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

By Tianzheng Wang, Hideaki Kimura
Presented by Qing Wei

Goal

- Achieve orders of magnitude higher performance for dynamic workloads
- Avoid clobbered reads for high conflict workloads, without any centralized mechanisms or heavyweight inter thread communication

Gap

- Emerging servers will contain far more CPU cores a deeper, more complex memory hierarchy.
  - Extremely high contention (Read only YCSB workload)
    - 2PL 170x slower than OCC due to its overhead to take read locks
    - Slowdown quickly grows with the number of cores and the depth of the memory hierarchy due to physical lock contention.
  - Extremely high conflict
    - OCC scales well with low conflict, but suffers high abort ratio under high conflict.
    - With 1 write, 80% of transactions abort
    - With 10 writes, 98% of transactions abort
    - High abort ratio due to deadlocks for pessimistic protocols

Gap-cont.

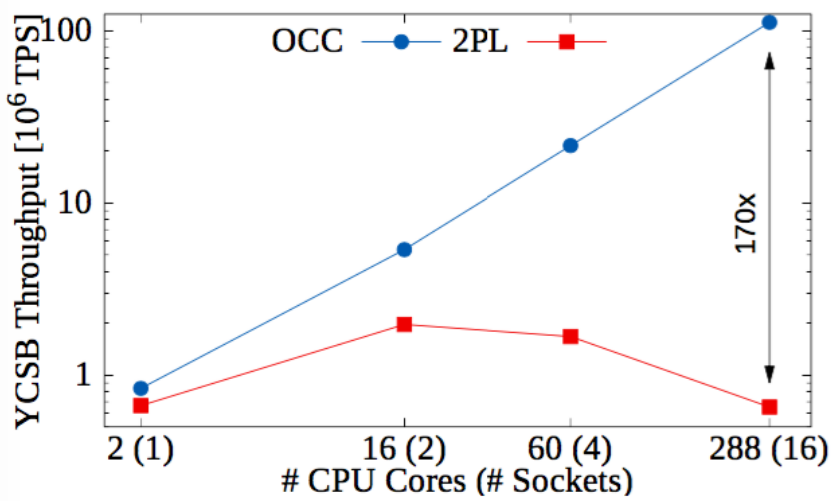

Figure 1: Read-only, highly contended YCSB. OCC scales better than 2PL on a deep memory hierarchy, which augments every single scalability issue to an unprecedented degree.

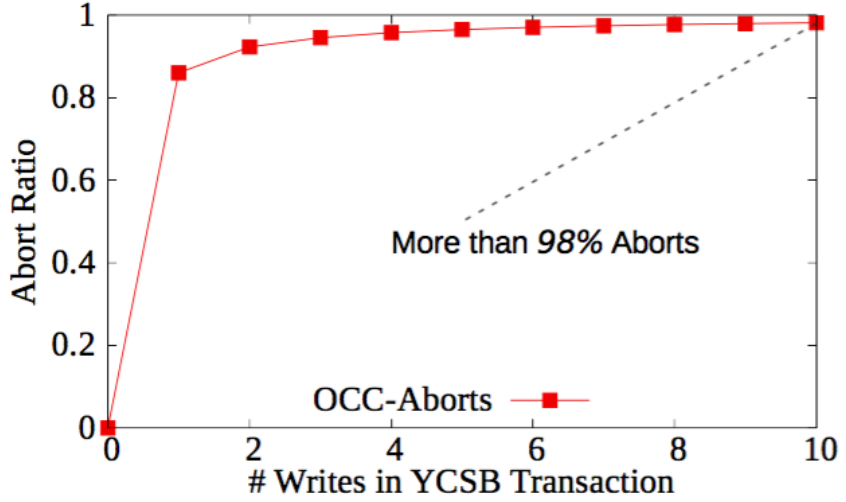

Figure 2: Read-write, highly contended YCSB. OCC is vulnerable to aborts under high conflict. In an extreme case, it makes almost no progress.

Gap-cont.

