A

# A Scalable Lock Manager for Multicores

HYUNGSOO JUNG, NICTA
HYUCK HAN, Dongduk Women's University
ALAN FEKETE, University of Sydney and NICTA
GERNOT HEISER, NICTA and UNSW
HEON Y. YEOM, Seoul National University

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 **into over-saturation (where there are more request threads than cores),** even for a workload with little or no logical contention for locks, **such as a read-only workload**. 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

Additional Key Words and Phrases: Transactions, lock manager, multicore, scalability, lock-free synchronization

**ACM Reference Format:**
Hyungsoo Jung, Hyuck Han, Alan Fekete, Gernot Heiser, and Heon Y. Yeom. A Scalable Lock Manager for Multicores. *ACM Trans. Datab. Syst.* V, N, Article A (January YYYY), 30 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

## 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 [Johnson et al. 2009b; Salomie et al. 2011] showed that many open-source and commercial database engines exhibit limited multi-core scalability, and even performance collapse, due to bottle-

Author's addresses: H. Jung, NICTA, Australia; H. Han, Department of Computer Science, Dongduk Women's University, Korea; A. Fekete, School of Information Technologies, University of Sydney, and NICTA, Australia; G. Heiser, NICTA and UNSW, Australia; H. Y. Yeom, School of Computer Science and Engineering, Seoul National University, Korea.

DOI: http://dx.doi.org/10.1145/0000000.0000000

necks in their lock, log or transaction manager. This has triggered significant research activity [Horikawa 2013; Boyd-Wickizer et al. 2010; Johnson et al. 2009a; Johnson et al. 2010; Clements et al. 2012; Johnson et al. 2012; Pandis et al. 2010; Pandis et al. 2011; Salomie et al. 2011; Pesterev et al. 2012; Sewall et al. 2011] 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.)

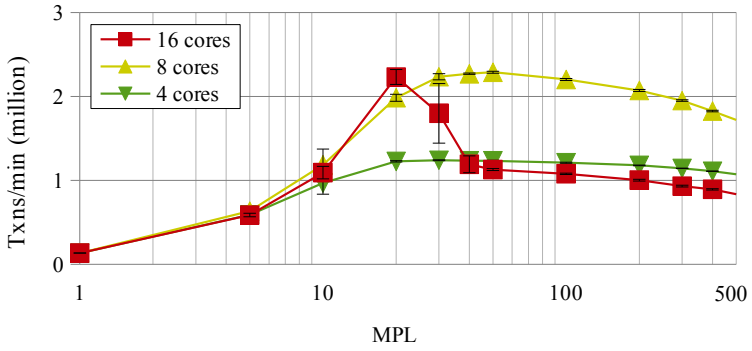



Fig. 1. MySQL throughput under varying multi-programming level (MPL) on 4 (green), 8 (yellow) and 16 cores (red).

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. **We note that these collapses all occur when the system is over-saturated, that is, the number of request threads being handled concurrently exceeds the number of available cores.** The peak throughput of 16 cores is also below that of 8.

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 **which in InnoDB protects the global table of locks**. Even though the mutex is only held briefly, *cache-line bouncing* [Schimmel 1994] makes acquiring or releasing it expensive under high contention [Boyd-Wickizer et al. 2012]. **While other systems use finer-grained latches, we have found that some important examples still suffer from a performance collapse when over-saturated.**

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 phases: 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 article 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 and Section 5.3).
— We propose a design for the lock manager in which much less latching is needed, and we describe its implementation in MySQL (Section 3). The key idea of our design is bulk allocation and de-allocation of locks, asynchronous to transaction processing (Section 4);
— We propose an epoch-based lock reclamation scheme to make the approach work with long-running transactions (Section 4).
— We evaluate our implementation, demonstrating that staged lock handling avoids the throughput collapse and also leads to improved peak throughput compared to the baseline MySQL platform (Section 5.2 and Section 5.3).

A preliminary version of this work appeared as [Jung et al. 2013]. We now extend the technique to support long-running transactions, and all experiments have been redone with a system including this feature.

## 2. BACKGROUND AND RELATED WORK

This article 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. [1987] and Weikum and Vossen [2001]. The internal design of a DBMS is described by Hellerstein et al. [2007]. 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 [1992].

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 [2010]. Some early open-source DBMS code used a coarse-grained latch on the whole of the kernel internals, but the current InnoDB system we used as the baseline for our evaluation has moved to a more sophisticated design with separate latches for different parts of the kernel, such as the lock manager, the buffer pool, etc. **It is reported that many commercial platforms use a latch for each hash-bucket of locks within the lock manager.**

**A substantial body of recent research has addressed some bottlenecks in transaction processing on multi-core platforms. An early study [Johnson et al. 2009b] identified a variety of bottlenecks across the storage manager, and proposed the design of Shore-MT to alleviate them. Within the lock manager, this design used a mutex per bucket (rather than a mutex for the lock manager as a whole), it rewrote several critical sections to shorten them and avoided nested latches, and it used a compare-and-swap on a latch-free pool of lock requests. Shore-MT has been the platform for many subsequent studies.**

**Several papers have improved multicore scalability of different parts of the storage manager. Latching for index structures was studied in Sewall et al. [2011], Pandis et al. [2011], and Graefe et al. [2012]. Scalability of logging was the focus of Johnson et al. [2012]. At a more detailed level of the system design, Johnson et al. [2010] proposed better scheduling for the requests on a latch, to mitigate the convoy phenomenon [Blasgen et al. 1979] (a quasi-deadlock to which queue-based spinlocks are vulnerable, that can happen if a latch is held by a thread that has been scheduled out, leaving all waiting threads unable to advance even when they are active). The convoy problem can be worse in multicores where it meets other aggravating factors (such as the cache invalidation storm).**

**Closer to our work has been proposals to improve the multicore scalability of locking. Johnson et al. [2009a] proposed a technique, called speculative lock inheritance (SLI), that identifies a working set of frequently accessed locks and transfers the ownership of such locks to different transactions without latching. In Kimura et al. [2012], a lightweight intent lock (LIL), which maintains a set of lightweight counters in a global lock table, is proposed to shorten code paths for locking. Ren et al. [2012] propose very lightweight locking (VLL) to avoid a global lock manager; instead of using a global lock manager, VLL associates a pair of counters with each tuple, tracking how many transactions request read/write accesses.**

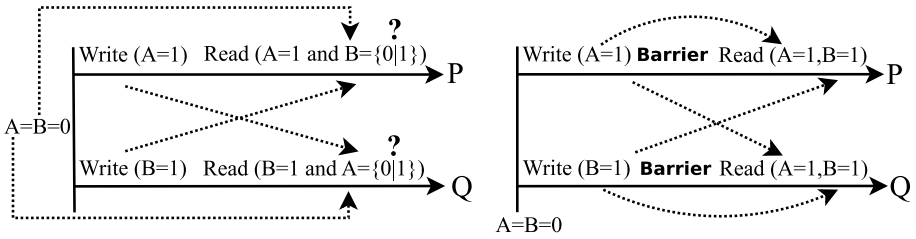
**Other recent works [Horikawa 2013; Han et al. 2014] have studied multicore scalability for systems with different concurrency control, such as those based on multiple versions and snapshots.**

**More radical yet are some proposals for complete re-architecting the storage system for a multicore platform. Salomie et al. [2011] suggests a distributed approach, where each core has specific roles, and so all logical concurrency control would be placed on a single core, different from where data access occurs. Pandis et al. [2010] introduced DORA, where the transaction execution moves among the threads, so that each thread can deal with localised data. This allows decentralising the lock management.**

**Unlike these works, our focus is to design a localized change to the lock manager of a platform using lock-based concurrency control, without changing what is locked or requiring other parts of the platform to be recoded. A preliminary conference version of some of our design has appeared [Jung et al. 2013] with the same goals, but it is more restricted because it does not deal with long-running transactions, and also the evaluation has now been improved and extended.**

### 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*




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

Fig. 2. Unsafe vs. safe RAW execution.

*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* (CAS) 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 the process writing to B in between [Attiya et al. 2011]. Use of the RAW pattern must ensure that concurrent processes only see a compatible (linearizable or sequentially consistent) state of shared variable. In reality, modern compilers may optimize the simple RAW code segment by reordering `load` and `store` instructions (i.e., out-of-order execution). In multi-threaded programs, different threads may see incompatible values owing to this instruction reordering. 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. The main role of memory barriers strictly prevents compilers from reordering instructions across the memory barrier.

