Opportunities for optimism in contended main-memory multicore transactions

VLDB manuscript No.
(will be inserted by the editor)

Opportunities for Optimism in Contended Main-Memory Multicore Transactions

Yihe Huang[1] · **William Qian**[1] · **Eddie Kohler**[1] · **Barbara Liskov**[2] · **Liuba Shrira**[3]

Received: date / Accepted: date

Abstract Main-memory multicore transactional systems have achieved excellent performance using single-version optimistic concurrency control (OCC), especially on uncontended workloads. Nevertheless, systems based on other concurrency control protocols, such as hybrid OCC/locking and variations on multiversion concurrency control (MVCC), are reported to outperform the best OCC systems, especially with increasing contention. This paper shows that implementation choices unrelated to concurrency control can explain some of these performance differences. Our evaluation shows the strengths and weaknesses of OCC, MVCC, and TicToc concurrency control under varying workloads and contention levels, and the importance of several implementation choices called *basis factors*. Given sensible basis factor choices, OCC performance does not collapse on high-contention TPC-C. We also present two optimization techniques, *deferred updates* and *timestamp splitting*, that can dramatically improve the high-contention performance of both OCC and MVCC. These techniques are known, but we apply them in a new context and highlight their potency: when combined, they lead to performance gains of 4.74× for MVCC and 5.01× for OCC in a TPC-C workload.

Yihe Huang
E-mail: yihehuang@g.harvard.edu

William Qian
E-mail: wqian@g.harvard.edu

Eddie Kohler
E-mail: kohler@seas.harvard.edu

Barbara Liskov
E-mail: liskov@piano.csail.mit.edu

Liuba Shrira
E-mail: liuba@brandeis.edu

[1]Harvard University, Cambridge, MA
[2]MIT, Cambridge, MA
[3]Brandeis University, Waltham, MA

1 Introduction

The performance of multicore main-memory transactional systems remains a subject of intense study [12,21,24,32,37,38,52–54,60]. Techniques based on optimistic concurrency control perform extremely well on low-contention workloads due to their efficient use of shared memory bandwidth and avoidance of unnecessary memory writes. On high-contention workloads, however, OCC abort rates rise, and in the worst case classes of transaction can experience *contention collapse*: repeated conflicts prevent the transactions from ever committing.

Designs targeted at high-contention workloads, including partially-pessimistic concurrency control [53], dynamic transaction reordering [60], and variants of multiversion concurrency control (MVCC) [25,32], adapt their core transactional concurrency control protocols to better support high-contention transactions. These designs are reported to show dramatic benefits over OCC on high-contention workloads, including TPC-C, and some report benefits over OCC even at low contention [32]. But many of these evaluations compare different code bases, meaning other implementation differences can influence the results.

Our analysis of main-memory transactional systems including Silo [52], DBx1000 [59], Cicada [32], ERMIA [25], and MOCC [53] shows that engineering choices separate from concurrency control protocol have dramatically affected these systems' performance under high contention. For instance, some systems' abort mechanisms exacerbate contention by obtaining a hidden lock in the language runtime. We call these engineering choices *basis factors*. To better isolate the impact of concurrency control (CC) on performance,

we implement and evaluate three CC mechanisms – OCC, TicToc [60], and MVCC – in a new system, *STOv2*, that makes consistent and reasonable implementation choices for all identified basis factors. We show results up to 64 cores and for several benchmarks, including low- and high-contention TPC-C, YCSB, and benchmarks based on Wikipedia and RUBiS. STOv2 OCC performance does not collapse on these benchmarks, even at high contention, and OCC and TicToc significantly outperform MVCC at low and medium contention. This contrasts with prior evaluations, which reported OCC collapsing at high contention [14] and MVCC performing as well as OCC at all contention levels [32].

With basis factors controlled, changes to core concurrency control protocol show limited benefits on high-contention workloads and nontrivial costs on low-contention workloads. We therefore introduce, implement, and evaluate two optimization techniques that safely eliminate classes of conflict common in our high-contention workloads. These techniques improve high-contention performance for all of OCC, TicToc, and MVCC, and by much larger margins than protocol choice alone. The *deferred update* technique eliminates conflicts that arise when read-modify-write operations, such as increments, are implemented using plain reads and writes, and the *timestamp splitting* technique avoids conflicts between transactions that read some parts of a record and transactions that write other parts of the same record. These techniques have little performance impact on low-contention workloads. Though they have workload-specific parameters, they are conceptually general and applied to every workload we investigated. The techniques are widely known, but our variants are new, and we are the first to report their application to TicToc and MVCC.

The rest of the paper is organized as follows. After describing related work (§2), we present our OCC, TicToc, and MVCC implementations (§3) and our experimental setup (§4). We identify the basis factors we discovered and characterize their effects on performance (§5), then compare the performance of the OCC, TicToc, and MVCC concurrency control protocols, with basis factors fixed, on a range of low- and high-contention benchmarks (§6). We then describe the deferred update and timestamp splitting techniques (§7) and evaluate their performance (§8). Finally we conclude (§9).

2 Related work

2.1 Optimistic concurrency control protocols

Concurrency control is a central concern for database research, with work going back many decades [17]. The best choice of concurrency control algorithm can depend on workload, and OCC has long been understood to work best for workloads "where transaction conflict is highly unlikely" [28]. Optimistic approaches can experience starvation of whole classes of transactions (see Figure 9 in §6.1). Locking approaches, such as two-phase locking (2PL), never abort transactions due to conflicts, but the shared-memory writes and stalls required to lock records can cause performance degradation close to starvation at even low conflict rates [38]. Since performance tradeoffs between OCC and locking depend on technology characteristics as well as workload characteristics, and multicore main-memory systems have high penalties for memory contention, OCC can perform surprisingly well even for relatively high-conflict workloads and long-running transactions. This work was motivated by a desire to better understand the limitations of OCC execution, especially on high-conflict workloads.

The main-memory Silo database [52, 61] introduced an OCC protocol that, unlike other implementations [7,28], had no per-transaction contention point, such as a shared timestamp counter. Though Silo addressed some starvation issues by introducing snapshots for read-only transactions, and performed reasonably on some high-contention workloads, subsequent work has reported that Silo still experiences performance collapse on other high-contention workloads. These discrepancies are due to its basis factor implementations, as discussed in §5.

Several concurrency control techniques have aimed to preserve OCC's advantages at low contention and mitigate its flaws at high contention. We focus our efforts on two, TicToc [60] and optimistic MVCC [4, 42].

TicToc records have two timestamps. Write timestamps resemble OCC record timestamps, while read timestamps allow TicToc to commit some apparently-conflicting transactions by reordering them. Timestamp maintenance becomes more expensive than OCC, but reordering has benefits for high-contention workloads. We present results for our implementation of TicToc.

MVCC systems such as ERMIA [25] and Cicada [32] keep multiple versions of each record. These versions allow more transactions to commit through reordering, and in particular, read-only transactions can always commit. We present results for ERMIA and Cicada, as well as for our MSTO MVCC system, which adopts some (but not all) of Cicada's optimizations.

ERMIA uses a novel commit-time validation mechanism called the Serial Safety Net (SSN) to ensure strict transaction serializability. ERMIA transactions perform a check at commit time that is intended to be cheaper than OCC-style read set validations. The check is also less conservative, and can allow more transaction schedules to commit. The SSN mechanisms in ERMIA, however, involve expensive global thread registration and deregistration operations that limit its scalability [53]. In our experiments, ERMIA's locking overhead further swamps any improvements from its commit protocol.

Cicada contains optimizations that reduce overhead common to many MVCC systems, and in its measurements, its MVCC outperforms single-version alternatives on both low- and high-contention workloads. This disagrees with our results, where our OCC, OSTO, outperforms Cicada at low contention (Figure 10b). This difference is due to basis factor choices in Cicada's OCC comparison systems.

Optimistic MVCC still suffers from many of the same problems as single-version OCC. When executing read-write transactions with serializability guarantees, read-write and write-write conflicts still result in aborts. Optimizations such as deferred updates and timestamp splitting can alleviate these conflicts.

2.2 Other concurrency control research

Hybrid concurrency control in MOCC [53] and ACC [49] uses online conflict measurements and statistics to switch between OCC-like and locking protocols dynamically. We evaluate MOCC. Locking can be expensive (it handicaps MOCC in our evaluation), but prevents starvation.