- Issues in Existing DB
  - Page latches
    - Most existing DBs use page latches to protect a physical page rather than logical records.
    - Poor performance on thousands of cores
  - Reads become writes
    - Limit scalability and performance
  - Expensive inter thread communication
  - Frequent deadlocks and aborts

Mostly-optimistic Concurrency Control

MOCC must be based on 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 inter-thread coordination

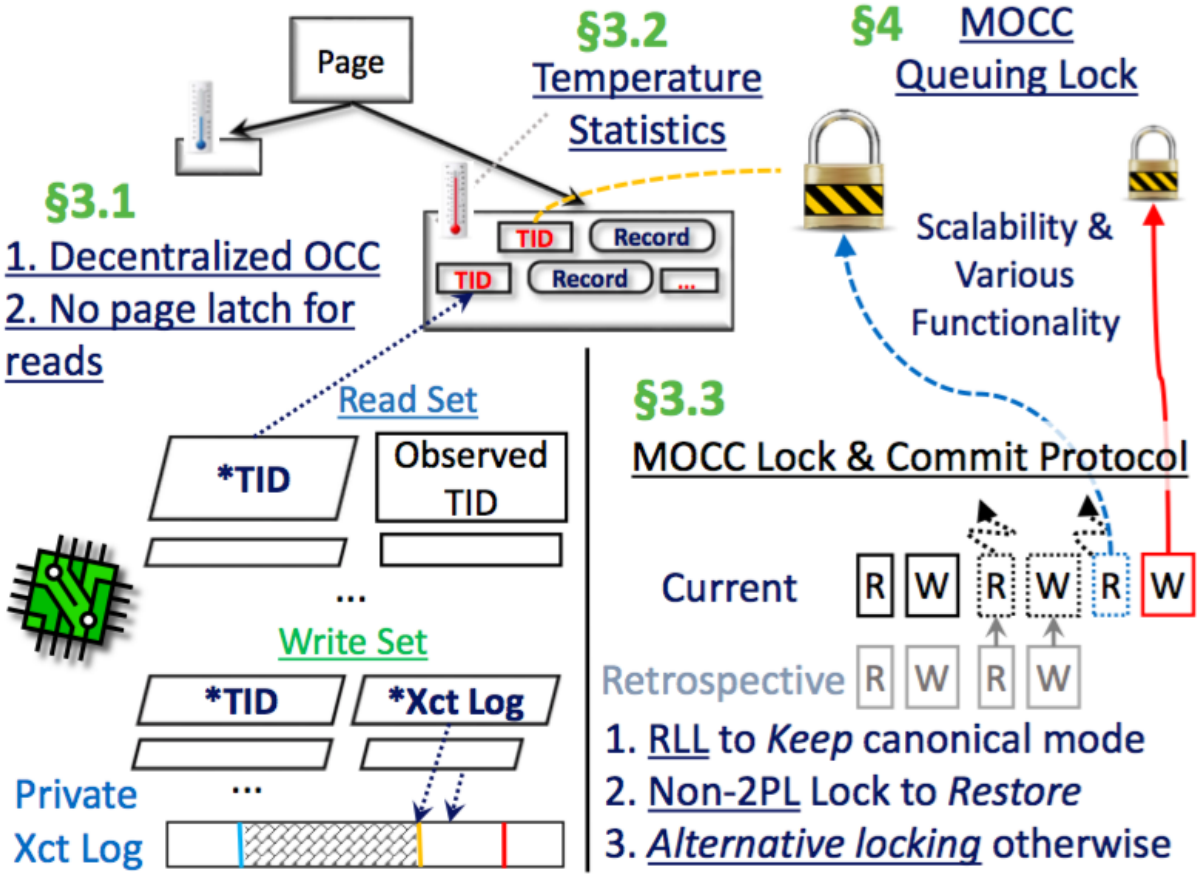

Figure 3: MOCC overview and paper outline.

Recap: Decentralized OCC

- No page latching for reads
- Apply-after-commit
- Deadlock-free verification Protocol
- Missing read locks

Temperature Statistics

- MOCC takes read locks on records whose temperature is above some threshold.
  - These records are likely to be updated by concurrent transactions, causing aborts
  - Tracks the number of aborts due to verification
  - Maintain statistics at page level to reduce space overhead
  - Verification failure will increase the temperature of the affected page.

MOCC Protocols

Canonical mode:

Let $l_m < l_n$ mean that the lock lm is ordered before the lock ln 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 $l_c < l_n : \forall l_n \in NL, l_c \in CLL$.

MOCC Protocols

Canonical mode:

- In canonical mode, no risk of deadlock
- Can unconditionally take locks like FOEDUS or Silo
- MOCC protocol is designed to:
  - Keep transactions in canonical mode as much as possible
  - Restore canonical mode when not in canonical mode
  - Try taking locks as efficiently as possible without risking deadlocks when canonical mode is not attainable

MOCC Protocols

Acquiring and Releasing Locks:

- In traditional 2PL architectures, if a transaction releases a lock before commit and then takes another lock, the execution could be non-serializable.
- MOCC is based on OCC, hence serializability is guaranteed no matter whether it holds a read lock or not.
- **MOCC can safely acquire, release, or re-acquire arbitrary locks in an arbitrary order**.