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 [2008] 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, and Figure 4 illustrates these two operations with a schematic flow diagram in the context of transaction execution (top); other database engines use a similar approach. Most of the acquisition

| Traditional Lock Manager | Our Lock Manager |
|---|---|
| Lock Acquire in Growing Phase | |
| **mutex_enter(lock_table->mutex);**<br>n_lock = lock_create();<br>n_lock->state = ACTIVE;<br>lock_insert(n_lock);<br>for all locks (lock) in hash_bucket<br>  if (lock is incompatible with n_ lock)<br>  n_lock->state = WAIT;<br>  if (deadlock_check() ==TRUE)<br>  abort Tx;<br>  break;<br>  else<br>  continue;<br>  end if<br>end for<br>**mutex_exit(lock_table->mutex);**<br>if (n_lock->state==WAIT)<br>  **mutex_enter(Tx->mutex);**<br>  Tx->state = WAIT;<br>  **os_cond_wait(Tx->mutex);**<br>  **mutex_exit(Tx->mutex);**<br>end if | **A1:** n_lock = lock_create();<br>**A2:** n_lock->state = ACTIVE;<br>**A3:** **atomic_lock_insert(n_lock);** **S1**<br>**A4:** for all locks (lock) in hash_bucket<br>**A5:** if (lock is incompatible with n_lock)<br>**A6:** n_lock->state = WAIT;<br>**A7:** **atomic_synchronize();** **S2**<br>**A8:** if (lock->state==OBSOLETE)<br>**A9:** n_lock->state=ACTIVE;<br>**A10:** **atomic_synchronize();** **S3**<br>**A11:** continue;<br>**A12:** if (**new_deadlock()**==TRUE)<br>**A13:** abort Tx;<br>**A14:** break;<br>**A15:** end if<br>**A16:** end for<br>**A17:** if (n_lock->state == WAIT)<br>**A18:** **mutex_enter(Tx->mutex);**<br>**A19:** **atomic_synchronize();** **S4**<br>**A20:** if ( n_lock has to wait )<br>**A21:** Tx->state = WAIT;<br>**A22:** **os_cond_wait(Tx->mutex);**<br>**A23:** else<br>**A24:** n_lock->state = ACTIVE;<br>**A25:** **atomic_synchronize();** **S5**<br>**A26:** end if<br>**A27:** **mutex_exit(Tx->mutex);**<br>**A28:** end if |
| Lock Release in Shrinking Phase | |
| **mutex_enter(lock_table->mutex);**<br>for all locks (lock1) in Tx<br>  lock_release(lock1);<br>  for all locks (lock2) following lock1<br>  if ( lock2 doesn't have to wait )<br>  lock_grant( lock2 );<br>  lock2->state=ACTIVE;<br>  end if<br>  end for<br>end for<br>**mutex_exit(lock_table->mutex);** | **R1:** for all locks (lock1) in Tx<br>**R2:** lock1->state = OBSOLETE;<br>**R3:** **atomic_synchronize();** **S6**<br>**R4:** for all locks (lock2) that follow lock1<br>**R5:** **mutex_enter(lock2->Tx->mutex);**<br>**R6:** if ( lock2->Tx->state==WAIT &&<br>**R7:** lock2 does not have to wait )<br>**R8:** lock2->Tx->state=ACTIVE;<br>**R9:** lock2->state=ACTIVE;<br>**R10:** **atomic_synchronize();** **S7**<br>**R11:** **os_cond_signal(lock2->Tx);**<br>**R12:** end if<br>**R13:** **mutex_exit(lock2->Tx->mutex);**<br>**R14:** end for<br>**R15:** end for |

Fig. 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*.

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.

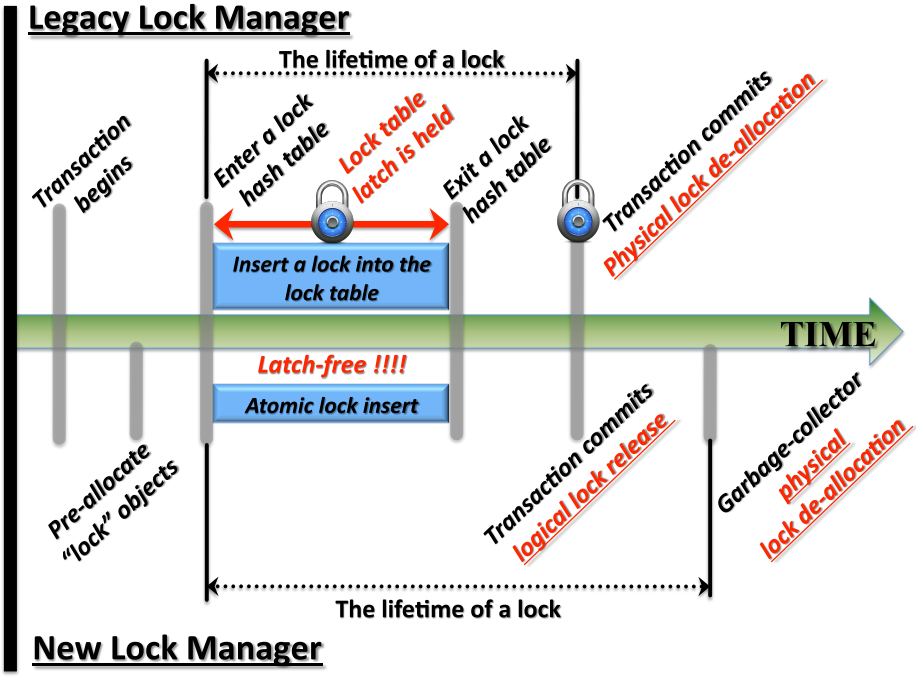



Fig. 4. A schematic flow diagram of lock acquisition and release operations in the legacy lock manager (MySQL) and the new lock manager.

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 and illustrated in Figure 4. 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[3]), 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

[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.
[3]We map this function to the full memory barrier instruction (i.e., `mfence`).

the lock is no longer in use. The deadlock-checking function (line **A12**) must be replaced by a race-tolerant implementation `new_deadlock()`.

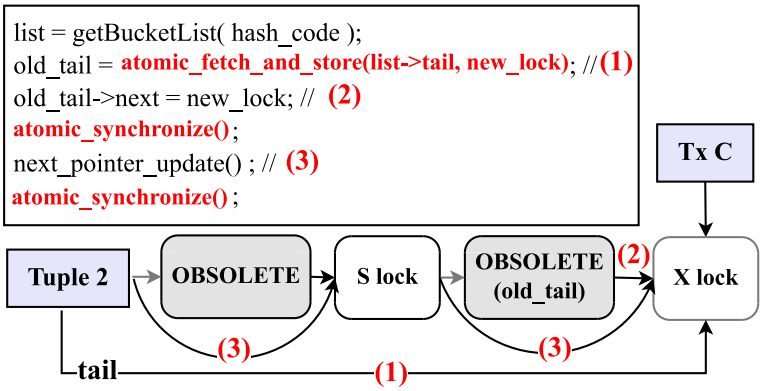



Fig. 5. Operation of `atomic_lock_insert()` on a lock hash chain.

Operation of `atomic_lock_insert()` is shown in Figure 5. 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*-based concurrent update [Herlihy and Shavit 2008] 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.

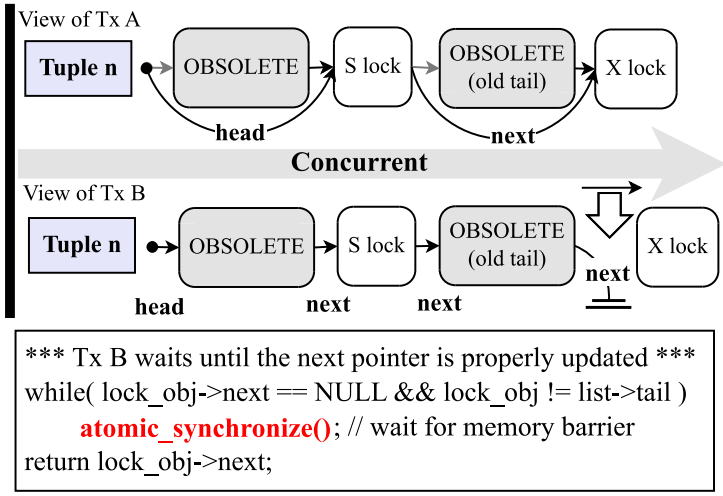



Fig. 6. Pseudo-code of latch-free list iteration.

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 6. 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 5, 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 contended 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. This affects to the lifetime of a lock object in the hash table. As shown in Figure 4, a lock object, after it's logically released, should remain in the lock hash table until a garbage-collector safely de-allocates it physically from the lock hash table.

