

# Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores

Tianzheng Wang  
University of Toronto  
tzwang@cs.toronto.edu

Hideaki Kimura  
Hewlett Packard Labs  
hideaki.kimura@hpe.com

## ABSTRACT

Future servers will be equipped with thousands of CPU cores and deep memory hierarchies. Traditional concurrency control (CC) schemes—both optimistic and pessimistic—slow down orders of magnitude in such environments for highly contended workloads. Optimistic CC (OCC) scales the best for workloads with few conflicts, but suffers from clobbered reads for high conflict workloads. Although pessimistic locking can protect reads, it floods cache-coherence backbones in deep memory hierarchies and can also cause numerous deadlock aborts.

This paper proposes a new CC scheme, *mostly-optimistic concurrency control* (MOCC), to address these problems. MOCC achieves orders of magnitude higher performance for dynamic workloads on modern servers. The key objective of MOCC is to avoid clobbered reads for high conflict workloads, without any centralized mechanisms or heavyweight interthread communication. To satisfy such needs, we devise a native, cancellable reader-writer spinlock and a serializable protocol that can acquire, release and re-acquire locks in any order without expensive interthread communication. For low conflict workloads, MOCC maintains OCC’s high performance without taking read locks.

Our experiments with high conflict YCSB workloads on a 288-core server reveal that MOCC performs  $8\times$  and  $23\times$  faster than OCC and pessimistic locking, respectively. It achieves 17 million TPS for TPC-C and more than 110 million TPS for YCSB without conflicts,  $170\times$  faster than pessimistic methods.

## 1. INTRODUCTION

Core counts in modern database servers range into the hundreds today [10, 38] and are expected to grow into the range of thousands soon [11]. Despite such advances, emerging software systems are struggling to utilize such modern hardware to generate, analyze, and store an exponentially growing amount of data. For example, GraphLab [27] parallelizes a graph query by internally splitting up the work into billions of ACID transactions that often contain read-write conflicts between cores. Other examples of growing concurrency demands include security monitoring, high frequency trading, and smart grid management.

Existing databases, however, cannot scale up to thousands of cores, especially when the workload is highly contended. Most databases today employ either pessimistic *two phase locking* (2PL) or *optimistic concurrency control* (OCC) to guarantee serializability. As we shall see, critical scalability bottlenecks slow down both 2PL and OCC by orders of magnitude in modern servers.

OCC has low overhead and scales up nicely on manycore servers for low-conflict/contention workloads. When the workload contains frequent read-write conflicts, however, OCC can exhibit high abort rates that strongly limit throughput [8, 16]. 2PL exhibits two scalability issues. First, pessimistic read locks severely limit scalability due to expensive cache-coherence traffic on deep memory hierarchies. Second, somewhat ironically, 2PL does *not* necessarily reduce aborts. Especially when record accesses are highly contended and in a random order, even 2PL causes massive aborts due to deadlocks. Moreover, existing 2PL architectures necessitate unscalable interthread communication.

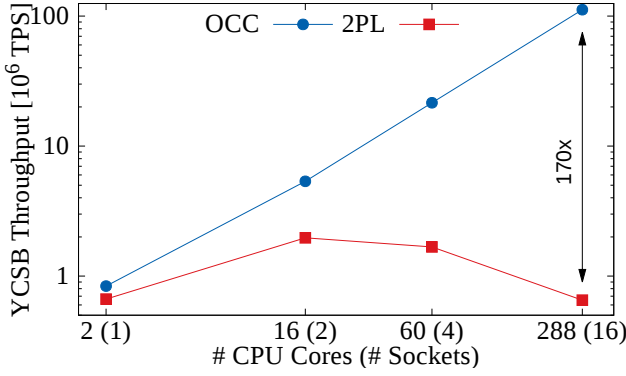
In this paper, we propose *mostly-optimistic concurrency control* (MOCC), a new concurrency control scheme that addresses these problems and scales up to thousands of cores. MOCC dramatically reduces the abort ratio and guarantees robust progress even when the workload has extremely frequent read-write conflicts. Yet, MOCC has extremely low overhead, performing orders of magnitude faster than 2PL and as fast as the state-of-the-art OCC even in its best case, low conflict workloads. Finally, MOCC dynamically and autonomously optimizes itself to retain its high performance in the most challenging yet common case, in which each transaction involves multiple data sets of opposite nature: records that have frequent read-write conflicts and records that are read-mostly.

MOCC is based on modern OCC that is highly decentralized. There are two key reasons why we design MOCC based on OCC: First, its decentralized architecture scales far better on modern servers when the workload does not have too frequent conflicts. It is adopted by state-of-the-art main-memory optimized systems, such as FOEDUS [17] and Silo [41]. MOCC behaves the same as FOEDUS for such workloads and retains its performance. Second, we found that OCC’s commit protocol fits well with the non-2PL protocol proposed in this paper, which must acquire and release locks in arbitrary orders without violating serializability.

Like 2PL, MOCC incorporates pessimistic read locks to prevent writers from clobbering readers. Unlike 2PL, however, an MOCC transaction might release locks before commit, then re-acquire them in a different order during pre-commit. Doing so allows MOCC to effectively avoid aborts and deadlocks, without relying on any unscalable component such as a centralized lock manager. Section 3 describes the MOCC protocol in detail, including how MOCC dynamically and selectively adds pessimistic locking onto OCC and how MOCC avoids deadlocks.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org).

*Proceedings of the VLDB Endowment*, Vol. 10, No. 2  
Copyright 2016 VLDB Endowment 2150-8097/16/10.



**Figure 1: Read-only, highly contended YCSB. OCC scales better than 2PL on a deep memory hierarchy, which augments every single scalability issue to an unprecedented degree.**

Another key technique to enable MOCC is the MOCC Queuing Lock (MQL), a reader-writer queue lock that allows parallel, asynchronous locking and cancelling. Compared to the lock algorithm in traditional databases, MQL scales far better on deep memory hierarchies and also supports modes that MOCC exploits: reader/writer and unconditional/try/asynchronous locking. Section 4 describes MQL and how MOCC exploits it.

The experiments in Section 5 show that MOCC achieves orders of magnitude higher performance than existing approaches on a modern 288-core machine serving high contention workloads.

## 2. KEY CHALLENGES AND PRINCIPLES

Scalability tradeoffs surrounding both optimistic and pessimistic concurrency control have been studied extensively, as reviewed in Section 6. This section highlights the key challenges that we face and the principles that differentiate our work from prior research.

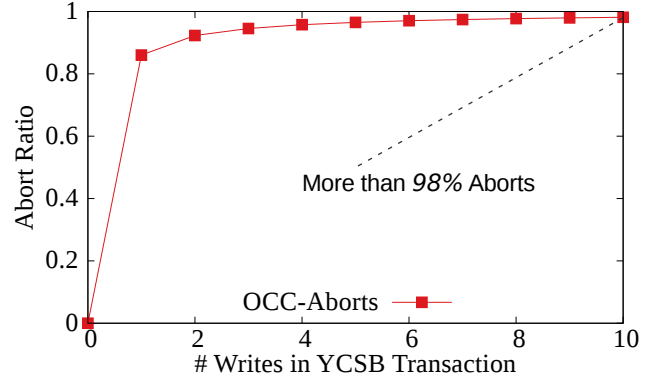
### 2.1 Not Your Father’s Hardware!

Compared with today’s hardware, emerging servers will contain far more CPU cores and a deeper, more complex memory hierarchy to interconnect them. Figures 1 and 2 illustrate the shocking consequences for performance.

This experiment uses FOEDUS [17], a manycore optimized database that uses modern decentralized OCC. We added a new 2PL variant to it and ran highly contended (50 records) YCSB benchmarks. We tested on various machines, ranging from a single-socket dual-core laptop to a modern server with 16 sockets and 288 cores. We present the details of this experiment in Section 5 and summarize the main observations here.

**Extremely High Contention:** Figure 1 shows a read-only workload where 2PL performs more slowly than OCC due to its overhead to take read locks, which is expected. Nonetheless, it is surprising to observe that the read lock overhead is not merely a nuisance, but an *abyss* [46], slowing down the seemingly easy read-only workload by **170×** in a modern server. The slowdown quickly grows with the number of cores and the depth of the memory hierarchy due to *physical lock contention* even though all of the locks are logically compatible. Prior literature has studied physical lock contention [12, 18], but none has observed this much effect (e.g., Johnson et. al [12] report up to 15% slowdown on 64 threads) or designed a system for such a high level of contention.

We also observed several other scalability issues in a far larger degree than prior studies, such as the ever growing cost of remote



**Figure 2: Read-write, highly contended YCSB. OCC is vulnerable to aborts under high conflict. In an extreme case, it makes almost no progress.**

NUMA access and atomic operations. To the best of our knowledge, the only empirical performance studies at similar scales, except simulation-based [46], were found outside of the context of databases, such as those in the locking community [7, 42].

At this scale, we found that strict 2PL is not a viable choice because it causes severe cacheline ping-pong in the already moribund cache-coherence backbone.