Transaction batching and reordering [10] aims to discover more reordering opportunities by globally analyzing dependencies within small batches of transactions. It improves OLTP performance at high contention, but requires more extensive changes to the commit protocol to accommodate batching and intra-batch dependency analyses. Our workload-specific optimizations are orthogonal to these techniques; our optimizations can eliminate some dependency edges, while batching and reordering can work around others.

Static analysis can improve the performance of high-contention workloads. Given an entire workload, a system can discover equivalent alternative executions that generate many fewer conflicts. Transaction chopping [45] uses global static analysis of all possible transactions to break up long-running transactions, allowing subsequent pieces in a transaction to execute conflict-free. Systems like IC3 [54] combine static analysis with dynamic admission control to support more workloads. Static analysis techniques are complementary to our work; for instance, static analyses could help automate the application of deferred updates and timestamp splitting to a given workload.

2.3 Basis factors

Prior studies have measured the effects of some basis factors on database performance. One study found that a good memory allocator alone can improve analytical query processing performance by 2.7× [13]. A separate study presented a detailed evaluation of implementation and design choices in main-memory database systems, with a heavy focus on MVCC [58]; similar to our findings, the results acknowledge that CC is not the only contributing factor to performance, and lower-level factors like the memory allocator and index design (physical vs. logical pointers) can play a role in database performance. While we make similar claims in our work, we also describe more factors and expand the scope of our investigation beyond OLAP and MVCC.

Contention regulation [16] provides dynamic mechanisms, often orthogonal to concurrency control, that aim to avoid scheduling conflicting transactions together. Cicada includes a contention regulator. Despite being acknowledged as an important factor in the database research community, our work demonstrates instances in prior performance studies where contention regulation is left uncontrolled, leading to potentially misleading results.

A review of database performance studies in the 1980s [2] acknowledged conflicting performance results and attributed much of the discrepancy to the implicit assumptions made in different studies about how transactions behave in a system. These assumptions – which concerned, for example, how transactions restart and how system resources are handled – are analogous to our basis factors in that they do not concern the core CC algorithm, but significantly affect performance results. Our study highlights the significance of basis factors in the modern context, despite the evolution of database system architecture and hardware capabilities.

2.4 High-contention optimizations

Our deferred update and timestamp splitting optimizations have extensive precursors in other work. Timestamp splitting resembles row splitting, or vertical partitioning [39], which splits records based on workload characteristics in order to optimize I/O. Taken to an extreme, row splitting leads to column stores [29,48] or attribute-level locking [33]. Compared to these techniques, timestamp splitting has coarser granularity; this reduces fine-grained locking overhead, and suffices to reduce conflicts, but does not facilitate column-store-like compressed data storage.

Deferred updates obtain some of the same benefits as those obtained by commutative operators in other contexts, such as databases, file systems, and distributed systems [3, 27, 38, 43, 44, 57]. We know of no other work that applies commutativity or deferred updates to MVCC records, though many systems reason about the commutativity properties of modifications to MVCC indexes. Upserts in BetrFS [23, §2.2] resemble how we encode deferred updates; they are used to avoid expensive key-value lookups in lower-layer LSMs rather than for conflict reduction. Differential techniques used in column store databases [18] involve techniques and data structures that resemble deferred updates, though their goal

is to reduce I/O bandwidth usage in a read-mostly OLAP system.

2.5 Transactional memory

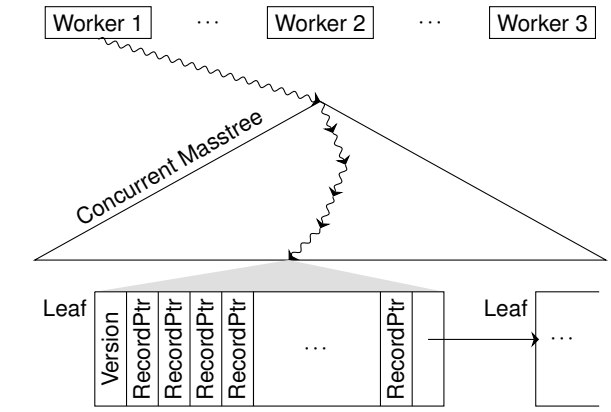
Transactional memory systems [20] provide a transaction abstraction that operates not on typed database records, but on words of primary memory. This simplifies the development of concurrent main-memory data structures, since transactions can be much easier to use and reason about than locks, compare-and-swap instructions, and other low-level synchronization primitives. The first transactional memories required hardware support, but pure software implementations are now common. These software transactional memories (STMs) use atomic instructions to build transaction abstractions that operate on general memory words [8, 46]. There are even multiversion STMs [5, 15].

Though a main-memory database transaction could correspond to a single transaction provided by a transactional memory, such an implementation would have poor performance [19, 21]. Transactional memories can perform well when transactions have a small memory footprint, as is typical for concurrent data structure operations; they struggle when transactions access hundreds or thousands of memory words, as is typical for main-memory database transactions. However, many of the basis factors we present have been explored in the context of software transactional memory. Some of our baseline choices were inspired by prior STM work [8], such as SwissTM's contention regulation [11].

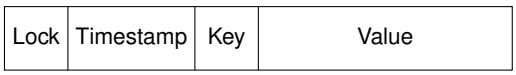
Recent STM systems bridge the gap between TMs and main-memory databases by implementing semantically-aware transactions on data structures, rather than on raw memory [21, 47]. We base our platform on our prior semantically-aware STM system, STOv1 [21], an OCC-only system that had good overall performance on some benchmarks and partial support for deferred updates and timestamp splitting.

STO has also been used as a baseline for other systems that address OCC's problems on high-contention workloads, such as DRP [37]. DRP effectively changes large portions of OCC transactions into deferred updates by using lazy evaluation, automatically implemented by C++ operator overloading. This moves most computation into OCC's commit phase, which works well at high contention, but imposes additional runtime overhead that our simpler implementation avoids.

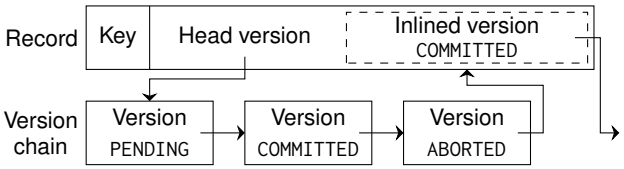
Several systems have achieved benefits by augmenting software CC mechanisms with hardware transactional memory [30, 55, 56]. HTM can also be used to implement efficient deadlock avoidance as an alternative to bounded spinning [55].



(a) *Retrieving a record by key.*



(b) *Record structure in OSTO.*



(c) *Record structure in MSTO. The record contains a pointer to the head of the version chain, which may include the inlined version.*

(d) *Version chain element in MSTO.*

Fig. 1: *STOv2 overview.*

2.6 Previous versions of this work

The measurements presented here include the results of some changes made since our original publication [22]. Our implementations of deferred updates (§7.1), timestamp splitting (§7.2), and garbage collection (§3.4) have been overhauled, reducing aborts and improving baseline performance. The high-contention optimizations also have much lower performance costs in low-contention scenarios. Several errors were corrected, including an error with our YCSB implementation that led to improper commits in certain configurations and an instance of false sharing on benchmark metadata in our RUBiS implementation.

3 STOv2 design

This work uses our STOv2 main-memory database. STOv2 is a redesign of STOv1 (Software Transactional Objects) [21], a type-aware software transactional memory with some support for database workloads. Compared with STOv1, STOv2 supports TicToc and MVCC as well as OCC, makes better

choices for basis factors (as explained in §5), and focuses solely on database workloads, where it offers better overall performance and scalability than STOv1.

We evaluate OSTO, MSTO, and TSTO, which are versions of STOv2 built around three concurrency control protocols. OSTO implements optimistic single-version concurrency control, MSTO implements optimistic multi-version concurrency control, and TSTO implements TicToc concurrency control. (TicToc [60] is an optimistic concurrency control protocol with characteristics of both single- and multi-version systems; each record has exactly two timestamps, and transaction ordering flexibility lies between that of OCC and MVCC.)

Figure 1 shows an overview of the STOv2 system and architecture. Primary and secondary indexes are implemented using hash tables and trees. Unordered indexes use hash tables to map keys to records; for ordered indexes, STO uses Masstree [34], a highly-concurrent B-tree variant that adopts some aspects of tries. All concurrency control mechanisms use identical trees and hash tables, differing in the structure of stored records.