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).

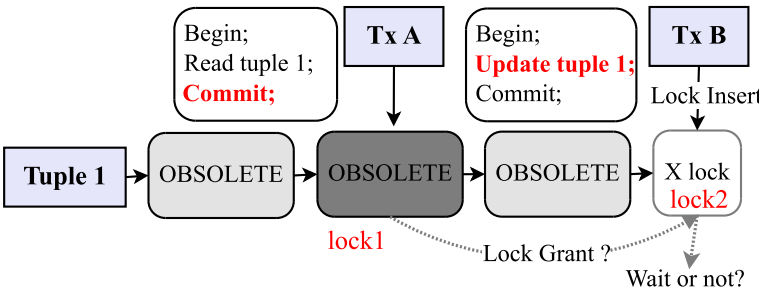
In our latch-free implementation, safety prevents a thread from changing the transaction state while other threads may access the state. Hence, the thread must abort its own transaction when detecting a deadlock, i.e., we can only implement the above policy (3).

### 3.4. Correctness

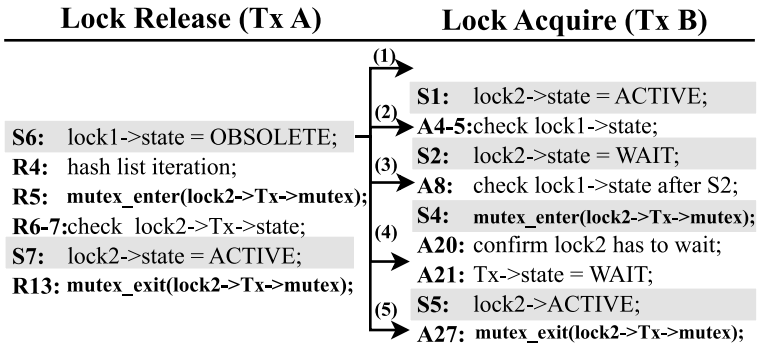
The main correctness concern is that concurrent acquisition and release operations operate correctly, and that deadlock detection is correct.

Figure 7(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 7(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 7(b).




(a) Illustration




(b) All possible schedules

| Schedule | Result |
|---|---|
| (1) S6 < S1 .. S7<br>(2) S1 < S6 < S2 .. S7<br>(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)<br>(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

Fig. 7. Concurrent execution of `Lock Acquire` and `Lock Release`.

The correctness argument is summarised in Figure 7(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 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.

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? Each lock belongs to a unique transaction which created it, and transactions do not pass to other transactions references to locks. We therefore previously formulated a sufficient safety criterion [Jung et al. 2013]:

THEOREM 4.1. *A lock L can be safely de-allocated if the oldest transaction in the system has started after L's transaction terminated.*

By Theorem 4.1, we can always de-allocate any locks released before the oldest active transaction from the lock table without risking the correctness, and we can progress as long as the oldest active transaction ID keeps increasing monotonically. However, if we have a long running transaction or a stalled transaction, Theorem 4.1 cannot be satisfied for locks released after the long running transaction started, and as a result database systems will sooner or later face memory exhaustion because the volume of released locks will continue to grow. In order to support long-running transactions, we provide a stronger and more general sufficient criterion:

THEOREM 4.2. *A lock L can be safely de-allocated if no other process (or thread) in a system holds any reference to L.*

Theorem 4.2 is stronger than Theorem 4.1 in a sense that Theorem 4.2 can provide a sufficient condition to the correct memory reclamation of released locks whereas Theorem 4.1 cannot: Even though there are long running transactions in a system, locks can be de-allocated by a garbage-collector if we can detect safe points where no other transactions hold references to a released lock.

This type of safe de-allocation, while reads are performed on dynamic data structures, is a challenging problem in lockless synchronization, and [Hart et al. 2007] conducted a performance study on three well-established memory reclamation schemes.

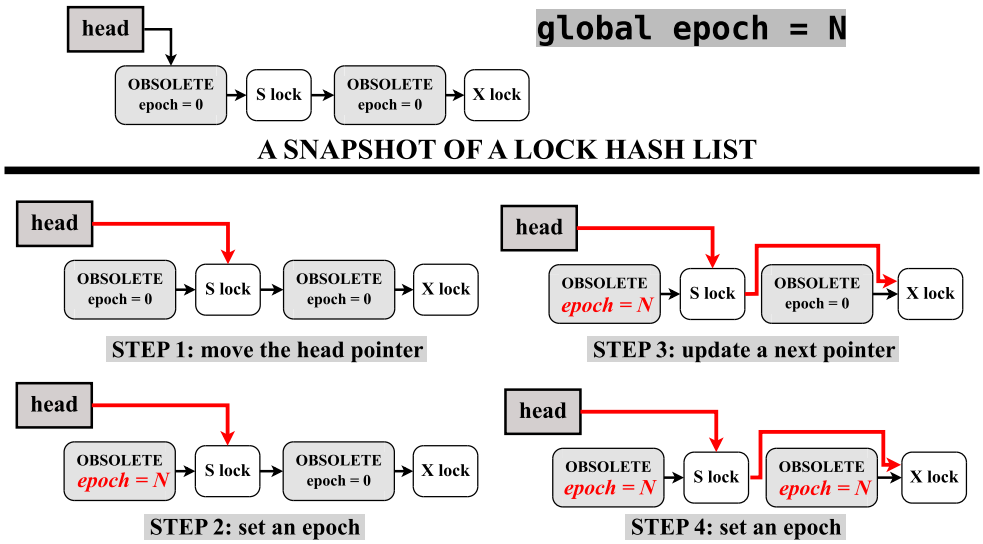



Fig. 8. Update on pointers and epochs.

In this article, we use an *epoch-based reclamation* (EBR) scheme [Fraser 2004] and adapt it to our lock manager when reclaiming (or de-allocating) a group of lock objects.

We first define the code section that accesses (or maneuvers) a lock (list) in the lock table as a *critical region*. When a lock L is released and is made invisible by manipulating pointers, we associate L with an *epoch* copied from the *global epoch* shared by all concurrent transactions. A transaction, when entering a critical region, first sets a local (per-transaction) epoch value to the global epoch, accesses lock objects, and clears the local epoch after executing the critical region. The global epoch is incremented by a transaction after a pre-defined number of executions of the critical region.

Incrementing the global epoch only succeeds if the local epoch of each transaction in a critical region is equal to the global epoch. This type of synchronization can be viewed as a *fuzzy barrier* [Gupta 1989]. It ensures that if the current value of the global epoch is set to $e$, then no other transaction can have a local epoch value of less than $e-1$, and it also implies that obsolete locks with an epoch of less than $e-1$ cannot be visible to any current or future transactions. As the global epoch increases, lock objects whose epoch is at least two less than the global epoch can be reclaimed [Fraser 2004].

In the EBR scheme, as in similar schemes, reclamation can stall when a transaction stalls before it clears its local epoch. Unless the database system aborts a stalled transaction which holds a reference to a released lock, reclamation will also stall.

The overall operation logic for update on pointers and epochs is described in Figure 8, which explains how pointers and epochs are updated upon detecting obsolete lock objects in the lock table. We use a snapshot of a lock hash list, which contains two obsolete locks, one is at the beginning and the other is in the middle of the list. In the `next_pointer_update()` function we move the head pointer, as it points to an obsolete lock, then we set that lock's local epoch to $N$, the global epoch. Advancing the head pointer continues until it finds an active lock.

Once the update on the head pointer is done, we update all next-pointers to skip obsolete locks in the middle of the list. We use the same approach to updating epochs: first update the next pointer and then set the epoch of the obsolete lock. Care must be taken when updating a next-pointer: We must never update the next-pointer of an obsolete lock, as this could lead to de-referencing a `null` pointer. Once pointers are

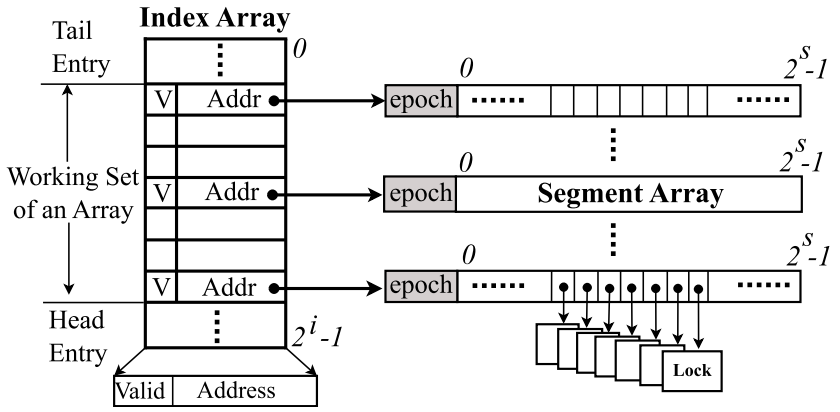



Fig. 9. Lock index data structure.

updated (red), transactions accessing the lock hash list can maneuver pointers without seeing any obsolete locks.