MOCC Protocols

Acquiring and Releasing Locks Example:

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, then unconditionally take l_3. (No re-take of released locks)

Does not violate serializability (MOCC verifies reads at commit time)

MOCC Protocols

Acquiring and Releasing Locks Example:

CLL : {l_1, l_2, l_4, l_5, l_6,...l_{1000}} and intends to take a read lock t = l_3

Cost to release a large number of locks is high. In such case, MOCC tries to take l_3 in non-canonical mode without releasing the affected locks.

MOCC Protocols

Retrospective lock list (RLL): a sorted list of locks with their preferred lock modes that will likely be required when the thread retries the aborted transaction.

- Constructing RLL
  - All records in the write set are added to RLL in write mode
  - Records in the read set that caused verification failures or in hot pages are added to RLL in read mode.
  - If in both read and write set, then maintains a single entry as write mode.
- Using RLL
  - When either of them implies that a pessimistic lock on the record is beneficial, immediately take all locks in RLL ordered before the requested lock.
  - The preferred lock mode in RLL overrides the requested lock mode

Algorithm 1 MOCC Protocols.

```
1  class MoccTransaction:
     const H # Temperature Threshold
3    R := {} # Read Set
     W := {} # Write Set
5    RLL := {} # Retrospective Lock List
     CLL := {} # Current Lock List
7
     def read(t: Record):
9      if temp(t) >= H or t in RLL:
         lock(t, max(preferred mode in RLL, R-mode))
11     R.add(t, t.TID)
       Read t
13
     def read_write(t: Record):
15     if temp(t) >= H or t in RLL:
         lock(t, W-mode)
17     R.add(t, t.TID)
       Read t
19     Construct log in private buffer
       W.add(t, log)
21   # Blind-write, same as in OCC

23   def lock(t: Record, m: Mode):
       if CLL already has t in mode m or stronger:
25       return
       violations := {l ∈ CLL, l.mode ≠ null, l ≥ t}
27     if too many violations:
         alternative_lock(t, m) # See Section 4
29       return or abort
       elif violations not empty:
31       # Not in canonical mode. Restore.
         CLL.unlock({violations})
33
       # Unconditional lock in canonical mode.
35     CLL.unconditional_lock({l ∈ RLL, l < t})
       CLL.unconditional_lock(t, m)
37
     def construct_rll(): # Invoked on abort
39     RLL := {}
       for w in W:
41       RLL.add(w, W-mode)
       for r in R:
43       if r not in RLL:
           if temp(r) >= H or r failed verification:
45           RLL.add(r, R-mode)
       RLL.sort()
47
     def commit():
49     W.sort()
       for w in W:
51       lock(w, W-mode)
       for r in R:
53       if r.observed_tid not equal r.tid:
           temp(r).hotter() # See Section 3.2
55         abort
       # Committed
57     Determine TID and apply/publish W # Silo/FOEDUS protocol
       CLL.unlock_all()
59     RLL, CLL, R, W := {}

61   def on_abort():
       CLL.unlock_all()
63     if user will retry the transaction:
         construct_rll()
65     else
         RLL := {}
67     CLL, R, W := {}
```

MOCC Queuing Lock

A scalable, queue-based reader-writer lock with flexible interfaces and cancellation support.

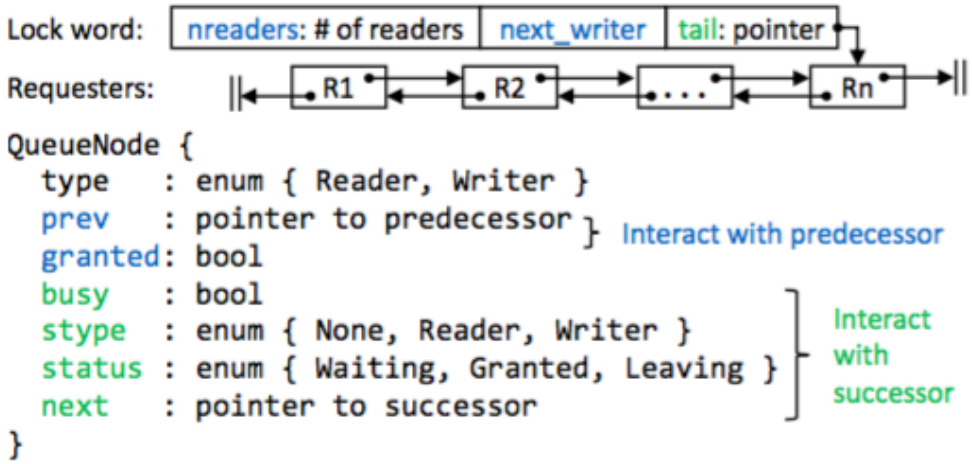

Figure 4: MQL data structures. Requesters form a doubly-linked list of qnodes to handle cancellation.