**Extremely High Conflict:** OCC scales well under low conflict, but it suffers a different drawback: high abort ratio under high conflict. Figure 2 varies the number of writes from zero (read-only) to 10 (read-modify-write only). The extremely high concurrency in hardware brings extremely high read-write conflicts that severely degrade performance. With just one write per transaction, more than 80% of transactions abort in OCC’s commit phase due to clobbered reads. With 10 writes, as many as 98% of transactions abort. Many transactional applications have some *hot spot* that receives high conflict accesses. Such a high abort ratio can completely stifle the performance of OCC. Furthermore, somewhat ironically, even pessimistic protocols have a high abort ratio due to deadlocks.

### 2.2 Issues in Existing Databases

Emerging server hardware demands a departure from previous database architectures that contain the following bottlenecks.

**Page Latches:** Most existing databases use *page latches*, or a lock to protect a physical page rather than logical records. Even some databases that use OCC for individual records take read latches for pages. Furthermore, read latches must be *coupled* to guarantee consistent page traversals [28]. Although previous work at smaller scales did not find physical contention on page latches as a central bottleneck, it severely stifles performance on thousands of cores.

**Reads become Writes:** Some prior OCC-based systems (e.g., the BwTree [23] in Hekaton [6]) maintain a readers count to deter writers from clobbering reads. Although such schemes ameliorates aborts in OCC, a read operation must *write* to a contended location, limiting scalability and performance.

**Expensive Interthread Communication:** Traditional architectures, whether OCC, 2PL, or multi-versioned, require interthread communication that limits scalability in modern servers. One example is the tracking of anti-dependencies to guarantee serializability on top of snapshot isolation, where reads should leave notes in shared locations for other threads to detect possible serializability violations [2]. Another instance is interthread communication to detect and resolve deadlocks, which we discuss next.

**Frequent Deadlocks and Aborts:** Aborts are the main scalability issue in OCC for highly contended workloads. However, even 2PL significantly slows down for these workloads due to aborts caused by deadlocks [34]. Using strict discipline, it is possible for a developer to design and implement a simple application where *all* record accesses strictly follow the same order to prevent deadlocks. However, this is impractical for database applications because of the complexity and hidden object accesses, such as secondary indexes and foreign key constraints. Hence, the high concurrency on modern servers often results in a large number of deadlocks and aborts. To make things even worse, due to the repetitive nature of transactional workloads, deadlocks will keep occurring in the same contended place, continually aggravating the issues described above.

## 2.3 Key Principles of MOCC

The above observations constitute our key principles:

- MOCC must be based on an architecture without page latches for reads, like pure OCC does.
- To scale better, the only mechanism that kicks in for the majority of reads must be OCC without any writes to contended memory locations.
- On top of OCC, MOCC must selectively acquire read locks on records that would cause an abort without locks.
- MOCC must avoid deadlocks without any unscalable interthread coordination.

## 3. MOSTLY-OPTIMISTIC CC

Figure 3 gives a high level overview of MOCC. MOCC is based on a modern decentralized OCC without page latches for reads. Section 3.1 recaps the relevant aspects of OCC.

MOCC employs pessimistic read locks to overcome the drawback of OCC, aborts during read verification. However, if MOCC issues read locks for all reads, its scalability will degenerate to that of 2PL. MOCC thus maintains *temperatures* to selectively issue read locks only to reads that will likely cause read-write conflicts (Section 3.2).

Although it sounds trivial to acquire read locks, doing so naively could revive all the bottlenecks and complexities that OCC avoided to achieve high scalability. The crux of implementing MOCC lies in the locking and commit protocol to avoid deadlocks caused by the additional locks taken *before* the commit phase. Section 3.3 covers the details of the MOCC protocol.

### 3.1 Recap: Decentralized OCC

MOCC inherits several ideas from decentralized OCC-based systems, most notably the following.

**No Page Latching for Reads:** Same as FOEDUS [17], MOCC avoids acquiring page latches for reads. For example, FOEDUS employs a variant of Masstree [29] that is not only cache-friendly, but also has a performance advantage on manycore servers because it does not require page latches for read operations. Masstree employs RCU (read-copy-update) to create a new version of a page and atomically switches a pointer to it. Hence, a read operation does not need any page latch to safely access a page. In MOCC, other storage representations (e.g., lock-free hash tables) do not require page latches for reads, either.

**Apply-after-Commit:** One key principle in modern OCC is to decentralize its transactional processing as much as possible for higher scalability. Each worker thread maintains its own log buffer and merely appends its uncommitted write operations to the log buffer. The thread does not apply the writes to data pages until the transaction is guaranteed to commit. The log buffers are completely thread private, hence threads synchronize with each other only when there is a potential conflict in their data accesses or an epoch-based

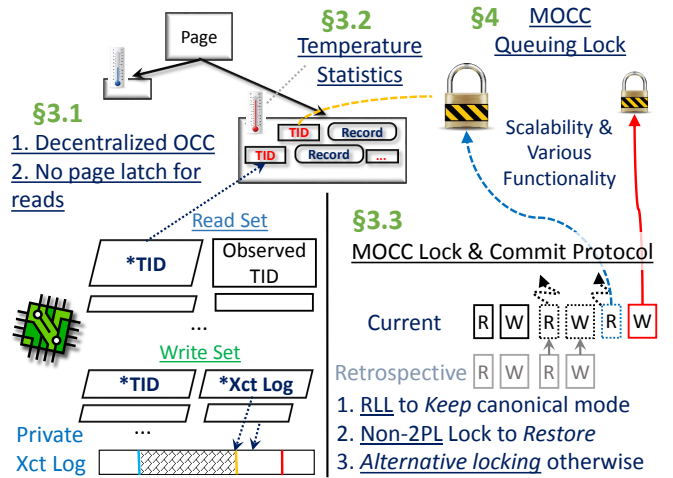


Figure 3: MOCC overview and paper outline.

group commit happens. Another important aspect of this design is the *redo-only* protocol that has no uncommitted data in pages even when a transaction aborts. This enables the non-2PL yet serializable lock/unlock protocol MOCC employs.

**Deadlock-free Verification Protocol:** All OCC variants, such as Silo and FOEDUS, verify reads at transaction commit time to guarantee serializability. Each transaction remembers the state of records as of reading and compares it with the current state at the commit procedure. If any of the reads observes a different state, the transaction aborts and either retries or returns an error to the client.

An OCC transaction finalizes the order it is serialized by locking all records it will modify (its write set) before it verifies the reads and determines the serialization order from its read/write set.

Most importantly, this protocol is completely deadlock-free. The locking happens during the commit phase when the transaction already knows all of its read/write set. As a result, we can lock the write set in any globally consistent order (e.g., by tuple virtual addresses), thus guaranteeing deadlock freedom and serializability without any interthread communication or relying on a centralized lock manager. Assuming this property, we can unconditionally take each lock, meaning the transaction indefinitely waits for lock acquisition.

**Missing Read Locks:** The above property perfectly satisfies our goal *if* we do not take any read locks. However, a transaction must take read locks to prevent read-clobber aborts, and it must take them *as of* the read operations to be protected, at which point the full footprint is unknown. This breaks the deadlock-free property, and the existing solution is to revive heavyweight, unscalable interthread communication, falling back to the scalability of 2PL. This is exactly why adding read locks to OCC, albeit apparently straightforward, is the crux of implementing MOCC.

### 3.2 Temperature Statistics

MOCC maintains *temperature statistics*, illustrated at the top of Figure 3, to predict how likely a read on a record will be clobbered by concurrent transactions. MOCC takes read locks on records whose temperature is above some threshold. Such records are likely to be updated by concurrent transactions, causing aborts. The temperature statistics track the number of aborts due to verification failures. To reduce space overhead, we maintain the statistics at page level, instead of in each record. Verification failures will increase the temperature of affected pages containing the clobbered records.

Since all the records belonging to the same page share the same temperature statistic, frequent writes to it might cause physical contention, defeating our purpose. To solve this problem, we maintain page temperature (i.e., number of aborts) using a variant of approximate counters [33]. Each page  $p$  maintains a temperature  $temp_p$ , which probabilistically implies that there were about  $2^{temp_p}$  aborts in the page. Whenever a transaction is aborted due to a verification failure on a record in  $p$ , it checks the current value of  $temp_p$  and increments it with probability  $2^{-temp_p}$ . Upon observing an abort, a page that has had frequent aborts (hot page) is likely to have a higher value in  $temp_p$ , significantly lowering the probability of cacheline invalidation due to frequent writes to a centralized location.

### 3.3 MOCC Protocols

Guided by the temperature statistics, MOCC takes pessimistic locks on hot records as described in Algorithm 1. Lines 9–10 and 15–16 show how we check the temperature statistics and trigger pessimistic locking. The OCC protocol is no longer deadlock-free as soon as we take locks *during* transaction execution in addition to taking locks at commit time. To avoid deadlocks and repeated aborts due to pessimistic locking, MOCC follows a non-2PL approach, described next.