As we briefly described in Section 3.2, the new lock manager manages lock objects based on bulk allocation and de-allocation. This indicates that the lock manager does not have to assign an epoch number to each lock object. For bulk de-allocation of a group of locks, we only need to keep track of one epoch number for the group, and check that this epoch is at least two less than the global epoch. A simple way to do this is to copy the global epoch $e_g$, to the group-local epoch, $e_l$, whenever one of the group's transaction terminates. The group can be cleaned up when (i) all of its transactions have terminated and (ii) $e_l < e_g - 1$ for the group.

### 4.2. Lock Pool Implementation

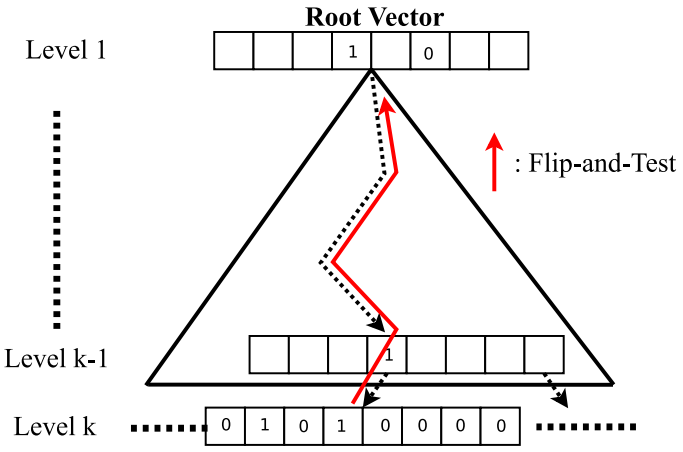
To support bulk allocation and de-allocation of locks, we use a data structure which can grow and shrink on demand and naturally groups locks of similar age, and allows us to locate locks quickly. Specifically, we use a page-table-like index structure to locate a lock from its ID, as shown in Figure 9.

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 epoch number for each segment array and 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 $\leq$head as well as those $\geq$tail. A third pointer (not shown) points to the highest (newest) used segment, all segments between that pointer and the head are unused.

### 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.




(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

Fig. 10. Hierarchical free-list maintenance.

### 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

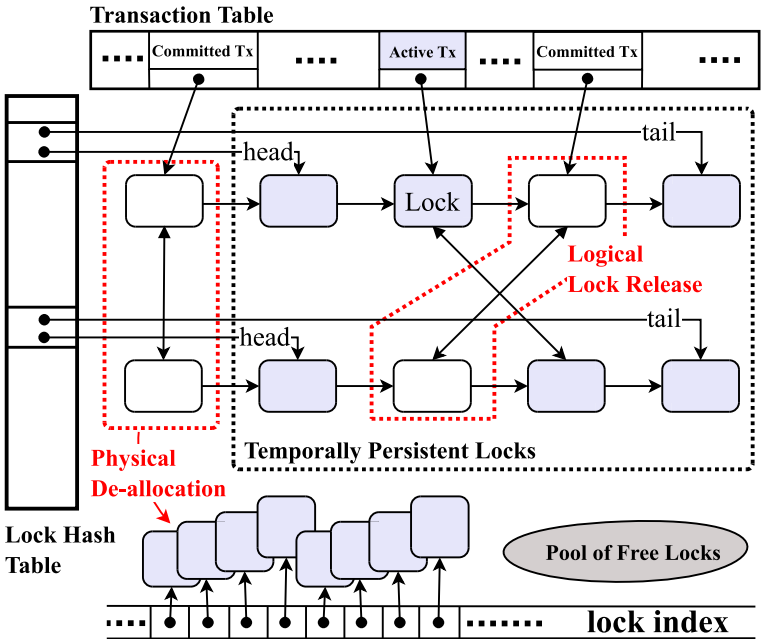



Fig. 11. High-level view of the lock manager data structures.

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 10.

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 10(a)). Flags are encoded as bytes (rather than bits) as the x86 architecture supports atomic updates of bytes but not of individual bits. Figure 10(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 5). 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 (i.e., Theorem 4.2) holds*. As indicated in Section 4.1, we do this by keeping, for each segment, track of the highest epoch number, $e_l$, whenever a lock becomes invisible by `next_pointer_update()` operating as described in Figure 8. Updating the global epoch is done only if the local

Table I. Dell PowerEdge R810 hardware specifications.

| 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 |

epoch of each transaction in the critical region is equal to the current global epoch, as described in Section 4.1. This synchronized behavior guarantees that none of the transactions concurrently running in the critical region have epoch values that differ by more than 1. Segment arrays whose $e_l$ is less than $e_g - 1$ therefore satisfy Theorem 4.2 and can be safely reclaimed by the garbage collector. This EBR-based reclamation therefore removes the limitation of the prior work [Jung et al. 2013] with respect to long-running transactions.

### 4.5. Lock Manager Architecture

The resulting lock manager architecture is shown in Figure 11. The traditional hash-based lock manager architecture [Gray and Reuter 1992], 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* [Boyd-Wickizer et al. 2010], 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 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-7.0 (the official EPFL branch[4]), and a commercial DBMS X.[5] 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 with EBR-based garbage collection; 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 I. The operating system is Linux

[4]The EPFL branch supports more recent techniques, such as *speculative lock inheritance* [Johnson et al. 2009a].
[5]The commercial system's license prohibits us from identifying it.

3.1.5, which incorporates the scalability improvement patch by Boyd-Wickizer et al. [2010].

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." This means that log flushes do not become a bottleneck, the same benefit as obtained in more sophisticated systems from group commit. 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 follow the lazy log-flushing policy. Shore-MT does offer the "group commit" function that enables multiple transactions to commit together with a single log flush, but we do not use it in any Shore-MT experiments. DBMS X is configured with similar parameters: the same size of buffer pool, log file, and log buffer, and using lazy flushing rather than its group commit policy.

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. We also enable *speculative lock inheritance* [Johnson et al. 2009a] for all Shore-MT experiments to see the behavior of the most recent techniques developed for improving scalability.

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. [2009b], which we retain to make our results comparable to theirs. This gives Shore-MT a potential performance advantage over the other systems, but we are not aiming to compare performance across unrelated platforms; instead the purpose of our evaluation of those other platforms is to examine scalability **when there is no logical contention, and especially to determine whether problems similar to those of MySQL exist**.

*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, and the total data size of three tables is 0.5 GiB.

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 $S = 100$. 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 $\leq 8$ 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. Scalability of Platforms

*5.2.1. Performance under read-only 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 **(such as MySQL's use of a single mutex for the whole lock table)**, but rather the collapse applies to a diversity of platforms. We also show that our design eliminates the collapse.

**These experiments are all done with a workload that is entirely read-only, so that there is no limitation arising from logical contention on the locks (which could not be eliminated by a different implementation of the lock manager). By using a read-only workload, we focus attention on the impact of the latching and implementation choices in designing the lock managers in-**