MOCC Queuing Lock - Supporting Readers/Writers

Fair variant of the reader-writer MCS lock

A requester (reader or writer) R brings a qnode and joins the queue using the wait-free doorway by using an atomic-swap (XCHG) instruction to install pointer to point to its qnode on lock.tail. XCHG will return a pointer to the predecessor P.

If P conflicts with R, R must wait for its predecessor to wake it up.

If P and R are both readers, R can enter the critical section if the lock is free or P is also a reader and is holding the lock

Table 1: Using MQL in MOCC.

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

Evaluation

Set up:

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

Evaluation-cont.

CC schemes:

- MOCC/OCC
- PCC/Dreadlock/WaitDie/BlindDie
- Orthrus (A recent proposal separates CC and transaction worker threads for high contention scenarios)
- ERMIA (MVCC)

Evaluation-cont.

Workloads:

- TPC-C
  - Widely used OLTP benchmark
  - Six tables and five transactions generate moderate read-write conflicts
- YCSB
  - One table and simple, short transactions

Evaluation-cont.

TPC-C Low Contention, Low Conflict

Low contention: some read-write conflicts.

MOCC: temperature statistics of almost all data pages are below the threshold, no read locks

ERMIA performs lowest due to its centralized design

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%

Evaluation-cont.

YCSB High Contention, No Conflict

Pessimistic approaches, are slower than MOCC/FOEDUS because read locks and severe physical contention.

Orthrus scales better than PCC, but it still needs frequent interthread communication.

Ermia‘s centralized thread registration becomes a major bottleneck.

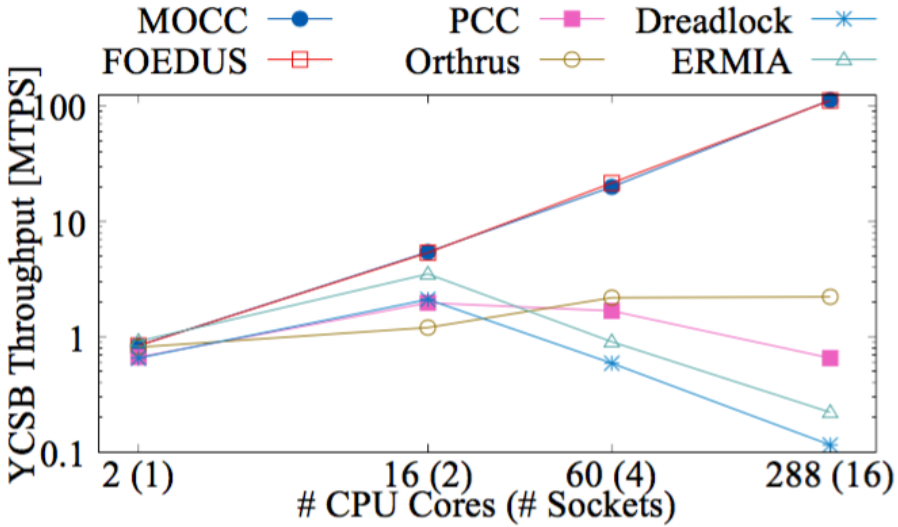

Figure 6: Throughput of a read-only YCSB workload with high contention and no conflict on four machines with different scales. MOCC adds no overhead to FOEDUS (OCC), performing orders of magnitude faster than the other CC schemes.

High Contention, High Conflict YCSB

Vary the amount of read-modify-operation from 0-10

FOEDUS's throughput significantly drops due to massive aborts.

Pessimistic approaches' performance still drops due to aborts caused by deadlocks

MOCC dramatically reduces aborts without adding noticeable overhead.