Transactions are implemented as workload-specific C++ programs that access transactional data structures. Data structure code and the STOv2 core library work together to ensure transaction serializability. Transactions execute in "one-shot" style, meaning that all user inputs are available when transactions begin and transactions run without external communication until they commit. However, transactions build their read and write sets dynamically as they run (there is no need to pre-declare read or write sets). We do not support durability or networking, as they are not primary concerns of this work.

We now for completeness describe the designs of OSTO, MSTO, and TSTO. Many aspects of these designs are shared with other systems; important differences are highlighted. STOv2's read-copy-update-based garbage collection system (§3.4) is particularly important for its performance. Objects such as records and index nodes are enqueued for garbage collection when they become obsolete, but physically deleted only after all parallel transactions that could be accessing the corresponding memory have completed. The commit protocols and garbage collection system depend on the per-thread and global timestamps listed in Figure 2.

Name	Description
cts_g	Global commit timestamp counter; atomically incremented for each commit attempt
wts_{th}	Per-thread commit bound; must be $<$ the timestamps of all concurrent or future updates committed by transactions on this thread ($\max wts_{th} < cts_g$)
wts_g	Global commit bound; must be $<$ the timestamps of all concurrent or future updates committed by *any* transaction ($wts_g \leq \min wts_{th}$)
rts_{th}	Per-thread read bound; must be $\leq$ the timestamp used for all future observations made by transactions on this thread ($\max rts_{th} \leq wts_g$)
rts_g	Global garbage collection bound; must be $\leq$ the timestamp used for any future observation made by *any* transaction ($rts_g \leq \min rts_{th}$)

Fig. 2: *Timestamp variables in STOv2. Per-thread variables and cts_g are maintained by each worker thread that executes transactions, while wts_g and rts_g are maintained by a periodic maintenance thread. See §3.4 for details.*

3.1 OSTO

In OSTO, the STOv2 implementation of single-version optimistic concurrency control, each record contains one timestamp, that of the most recently committed transaction to modify that record. Transaction execution generates read and write sets in the usual way; a transaction's read set contains the timestamps of records it read, and its write set contains record modifications that will be installed upon commit. Core library code aborts the enclosing transaction if a conflict is detected during execution, for instance if a record is observed multiple times with different timestamps, or if a record remains locked for so long that the library suspects deadlock. If the transaction completes its execution successfully, it invokes the commit protocol. This core library code ensures serializability and exposes modifications to other transactions.

The commit protocol runs in three phases. In Phase 1, all records in the write set are locked. An abort occurs if deadlock is suspected. After Phase 1, the commit protocol selects the transaction's unique timestamp, marking the transaction's serialization point. This timestamp is selected by atomically incrementing a globally-accessible *commit timestamp* counter cts_g, ensuring that transactions always commit at distinct increasing timestamps. In Phase 2, the timestamps of records in the read set are validated. This checks that the versions are unchanged and that records are not locked by other transactions; abort occurs on validation failure. If Phase 2 succeeds, then the transaction will definitely commit. In Phase 3, new versions of the records in the write set are installed. The protocol also updates modified records' timestamps to the commit timestamp, and releases record locks.

When a record is inserted, OSTO adds it eagerly to the corresponding index while the transaction is running. This approach, which is shared with STOv1, has the important advantage that the commit protocol usually need not access indexes at all. The eagerly-added record is marked as DELETED, and concurrent transactions abort if they access such a key. If the transaction commits, OSTO updates the record to contain the written value and the commit timestamp, and removes the DELETED flag, allowing concurrent transactions

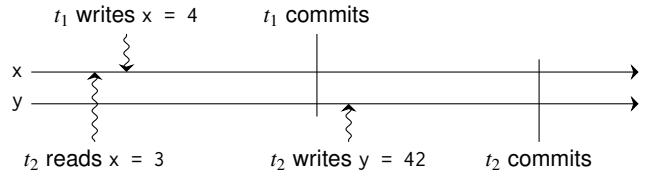


Fig. 3: *Although t_2 finishes later in time, it can still commit if placed earlier than t_1 in the serial order. OCC will abort t_2; MVCC and TicToc can commit it.*

to access the new value. If the transaction aborts, however, the record is removed from its index and garbage collected.

When a transaction that deletes a key commits, the record is assigned the transaction's timestamp and is marked as DELETED; after the commit protocol completes, the record is removed from its index and garbage collected. If the transaction aborts nothing needs to be done.

3.2 MSTO

MSTO is our multi-version optimistic concurrency control implementation. MSTO maintains multiple versions of each record so that transactions can access recent-past states as well as present states. Like any MVCC protocol, MSTO can run transactions declared as read-only in the recent past, allowing them to execute conflict-free and always commit. In addition it can commit read/write transactions in schedules that OCC and OSTO cannot, such as the one in Figure 3. However, these benefits come at the cost of increased memory usage for versions, increased cache pressure during transaction execution, and increased garbage collection overhead.

MSTO, like OSTO, uses indexes to map primary keys to records, but rather than storing data directly in records, it introduces a layer of indirection called a *version chain* (Figure 1c). A record comprises a key and a pointer to the head version in the chain. Each version carries a *write timestamp*, a *read timestamp*, and a *status*, as well as the record data and a chain pointer. The write timestamp, like an OSTO record's timestamp, is the timestamp of the transaction that created the version. The chain is sorted by write timestamp: a committed chain $v_n, \ldots, v_1$, where v_n is the most recent version, will have $v_n.wts > v_{n-1}.wts > \cdots > v_1.wts$. In addition, each version v has $v.rts \geq v.wts$, and $v.rts$ will be no smaller than the timestamp of the latest committed transaction that observed the version. A version's status is COMMITTED if the transaction that added the version committed, ABORTED if it aborted, and PENDING if the transaction is trying to commit. The version chain for a freshly-inserted record contains a single COMMITTED version that is marked as DELETED. Unlike in OSTO, transactions observing a deleted version need not abort (a deleted version is treated like an absent record).

Each MSTO transaction executes at a timestamp ts_{tx} that determines which versions it observes. When the transaction reads a record, MSTO scans the record's chain and uses the version visible at ts_{tx}. This is the most recent non-aborted version v_i that satisfies $v_i.wts \leq ts_{tx}$. If that version has PENDING status, the transaction waits for its status to resolve to either COMMITTED or ABORTED.

For transactions declared to be read-only, the transaction timestamp ts_{tx} is set less than or equal to wts_g (see Figure 2). This timestamp is by construction less than the commit timestamp of all concurrent or future transactions, so all read-/write transactions at or prior to that timestamp have already committed or aborted, and reading at this timestamp will always succeed without conflicts. Declared read-only transactions are therefore certain to commit, and there is no need to maintain a read set or to update observed versions' $v.rts$ fields.

Transactions not declared to be read-only execute as follows. Transaction read sets contain versions of records, not the records themselves. The transaction timestamp ts_{tx} is chosen the first time a record is accessed by atomically incrementing cts_g; the transaction observes versions at that timestamp and will commit its modifications at that timestamp. The transaction is guaranteed to serialize after any transactions that completed before this first access.

The commit protocol again runs in three phases; Algorithm 1 describes it in detail. Phase 1 processes the write set, inserting for each item a new *write version* at the correct location in the corresponding record's version chain. (This is often at the head of the version chain – the write version most likely has the largest timestamp of any record version – but not always.) The write version wv has $wv.status =$ PENDING, indicating that the relevant transaction is trying to commit, and read and write timestamps both set to ts_{tx}. The insertion protocol uses lock-free atomic operations to ensure the version chain stays in correct order. It also checks for an irreconcilable conflict caused by an *anti-dependency*: a concurrent transaction that observed a value this transaction would overwrite, but that will commit after this transaction in the serial order. It finds these conflicts by examining the read timestamps of the version vp that was committed most recently before ts_{tx}. If this version has $vp.rts > ts_{tx}$, tx must abort.

