <= :id+S
```

This means that the DBMS scans S rows in the table and aggregates their integer column. This keeps the size of the result small (a single integer), to minimise network traffic.

The read-update transaction first reads S rows from `txbench-i` and updates $0.2 \times S$ rows from `txbench-((i+1)%3)`. The reading part of this transaction uses the same range query as the read-only transaction, and the update part consists of just a single SQL statement (again to minimize network cost):

```
UPDATE txbench-((i+1)%3) SET b_value-k = rand_str
 WHERE b_int_key = :id1
       OR b_int_key = :id2
```

Note that the read-update transactions are not serializable in `REPEATABLE READ` isolation due to cyclic dependency among these transactions. Our measurements are done with a limited implementation, to read and write records (but not to insert or delete).

We vary the multiprogramming level (MPL) from 1 to 500, with all clients trying to execute transactions as fast as possible without think time. Each experiment was repeated five times, with each run consisting of one minute ramp-up period and one minute measurement period. All plotted points in the figures of Section 5 are the average of these five test runs.

5.1.3 *System configuration and profiling*

To profile various system activities, we used OProfile, a system-wide, statistical, continuous profiler for Linux systems. We also used Intel VTune to verify function call traces. We use profiling to break total execution time into four categories: Idle for the idle state, Kernel for Linux kernel functions, Database for non-mutex related database functions, and Mutex for all mutex related functions in the DBMS. For DBMS X we cannot break down the time spent inside the database, so we use DBMS X for the total time spent inside the database code (corresponding to the sum of Database and Mutex).

To evaluate the impact of various CPU configurations, we use the CPU hotplug feature of modern processors. The hotplug function can make CPUs (or cores) available or unavailable to the Linux kernel. For a given number of cores, we use the minimum number of sockets (i.e., 1 socket for ≤ 8 cores and 2 sockets for $9 \cdots 16$ cores). This minimises cache coherence costs and thus represents a best case for latch-based approaches.

5.2 Experimental Results

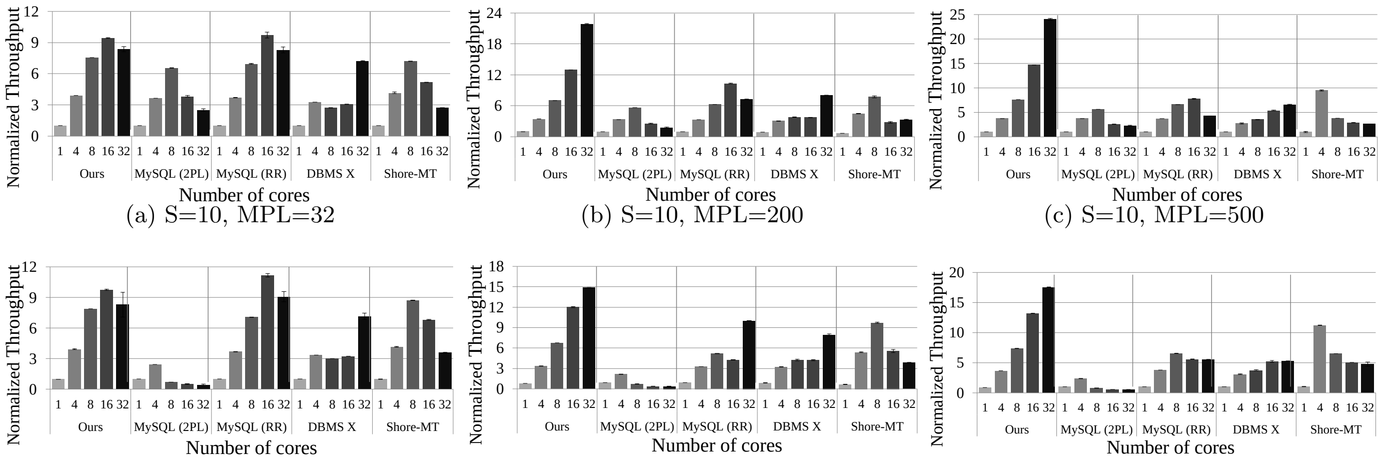
5.2.1 *Performance under load*

We evaluate the scalability of the various database systems with respect to the number of cores utilized, and to the workload. A partial result was shown in Figure 1, which motivated our reduced-latch approach to lock management. Our purpose here is to confirm that the collapse is not an artefact of a poor implementation of one system, but rather applies to a diversity of platforms, and also to show that our design eliminates the collapse.