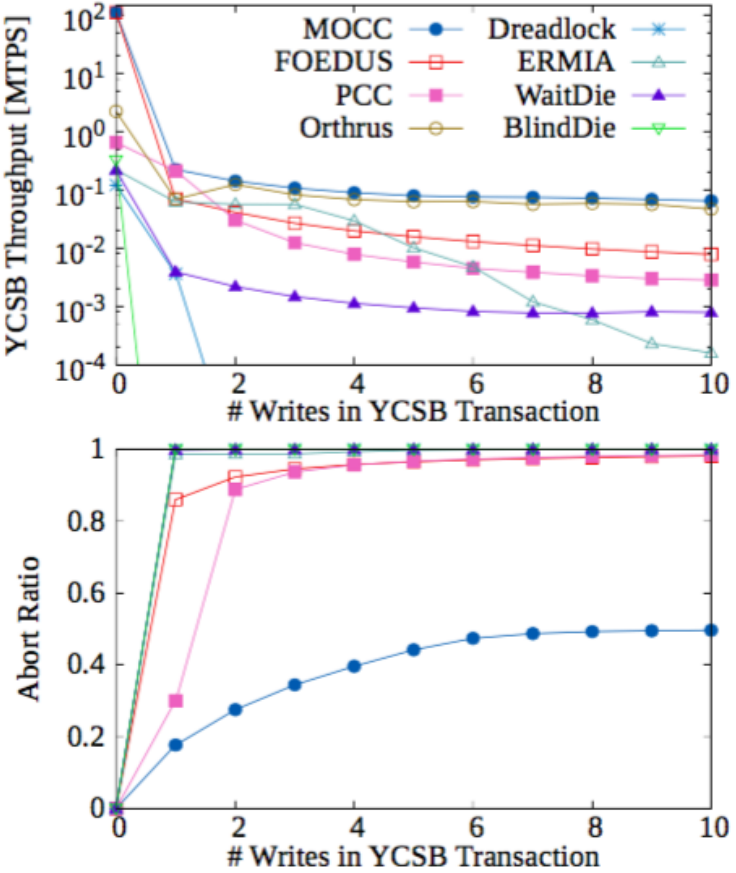

Figure 7: Read-write YCSB on GryphonHawk with high contention and/or high conflict. MOCC achieves dramatically lower abort ratio than OCC and even PCC, while maintaining robust and the highest throughput for all cases.

Multi-table, shifting workloads

Two table experiment: First table contains one record, second table contains one million

Shifting workload: Dynamically switches the nature of the small table every 0.1 second.

Throughput and abort ratio of MOCC over time with different temperature thresholds

H=0, significantly (24x) slower due to read locks like pessimistic schemes

Lower thresholds result in quicker learning and abort ratio drops quickly while large thresholds are unstable.

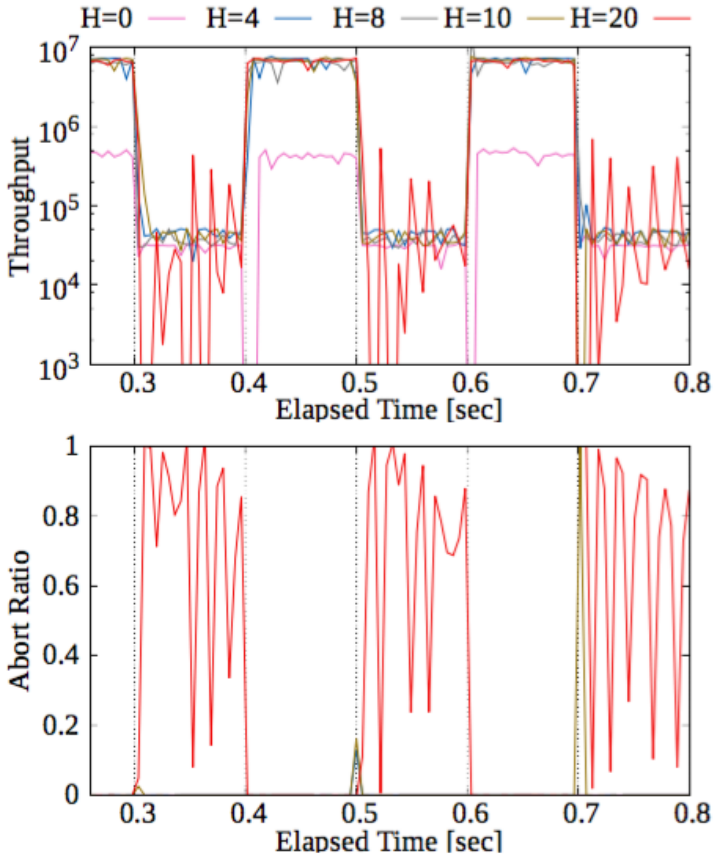

Figure 8: Throughput (top) and abort ratio (bottom) of the multi-table, shifting workload on GryphonHawk. The workload changes its access pattern on a contended table every 0.1 second. Larger thresholds take longer to learn new hot spots.

Long Scan Workloads

Every transaction reads one record in the small table, and scans 1000 record in the larger table, then updates the record in the small table.

MOCC performs an order of magnitude better than all others because of its temperature statistics

Table 4: Long Scan Workload on GryphonHawk.

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

Conclusion

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

Thank You!