#### 3.3.1 Canonical Mode

Central to the entire protocol is the concept of canonical lock acquisition. The idea is a generalization of the consistent sorting of write sets in Silo and FOEDUS, such as the virtual address order and lock ID order. Let  $l_m < l_n$  mean that the lock  $l_m$  is ordered before the lock  $l_n$  in some universally consistent order.

Let  $CLL$  be the list of locks the transaction has taken so far. Suppose the transaction now tries to acquire a set of new locks  $NL$ . The transaction is said to be in canonical mode if and only if  $l_c < l_n : \forall l_n \in NL, l_c \in CLL$ .

When a transaction is in canonical mode, there is no risk of deadlock. Thus, the transaction can unconditionally take the locks just like FOEDUS or Silo does. Unconditionally taking a lock is the most desirable way of acquiring locks especially in queue-based locks as Section 4 describes later.

When MOCC does not take any read locks, it is always in canonical mode because the only locking happens in commit phase with an empty  $CLL$  and  $NL$  being the write set. Read locks, however, can cause the transaction to leave canonical mode. In short, the MOCC protocol is designed to: (1) *keep* transactions in canonical mode as much as possible, (2) *restore* canonical mode when not in canonical mode, and (3) *try* taking locks as efficiently as possible without risking deadlocks when canonical mode is not attainable.

#### 3.3.2 Acquiring and Releasing Locks

In traditional 2PL architectures, there is not much one can do to reach the above goals because locks must be held in *two phases*; the growing phase takes locks and the shrinking phase releases locks. If a transaction releases a lock before commit and then takes another lock, the execution could be non-serializable.

However, MOCC is based on OCC, which gives far more flexibility in this regard. MOCC verifies reads at commit, hence serializability is guaranteed no matter whether it holds a read lock or not. MOCC, like the original OCC protocol, determines the serialization order and verifies/applies the read/write set only after it takes all write locks. Hence, until MOCC finalizes the transaction in commit phase, **MOCC can safely acquire, release, or re-acquire arbitrary locks in an arbitrary order**. MOCC fully exploits this flexibility, which is one reason why MOCC is based on OCC.

---

#### Algorithm 1 MOCC Protocols.

---

```

1 class MoccTransaction:
2     const H # Temperature Threshold
3     R := {} # Read Set
4     W := {} # Write Set
5     RLL := {} # Retrospective Lock List
6     CLL := {} # Current Lock List
7
8     def read(t: Record):
9         if temp(t) >= H or t in RLL:
10             lock(t, max(preferred mode in RLL, R-mode))
11         R.add(t, t.TID)
12         Read t
13
14     def read_write(t: Record):
15         if temp(t) >= H or t in RLL:
16             lock(t, W-mode)
17         R.add(t, t.TID)
18         Read t
19         Construct log in private buffer
20         W.add(t, log)
21     # Blind-write, same as in OCC
22
23     def lock(t: Record, m: Mode):
24         if CLL already has t in mode m or stronger:
25             return
26         violations := {l ∈ CLL, l.mode ≠ null, l ≥ t}
27         if too many violations:
28             alternative_lock(t, m) # See Section 4
29             return or abort
30         elif violations not empty:
31             # Not in canonical mode. Restore.
32             CLL.unlock({violations})
33
34         # Unconditional lock in canonical mode.
35         CLL.unconditional_lock({l ∈ RLL, l < t})
36         CLL.unconditional_lock(t, m)
37
38     def construct_rll(): # Invoked on abort
39         RLL := {}
40         for w in W:
41             RLL.add(w, W-mode)
42         for r in R:
43             if r not in RLL:
44                 if temp(r) >= H or r failed verification:
45                     RLL.add(r, R-mode)
46             RLL.sort()
47
48     def commit():
49         W.sort()
50         for w in W:
51             lock(w, W-mode)
52         for r in R:
53             if r.observed_tid not equal r.tid:
54                 temp(r).hotter() # See Section 3.2
55             abort
56         # Committed
57         Determine TID and apply/publish W # Silo/FOEDUS protocol
58         CLL.unlock_all()
59         RLL, CLL, R, W := {}
60
61     def on_abort():
62         CLL.unlock_all()
63         if user will retry the transaction:
64             construct_rll()
65         else
66             RLL := {}
67             CLL, R, W := {}

```

---

For example, in line 26 of Algorithm 1, suppose the transaction has a current lock list  $CLL : \{l_1, l_2, l_4\}$  and intends to take a read lock  $t = l_3$ . Since the transaction is already holding  $l_4$ , taking a lock ordered before it will leave the transaction in non-canonical mode. MOCC can restore canonical mode by releasing  $l_4$  first (lines 30–32), then unconditionally take  $l_3$  (lines 35–36). We do not re-take the released locks unless it is explicitly requested later.



This does not violate serializability because MOCC verifies reads at commit time (same as in OCC). There is only a slightly higher risk of verification failure, but correctness is never compromised. The same technique applies to write locks during the commit phase, too. Whenever we take write locks in non-canonical mode, for example when the record is moved [17] to a different page and the sort order of its write set we initially assumed has changed, we can safely release some of the locks to restore canonical mode.

The restoration of canonical mode is a powerful tool in MOCC to avoid deadlocks without any heavyweight modules or interthread communication. However, releasing too many locks is also costly. In the above example, if *CLL* also contains  $l_5, l_6, \dots, l_{1000}$ , the cost to release and re-acquire a large number of locks is fairly high. In such a case (lines 27–28), MOCC *tries* to take  $l_3$  in non-canonical mode without releasing the affected locks.

Implementing the canonical mode requires a cancellable lock interface discussed in Section 4. When the lock cannot be granted immediately, the acquiring thread can choose to abort and retry because long lock acquire delays might indicate a deadlock (line 29). If the lock is requested in read mode (thus not a mandatory acquisition), then one can ignore the lock request and move on. For write locks, however, a long delay might demand an immediate abort because acquiring the lock is mandatory for determining serialization order.

### 3.3.3 Retrospective Lock List

Whenever a transaction in MOCC aborts, due to either the aforementioned conservative aborts in non-canonical mode or simply a read verification failure, MOCC instantiates a Retrospective Lock List (RLL) for the transaction. An RLL is a sorted list of locks with their preferred lock modes that will likely be required when the thread retries the aborted transaction. The RLL keeps the retried transaction in canonical mode for most cases.

**Constructing RLL:** Function `construct_rll()` in Algorithm 1 shows how MOCC constructs RLL from the read/write set of the aborted transaction and the temperature statistics. All records in the write set are added to RLL in write mode (lines 40–41). Records in the read set that caused verification failures or in hot pages are added to RLL in read mode (lines 42–45). When a record is both in the read set and write set, which happens often, RLL maintains a single entry for the record in write mode. At the end of the construction, MOCC sorts entries in RLL for the next run.

**Using RLL:** Function `read()/read_write()` in Algorithm 1 is invoked whenever a transaction accesses a record. The transaction checks the temperature statistics and queries the RLL. When either of them implies that a pessimistic lock on the record is beneficial, we immediately take all locks in RLL ordered before the requested lock. The transaction also respects the recommended lock mode in the RLL. In other words, the preferred lock mode in RLL overrides the requested lock mode. It takes a write lock instead of read lock when the requested lock itself exists in RLL. Even if the transaction does not see a high temperature of the page, it takes a read lock on the record when RLL advises so, which means the previous run was aborted due to a conflict on the record despite low temperature.

RLL often keeps the retried transaction in canonical mode, making the transaction completely free of deadlocks and aborts because all the contended reads are protected with read locks. As we verify in Section 5, RLL achieves significantly more robust progress even for the most contented transactions than either OCC or 2PL.

However, there are a few cases where even RLL might not prevent aborts. First, the user code might have non-determinism that changes its behavior in another run and accesses a different set of records. Such applications exist, but removing non-determinism is relatively

easy for most applications and will provide additional opportunities to improve performance [36]. Second, even when the user code is *logically* deterministic, it might access different *physical* records due to various structural modification operations (SMOs), such as page splits. When these cases occur, MOCC falls back to the behavior in non-canonical mode regarding the particular record that violates canonical mode. It might cause another abort, but it is unlikely for the same thread to keep observing an SMO for the same page.

### 3.3.4 Commit Protocol

An MOCC transaction invokes `commit()` and `abort()` functions in Algorithm 1 when it commits. Like OCC, MOCC verifies reads after taking write locks.

We verify all reads no matter whether they are protected by read locks. There are two practical reasons for us to take this approach. First, even if the record is already being protected by a read lock, it might not be acquired at the point this tuple was first read during the transaction because MOCC eagerly releases and re-acquires locks. Second, once we have acquired a read lock, no other transactions can modify the record’s version number, which should be cached by the CPU, making the verification process cheap.

When verification fails due to clobbered reads, MOCC updates the temperature statistics of the corresponding page. MOCC then constructs an RLL from the now-complete read/write sets. MOCC uses RLL only when the previous run aborts and the client chooses to retry the same transaction.