In Phase 2, MSTO checks the read set. For each observed version v, MSTO first updates the version's read timestamp $v.rts$ to $\max\{v.rts, ts_{tx}\}$ using atomic operations. Then MSTO scans the record's chain for the version visible at ts_{tx}. If that version differs from the version in the read set, a concurrent transaction overwrote the value this transaction observed, and this transaction must abort. The atomic update of the read timestamp in Phase 2 synchronizes with the atomic

insertion of the write version in Phase 1 to ensure that anti-dependencies are detected. Every thread will observe these atomic operations in the same order. If the *rts* update happens first, a concurrent transaction running Phase 1 will detect that update and abort; if the insertion happens first, a concurrent transaction running Phase 2 will wait for the new version to resolve and abort if necessary.

Finally, in Phase 3, all PENDING versions are changed to COMMITTED or ABORTED as appropriate, and redundant parts of version chains are cleaned for garbage collection (§3.4).

Algorithm 1 MSTO commit protocol.

Every version chain is created with an initial version xv representing the absence of a value. It has $xv.wts = xv.rts = 0$, $xv.status =$ COMMITTED, and is marked as DELETED.

```
TRY-INSERT-WRITE-VERSION(wv)
 1  vptr = &wv.record.head
 2  forever:
 3      v = *vptr
 4      if v.wts > wv.wts:
 5          // move down version chain
 6          vptr = &v.prev
 7      elseif v.rts > wv.wts and v.status ≠ ABORTED:
 8          return FALSE
 9      else
10          // insert wv before v; retry on concurrent modification
11          wv.prev = v
12          if COMPARE-EXCHANGE(*vptr, v, wv) succeeds:
13              return TRUE

MSTO-PHASE1(tx)
 1  for each wv ∈ tx.writeset:
 2      wv.wts = wv.rts = tx.ts
 3      wv.status = PENDING
 4      if ¬TRY-INSERT-WRITE-VERSION(wv):
 5          ABORT(tx); return
 6      // Now wv is in the version chain at the proper location.
 7      // Check for concurrent read
 8      vp = wv.prev
 9      while vp.status ≠ COMMITTED:
10          vp = vp.prev
11      if vp.rts > wv.wts:
12          ABORT(tx); return

MSTO-PHASE2(tx)
1  for each rv ∈ tx.readset:
2      rv.rts = atomic max{rv.rts, tx.ts}
3      v = rv.record.head
4      while v ≠ rv:
5          if v.wts < rv.rts and v.status ≠ ABORTED:
6              ABORT(tx); return
7          v = v.prev
```

A transaction that deletes a record adds a DELETED version to the chain. The record cannot be immediately removed from its index because it may still be needed by other transactions, including read-only transactions that run in the past; furthermore, if the record were reinserted, we might need to access versions both before and after the delete. Long-deleted records are garbage collected from indexes using a cleaning procedure (§3.4).

Our MSTO implementation was influenced by Cicada, but has important differences. Our commit protocol does not limit new versions to being added only at the top of the chain; it is highly concurrent and avoids the use of locks. Another important difference is the way we manage record inserts and deletes: we continue to use the same chain if a key is deleted and then re-inserted, and we do not use version chains for tree nodes. We do take advantage of Cicada's *inlined versions* optimization. One version can be stored inline with the record, which reduces memory indirections, and therefore cache pressure, for values that change infrequently. MSTO fills the inline version slot when it is empty or has been garbage collected.

3.3 TSTO

TSTO is an OSTO variant that uses TicToc [60] in place of plain OCC as the CC mechanism. TicToc, like MVCC, uses separate read and write timestamps for each record, but maintains only the most recent version. It computes transactions' commit timestamps based on read and write set information, allowing for more flexible transaction schedules than simple OCC at the cost of more complex timestamp management. Except for concurrency control, TSTO and OSTO share identical infrastructure. We use full 64-bit words for *wts* and *rts* rather than TicToc's delta-*rts* encoding [60, §3.6]; in our benchmarks the delta-*rts* encoding caused many false aborts and worse performance, especially in read-heavy workloads.

3.4 Garbage collection

Memory deallocation in STOv2 is managed using read-copy-update (RCU) techniques [35]. This lets transactions access records safely after their logical deletion and largely eliminates readers-writer locks. RCU requires a mechanism for determining when logically deleted objects are safe to free, so STOv2 maintains a set of thread-local variables and several global variables that are used by the concurrency control implementation both to run transactions and to inform STOv2 when records or tree nodes are safe to delete. Figure 2 lists these variables.

The commit bounds wts_{th} and wts_g define timestamps below which no concurrent or future read/write transaction will commit. Each worker thread (that is, each thread that can execute transactions) maintains its wts_{th} variable by periodically setting $wts_{th} = cts_g$, and every millisecond a main-

tenance thread computes wts_g by taking the minimum of all threads' wts_{th} values. The read bounds rts_{th} and rts_g define timestamps below which no concurrent or future transaction will read. Each worker thread maintains its rts_{th} by periodically setting $rts_{th} = wts_g$, and the maintenance thread maintains rts_g by periodically setting $rts_g = \min rts_{th}$.

These variables let OSTO and TSTO safely free records and versions once their contents become inaccessible to concurrent transactions. Say that a transaction with commit timestamp ts_{tx} makes an object redundant (for instance, by deleting a record from a shared data structure). The object cannot be deleted right away, since concurrent transactions might be accessing it or have stored it in their read sets. However, it is safe to delete the object once all concurrent and future transactions are guaranteed to access the object – that is, when $rts_g > ts_{tx}$. Thus, the thread running the transaction enqueues the object on a thread-local garbage collection list, marking it with a *freeing timestamp fts* that equals the transaction's commit timestamp ts_{tx}. The object is garbage collected later, when the global garbage collection bound rts_g is $> fts$.[1]

Garbage collection of redundant versions in MSTO version chains takes two rounds because MSTO can commit versions out of order. When MSTO commits a new version v, it enqueues that version for *cleaning* with freeing timestamp *fts* equal to the transaction timestamp ts_{tx}. The cleaning procedure runs when $wts_g > fts$. At this point the chain of versions older than v has become fixed (whereas at v's commit time, concurrent transactions might be modifying the chain). The cleaning procedure traverses the chain starting *just below* v and enqueues those versions, up to and including the previous committed version, for second-round garbage collection. The versions are logged with *fts* and will be garbage collected as described above. This two-round structure allows segments of the version chain to be cleaned independently: every committed version's first-round cleaning will run before that committed version is freed in the second round. Another two-round procedure is used to safely remove deleted records from data structures, ensuring that trees and hash tables do not grow without bound.

4 Experimental setup

We conducted our experiments on Amazon EC2 m4.16xlarge dedicated instances, each powered by two Intel Xeon E5-2686 v4 CPUs (16 cores/32 threads each, 32 cores/64 threads per machine) with 256GiB of RAM. For our experiments, we enable support for 200GiB in 2MiB hugepages. The operating system is an Ubuntu 18.04 image on the Linux 4.18.0-25-generic kernel with all security mitigations enabled. (This performed better than kernel version 5.4.)

Medians of 5 runs (or 10 for RUBiS experiments) are reported with mins and maxes shown as error bars. Some results show very little variation so error bars are not always visible. In all experiments, aborted transactions are automatically retried on the same worker thread until they commit.

4.1 Workloads

We measure two standard benchmarks, YCSB (A, B, and C) [6] and TPC-C [50] (high and low contention settings). We also measure two additional high-contention workloads modeled after Wikipedia and RUBiS.

The TPC-C benchmark models an inventory management workload. We implement the full mix and report the total number of transactions committed per second across all transaction types, including 45% new-order transactions. As required by the TPC-C specification, we implement a queue per warehouse for delivery transactions, and assign one thread per warehouse to preferentially execute from this queue. ("[T]he Delivery transaction must be executed in deferred mode ... by queuing the transaction for deferred execution" [51, §2.7].) Delivery transactions for the same warehouse always conflict, so there is no point in trying to execute them in parallel on different cores. TPC-C contention is controlled by varying the number of warehouses. With one warehouse per core, contention is relatively rare (cross-warehouse transactions still introduce some conflicts); when many cores access one warehouse, many transactions conflict. We enable Silo's fast order-ID optimization [52], which reduces unnecessary conflicts between new-order transactions. We implement contention-aware range indexes (§5.5) and use hash tables to implement indexes that are never range-scanned. On MVCC systems (MSTO and Cicada), we run read-only TPC-C transactions slightly in the past, allowing them to commit with no conflict every time.