Figure 10 shows the complete results for read-only transactions. The leftmost graphs (Our system) show MySQL with the implementation of our reduced-latch, staged lock manager, while the other graphs show unmodified existing systems. Note that the scales on the y-axis differ, as attempting to display all graphs on the same scale would make the results of the less-performing systems unreadable. In any case, our purpose is not to compare performance across platforms (except between MySQL and Our System), but to show which systems experience performance collapse, at high (but not logically contending) load and on many cores.

We can see that even with 4 cores, all the vanilla configurations show a throughput peak, with performance degrading when load increases further.

At 16 cores (2 sockets) the performance degradation turns into an outright collapse for both open-source systems. Interestingly, DBMS X roughly maintains peak performance, but its peak is about an order of magnitude less than that



(a) S=10, MPL=32 (b) S=10, MPL=200 (c) S=10, MPL=500

(d) S=100, MPL=32 (e) S=100, MPL=200 (f) S=100, MPL=500

Figure 11: Read-only throughput normalised to single-core performance.

of the open-source systems, and throughput remains below that of the open-source systems even at the highest loads we applied.

At 32 cores (4 sockets), the performance collapse can be observed for all vanilla configurations, including DBMS X. For MySQL, even the peak performance is degraded below that of configurations with fewer cores. The best scalability is exhibited by MySQL in the RR configuration with the larger (S=100) transactions.

In contrast, our system never shows a performance collapse, at worst there is a small degradation from the peak at increasing load. Furthermore, adding cores increases the peak throughput in all cases. The throughput of our systems is also always at least as high as that of the vanilla MySQL(2PL) system it is based on. For the smaller transactions it even matches or exceeds the performance of the RR configuration. The reason why our system outperforms MySQL(RR) is because MySQL(RR) uses the transaction-system mutex whenever it opens a consistent read-view, when a transaction begins or commits. As MPL grows, the mutex duration also increases and incurs high contention among transactions. This leads to the unexpected performance behavior under high load of smaller transactions.

5.2.2 *Core scalability*

For a more detailed analysis of the scalability of the different core configurations with increasing core count, we zoom in on three "interesting" values of MPL: 32, 200 and 500.

We choose MPL 32 as a good proxy for peak throughput of the vanilla databases, as all of them, except MySQL(RR), peaked in the vicinity of that load. Furthermore, 32 is the maximum core count, so in the full hardware configuration there would be one transaction per core, representing something of an ideal case. MPL 200 is interesting as our system is at or near peak performance there. MPL 500 is the most extreme load we ran, and the one which triggered the most serious performance collapses.

The results are shown in Figure 11. MPL 32, subfigures (a) and (d), is fairly benign, all systems exhibit at worst a small degradation of throughput with increasing core count (although MySQL(2PL) with S=100 degrades starting with as few as 8 cores).

At MPL 200, MySQL shows a significant degradation, and Shore-MT performance plateaus for small transactions (b), while the system degrades for the large transactions (e). DBMS X shows a somewhat strange behaviour, with flat performance at intermediate core counts but roughly doubling from 16 to 32 cores. Our system is well behaved, scaling well with small transactions and saturating but not degrading for the large transactions.

At the highest load, our system scales almost perfectly with the small transactions until 16 cores and then saturates, with the large transactions the saturation starts earlier. DBMS X scales similar to or system for large transactions, but much worse for small ones. Shore-MT scales to 4 cores but then saturates (but remember that this system shows performance degradation under high load even on a single core). MySQL shows performance degradation with increasing core numbers.

5.2.3 *Where does the time go?*

Profiling allows us to peek inside the systems, as shown in Figure 12. The leftmost column stands out for its virtual absence of the (yellow) mutex times, thanks to latch-free implementation of lock management. This is in stark contrast to the second column, showing the MySQL system ours was derived from, which has in excess of 50 % of time spent spinning on mutexes. The lowest fraction of spinning is in the 8-core configuration with small transactions. As Figure 10 shows, this is the configuration where MySQL performs best. It is obvious that the performance degradation of MySQL results from high mutex contention.