## 3.4 Discussion

**Temperature Statistics:** Temperature statistics can be either per-page or per-record. Per-record statistics would be more accurate to predict verification failures on individual records, but it requires a larger footprint for the statistics. Our current implementation chose per-page statistics because in many cases RLL can recommend read locks on individual records that caused aborts, which overrides what the (low) page temperature recommends.

The MOCC protocol is not sensitive to the threshold unless it is an extremely low (e.g., 0) or high (e.g., 20) as shown in Section 5. Highly conflicting pages, by definition, quickly reach the threshold and trigger read locks. In most cases, we recommend setting the threshold to 10.

The counter must be either reset or decremented because the nature of the page might change over time. Our experiments in Section 5 occasionally reset the value of the counter to zero. The statistics will be inaccurate immediately after the reset, but the value quickly becomes large again if the page still is hot. MOCC can also use alternative approaches, such as occasionally decrementing the counter when the read lock turns out to be unnecessary by checking the state of lock queues at commit time.

**Consistent Ordering:** The ordering scheme can be anything so long as it is universally consistent among all threads as discussed in [41], and we follow the same principle. We note, however, that the order based on virtual addresses does not work in some situations. FOEDUS, like most major databases, run multiple processes using multiple shared memory segments. The order of virtual address is not consistent among processes in such an environment. For instance, shared memory segments A and B might be ordered  $A < B$  in process-1’s virtual address space while they might be ordered  $B < A$  in process-2. Furthermore, even in a single process, memory-mapping APIs, such as `shmat` and `mmap`, give aliases to the same physical memory, leading to inconsistent order and deadlocks. We thus use a logical identifier of each lock, which consists of a shared memory segment ID and an offset of the lock object from the beginning of the memory segment.

## 4. MOCC QUEUING LOCK

In this section, we propose the MOCC Queuing Lock (MQL) to realize MOCC without the prohibitive overheads incurred by traditional locking. MQL is a scalable, queue-based reader-writer lock with flexible interfaces and cancellation support. This section focuses on describing:

- Why MOCC demands yet another lock (Section 4.1);
- A high level summary of MQL (Section 4.2);
- How MOCC uses MQL (Section 4.3).

### 4.1 Background

**Native Locks:** MQL must be a *native* lock. Traditional databases use record locking that supports reader-writer modes and cancellation. But, they often use two levels of synchronization primitives: (1) an exclusive spinlock to protect the lock state itself, and (2) a logical lock to protect a record in various modes.

Traditional lock managers often use basic, centralized spinlocks (e.g., test-and-set and ticket locks) for (1) and sleep-wait (e.g., pthread mutex) for (2). Lock managers typically employ a hash-partitioned lock table [13] to manage locks. Any access to lock state must be protected by a spinlock in the corresponding partition. A transaction must acquire the spinlock every time no matter whether it merely checks lock compatibility or inserts a new request queue. Because all accesses to the lock state are serialized, the database can easily provide various lock modes and support cancellation.

This approach works well enough in disk-based databases with a few CPU cores, but recent databases optimized for main memory and manycore servers observed bottlenecks in two-tier locking. Thus, recent systems have started to employ low-level synchronization primitives directly as record locks. For example, FOEDUS places a spinlock in each record header. Transactions directly synchronize with each other using spinlocks. We call this approach “native locking”. MOCC demands a native lock to scale up to 1000 cores.

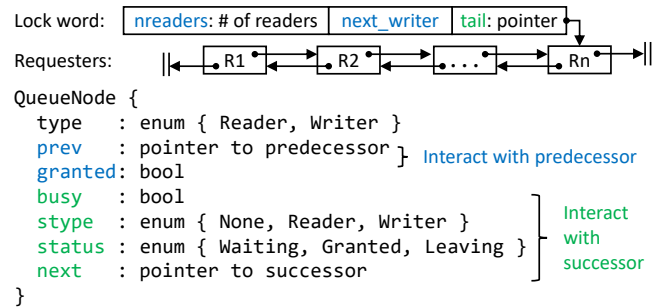
**Queue-based Locks:** In deep memory hierarchies, queue-based locks (e.g., MCS locks [30]) exhibit much better scalability than centralized locks. For example, prior work [42] on a 240-core server observed that the MCS lock has two orders of magnitude better performance than centralized spinlocks. However, existing native queue-based locks have limited functionality and cannot fully replace logical locks found in traditional databases.

For example, FOEDUS uses the MCS lock for records. The reason it could do so is because its commit protocol needs only exclusive and unconditional record locking. MOCC needs to issue read locks, which inherently demands reader-writer modes. MOCC also requires the cancellation (timeout) functionality to avoid deadlocks. Existing queue-based lock algorithms [15, 22, 31, 37, 42] can only partially satisfy MOCC’s requirements: they lack either the needed functionality or performance. For instance, exclusive-only locks would lower concurrency while non-cancellable locks would cause unrecoverable deadlocks. We thus develop MQL to support both cancellation and reader-writer modes.

### 4.2 Summary of MQL

**Key Challenge:** The key challenge, which makes the full algorithm complex, is the increased amount of lock states. Reader-writer [31] and cancellation [37] features fundamentally necessitate more lock states. To achieve both, MQL maintains three variables in the lock, a doubly-linked list of requesters, and five additional variables in each queue node (qnode) that represents a requester.

MQL uses a collection of lock-free protocols to carefully coordinate reads and writes on these lock states and to handshake between threads. MQL follows the lock-free principle because it defeats the whole purpose to take a lock within a scalable lock. MQL is



**Figure 4: MQL data structures.** Requesters form a doubly-linked list of qnodes to handle cancellation.

complex because the complexity of lock-free programming, unfortunately, is often superlinear to the complexity of shared information. It must carefully handle data races that lead to incorrect concurrency, deadlocks, dangling pointers, and/or corrupted lock states.

We now present the data structures of MQL and how MQL uses them to support reader-writer modes and cancellation. We omit details here, such as required memory barriers in each shared access. Interested readers may refer to the extended version of this paper [44] for the full MQL algorithm.

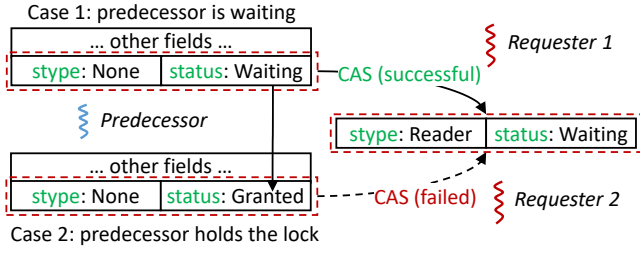
**Data Structures:** MQL’s data structures are based on the fair variant of the reader-writer MCS lock [31]. If a read requester R is queued up after a writer W, whose predecessor is another *active* reader holding the lock, R still has to wait until W hands over the lock. This property ensures that MOCC treats reads and writes fairly.

As Figure 4 shows, *tail* always points to the last requester (if any). The *nreaders* field records the number of readers currently holding the lock, and *next\_writer* points to the qnode of the upcoming writer waiting for the lock. In qnode, *type* and *granted* indicate the qnode owner’s type (reader or writer), and whether it has acquired the lock, respectively. In addition to the *next* field found in the MCS lock’s qnode, MQL’s qnode records more information to support concurrent readers: *stype* records the successor’s type (if any) and a reader requester can get the lock immediately if its predecessor is an active reader (described later).

Supporting cancellation essentially requires a doubly-linked list of qnodes. Each qnode contains an additional *prev* field to track the predecessor and support cancellation, similar to the design of the MCS lock with timeout [37]. The *granted* and *prev* fields are used to interact with the predecessor, whereas the last four fields (in green color in Figure 4) are for interacting with the successor.

**Supporting Readers and Writers:** Without cancellation, MQL works similarly to the fair reader-writer variant of the MCS lock. A requester (reader or writer) R brings a qnode and joins the queue using the wait-free doorway [21]: R issues an atomic-swap (XCHG) instruction to install a pointer to its qnode on *lock.tail*, after initializing its qnode with the proper values shown in Figure 4. The XCHG will return a pointer to the predecessor P, if any.

If P conflicts with R (i.e., any of them is a writer), R must wait for its predecessor to wake it up. If both P and R are readers, R can enter the critical section if the lock is free or P is also a reader *and* is holding the lock. As shown in Figure 5, R should handle two principal cases: (1) P is also waiting for the lock and (2) P has acquired the lock. In case 1, R must wait for the predecessor to awaken it; in case 2, R should be granted the lock immediately because P is compatible with R (both readers). Note that P might be going through the same process and have its *status* changed to *Granted* at any time. Therefore, R must use a compare-and-swap (CAS) in-



**Figure 5: The two principal cases a reader requester should handle if it has a reader predecessor.**

struction to try “registering” itself on P’s `stype` field, by changing the composite of `[busy, stype, status]` from `[False, None, Waiting]` to `[False, Reader, Waiting]`. If the CAS succeeds, R will continue to spin on its `granted` field until it is toggled by P, which should check if it has any immediate reader successor waiting to be waken up once granted the lock. Failing the CAS indicates P has already acquired the lock, so R should toggle its `granted` field and is granted the lock immediately. These three fields must fit within the same 8-byte word to be used with CAS. The `busy` field is used to indicate whether the `qnode` owner is open to state change made by the successor. We omit it in Figure 5 for clarity.