YCSB models key-value store workloads. YCSB-A is update-heavy, YCSB-B is read-heavy, and YCSB-C is read-only. YCSB contention is controlled by a skew parameter. We set this relatively high, resulting in high contention on YCSB-A and moderate contention on YCSB-B (the benchmark is read-heavy, so most shared accesses do not cause conflicts). We use a uniform distribution for YCSB-C. All YCSB indexes use hash tables.

Our Wikipedia workload is modeled after OLTP-bench [9]. By default, the workload mix is around 5% writes and 95% reads. Among the writes, almost all transactions are `UpdatePage`, which selects a random page from among a skewed distribution, creating high contention on a few key pages.

Our RUBiS workload is the core bidding component of the RUBiS benchmark [40], which models an online auction site. The workload mix is 50% reads and 50% writes.

[1] The distinction between commit and read bounds matters only for MSTO, which executes declared read-only transactions at timestamp wts_g. OSTO and TSTO never execute transactions in the past and could alternatively free objects sooner, when $wts_g > fts$.

60% of writes are `PlaceBid` and 40% are `BuyNow`, with both transactions being somewhat similar. Both transactions contend heavily on updating the `Items` table, and each contends with other transactions of the same type on appending the bid or buy to the corresponding table. Whenever necessary, indexes use Masstree to support range queries.

We also evaluate other implementations' TPC-C benchmarks, specifically Cicada, MOCC, and ERMIA. All systems use Silo's fast order-ID optimization (we enabled it when present and implemented it when not present). We modified Cicada to support delivery queuing, but did not modify MOCC or ERMIA.

5 Basis factors

Main-memory transaction processing systems differ in concurrency control, but also differ in implementation choices such as memory allocation, index types, and backoff strategy. In years of running experiments on such systems, we developed a list of *basis factors* where different choices can have significant impact on performance. This section describes the most impactful basis factors. For instance, OCC's contention collapse on TPC-C can stem not from inherent limitations, but from basis factor choices. For each factor we suggest a specific choice that performs well, and conduct experiments using both high- and low-contention TPC-C to show the effects of the choice on performance. We end the section by describing how other systems implement the factors, calling out important divergences.

Figure 4 shows an overview of our results for OSTO, TSTO, and MSTO. The light-blue lines represent the baselines in which all basis factors are implemented according to our guidelines. In every other line, a single factor's implementation is replaced with a different choice taken from previous work. The impact of the factors varies, but on high-contention TPC-C with OSTO, four factors have 20% or more impact on performance, and two factors can cause collapse. In TSTO and MSTO, the basis factors have similar impact, except that a slow allocator has even more impact on MSTO because multi-versioning causes more allocation and deallocation.

5.1 Contention regulation

Contention regulation avoids repeated cache line invalidations by delaying retry after a transaction experiences a conflict. Over-eager retry can exacerbate contention, while over-delayed retry can leave cores idle. We recommend *randomized exponential backoff* as a baseline for contention regulation. This is not optimal at all contention levels – under medium contention, it can cause some idleness – but as with spinlock implementations [36] and network congestion [1], exponential backoff balances quick retry at low contention with low invalidation overhead at high contention.

The "No contention regulation" lines in Figure 4 show OSTO, TSTO, and MSTO performances with no backoff. Lack of contention regulation can severely impact performance as contention rises. (We have observed high performance variability and even complete performance collapse in other benchmarks, though not in Figure 4.) Silo supports exponential backoff through configuration, though unfortunately it does not enable backoff by default [52]. Some comparisons using Silo have explicitly disabled backoff, citing mild effects at medium contention [31]. This provides a misleading picture of OCC's inherent performance in evaluations including high-contention experiments.

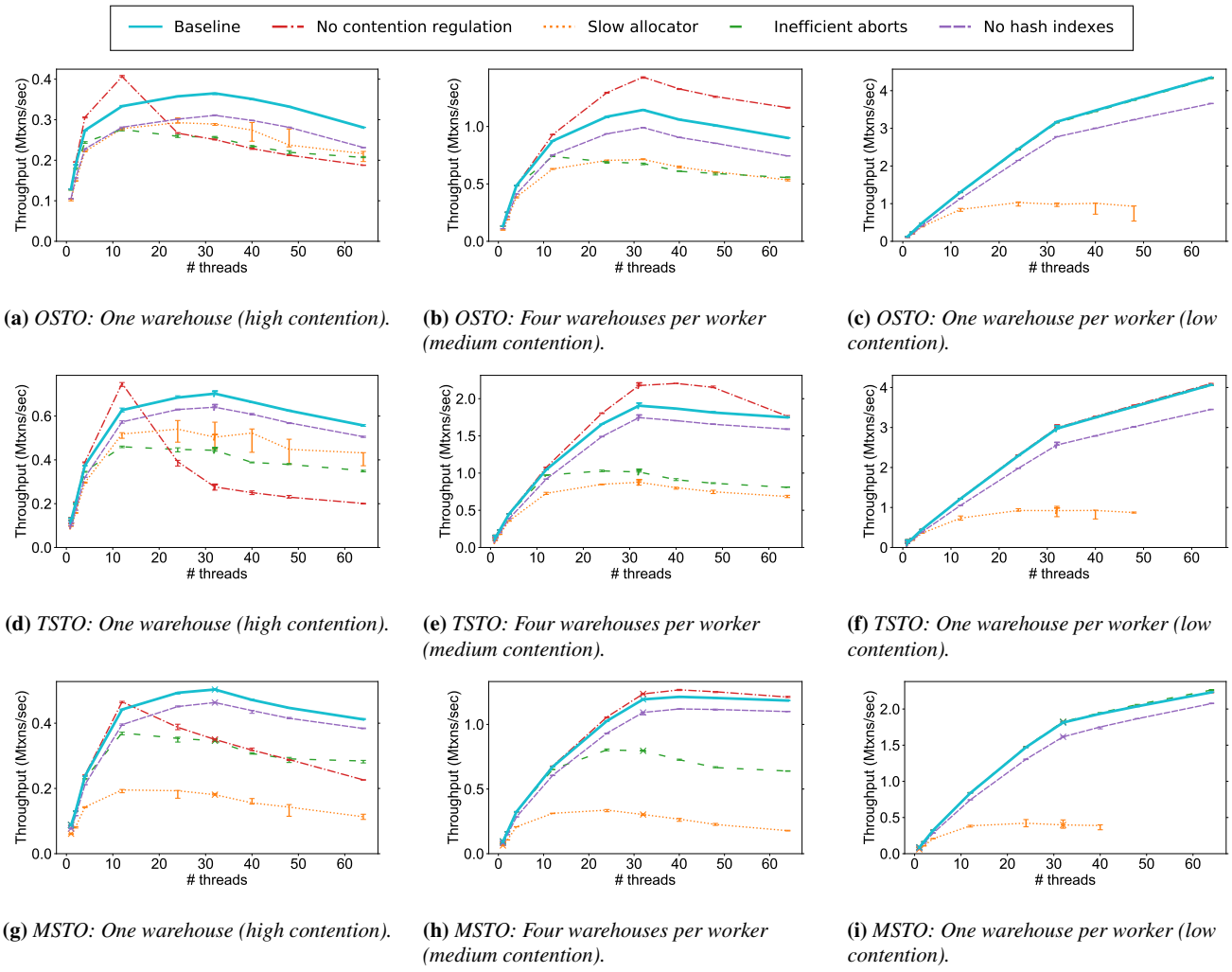
5.2 Memory allocation

Transactional systems stress *memory allocation* by allocating and freeing many records and index structures. This is particularly true for MVCC-based systems, where every update allocates memory so as to preserve old versions. Memory allocators can impose hidden additional contention (on memory pools) as well as other overheads, such as TLB pressure and memory being returned prematurely to the operating system. We recommend using a *fast general-purpose scalable memory allocator* as a baseline, and have had good experience with rpmalloc [41]. (The popular jemalloc and tcmalloc allocators are not nearly as fast.) A special-purpose allocator could potentially perform even better, and several systems implement their own allocators, but efficient general-purpose allocators are now available, scalable allocators are complex, and we found bugs in Cicada's allocator that hobbled performance at high core counts (§6.3). Some systems, such as DBx1000, reduce allocator overhead to zero by pre-allocating all record and index memory before experiments begin. We believe this form of preallocation changes system dynamics significantly – for instance, preallocated indexes never change size – and should be avoided.