Surprisingly, at 32 cores the overloaded MySQL system actually has significant idle time! MySQL implements all mutexes using the atomic Test-and-Set instruction, with an additional random backoff mechanism. After a failed attempt to acquire a mutex it busy-waits. After the predefined (but configurable) number of failures, the mutex routine puts the calling thread to sleep until the mutex is released. This explains the idle time at times of high mutex contention.

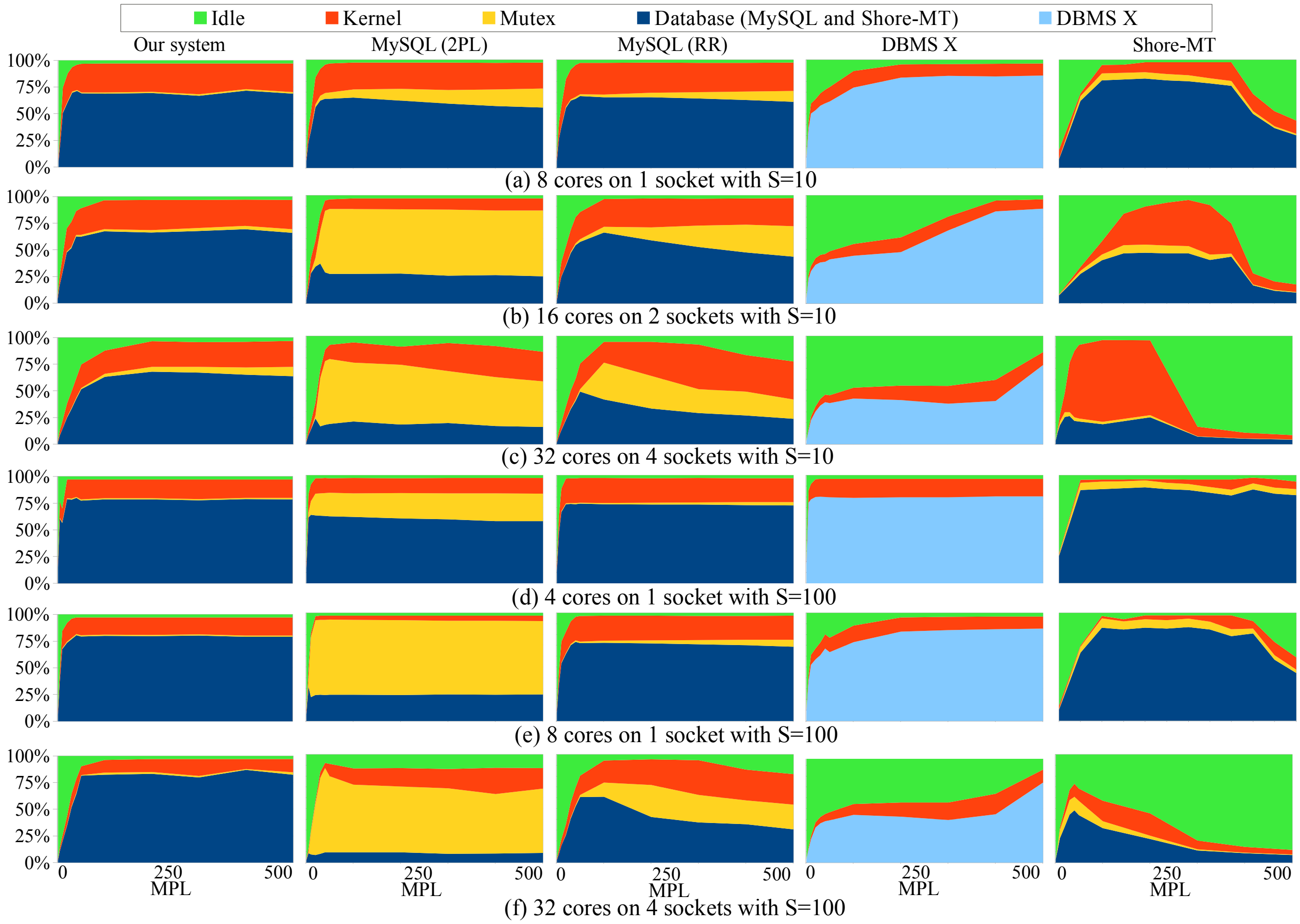


Figure 12: Breakdown of profiled results of our system, MySQL (2PL), MySQL (RR), DBMS X, and Shore-MT (from left to right) with different core/socket configurations (from top to bottom) and 100% read-only workloads.