**Supporting Cancellation:** Cancelling a lock request introduces additional races: the predecessor, successor, or both might be trying to cancel. In MQL, we use the `prev` field in the `qnode` as a channel to coordinate with the predecessor, and the `next` and `status` fields to coordinate with the successor. We next explain how MQL handles cancellation of a reader-lock request with a reader predecessor. The other cases (e.g., cancelling a writer-lock request) work similarly.

To cancel, a reader R must: (1) notify the predecessor about the leaving, (2) give the predecessor a new successor (if any), and (3) tell the successor that it has a new predecessor (if any). We devise a handshake protocol for (1) and (2) to work: when a reader predecessor P is granted the lock and trying to pass the lock to R, P must CAS `R.prev` from a pointer to P’s `qnode` to a sentinel value `Acquired` and retries if the CAS failed. To cancel, R atomically installs `NULL` on `R.prev` using `XCHG`. If the `XCHG` returns `Acquire`, then R is granted the lock. Otherwise, R continues by trying to change the composite field of `[busy, stype, status, next]` in P’s `qnode` from `[False, Reader, Waiting, R]` to `[False, Reader, Waiting, SuccessorLeaving]` using a CAS. `SuccessorLeaving` is a sentinel value indicating the successor is cancelling. If the CAS succeeded, then R has successfully marked its departure and can proceed to step 3; if the CAS failed, R must check the current value of the composite field and decide how to proceed. For example, if the predecessor’s `status` is now `Granted`, then R can be granted the lock as well.

After marking the departing status in the predecessor’s `qnode`, R must ensure its successor S is stable before making S visible to S. R accomplishes this by setting its `status` to `Leaving`. Recall that a cancelling successor must CAS the predecessor’s composite field including `status`. Thus, this state change can be an atomic write, and it ensures S will not leave the queue until R finishes its operations. R then links its predecessor and successor using a series of atomic operations before it leaves.

### 4.3 Using MQL in MOCC

Table 1 summarizes how MOCC uses MQL in various situations. MOCC requests either a read lock or write lock depending on the type of the record access. The primary way to acquire locks in MOCC is the unconditional mode because it is the most performant

**Table 1: Using MQL in MOCC.**

Mode	Description	Use in MOCC
Read/-Write	Allows concurrent readers. Write is exclusive.	All cases
Unconditional	Indefinitely wait until acquisition.	Canonical mode.
Try	Instantaneously gives up. Does not leave <code>qnode</code> .	Non-canonical mode. Record access.
Asynchronous	Leaves <code>qnode</code> for later check. Allows multiple requests in parallel.	Non-canonical mode. Record access and pre-commit (write set).

code path; it simply waits on the `qnode`’s `granted` field.

When a transaction is not in canonical mode, however, we might need to use the cancellation functionality. Specifically, a transaction issues a lock request in either *try* or *asynchronous* mode. Both of them push the `qnode` into the requesters list. The try mode then instantaneously gives up and removes the queue node when the lock is not immediately acquirable. The asynchronous mode, on the other hand, leaves the queue node in the case so that we can later check whether we acquired the lock.

## 5. EVALUATION

We have implemented MOCC with MQL in FOEDUS [17]. We empirically compare the performance of MOCC with other methods on a variety of workloads. In particular, we confirm that:

- MOCC keeps OCC’s low overhead in **low contention** workloads (Section 5.2);
- MOCC’s selective locking achieves high scalability in **high contention, low conflict** workloads (Section 5.3);
- MOCC with MQL achieves significantly lower abort ratio and higher performance than both OCC and pessimistic CC in **high contention, high conflict** workloads (Section 5.4);
- MOCC can autonomously and quickly adjust itself in more realistic, **dynamically shifting** workloads on multiple tables with different nature (Section 5.5).
- MOCC is especially beneficial for long-running transactions (e.g., scan) with high conflict operations (Section 5.6).

### 5.1 Setup

We run experiments on machines listed in Table 2. Most experiments use the largest server, GryphonHawk, which is equipped with 16 Intel Xeon E7-8890 processors, each with 18 physical cores. In total the server has 288 physical cores and 12 TB of main memory. The processor has 256 KB of L2 cache per core and 45 MB of L3 cache shared among all cores in each socket. For all experiments, we fit the whole database in memory. Each run lasts for 10 seconds, and is repeated for 3 (YCSB) or 10 (TPC-C) times.

**Table 2: Hardware for Experiments.**

HP Model	EB840	Z820	DL580	GryphonHawk
Sockets	1	2	4	16
Cores (w/HT)	2 (4)	16 (32)	60 (120)	288 (576)
CPU [GHz]	1.90	3.40	2.80	2.50
DRAM	DDR3			DDR4

#### 5.1.1 CC schemes and systems

We compare MOCC with a variety of CC schemes and systems. All the systems used in our experiments are serializable *embedded*



databases. The user writes transactions directly using the system-provided APIs without the overhead to interpret SQL queries over the network. SQL-based systems (e.g., MySQL) could perform orders of magnitude slower due to these overheads, hence we conduct experiments on embedded databases for fair comparison.<sup>1</sup>

**MOCC/OCC:** Throughout this section, **FOEDUS** denotes the original FOEDUS to represent the performance of pure OCC. **MOCC** denotes the modified version of FOEDUS that implements the MOCC protocol with MQL. We use per-page temperature and  $H=10$  as the temperature threshold unless noted otherwise.

**PCC/Dreadlock/WaitDie/BlindDie:** To compare with pessimistic approaches and implementations, we evaluate a few pessimistic schemes, one in FOEDUS and others in Orthrus [34], a recent system that targets high contention workloads. **PCC** denotes a 2PL variant we implemented in FOEDUS. **Dreadlock/WaitDie/BlindDie** denote 2PL variants in Orthrus. They take locks before every access to guarantee serializability. All of them *avoid* deadlocks by aborting threads that see themselves potentially risking deadlock.

PCC uses MQL’s try interface to opportunistically take locks. If the lock cannot be granted immediately, the transaction will continue executing without taking the lock, thus is deadlock-free. At commit time, PCC tries to acquire write locks for records in the write set, but aborts if the lock cannot be granted immediately.

WaitDie is a traditional deadlock avoidance protocol where a lock-waiting transaction with older timestamp is immediately aborted. BlindDie is a simplistic variant of WaitDie where the transaction is aborted regardless of timestamp. Dreadlock [19] detects potential deadlocks by maintaining and spreading bitmaps that summarize recursive dependencies between waiting threads. When a thread finds its own fingerprint in the bitmap, Dreadlock predicts a risk of deadlock. Dreadlock might have false positives, but has no false negatives after sufficient cycles of checking and spreading bitmaps.

**Orthrus:** We also compare MOCC with Orthrus [34], a recent proposal that separates CC and transaction worker threads for high contention scenarios. We configured Orthrus on our hardware with the kind help from the authors and verified its performance results with them. We set the ratio of CC:executor threads to 1:4. Section 6 discusses Orthrus in more detail.

**ERMIA:** Serializable MVCC is a popular choice in recent main-memory systems because of its read-friendliness [6, 16, 24]. ERMIA [16] is based on snapshot isolation and uses the serial safety net (SSN) [43] to guarantee serializability. We use ERMIA as a representative for both certification and MVCC based systems.

### 5.1.2 Workloads

Experiments in this paper use the following benchmarks.

**TPC-C:** TPC-C is arguably the most widely used OLTP benchmark. It involves six tables and five transactions that generate moderate read-write conflicts. We place the same number of warehouses as the number of threads. About 10% transactions access remote warehouses, following the TPC-C specification. The workload is known to be relatively easy to partition such that most records are either read-only or read/written only by the owner of the partition [5].

**YCSB:** YCSB has only one table and simple, short transactions. Unlike TPC-C, it has no locality, thus no partitioning makes sense. We have implemented YCSB in FOEDUS and ERMIA. We also modified the YCSB implementation in ERMIA and Orthrus (the “microbenchmark”) to be equivalent to ours.

<sup>1</sup>All the source code and experimental results of MOCC presented in this paper are available at <https://github.com/hkimura/foedus>. We downloaded the latest version of ERMIA source code from <https://github.com/ermia-db/ermia>. The authors of Orthrus kindly shared their latest source code with us.

**Table 3: TPC-C throughput (low contention, low conflict) on GryphonHawk. MOCC behaves like OCC. PCC has a moderate overhead for read locks. ERMIA is slower due to frequent and costly interthread communication.**

Scheme	Throughput [MTPS+Stdev]	Abort Ratio
MOCC	16.9±0.13	0.12%
FOEDUS	16.9±0.14	0.12%
PCC	9.1±0.37	0.07%
ERMIA	3.9±0.4	0.01%

## 5.2 TPC-C: Low Contention, Low Conflict

We run TPC-C under MOCC, FOEDUS, PCC, and ERMIA. We have not tested TPC-C in Orthrus because its current implementation lacks support for range queries. Table 3 shows the results.