The "Slow allocator" lines in Figure 4 show performance using the default glibc memory allocator; jemalloc outperforms glibc, but not by much. The glibc allocator is Silo's default choice [52]. OSTO and TSTO with rpmalloc perform 1.6× better at high contention, whereas MSTO with rpmalloc performs about 2.5× better at high contention. On low contention benchmarks, the glibc allocator hamstrings performance on all three protocols.

5.3 Abort mechanism

High-contention workloads stress the *abort mechanism* in transaction systems. High abort rates do not necessarily correspond to lower throughput on modern systems, and reduc-



(a) *OSTO: One warehouse (high contention).*

(b) *OSTO: Four warehouses per worker (medium contention).*

(c) *OSTO: One warehouse per worker (low contention).*

(d) *TSTO: One warehouse (high contention).*

(e) *TSTO: Four warehouses per worker (medium contention).*

(f) *TSTO: One warehouse per worker (low contention).*

(g) *MSTO: One warehouse (high contention).*

(h) *MSTO: Four warehouses per worker (medium contention).*

(i) *MSTO: One warehouse per worker (low contention).*

Fig. 4: *OSTO, TSTO, MSTO throughput under full-mix TPC-C showing impact of basis factors. The thick blue line represents our baseline implementation with reasonable basis factor choices; the other lines show the impact of changing one base factor choice (contention regulation, allocator, abort mechanism, or index type) relative to that baseline.*

ing abort rates does not always improve performance [32]; 50% abort rates are not necessarily bad for overall performance on very fast systems. However, some abort mechanisms impose surprisingly high hidden overheads. C++ exceptions – a tempting abort mechanism for programmability reasons – can acquire a global lock in the language runtime to protect exception-handling data structures from concurrent modification by the dynamic linker. This lock then causes all aborted transactions to contend! We recommend implementing aborts using *checked return values* instead.

The "Inefficient aborts" lines in Figure 4a, Figure 4d, and Figure 4g show performance using C++ exceptions for aborts. STOv1, Silo, and ERMIA abort using exceptions. Fast abort support offers 1.2–1.5× higher throughput at high contention for all CCs.

5.4 Index types

Transaction systems support different *index types* for table indexes. Silo, for instance, uses Masstree [34], a B-tree-like structure, for all indexes. Other systems can choose different structures based on transaction requirements. Most TPC-C implementations we have examined use hash tables for indexes unused in range queries; some implementations use hash tables for *all* indexes and implement complex workarounds for range queries [59]. Hash tables offer $O(1)$ access time where B-trees offer $O(\log N)$, and a hash table can perform 2.5× or more operations per second than a B-tree even for a relatively easy workload. We recommend using *hash tables* when the workload allows it, and B-tree-like indexes elsewhere.

The "No hash index" lines in Figure 4 show performance when all indexes use Masstree, whether or not range scans

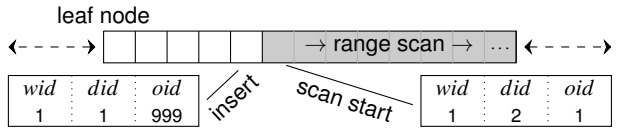


Fig. 5: *Illustration of index contention on the TPC-C* NEW ORDER *table. An insert to one district in a new-order transaction has a physical conflict with a range scan in a delivery transaction on the adjacent district, though these transactions do not logically conflict.*

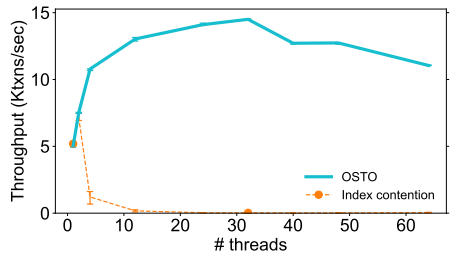


Fig. 6: *Throughput of delivery transactions with and without contention-aware indexes. OSTO, full-mix TPC-C, one warehouse (high contention).*

are required. Silo and ERMIA lack hash table support. Hash indexes offer at most 1.2× higher throughput in this experiment; this is less than 2.5× because data structure lookups are not the dominant factor in TPC-C transaction execution.

5.5 Contention-aware indexes

Some basis factors do not greatly affect overall TPC-C performance, but hugely impact the performance of some classes of transaction. For instance, the NEW ORDER table in the TPC-C benchmark is keyed by $\langle wid, did, oid \rangle$, a combination of warehouse ID, district ID, and order ID. Each new-order transaction inserts records at the end of a $\langle wid, did \rangle$ range, while each delivery transaction scans a $\langle wid, did \rangle$ range from its beginning. Ideally, new-order and delivery transactions would conflict only if they used the same district (the same $\langle wid, did \rangle$ pair). However, if a district boundary falls *within* a B-tree node, phantom protection can cause new-order and delivery for *adjacent* districts to appear to conflict, inducing aborts (see Figure 5).

A *contention-aware index* is an index that avoids contention between important disjoint ranges. We recommend implementing contention-aware indexing, either automatically or by taking advantage of static workload properties. Our baselines implement contention-aware indexing via a side effect of Masstree's trie-like structure [34, §4.1]: certain key ranges in Masstree completely avoid phantom-protection conflicts. If, for example, a $\langle wid, did \rangle$ pair is represented using an exact multiple of 8 bytes, then scans on one such range will never conflict with inserts into any other range. To implement contention-aware indexing, we reserve eight bytes for each key component in a multi-key index. This maps each key component to distinct layers of B-trees, avoiding false index contention at the cost of larger key size (24 bytes instead of 8 bytes). We observe negligible performance overhead under low contention due to this increase in key size.

Figure 6 shows the impact of contention-aware indexes on delivery transactions in OSTO full-mix TPC-C. Without contention-aware indexes (the "Index contention" line in the figure), delivery transactions almost completely starve at high contention. This starvation is similar to the OCC performance collapse under high contention reported in prior work [32]. When executing delivery transactions in deferred mode, as required by the TPC-C specification, this starvation of delivery transactions may not actually lead to a collapse in overall transaction throughput, because other transactions can still proceed as normal while delivery transactions are being starved in the background.

5.6 Deadlock avoidance

Every system that can hold more than one lock at a time must include a *deadlock avoidance or detection strategy*. Early OCC database implementations avoided deadlock by sorting their write sets into a globally consistent order [26, 52, 60]. Fast sorts are available; for instance, the memory addresses of records and nodes are satisfactory sort keys. Transactional memory systems have long relied instead on bounded spinning, where a transaction that waits too long to acquire a lock assumes it's deadlocked, aborts, and tries again. Bounded spinning can have false positives – it can detect deadlock where there is none – but it has low overhead, and when two OCC transactions try to lock the same record, the second transaction often benefits from aborting early. (The lock indicates upcoming changes to the underlying record, and if those changes happen the second transaction will typically abort anyway.) Our experience as well as prior study [55, §7.2] finds that write set sorting is expensive and we recommend *bounded spinning* for deadlock avoidance. Although write set sorting generally had relatively low impact ($\approx$ 10%) on TPC-C, DBx1000 OCC [60] prevents deadlock using an unusually expensive form of write set sorting. In that system, comparisons use records' primary keys rather than their addresses, which causes many additional cache misses, and the sort algorithm is $O(n^2)$ bubble sort; as a result, deadlock avoidance takes close to 30% of the total run time of DBx1000's "Silo" TPC-C under high contention.

System	Contention regulation	Memory allocation	Aborts	Index types	Transaction internals	Deadlock avoidance	Contention-aware index
Silo [52]	−−	−−	−−	−	−	+	+
STO [21]	−−	−−	−−	+	+	+	+
DBx1000 OCC [59]	+	N/A	+	+	−	−−	−−
DBx1000 TicToc [60]	+	N/A	+	+	−	+	−−
MOCC [53]	N/A	+	+	+	+	+	−−
ERMIA [25]	+	+	−−	−	+	+	+
Cicada [32]	+	+	+	+	+	N/A	N/A
STOv2 (this work)	+	+	+	+	+	+	+

Fig. 7: *Overview of basis factor impact for our work and seven comparison systems. On high-contention TPC-C at 64 cores, "+" choices have at least 0.9× STOv2's performance, while "−" choices have 0.7–0.9× and "−−" choices have less than 0.7×. For instance, Silo's default implementation of contention regulation (§5.1) dramatically reduces its performance on 64-core high-contention TPC-C.*

5.7 Transaction internals

Transaction internals refers to a transaction library's mechanisms for maintaining read sets and write sets. The best internals use fast hash tables that map logical record identifiers to their physical in-memory locations, and that can be cleared efficiently on transaction completion. We recommend strong transaction internals by default, but found to some surprise that the factors listed above have more performance impact. Replacing STOv2's highly-engineered internals with Cicada's somewhat simpler versions reduced performance by just 2%. (Using DBx1000's internals reduced performance by somewhat more.) Engineering effort spent on transaction internals seems to quickly reach a point of diminishing returns.