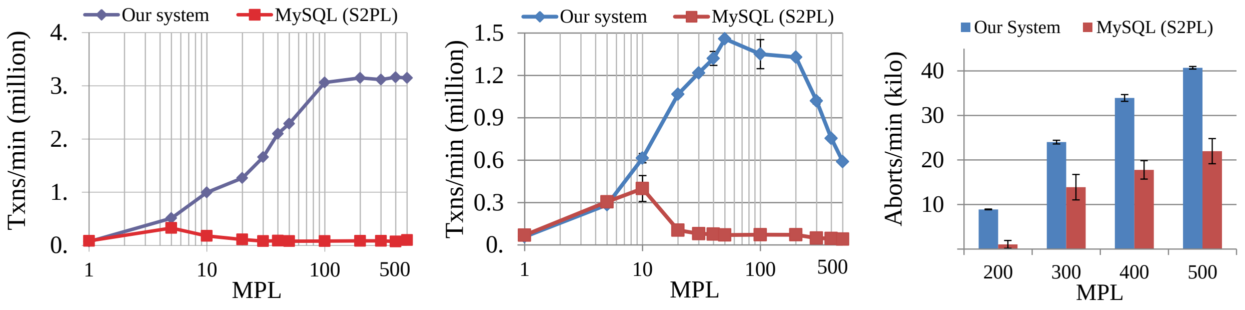
Also noticeable is that for small transactions, MySQL(RR) also shows significant mutex overhead (as well as idle time), despite the lower isolation level. The explanation is that in RR isolation, when a read view is to be opened, MySQL uses the transaction-system mutex to find a serialisable point. Consequently, for the larger transaction size the relative mutex overhead is much reduced.

DBMS X, where we could not break down the in-database time due to lack of access to internals, also exhibits a large amount of idle time in overload situations.

Shore-MT does not show much mutex time, but large amounts of OS time. At early stages of MPL, Shore-MT spends most time in the operating system. At very high load, the system is mostly idle. Further analysis can explain these two phenomena. Shore-MT uses the `pthread_mutex_lock/unlock` routines as default latch functions, and those pthread functions are based on the `futex` system call. If the latch is uncontended, `futex` can acquire the latch with simply calling `atomic_swap` and memory barrier functions and return the locked latch without context switch. This explains why Shore-MT spends most of time in Kernel, especially in the kernel function `cpu_relax()`, which keeps invoking a memory barrier until it gets to hold a mutex. However, if latch contention is detected during the futex call, the calling process or thread sleeps until the latch is released. This explains the huge idle time in high load conditions. If Shore-MT is configured to use MCS locks [13] instead of pthread mutex functions, the throughput at MPL 500 on 32 cores drops dramatically to about 5,000 tpm, with >95% of time spent in the idle state.

5.2.4 *Read-update workload*

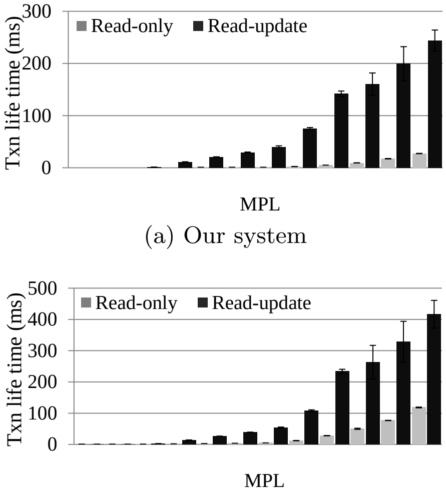
Our previous experiments included no updates, so that there was no logical contention at all between locks, in order to focus attention on the contention between latches. However, all OLTP workloads do include some updates, which are the reason even read-only queries need to set locks at all. Figure 13 shows a direct comparison of our system with MySQL(2PL). Figure 13(a) repeats the throughput data for read-only workloads (32-core lines of the two leftmost graphs in the bottom row of Figure 10). Figure 13(b) shows the same configuration, except that we now use a read-update workload with 20 % update transactions. Update transactions acquire locks which are incompatible with those of both read-only and update transactions, and this leads to blocking and deadlocks. Figure 13(c) shows the abort rates for the workload with 20% update transactions; these aborts in-



(a) Throughput, R/O workload (b) Throughput, with 20% updates (c) Aborts, with 20% updates

Figure 13: Our system vs. MySQL(2PL) on 32 cores, S=100.