TPC-C has low contention. It contains some read-write conflicts, such as remote-warehouse accesses in `payment` and `neworder`, but still there are on average a very small number of threads touching the same records concurrently. Hence, even FOEDUS experiences only one in thousand aborts.<sup>2</sup> MOCC behaves exactly the same as OCC because the temperature statistics of almost all data pages are below the threshold, rarely triggering read locks.

For such low contention, low conflict workloads, PCC adds unwanted overhead of read locks for only slight reduction in abort ratio. However, PCC, unlike the experiments that follow, is only  $\sim 2\times$  slower than FOEDUS/MOCC. The reason is TPC-C does not have much physical contention, thus locks are not heavily contended.

ERMIA performs the lowest ( $\sim 3.9$  MTPS) among the four systems we tested for TPC-C, due to its centralized design. It frequently issues atomic instructions such as CAS on centralized memory locations to manage resources (e.g., transaction IDs). To guarantee serializability, ERMIA needs to stamp most of the tuples read, making reads become writes. The experiments confirm that having frequent interthread communication prevents the system from scaling up to hundreds of cores, even for low-contention workloads.

In summary, avoiding frequent interthread communication allows scaling up low contention/conflict workloads easily. Pessimistic schemes might incur additional overheads of read locks, but this is a fixed overhead rather than a scalability bottleneck.

## 5.3 High Contention, No Conflict YCSB

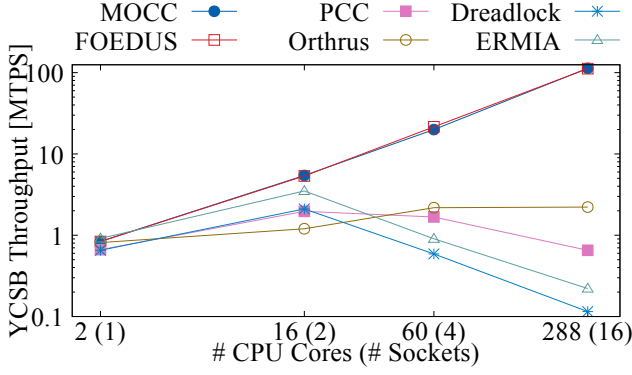
The rest of the experiments focus on high contention cases: a large number of concurrent threads touch a small number of records. There are two kinds of highly contended workloads: low read-write conflict and high read-write conflict. This section evaluates the performance under the former type of workloads.

To show the effect of contention, we vary the scale of the hardware ranging from a laptop (HP EB840) to GryphonHawk as listed in Table 2. Part of this experiment was shown in Figure 1. In this section, we also show additional throughput numbers for the other systems. Figure 6 plots the throughput of a read-only YCSB workload where each transaction reads 10 records randomly from 50 records. Both MOCC and FOEDUS scale up perfectly because they do not cause any interthread communication.

Pessimistic approaches—PCC and Dreadlock—are again slower than MOCC/FOEDUS, but this time they are  $170\times$  slower because a huge number of concurrent threads must take read locks on the same record, causing severe physical contention. Furthermore, Dreadlock

<sup>2</sup>The table does not count deliberate 1% aborts in `neworder` defined in the TPC-C spec because they are not race aborts.





**Figure 6: Throughput of a read-only YCSB workload with high contention and no conflict on four machines with different scales. MOCC adds no overhead to FOEDUS (OCC), performing orders of magnitude faster than the other CC schemes.**

adds one more scalability bottleneck: frequent interthread communication to check and disseminate thread fingerprints. It is not a significant overhead on machines with up to two sockets, but it causes expensive cache-coherence communication on deep memory hierarchies that interconnect hundreds of cores. In the remaining experiments, we observed orders of magnitude slower performance on Dreadlock for this reason. This result clearly shows the limitation of pessimistic approaches under highly contended workloads.

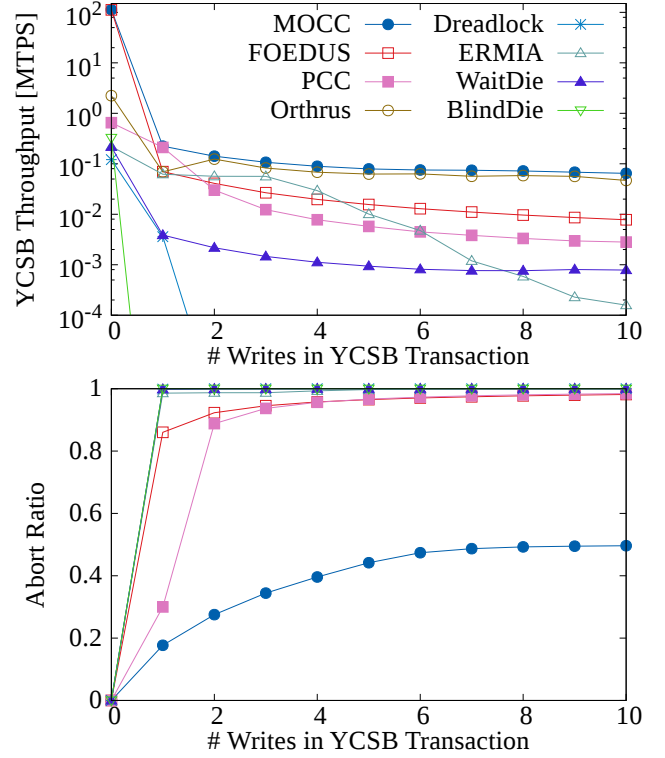
Orthrus scales better than PCC, but it still needs frequent interthread communication between the execution threads and CC-threads. YCSB accesses a completely random set of records per transaction with no locality whatsoever, which means that the number of lock manager threads involved in each transaction will be approximately equal to the number of records accessed. This represents the worst case scenario in Orthrus, since the number of messages sent per transaction is linear to the number of lock manager threads involved in that transaction. For this workload without any read-write conflicts, execution threads can quickly saturate CC threads, limiting the throughput to 2 MTPS. We also varied the fraction of executor and CC threads, but the throughput is not comparable to that of FOEDUS/MOCC ( $> 100$  MTPS).

On smaller scales with at most two sockets, ERMIA is only slower than FOEDUS and MOCC. As we move to larger machines, ERMIA's throughput drops significantly: from 3.5 MTPS (2 sockets) to 0.23 MTPS (16 sockets). But it is still much faster than Dreadlock for all cases. On large machines such as GryphonHawk, ERMIA's centralized thread registration mechanism used by SSN (for guaranteeing serializability) becomes a major bottleneck, although it is not necessary for this workload. Without SSN, ERMIA can achieve as high as  $\sim 4.6$  MTPS for the same workload on GryphonHawk under snapshot isolation. Interthread communication required by the other centralized mechanisms (e.g., transaction ID management) prevented ERMIA from scaling further.

Although we omit the details due to limited space, we observed similar effects on our partial implementation of TPC-E [40]. TPC-E contains frequent writes, but they usually occur in less contended records. Instead, small read-mostly/read-only dimension tables, such as TRADE.TYPE, receive highly contended reads as above.

## 5.4 High Contention, High Conflict YCSB

The next experiment also uses YCSB under high contention, but also includes read-modify-write (RMW) operations that can cause frequent read-write conflicts. Again we use a table of 50 records

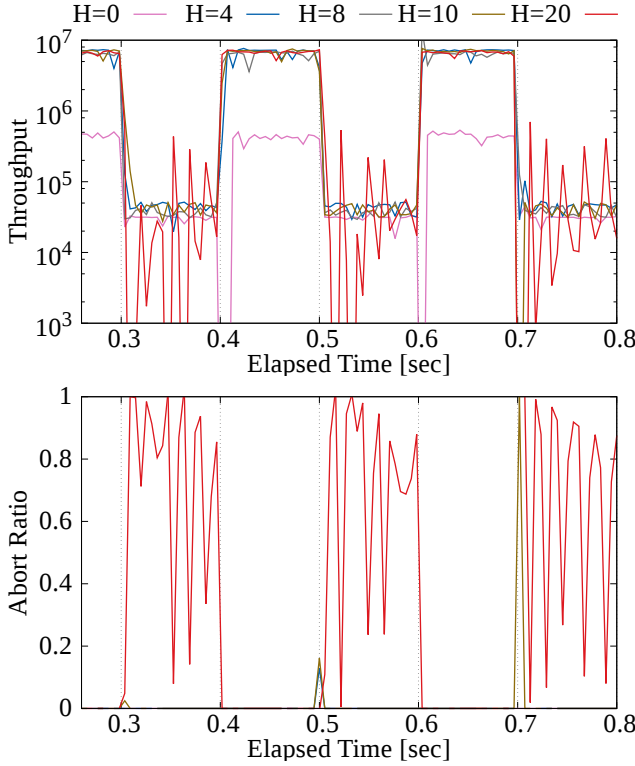


**Figure 7: Read-write YCSB on GryphonHawk with high contention and/or high conflict. MOCC achieves dramatically lower abort ratio than OCC and even PCC, while maintaining robust and the highest throughput for all cases.**

and let each transaction access 10 records. We vary the amount of RMWs between 0 and 10; the rest operations are pure reads. We access records in random order to evaluate how pessimistic approaches handle deadlocks.