5.8 Global timestamps

Many concurrency control designs assign each read/write transaction a unique timestamp. STOv2 computes these timestamps by atomically incrementing a shared global counter. This introduces a point of contention that can be avoided: other systems, including Silo and Cicada, have engineered mechanisms that compute unique timestamps with little or no access to contended global state. However, we have observed little impact from this contention point except on very small transactions. For instance, YCSB's mandated record size, 1000 bytes, is large enough that data movement costs overshadow global timestamp contention in our experiments. Previous results that report significant benefits from local timestamp calculation use records that are an order of magnitude smaller [32].

5.9 Summary

Figure 7 summarizes our investigation of basis factors by listing each factor and qualitatively evaluating 8 systems, including STOv2, according to their implementations of these factors. We performed this evaluation through experiment and code analysis. Each system's choice is evaluated relative to STOv2's and characterized as either good ("+", achieving at least 0.9× STO's performance), poor ("−", 0.7–0.9×), or very poor ("−−", less than 0.7×).

6 Evaluation of concurrency control protocols

Armed with this thorough evaluation of effective basis factor choices, we now evaluate STOv2's three concurrency control mechanisms on our suite of benchmarks at different contention levels. Our goal is to isolate the performance impacts of concurrency control choice, rather than basis factor choice. Prior work showed OCC performance collapsing at high contention on TPC-C, but our findings show otherwise. None of OSTO, TSTO, and MSTO collapse on high-contention TPC-C; neither do they scale. OSTO's high-contention TPC-C throughput is approximately 0.6× that of MSTO, even at 64 threads, but at low contention, OSTO throughput is approximately 2× that of MSTO. For our other benchmarks, OCC sometimes performs on par with MVCC (Figure 8h) or even better than MVCC (Figure 8d and Figure 8g) at high contention.

6.1 Overview

Figure 8 shows the transaction throughput of all three STOv2 variants on all our benchmarks with thread counts varying from 1 to 64. The committed mix of transactions conforms to the TPC-C specification except in one-warehouse, high core count settings. (The warehouse delivery thread mandated by the specification cannot execute enough transactions to reach 4% of the mix when 63 other threads are performing transactions on the same warehouse; we observe a mix of 1.7% delivery transactions on some one-warehouse TPC-C experiments.)

Only low-contention benchmarks (TPC-C with one warehouse per worker, Figure 8c, and YCSB-B, Figure 8e) approach perfect scalability. (The change in slope at 32 threads is due to our machine having 2 hyperthreads per core.) On high-contention benchmarks, each mechanism scales up to 4 or 8 threads, then levels off. Performance declines at higher thread counts, but does not collapse.

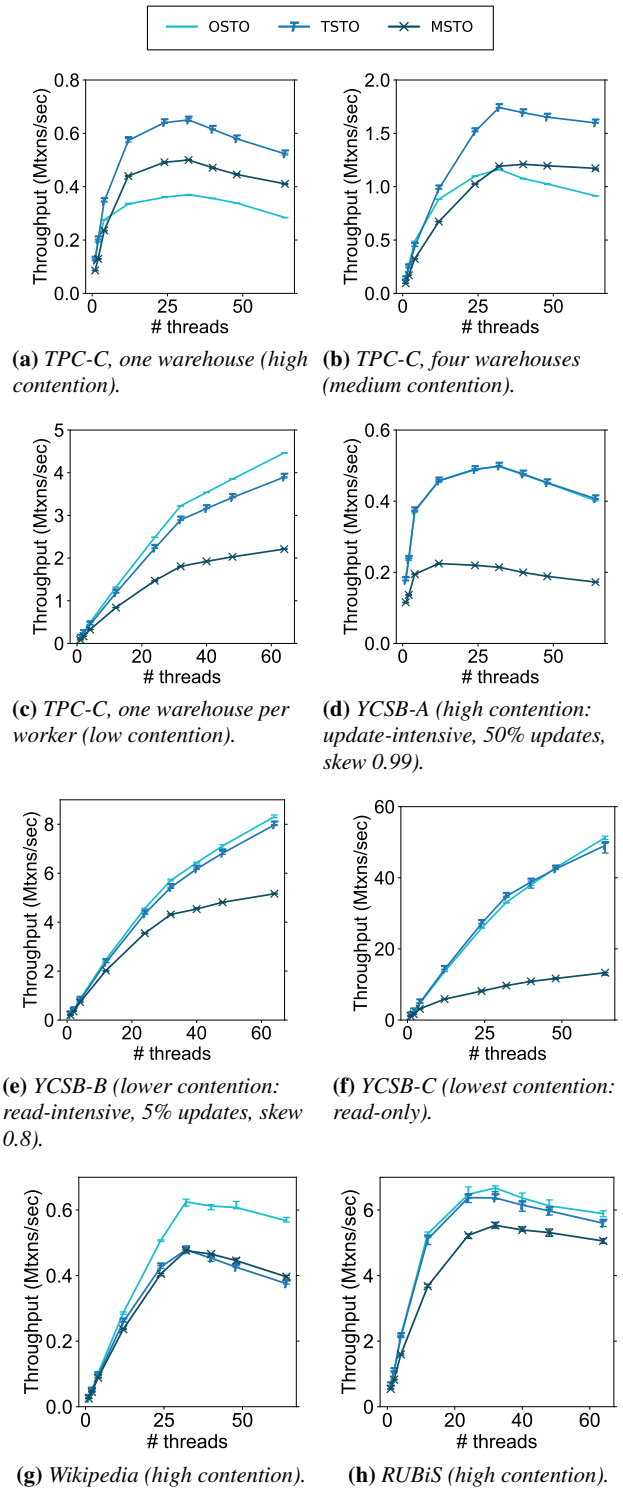
When scalability is good, performance differences can be attributed primarily to the inherent overhead of each mechanism. In Figure 8c, for example, TSTO's more complex timestamp management causes it to slightly underperform low-overhead OSTO, while MSTO's considerably more complex version chain limits its throughput to 0.52× that of OSTO.

Some of the high-contention benchmarks impose conflicts that affect all mechanisms equally. For example, YCSB-A has fewer than 0.1% read-only transactions and high key skew (many transactions touch the same keys). This prevents TicToc and MVCC from discovering safe commit orders, so OSTO, TSTO, and MSTO all scale similarly, and OSTO outperforms MSTO by 1.5–1.7× due to MSTO overhead (Figure 8d). On other benchmarks, the mechanisms scale differently. For example, in high-contention TPC-C (Figure 8a), OSTO levels off after 4 threads, while MSTO and TSTO scale well to 8 threads. This is due to OSTO observing more irreconcilable conflicts and aborting more transactions, allowing MSTO to overcome its higher overhead and outperform OSTO. At 12 threads with 1 warehouse, 47% of new-order/payment transactions that successfully commit in MSTO would have been aborted by an OCC-style timestamp validation.

In summary, we do not observe contention collapse, and our MVCC implementation has significant overhead over OCC at low contention and even some high-contention scenarios. All these results differ from previous reports.

Some differences from prior results are worth mentioning. Our YCSB-A results are lower than those reported previously [32]. This can be attributed to our use of the YCSB-mandated 1000-byte records; DBx1000 uses 100-byte records. Cicada's reported results for Silo and "Silo′" (DBx1000 Silo) show total or near performance collapse at high contention, but our OCC measurements show no such collapse. We attribute this difference to Silo's lack of contention regulation, inefficient aborts, and general lack of optimization, and to DBx1000's unnecessarily expensive deadlock avoidance and lack of contention-aware indexing.

In these real-world-inspired benchmarks, OCC's performance did not collapse; on some of the benchmarks, MVCC had similar scaling behavior as OCC. However, we do not claim that OCC will *never* collapse: there are workloads that can cause any optimistic concurrency control protocol to experience contention collapse, at least for some transaction classes in a workload. Figure 9 shows an example.