crease dramatically at high loads, with up to 46 % of MySQL transactions aborted at top load. Our system has similar overall abort rates, but due to the higher throughput, the fraction of aborts is much lower; at worst about 6.7 % of transactions are aborted with our system. This shows the benefit of concurrent deadlock checking. Our system shows performance degradation in high workload due to lock conflicts, not because of latch contention.



(a) Our system

(b) MySQL(2PL)

Figure 14: Transaction lifetimes for configuration of Figure 13(b).

An examination of transaction lifetimes, shown in Figure 14, provides additional insight. The duration of read-only as well as update transactions increases dramatically with load in MySQL, indicating long periods of spinning on mutexes. In contrast, in our system the durations of R/O transactions do not show unexpected increase by load, and even the duration of update transactions increases only moderately with load, as one would expect in an ideal system.

5.2.5 *Lock hash chain*

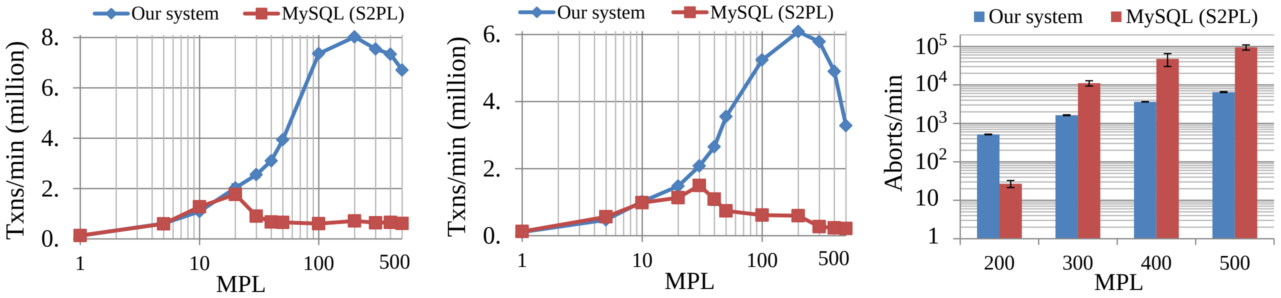
A potential bottleneck of our staged lock management is the lock hash list, which needs to be scanned during lock acquisition (lines **A4. . . A15** of Figure 3). In our implementation, this list is potentially much longer than in vanilla MySQL, as it may contain many obsolete locks. We therefore devised a benchmark which stresses this part of the code, by increasing contention for data items.

Our *hotspot* benchmark is a variant of our standard $S = 10$ benchmark with 20 % update transactions. In this case we restrict transactions to access only 5 % of all table rows, thus increasing logical contention twentyfold. Figure 15(b) shows the results for throughput and Figure 15(c) shows abort rates. The standard $S = 10$, R/O benchmark throughput is shown in Figure 15(a) for comparison. We can see from Figure 15 that the hotspot load degrades throughput of our system moderately again due to lock conflicts, the degradation is actually more pronounced in MySQL(2PL). Under the read-only workload (Figure 15(a)), the throughput of our system hits the peak of 8 million tpm (at MPL 200) and decreases to 6.8 million tpm (at MPL 500), while that of MySQL(2PL) reaches the peak of 1.8 million tpm (at MPL 20) and collapses to 0.7 million tpm (at MPL 500). With the read-update workload (Figure 15(b)), the throughput of our system has the peak of 6.1 million (at MPL 200, 75 % of that of the R/O workload) and then decreases to 3.3 million tpm (at MPL 500, again 48 % of R/O). In contrast, MySQL(2PL) achieves a peak of 1.6 million tpm (at MPL 20, 90 % of R/O) and collapses to 0.2 million tpm (at MPL 500, 30 % of R/O). This is also reflected in the abort rates, which for our system are a tiny 0.19 %, while MySQL(2PL) suffers 44 % aborted transactions. In other words, there is no indication that the increased length of our hash chains is becoming a bottleneck.