Figure 7 shows the throughput and abort ratio of each CC scheme. As expected, FOEDUS's throughput significantly drops due to massive aborts: as many as 98% of transactions abort on the highest conflict level. Somewhat ironically, although the pessimistic variants (PCC, Dreadlock, WaitDie, and BlindDie) protect reads using read locks, their performance still drops due to aborts caused by deadlocks. The workload accesses records in a random order. Therefore, traditional pessimistic approaches also suffer. Further, although the abort ratio of PCC is slightly lower than FOEDUS, each retry in PCC takes significantly longer than in FOEDUS, mostly busy-waiting for the lock. Thus, PCC's throughput is a few times lower than FOEDUS except with one RMW where pessimistic waiting significantly lowers aborts. Again, ERMIA's centralized design does not work well on large machines. With more writes, its performance gradually converges to PCC's and becomes worse than it as the workload becomes write dominant with more than 5 RMWs. Most aborts are caused by write-write conflicts (instead of serializability violations), which abort at least one of the participating transactions.

MOCC and Orthrus sustain high performance in this workload. MOCC dramatically reduces aborts without adding noticeable overhead, resulting in orders of magnitude higher throughput than FOEDUS and PCC. Although Orthrus has zero deadlocks, MOCC is an order of magnitude faster than Orthrus for transactions with up to one RMW because Orthrus needs frequent interthread communication between executor and CC threads. For transactions with two or



**Figure 8: Throughput (top) and abort ratio (bottom) of the multi-table, shifting workload on GryphonHawk. The workload changes its access pattern on a contended table every 0.1 second. Larger thresholds take longer to learn new hot spots.**

more RMWs, the maximal concurrency of the workload becomes fundamentally low. This shrinks the difference between MOCC and Orthrus to less than  $2\times$ , but still MOCC is consistently faster. Interestingly, the performance of Orthrus increases from 1 RMW to 2 RMWs because contention on the lock manager/CC queues decreases, leading to faster interthread communication.

## 5.5 Multi-table, Shifting Workloads

**Multi-table Environment:** Real-world transactions often access many tables of varying sizes and read-write patterns. Most workloads access large tables without high contention or conflict. Nevertheless, they also involve highly contended tables, either small tables or tables that receive skewed accesses. A monolithic scheme that works well for one might not for another.

To test MOCC in more realistic situations, we devise a two-table experiment. The first table contains one record, while the other contains one million. In each transaction, we issue one operation (initially a read) to the small table and 20 reads to the large table.

**Shifting Workload:** Real workloads are also often *dynamic*, i.e., the contention level and access patterns change over time. Hence, this experiment also dynamically switches the nature of the small table every 0.1 second, issuing an RMW instead of a read. During the first 0.1 second the small table is highly contended but has no conflict, hence the optimal behavior is expected to be similar to what we have shown in Section 5.3. During the next 0.1 second it receives frequent conflicts, hence the optimal behavior is like the one in Section 5.4. We keep switching between the two states.

Figure 8 shows the throughput and abort ratio of MOCC over time with different temperature thresholds ( $H$ ) for MOCC to trigger

**Table 4: Long Scan Workload on GryphonHawk.**

Scheme	Throughput [kTPS+Stdev]	Abort Ratio
MOCC	199.6 $\pm$ 3.1	0.4%
FOEDUS	10.5 $\pm$ 0.0	99.55%
PCC	25.7 $\pm$ 1.7	0%
Thomasian	20.8 $\pm$ 1.5	50.1%

pessimistic locking. We periodically reset the temperature counters.

During the no-conflict period, all threshold values perform the same except  $H = 0$ .  $H = 0$  is significantly ( $24\times$ ) slower because it always takes read locks like pessimistic schemes. We emphasize that the transaction issues most (20 out of 21) operations on the low-contention table. A monolithic, pessimistic scheme slows down for  $24\times$  due to the single operation on the high-contention table. This is very similar to our observations in TPC-E’s `TRADE_TYPE` table, which contains only five records and is very frequently read. Real workloads often contain such tables.

After switching to high conflict cases (at 0.3, 0.5, 0.7 seconds), all thresholds except  $H = 0$  observed aborts. Lower thresholds (4, 8, 10) result in quicker learning and the abort ratio immediately drops to zero. Extremely large thresholds keep causing aborts and unstable throughputs. For example,  $H = 20$  requires  $2^{20} \approx 1,000,000$  aborts to trigger read locks, which is virtually a monolithic OCC. A monolithic scheme, again, does not work well due to a single high conflict operation, which is not rare in real workloads.

We observe that as long as the threshold is reasonably small, all MOCC runs autonomously adjust their behaviors to converge to the same robust throughput. When the threshold is too small, however, MOCC can be confused by just a small number of sporadic aborts, leading to lower performance. Hence, we recommend setting  $H$  to 5–10, depending on the scheme to reset/decrement the counter and on whether the counter is per-page or per-record.

## 5.6 Long Scan Workloads

**Combining OLAP and OLTP:** Real workloads often contain both analytic accesses (e.g., a long cursor scan) and transactional updates. The last experiment evaluates MOCC’s benefits on such hybrid transactions. Every transaction reads one record in the small table of the previous experiment, scans 1,000 records in the larger table, then updates the record in the small table. This time, we also compare with a hybrid OCC method proposed by Thomasian [39], which switches from pure OCC to pure 2PL after an abort. We implemented the algorithm on FOEDUS’s codebase.

Table 4 shows the result. The long running transaction with high read-write conflict is especially challenging for pure OCC. FOEDUS aborts more than 99.5% of the time, being the worst among all schemes. PCC completely avoids aborts and performs faster, but it incurs unwanted locking overheads for scanning. Thomasian’s method [39] sits exactly in-between. It aborts the initial OCC run (thus exactly 50% abort ratio) and pessimistically locks all records in read/write sets in the second run. MOCC performs an order of magnitude better than all others because of its temperature statistics. Rather than statically switching from pure OCC to pure 2PL, all runs in MOCC are hybrid, taking beneficial read locks *within* an OCC transaction. The result verifies that MOCC is especially beneficial for long-running transactions in real workloads.

## 6. RELATED WORK

**Scalable Locks:** Efficient lock algorithm is crucial on many-core servers. Mellor-Crummey and Scott [30] invented the MCS

lock, which scales far better than centralized locks. MCSg [42] allows certain code paths to use the MCS lock without qnodes. The CLH lock [4] is similar to the MCS lock. Each requester spins on its predecessor’s qnode. Cohort locks [7] combine two different synchronization protocols for further scalability. All these locks are queue-based because scalability is fundamentally limited when multiple threads modifies a central, contended memory location.

**Locking in Databases:** Many existing databases use 2PL, which acquires a read or write lock before every data access. To guarantee serializability, no locks can be acquired after the transaction starts to release any lock. Additional locks (e.g., next-key locks [32] and gap locks [26]) prevent phantoms. Compared to 2PL, MOCC allows a transaction to acquire and release locks in an arbitrary order. The validation phase at pre-commit guarantees serializability.

Traditional databases usually use a centralized lock manager [9], which gradually becomes a bottleneck under high parallelism. Speculative lock inheritance [12] alleviates this problem by passing hot locks between transactions without going through the lock manager. Early lock release [14] allows a transaction to release all its locks without waiting for log flush, thus shrinking the size of the critical section. Like very lightweight locking [35], MOCC eliminates the centralized lock manager and co-locates the locks with records.

**Optimistic CC:** OCC is lightweight and reduces cache line invalidation, so it is especially attractive for parallel hardware. After formulated by Kung et al. [20], OCC has been adopted by databases for manycore servers [17, 41]. TicToc [47] avoid centralized timestamp allocation and can commit certain transactions that would be aborted by traditional timestamp ordering schemes. OCC is vulnerable to aborts under high conflict. MOCC solves this problem by selectively using pessimistic locking to protect reads.

**MVCC:** MVCC maintains multiple versions of each tuple and determines whether a transaction can access a version by comparing its begin timestamp and the version’s timestamp. MVCC has been widely adopted in recent main-memory systems [6, 16, 24]. However, a straightforward implementation of MVCC, such as snapshot isolation (SI), is not serializable [1]. Most recent efforts on adding serializability to SI are certification-based. Serializable SI [2] tracks and avoids the “dangerous structures” that will appear in non-serializable executions. ERmia [16] adopts the serial safety net [43], which can add serializability to both SI and read committed. Lomet et al. [25] manage a range of possible timestamps that capture conflicts to allow high concurrency. BOHM [8] analyzes transactions before execution to determine serializable schedules.