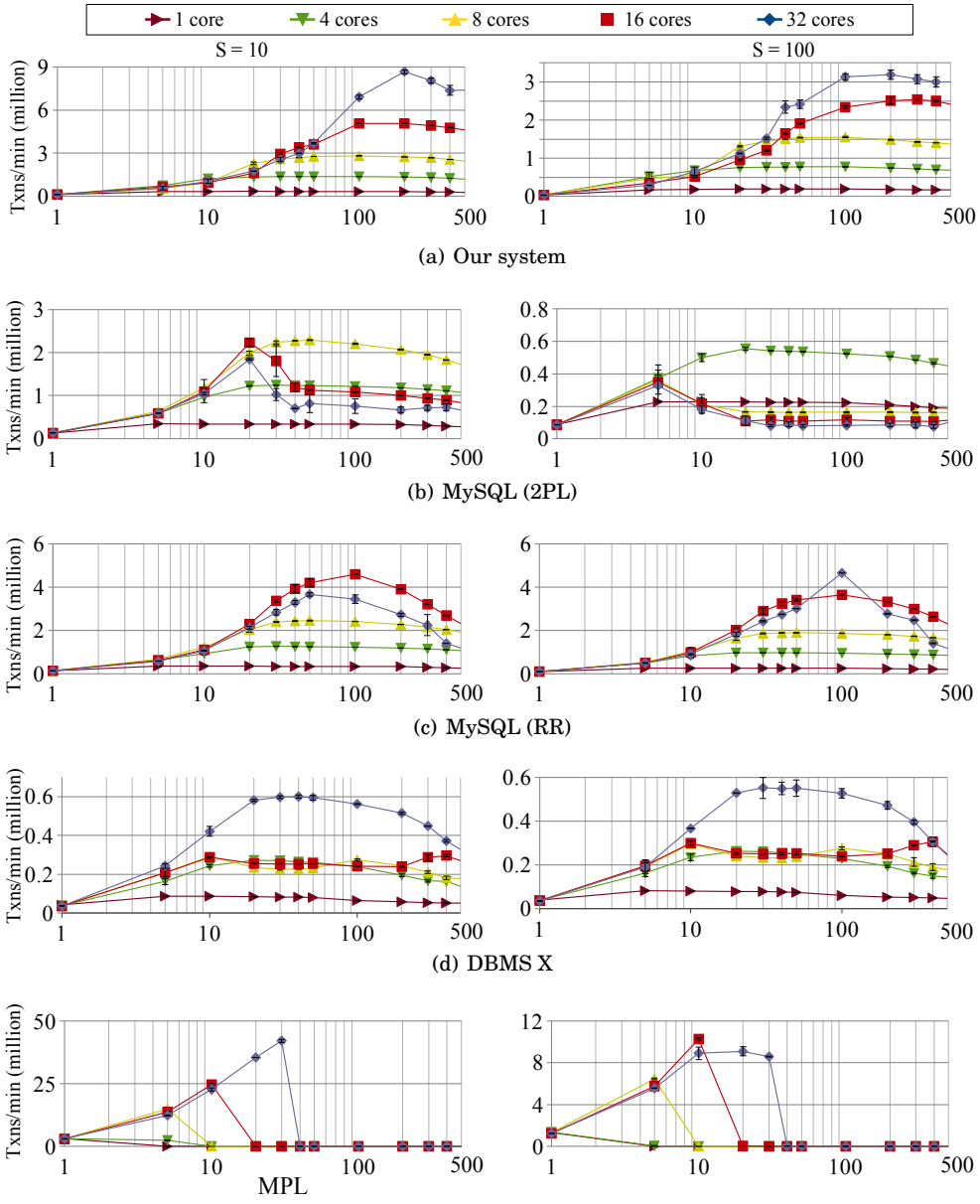



Fig. 12. Throughput as a function of load (MPL) for varying number of cores.

**ternals. In all these experiments, we scale up the MPL into the over-saturated situation, with more concurrent request threads than the number of cores. While some system designs may use a limited thread pool to reduce over-saturation and its problems, finding safe points to park a running connection (or session) and to resume the connection later is one of key mechanisms for the efficient implementation of thread pooling, and it is quite delicate to code. Thread pooling requires an efficient scheduling policy on those worker threads. In any event, our experiments show that the performance collapse at high client count and high core count remains an issue for a diverse set of real platforms.**

Figure 12 shows the complete results for read-only workload. The uppermost 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 **in the over-saturated situation**.

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, (althoughits peak is about an order of magnitude less than that 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 that the latter 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.

Shore-MT shows an impressive peak throughput ($\sim$42 million Txns/min) on 32 cores, with smaller (S=10) transactions, mainly due to the speculative lock inheritance technique. In terms of the peak value, the EPFL branch of Shore-MT achieves almost an order of magnitude higher throughput than that of the earlier system (i.e., Shore-MT-6.0) that we used in a preliminary version of our paper [Jung et al. 2013]. However, in all core configurations, the EPFL branch of Shore-MT shows worse throughput collapses than the previous version. We will investigate this collapse further using profiled data.

*5.2.2. Where does the time go?.* Profiling allows us to peek inside the systems, as shown in Figure 13. 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,

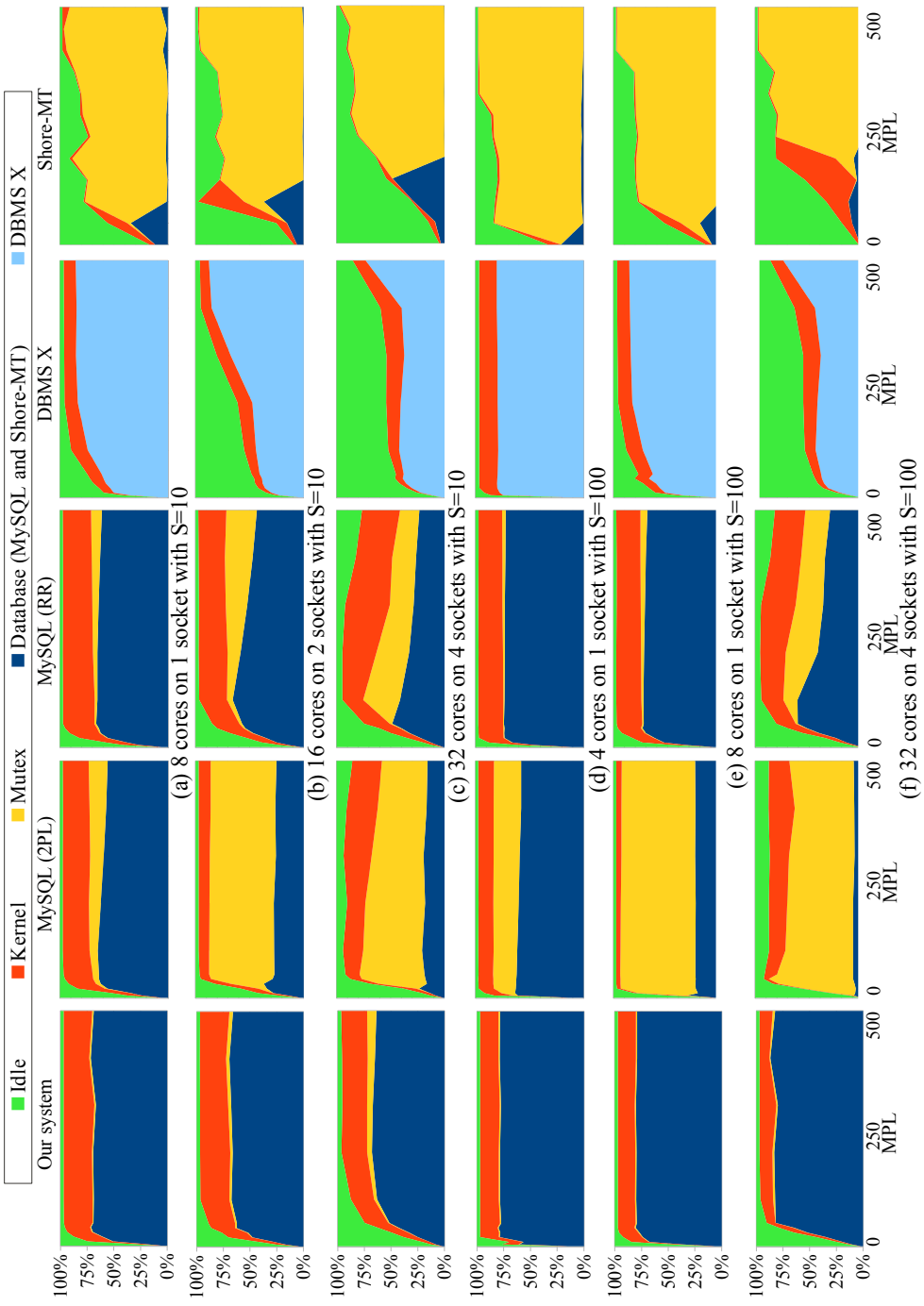
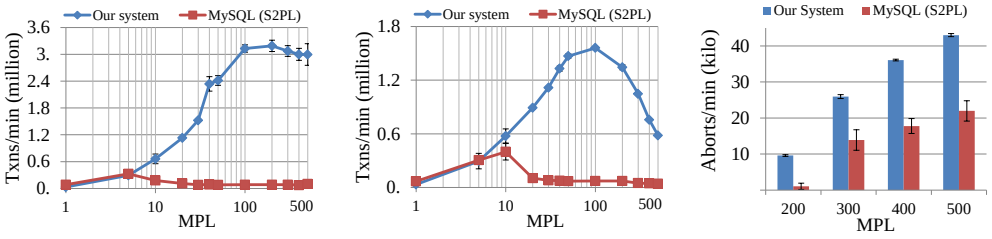



Fig. 13. Breakdown of profiled results of our system, MySQL (2PL), MySQL (RR), DBMS X, and Shore-MT (from left to right) with different core/socket configurations (from top to bottom) and 100% read-only workloads.