6. CONCLUSION

We have found that contemporary database systems are not yet ready for the multicore age. Our evaluation has demonstrated a collapse of transaction throughput under high load, even read-only load, for all the database systems we analysed. In the case of the open-source systems, MySQL and Shore-MT, we could identify latch contention in the lock manager as the bottleneck. While we could not perform the same in-depth analysis on the commercial DMBS X, its observable behaviour is similar enough to the open-source systems to suspect that the cause is similar.

We proposed to address this problem by an improved implementation of the lock manager, which adopts the RAW pattern and barrier synchronization to greatly reduce the use of latches. We have described the design for such a lock manager which is based on staged lock management, where locks are pre-allocated and lazily de-allocated in bulk,



(a) Throughput, R/O workload (b) Throughput, 20% updates, hotspot (c) Abort rate, 20% updates, hotspot

Figure 15: Exploring hotspot behaviour, 32 cores, S=10.

and deadlock checking can be done concurrently. We coded this design as a variant of the backend storage of MySQL. Our experiments show that this approach (i) eliminates the performance collapse by essentially eliminating latch contention, and (ii) always achieves at least the same performance as the baseline system.

The potential weak point of our staged approach is the longer hash chains, resulting from obsolete locks which have not yet been cleaned up. Our attempts to stress this part of the system by forcing a high rate of contention on individual data items did not indicate that this causes any performance problems. We conclude that the staged, reduced-latch approach to lock management looks promising.

7. ACKNOWLEDGMENTS

We would like to thank Michael Cahill for providing valuable comments on the initial draft, and SeongJae Park for helping in many places. We would like to thank the anonymous SIGMOD reviewers for insightful comments which helped to improve the paper.

8. REFERENCES

[1] H. Attiya, R. Guerraroui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated. In *POPL*, pages 487–498, 2011.

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

[3] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *OSDI*, pages 1–8, 2010.

[4] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable Locks are Dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[5] A. T. Clements, F. Kaashoek, and N. Zeldovich. Scalable Address Spaces Using RCU Balanced Trees. In *ASPLOS*, pages 199–210, 2012.

[6] G. Graefe. A Survey of B-tree Locking Techniques. *ACM Trans. Database Syst.*, 35(3):16:1–16:26, July 2010.

[7] G. Graefe, H. Kimura, and H. Kuno. Foster B-trees. *ACM Trans. Database Syst.*, 37(3):17:1–17:29, Sept. 2012.

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

[9] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. *Architecture of a Database System*. Now Publishers Inc., Hanover, MA, USA, 2007.

[10] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[11] 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. ACM, 2009.

[12] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Scalability of Write-ahead Logging on Multicore and Multisocket Hardware. *The VLDB Journal*, 21(2):239–263, 2012.

[13] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Computer Systems*, 9(1):21–65, 1991.

[14] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented Transaction Execution. *PVLDB*, 3(1-2):928–939, 2010.

[15] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: Page Latch-free Shared-everything OLTP. *Proc. VLDB Endow.*, 4(10):610–621, 2011.

[16] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *EuroSys*, pages 337–350, 2012.

[17] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database Engines on Multicores, Why Parallelize When You Can Distribute? In *EuroSys*, pages 17–30, 2011.

[18] C. Schimmel. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley Professional, 1994.

[19] J. Sewall, J. Chhugani, C. Kim, and P. Dubey. PALM: Parallel Architecture-Friendly Latch-Free Modification to B+Trees on Many-Core Processors. In *PVLDB*, pages 795–806, 2011.

[20] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.