**Hybrid Approaches:** There is prior work that combines locking and OCC. Static and dynamic OCC [45] switch from OCC to locking when retrying a transaction that was aborted under OCC. Static OCC acquires all locks before rerun, while dynamic OCC acquires locks upon data access. Thomasian [39] proposes to lock both reads and writes before the global validation in distributed OCC. Locks are retained and re-used should the validation fail. Despite their similarity to MOCC, most prior approaches are designed for disk-based or distributed systems, instead of modern main-memory systems. For instance, a committing transaction under OCC with broadcast during rerun [45] aborts conflicting rerun transactions but allows first-run transactions to run to completion so that all data are brought from disk to main memory. Compared to MOCC, prior hybrid approaches require a transaction use either pure locking or OCC, without considering data temperature.

Among recent systems, Hekaton [6] supports pessimistic locking on top of its MVCC protocol. Ziv et al. [49] formalize the theory of composing computations over multiple data structures using different CC schemes. Cao et al. [3] use OCC and pessimistic CC to handle low and high conflict accesses, respectively. It focuses on

single entry accesses, instead of serializable executions over multiple operations. Contrary to traditional pessimistic and optimistic CC methods characterized by whether there will be conflicts among transactions, LarkTM [48] combines pessimistic and optimistic methods that are characterized by whether a lock will be “required” by the same reader or writer. LarkTM operates on the granularity of objects and gives serializability over multiple operations.

**Orthrus:** Although based on a completely different architecture, Orthrus [34] is an intriguing, closely related approach. Orthrus completely avoids deadlocks and alleviates the overhead to dynamically coordinate between executor threads by handing off the coordination to CC threads via message passing and ordering locks in a consistent order like Silo and MOCC. Section 5 reproduced the very stable throughputs even against the highest read/write conflict as [34] observed, but at the same time we observed a few limitations.

First, although executor threads themselves do not have to coordinate with each other, they have to communicate with CC threads. In some sense, Orthrus shifts the issue of contentious communication to another place. In high throughput experiments where MOCC observes 100 MTPS or more, Orthrus is limited to around 2 MTPS because of the interthread communication. We varied several settings (e.g., the fraction of CC/executor threads and database size), but did not find cases where Orthrus exceeds 10 MTPS throughput.

Second, Orthrus is a *static* approach on assigning CC threads. To benefit from physical partitioning, the workload must have a partitionable locality, and the user must know it beforehand. For totally random workloads, Orthrus’ applicability is limited without oracles. Such totally random workloads (e.g., those represented by the YCSB workloads in Section 5) are the worst case for Orthrus.

## 7. CONCLUSIONS

We have proposed mostly-optimistic concurrency control (MOCC) to enhance OCC for highly contended dynamic workloads, by judiciously using pessimistic locking for suitable records. The main challenge is to add read locks *as of* the read with minimal overhead. Acquiring locks during forward processing breaks OCC’s deadlock-free property. To handle locking and deadlocks efficiently, we advocate native locking, which employs synchronization primitives directly as database locks. We have devised the MOCC Queuing Lock (MQL), a cancellable, queue-based reader-writer lock that works well on massively parallel machines, especially those with deep memory hierarchies. MOCC and MQL handle deadlocks without discarding OCC’s simplicity and scalability. MOCC incurs negligible overhead over OCC and achieves robust performance for a broad spectrum of workloads, including highly contended, conflicting, dynamically shifting ones. MOCC does not require any new modules or controller threads (such as those in Orthrus), thus is simple and easy to incorporate in various OCC databases. The core of our MOCC implementation adds fewer than 1000 LoC on top of the base system, which has more than 100k LoC in total.

## Acknowledgements

We appreciate the kind help from the authors of Orthrus, especially Kun Ren, to configure Orthrus on our hardware. We thank Ryan Johnson (LogicBlox) and our colleagues at Hewlett Packard Enterprise for their feedbacks to earlier versions of this paper and editorial help. We also thank the anonymous reviewers for their comments.

## 8. REFERENCES

- [1] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O’Neil, and P. E. O’Neil. A critique of ANSI SQL isolation levels. *SIGMOD*, pages 1–10, 1995.



- [2] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *TODS*, 34(4):20, 2009.
- [3] M. Cao, M. Zhang, A. Sengupta, and M. D. Bond. Drinking from both glasses: Combining pessimistic and optimistic tracking of cross-thread dependences. *PPoPP*, 2016.
- [4] T. Craig. Building FIFO and priority-queuing spin locks from atomic swap. *University of Washington Technical Report TR 93-02-02*, 1993. <ftp://ftp.cs.washington.edu/tr/1993/02/UW-CSE-93-02-02.pdf>.
- [5] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *VLDB*, 3(1-2):48–57, 2010.
- [6] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server’s memory-optimized OLTP engine. *SIGMOD*, 2013.
- [7] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: A general technique for designing NUMA locks. *PPoPP*, 2012.
- [8] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 2015.
- [9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. 1st edition, 1992.
- [10] Hewlett Packard Enterprise. *HPE Superdome Servers*. <https://www.hpe.com/us/en/servers/superdome.html>.
- [11] Hewlett Packard Labs. *The Machine: A new kind of computer*. <http://labs.hpe.com/research/themachine/>.
- [12] R. Johnson, I. Pandis, and A. Ailamaki. Improving OLTP scalability using speculative lock inheritance. *PVLDB*, 2009.
- [13] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. *EDBT*, pages 24–35, 2009.
- [14] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: A scalable approach to logging. *PVLDB*, 3(1-2):681–692, 2010.
- [15] G. K. R. Kakivaya, D. N. Cutler, and J. M. Lyon. Concurrency-safe reader-writer lock with time out support. *United States Patent US 6546443 B1*, 1999.
- [16] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: Fast memory-optimized database system for heterogeneous workloads. *SIGMOD*, 2016.
- [17] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. *SIGMOD*, pages 691–706, 2015.
- [18] H. Kimura, G. Graefe, and H. A. Kuno. Efficient locking techniques for databases on modern hardware. *ADMS’12*.
- [19] E. Koskinen and M. Herlihy. Dreadlocks: Efficient deadlock detection. *SPAA*, pages 297–303, 2008.
- [20] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, June 1981.
- [21] L. Lamport. A new solution of dijkstra’s concurrent programming problem. *CACM*, 17(8):453–455, Aug. 1974.
- [22] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. *SPAA*, pages 101–110, 2009.
- [23] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. *ICDE*, 2013.
- [24] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. *CIDR*, 2015.
- [25] D. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-version concurrency via timestamp range conflict management. *ICDE*, pages 714–725, 2012.
- [26] D. B. Lomet. Key range locking strategies for improved concurrency. *PVLDB*, pages 655–664, 1993.
- [27] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *PVLDB*, 2012.
- [28] D. Makreshanski, J. Levandoski, and R. Stutsman. To lock, swap, or elide: on the interplay of hardware transactional memory and lock-free indexing. *PVLDB*, 2015.
- [29] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. *EuroSys*, pages 183–196, 2012.
- [30] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS*, 9(1):21–65, 1991.
- [31] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *PPoPP*, pages 106–113, 1991.
- [32] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multi-action transactions operating on B-tree indexes. *PVLDB*, pages 392–405, 1990.
- [33] R. Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.
- [34] K. Ren, J. Faleiro, and D. J. Abadi. Design principles for scaling multi-core oltp under high contention. *SIGMOD’16*.
- [35] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *PVLDB*, 2012.
- [36] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *VLDB*, 7(10):821–832, 2014.
- [37] M. L. Scott and W. N. Scherer. Scalable queue-based spin locks with timeout. *PPoPP*, pages 44–52, 2001.
- [38] SGI. *UV3000*. [https://www.sgi.com/products/servers/uv/uv\\_3000\\_30.html](https://www.sgi.com/products/servers/uv/uv_3000_30.html).
- [39] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *IEEE TKDE*, 10(1):173–189, Jan 1998.
- [40] TPC. TPC benchmark E standard specification.
- [41] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. *SOSP*, pages 18–32, 2013.
- [42] T. Wang, M. Chabbi, and H. Kimura. Be my guest: MCS lock now welcomes guests. *PPoPP*, pages 21:1–21:12, 2016.
- [43] T. Wang, R. Johnson, A. Fekete, and I. Pandis. The serial safety net: Efficient concurrency control on modern hardware. *DaMoN*, pages 8:1–8:8, 2015.
- [44] T. Wang and H. Kimura. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores (Extended Version). *Hewlett Packard Labs Technical Report HPE-2016-58*, 2016. <http://www.labs.hpe.com/techreports/2016/HPE-2016-58.pdf>.
- [45] P. S. Yu and D. M. Dias. Analysis of hybrid concurrency control schemes for a high data contention environment. *IEEE Transactions on Software Engineering*, 18(2):118–129, 1992.
- [46] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 2015.
- [47] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. *SIGMOD*, pages 1629–1642, 2016.
- [48] M. Zhang, J. Huang, M. Cao, and M. D. Bond. Low-overhead software transactional memory with progress guarantees and strong semantics. *PPoPP*, pages 97–108, 2015.
- [49] O. Ziv et al. Composing concurrency control. *PLDI*, 2015.