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

Fig. 14. Our system vs. MySQL(2PL) on 32 cores, S=100.

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 12 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 pre-defined (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.

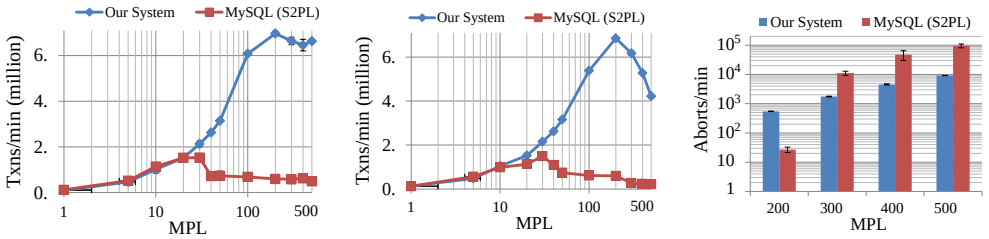
Also noticeable is that for small transactions, MySQL(RR) also shows significant mutex overhead (as well as idle time), despite the lower isolation level. The explanation is that in RR isolation, when a read view is to be opened, MySQL uses the transaction-system mutex to find a serialisable point. Consequently,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

**The EPFL branch of Shore-MT shows huge mutex time, with reduced idle time. This is mainly due to the default configuration that uses MCS locks [Mellor-Crummey and Scott 1991]. In our experiments, the `spin_on_waiting` routine for the MCS lock takes up-to 98% of running time. One characteristic of MCS locks we learnt from our experiments is a severe performance collapse when the number of competing threads outnumbers the core counts. We also observed that after MPL outnumbers core counts, Shore-MT spends most time in mutex contention. The performance collapses are still observable even if we configure Shore-MT to use the `pthread_mutex_lock/unlock()` routines as default latch functions. The main culprit in this case is the `join_xlist()` routine, not the `spin_on_waiting()` routine. In the `join_xlist()` routine, each transaction is added to the transaction list to get the transaction ID. To assign the transaction ID for a newly added transaction, Shore-MT increases the transaction ID of the predecessor by one. A newly added transaction spin-waits until the transaction ID of its predecessor is determined, and this spin-waiting time is lengthened quite severely as the number of competing threads outnumbers the core count by a wide margin.**

### 5.3. Performance Implications of Our Design

*5.3.1. 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,




(a) Throughput, 20% updates, uniform (b) Throughput, 20% updates, hotspot (c) Abort rate, 20% updates, hotspot

Fig. 15. Exploring hotspot behaviour, 32 cores, S=10.

which are the reason even read-only queries need to set locks at all. Figure 14 shows a direct comparison of our system with MySQL(2PL). As a baseline, Figure 14(a) repeats earlier throughput data for read-only workloads (32-core lines from $S = 100$ graphs, at the right of rows (a) and (b) of Figure 12). Figure 14(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 14(c) shows the abort numbers for the workload with 20% update transactions; these aborts increase dramatically at high loads, with up to 46% of MySQL transactions aborted at top load. Our system has somewhat higher number of aborts, but this must be considered against the much higher throughput: the fraction of transactions that abort at worst about 7.2%. This shows the benefit of concurrent deadlock checking. Our system shows performance degradation in high workload due to (inevitable) lock conflicts, not because of implementation-specific latch contention.

*5.3.2. Hotspot.* 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. For comparison, Figure 15(a) shows the performance with uniform data access in the $S = 10$ benchmark with 20% updates.

With low MPL, up to the peak, the presence of a hotspot makes no fundamental difference in our system nor with unmodified MySQL: throughputs for each are similar to what is found with uniform data access. With a hotspot in access patterns, our system reaches a peak throughput of 6.8 million tpm at MPL 200, compared to MySQL's peak of 1.5 million tpm at MPL 30. For MPL of 300 or greater, both systems show a lower throughput for the hotspot workload than with uniform access. At MPL 500, our system's throughput has dropped to 4.2 million tpm (that is, 63% of the workload with uniform access), while MySQL has only 0.2 million tpm which is 43% of the corresponding figure with uniform access. Similarly, the abort numbers for our system with a hotspot load are not very different from the abort numbers of MySQL; considered as a fraction of the committing transactions, our abort rate is remarkably low (below 0.25%), whereas MySQL has an abort rate up to 46%. Thus, there is no indication that the increased length of our hash chains is becoming a bottleneck even at high multiprogramming level.

*5.3.3. Long-running transaction.* Long-running transactions create substantial issues in databases, both for managing in-memory data structures as well as data kept on permanent storage. For example, under `REPEATABLE READ` isolation on a database with a multi-versioned storage engine (e.g., InnoDB), then the database must keep old versions of changed rows (or records) in permanent storage, as long as there are transactions accessing them. This is to ensure that a transaction always sees the data as it was when the transaction commenced. In our prior work [Jung et al. 2013], we reclaimed released locks based on Theorem 4.1, which could lead the system to run out of memory in the presence of long-running transactions.

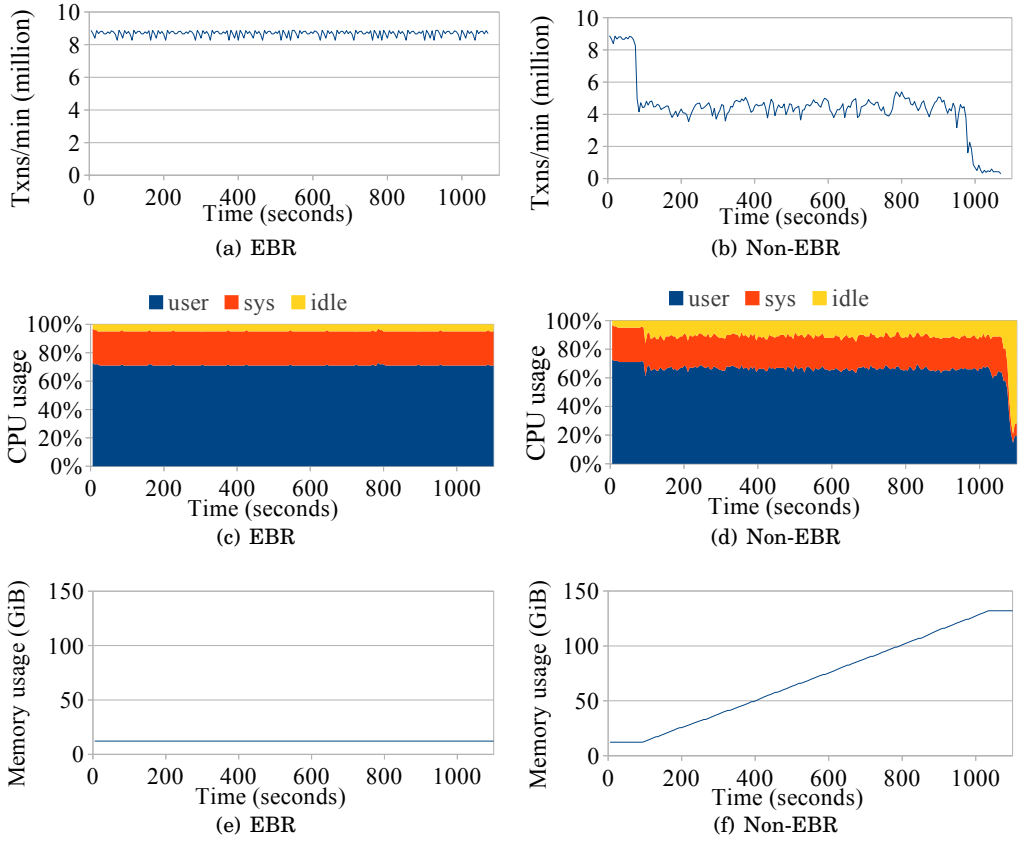



Fig. 16. Effect of a stalled transaction, on 32 cores, S=10, R/O workload at MPL 200.

To investigate the performance of the lock manager presented in Section 3 and Section 4, which has EBR-based garbage collection, we show the time-evolution of activity in a database system with a stalled transaction, showing throughput, CPU breakdown and memory usage. This is to show the limitation of our prior work (scalable lock manager without EBR, described in [Jung et al. 2013], and here labelled as Non-EBR) and the clear benefit of the current work (scalable lock manager with EBR). The workload consists of read-only transactions (S=10) at MPL 200, and a database system is running on 32 cores. Three metrics are measured every 5 seconds and plotted as a trajectory. We introduce a stalled transaction that begins at 100 seconds, long after initial

warm-up has completed, and it never ends. The oldest transaction ID in this experiment is fixed throughput the experiment, that leads the system without EBR-based garbage collection (from [Jung et al. 2013]) to *thrashing*. As shown in Figure 16(a), Figure 16(c) and Figure 16(e), our lock manager with EBR-based garbage collection does not show any noticeable changes in three metrics, even with a stalled transaction. However, Non-EBR shows throughput degradation right after a stalled transaction commenced (Figure 16(b)), and CPU time spent by the operating system is increased (Figure 16(d)), and the memory consumption for the database system gradually grows (Figure 16(f)). This behaviour of Non-EBR is due to the memory allocation overhead for new lock segment arrays and spurious page faults for the initial access to the lock objects. This pattern continues until the system consumes all available memory (i.e., ∼128 GiB) and faces *thrashing* around 1000 seconds. Then throughput collapses, and the idle time becomes dominant.

In summary, we see that the proposed EBR-based garbage collection can successfully support long-transaction workloads, without degrading throughput.

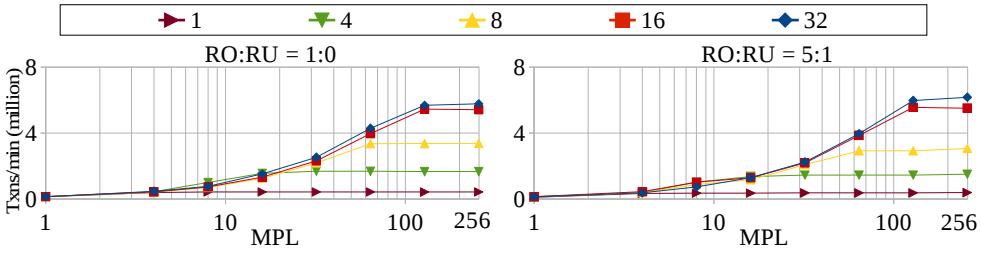
*5.3.4. More complex transaction code.* All the results we have seen so far are based on our micro-benchmark. Now we show core-scalability results using more complex transaction code. The workload we run is a subset of the transactions from the Telecommunication Application Transaction Processing (TATP) benchmark, which is an open source workload designed for high-throughput applications (e.g., in-memory database systems). **We first perform the experiments with the read-only transactions (i.e.,** `Get_Subscriber_Data`**,** `Get_Access_Data` **and** `Get_New_Destination`**); then we explore core-scalability under write workloads, by using update transactions (i.e.,** `Update_Subscriber_Data` **and** `Update_Location`**) as well as read-only transactions. We measure the performance of all systems used in Section 5.2 under varying load (from 1 to 256 MPL). We set the scale factor to 64 when we generate TATP data.**

As shown in Figure 17, original MySQL sysems (2PL and RR) show throughput collapses on 16 and 32 cores due to the latch contention around the lock table mutex. Our system does not show any collapse, but the peak throughput is the same both on 16 and 32 cores. The profiled data does not show any mutex overhead, except that idle time is increased on 32 cores. Further investigation on this behavior tells us that most of idle time is due to the read-write locks (i.e., `rw_s_lock`) on B-tree pages. **The profiled data shown in Figure 18 can explain performance collapses observed in MySQL (2PL), MySQL (RR), and Shore-MT. Our system does not show noticeable mutex time.**

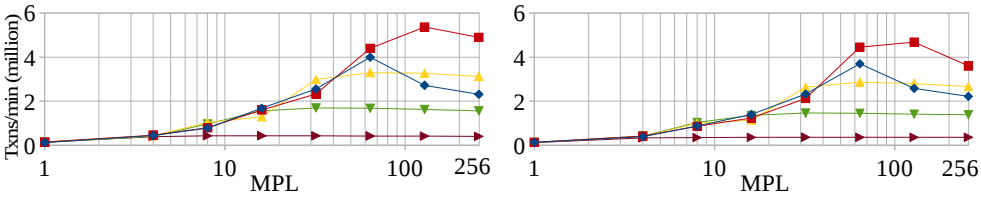
**We configure Shore-MT to use the MCS latch (default). Shore-MT shows a peak throughput (∼32 million Txns/min) on 32 cores with read-only transactions. When update transactions are mixed, the peak throughput is 5 millon Txns/min. Shore-MT does not show good scalability, mainly due to spin-waiting not inside the** `spin_on_waiting()` **routine, but in the the** `join_xlist()` **routine for the transaction initialization. This leads us to get a different profiled breakdown for Shore-MT compared to what we have shown in Figure 13.**

## 6. CONCLUSION

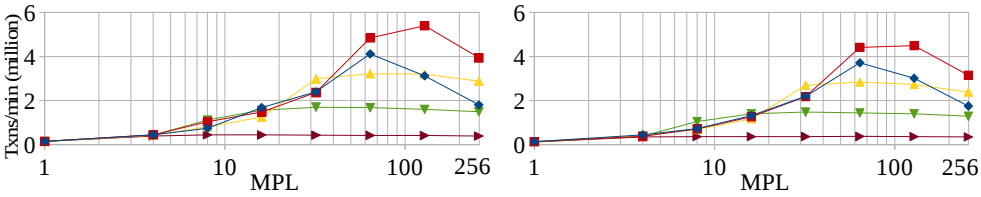
We have found that some contemporary database systems are not yet ready for the multicore age. Our evaluation has demonstrated a collapse of transaction throughput under high load, even read-only load, for the database systems we analysed. In the case of the open-source systems, MySQL and Shore-MT, we could identify latch contention in the lock manager as the bottleneck. While we could not perform the same in-depth



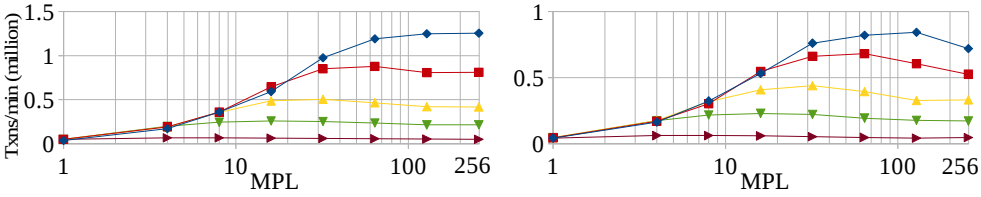

(a) Our system



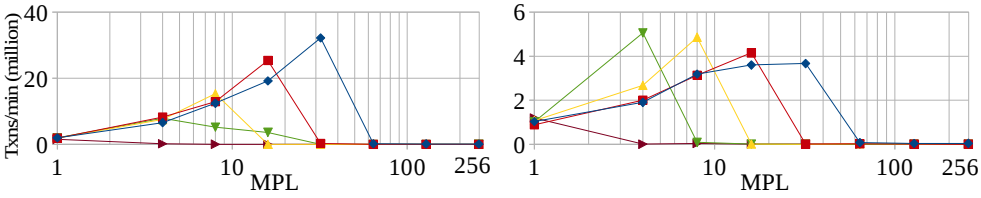

(b) MySQL (2PL)




(c) MySQL (RR)




(d) DBMS X




(e) Shore-MT

Fig. 17. TATP results

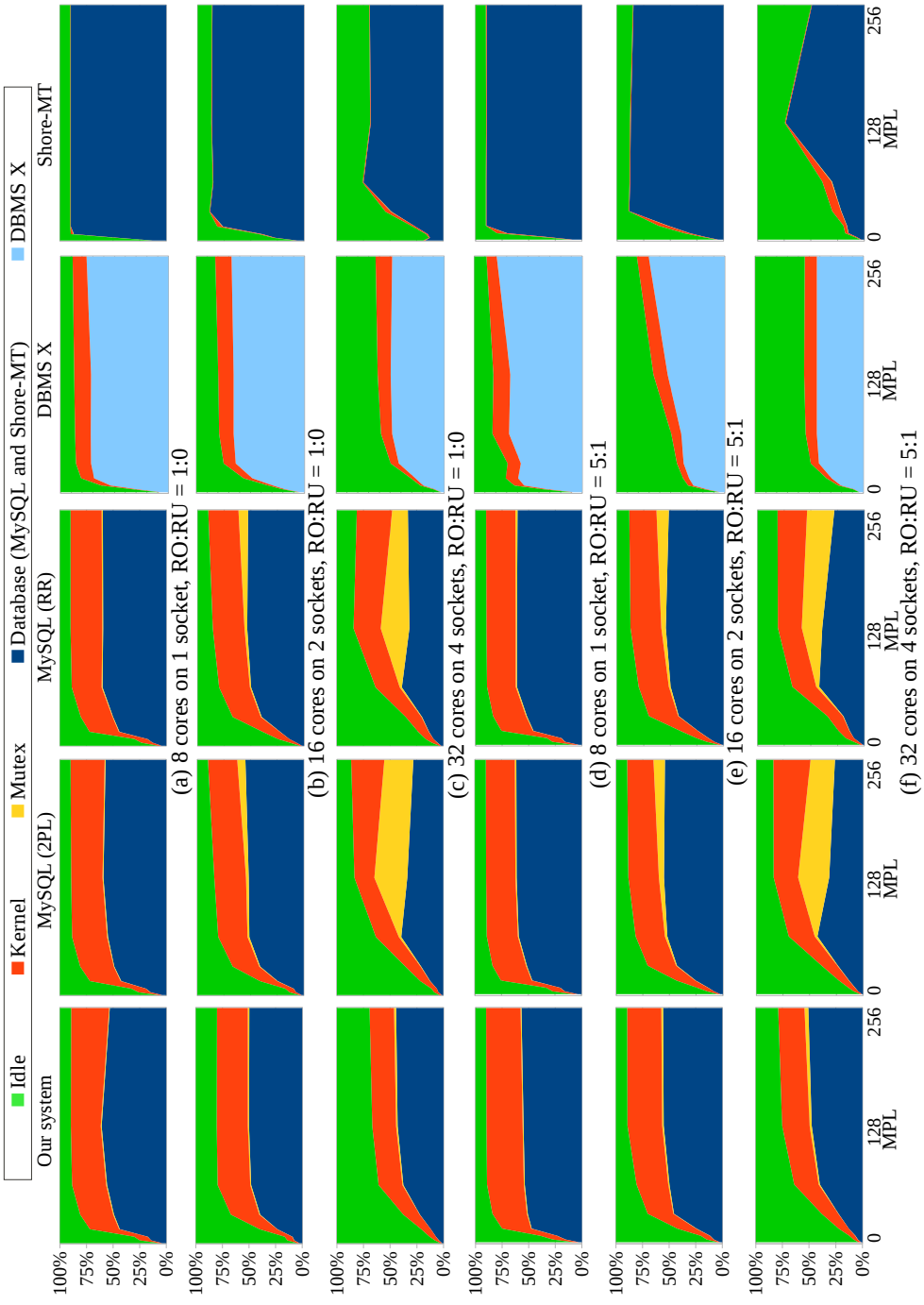



Fig. 18. Breakdown of profiled results of our system, MySQL (2PL), MySQL (RR), DBMS X, and Shore-MT (from left to right) with different core/socket configurations (from top to bottom).

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, 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) is not detrimental to the performance with fewer cores or lighter load.

Our proposal is modular, and can be introduced in a database management system with a conventional latch-based design for the lock manager, without affecting the index structures, query processing code, buffer management, or other parts of the system. For example, the early version of our design (originally published in [Jung et al. 2013]) was then implemented by researchers at HP, in the context of a system which is based on an older version of Shore-MT (incorporating some but not all of the improvements found in Shore-MT-7.0) and it uses the Foster B-tree index structure [Graefe et al. 2012]. On a 4-socket Intel machine, with RAMDisk storage, running TPC-C with 10 warehouses at MPL=12, their system had been bottlenecked on the lock manager, peaking at 15K TPS; on replacing the lock manager by one that was based on our design, these researchers moved the bottleneck to the log manager, and gained close to 40% in throughput, reaching 21K TPS [Kimura 2014].

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

## ACKNOWLEDGMENTS

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. Authors would like to thank SeongJae Park for helping in many places.

## REFERENCES

Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. 2011. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 487–498. DOI: http://dx.doi.org/10.1145/1926385.1926442

Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. 1979. The Convoy Phenomenon. *SIGOPS Oper. Syst. Rev.* 13, 2 (April 1979), 20–25. DOI: http://dx.doi.org/10.1145/850657.850659

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An analysis of Linux scalability to many cores. In *Proceedings of the 9th USENIX Syposium on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 1–8. http://dl.acm.org/citation.cfm?id=1924943.1924944

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2012. Non-scalable Locks are Dangerous. In *Proceedings of the Linux Symposium*. Ottawa, Canada, 121–132.

Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable address spaces using RCU balanced trees. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 199–210. DOI: http://dx.doi.org/10.1145/2150976.2150998

Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge Computer Laboratory.

Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM Transactions on Database Systems* 35, 3 (July 2010), 16:1–16:26. DOI:http://dx.doi.org/10.1145/1806907.1806908

Goetz Graefe, Hideaki Kimura, and Harumi Kuno. 2012. Foster B-trees. *ACM Transactions on Database Systems* 37, 3 (Sept. 2012), 17:1–17:29. DOI:http://dx.doi.org/10.1145/2338626.2338630

Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Rajiv Gupta. 1989. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*. ACM, New York, NY, USA, 54–63. DOI:http://dx.doi.org/10.1145/70082.68187

Hyuck Han, SeongJae Park, Hyungsoo Jung, Alan Fekete, Uwe Röhm, and Heon Y. Yeom. 2014. Scalable serializable snapshot isolation for multicore systems. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. 700–711. DOI:http://dx.doi.org/10.1109/ICDE.2014.6816693

Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel and Distrib. Comput.* 67, 12 (Dec. 2007), 1270–1285. DOI:http://dx.doi.org/10.1016/j.jpdc.2007.04.010

Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. *Architecture of a database system*. Now Publishers Inc., Hanover, MA, USA.

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Takashi Horikawa. 2013. Latch-free data structures for DBMS: design, implementation, and evaluation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 409–420. DOI:http://dx.doi.org/10.1145/2463676.2463720

Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. 2009a. Improving OLTP Scalability Using Speculative Lock Inheritance. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 479–489. http://dl.acm.org/citation.cfm?id=1687627.1687682

Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009b. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*. ACM, New York, NY, USA, 24–35. DOI:http://dx.doi.org/10.1145/1516360.1516365

Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2012. Scalability of write-ahead logging on multicore and multisocket hardware. *The VLDB Journal* 21, 2 (April 2012), 239–263. DOI:http://dx.doi.org/10.1007/s00778-011-0260-8

Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. 2010. Decoupling Contention Management from Scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 117–128. DOI:http://dx.doi.org/10.1145/1736020.1736035

Hyungsoo Jung, Hyuck Han, Alan D. Fekete, Gernot Heiser, and Heon Y. Yeom. 2013. A scalable lock manager for multicores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, New York, NY, USA, 73–84. DOI:http://dx.doi.org/10.1145/2463676.2465271

Hideaki Kimura. 2014. Private communication. (2014). Foster B-tree project at HP Labs.

Hideaki Kimura, Goetz Graefe, and Harumi A. Kuno. 2012. Efficient Locking Techniques for Databases on Modern Hardware. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2012, Istanbul, Turkey, August 27, 2012.* 1–12.

John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* 9, 1 (Feb. 1991), 21–65. DOI:http://dx.doi.org/10.1145/103727.103729

Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented transaction execution. *Proceedings of the VLDB Endowment* 3, 1-2 (Sept. 2010), 928–939. http://dl.acm.org/citation.cfm?id=1920841.1920959

Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. 2011. PLP: page latch-free shared-everything OLTP. *Proceedings of the VLDB Endowment* 4, 10 (July 2011), 610–621. http://dl.acm.org/citation.cfm?id=2021017.2021019

Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM EuroSys Conference (EuroSys '12)*. ACM, New York, NY, USA, 337–350. DOI:http://dx.doi.org/10.1145/2168836.2168870

Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2012. Lightweight Locking for Main Memory Database Systems. *Proc. VLDB Endow.* 6, 2 (Dec. 2012), 145–156. DOI:http://dx.doi.org/10.14778/2535568.2448947

Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. 2011. Database engines on multicores, why parallelize when you can distribute?. In *Proceedings of the 6th EuroSys Conference (EuroSys '11)*. ACM, New York, NY, USA, 17–30. DOI:http://dx.doi.org/10.1145/1966445.1966448

Curt Schimmel. 1994. *UNIX systems for modern architectures: symmetric multiprocessing and caching for kernel programmers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Jason Sewall, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2011. PALM: Parallel Architecture-Friendly Latch-Free Modification to B+ Trees on Many-Core Processors. *Proceedings of the VLDB Endowment* 4, 10 (July 2011), 795–806.

Gerhard Weikum and Gottfried Vossen. 2001. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Received October 2013; revised Month 20XX; accepted Month 20XX