(a) *TPC-C, one warehouse (high contention).*
(b) *TPC-C, four warehouses (medium contention).*
(c) *TPC-C, one warehouse per worker (low contention).*
(d) *YCSB-A (high contention: update-intensive, 50% updates, skew 0.99).*
(e) *YCSB-B (lower contention: read-intensive, 5% updates, skew 0.8).*
(f) *YCSB-C (lowest contention: read-only).*
(g) *Wikipedia (high contention).*
(h) *RUBiS (high contention).*

Fig. 8: *STOv2 performance on several workloads.*

The workload is YCSB-like, with a 16-record database and two transaction classes: class-A transactions update all 16 records, and class-B transactions update one of the records and read the other 15. In our experiment, one thread executes class-A transactions, while all other threads execute

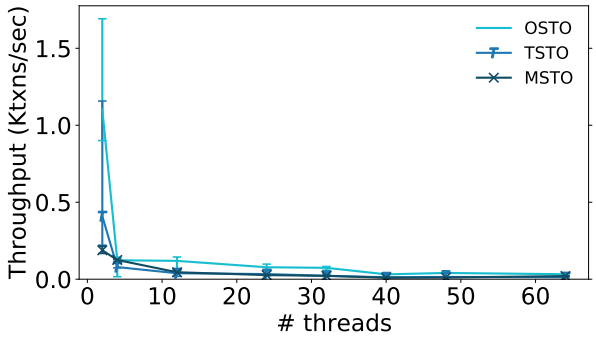


Fig. 9: *Contention collapse can affect OSTO, TSTO, and MSTO. In this workload, one thread runs "class-A" transactions that update all records in a small database, while the other threads run transactions that read the database and update one record each. The graph shows performance for class-A transactions. From 2 to 64 threads, performance collapses on all protocols.*

randomly-chosen class-B transactions. (This is somewhat similar to the setup for TPC-C's delivery transaction.) Performance on class-A transactions collapses under all three optimistic protocols.

6.2 Benefits of reordering

Figure 8a (high-contention TPC-C) shows that TSTO, which implements TicToc concurrency control, has an advantage even over MSTO. TSTO's dynamic transaction reordering avoids some conflicts on this benchmark, helping it outperform OSTO by up to 1.7×; since it keeps only one version per record, it avoids multi-version overheads and outperforms MSTO by up to 2×. This effect is limited to TPC-C: we observed no significant benefit of TSTO over OSTO in any other workload.

We believe this effect centers on a conflict between TPC-C's new-order and payment transactions. These transactions conflict while trying to access the same `WAREHOUSE` table row. New-order transactions read the tax rate of the warehouse, while payment transactions increment the year-to-date payment amount of the warehouse. This causes a conflict and attendant aborts on OCC, but the conflict is false – the transactions actually access distinct columns in the warehouse table – and TicToc and MVCC can reschedule the new-order transaction to commit with an earlier commit timestamp, reducing aborts and improving performance. However, this approach may not generalize well. Transactions that issue more reads than new-order are more difficult to reschedule, since reads constrain ordering, and TicToc cannot reschedule write-write conflicts. Neither TicToc nor MVCC addresses the true scalability issue, which is the false conflict. In §8 we will show that eliminating this class of conflicts with timestamp splitting is a more effective and generalizable approach that applies to all our benchmarks, not just TPC-C.

6.3 Cross-system comparisons

Figure 10 shows how STOv2 baseline systems compare with other state-of-the-art main-memory transaction systems on TPC-C. We use reference distributions of Cicada, ERMIA, and MOCC.

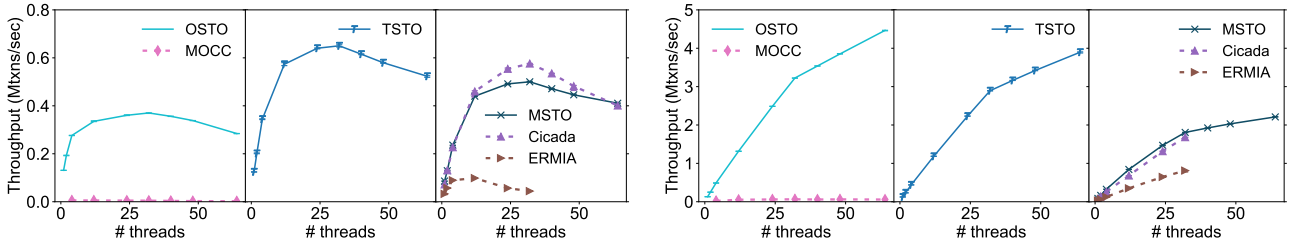
Figure 10a shows that both MOCC and ERMIA struggle at high contention; the reason is the overhead of locking. Cicada modestly outperforms both MSTO and OSTO. We expected Cicada to outperform our system, which lacks several Cicada optimizations. (For instance, Cicada assigns transaction timestamps using a scalable distributed algorithm – "loosely synchronized software clocks" – rather than a possibly-contended global variable, and its "early version consistency check" and "write set sorting by contention" optimizations attempt to abort doomed transactions as soon as possible, reducing wasted work.) However, we were surprised by the relatively small difference in performance, since in Cicada's own evaluation it outperformed all other systems, even on low contention benchmarks, by up to 3×. We believe that this is due to Cicada's evaluation comparing systems with different basis factors, which unfortunately leaves the relative importance of Cicada's optimizations in question.[2] Furthermore, in our measurements at low contention and high core counts, Cicada's performance collapses and it fails to complete some benchmarks due to memory exhaustion (Figure 10b). The reason is an issue with Cicada's special-purpose memory allocator (there is no exhaustion when that allocator is replaced with jemalloc), highlighting the costs as well as potential benefits of purpose-built allocators for in-memory databases.

7 High-contention optimizations

Our measurements show, once basis factors are controlled, that the choice of concurrency control protocol has limited effect on scalability. OSTO, TSTO, and MSTO have different relative performance on different benchmarks, but scale similarly on each benchmark. For high-contention read-write workloads, we see no evidence that a concurrency control protocol can remove a scalability bottleneck on its own.[3] To

[2] Since Cicada's basis factor choices are good, we doubt changes in basis factors would dramatically alter its performance.

[3] MVCC can remove some scalability bottlenecks involving read-only transactions since declared read-only transactions can always commit.



(a) *TPC-C, one warehouse (high contention).*

(b) *TPC-C, one warehouse per worker (low contention).*

Fig. 10: *Cross-system comparisons: STOv2 baselines and other state-of-the-art systems, TPC-C full mix.*

make a high-conflict transactional workload scale, programmers must remove some conflicts.

We now describe two general techniques, *deferred updates* (DU) and *timestamp splitting* (TS), that can eliminate many false conflicts from a broad range of transactional workloads, including every workload we tried. These techniques are based on previous work: deferred updates relates to ideas from transaction chopping [45] and timestamp splitting to vertical partitioning [39]. Our contributions lie in developing low-overhead versions of these techniques that perform effectively in memory, and that apply to all of OCC, TicToc, and MVCC. The techniques require workload-specific configuration by application programmers and/or schema designers, but they are conceptually general and not difficult to apply. They improved performance, sometimes significantly, for all of OCC, TicToc, and MVCC, and in some cases, they are more useful in combination than they are separately. The rest of this section describes the techniques; the next section evaluates them.

7.1 Deferred updates

Deferred updates aim to eliminate validation aborts associated with read-modify-write operations, which observe a record value and modify the record based on the previous value. Optimistic protocols implement these operations by first observing the relevant record versions (adding to the read set), then computing new values, and then, in the commit protocol, validating the read sets before committing the modifications. The validation step causes many aborts in high-contention workloads. However, many read-modify-write operations have limited interaction with the rest of a transaction, and such delimited operations can be *deferred* to a point in transaction execution when validation is no longer necessary.

Deferred updates are implemented using *updaters*, which are function objects that encode a read-modify-write operation and any parameters. Updaters are invoked on versions of the underlying record; they are allowed to observe the record value and their encoded parameters, and to modify the record value in place. OSTO and TSTO invoke updaters during Phase 3 of their commit protocols, when the corresponding records are locked and therefore safe to modify. When all modifications to a record are encoded in an updater, there is no need to validate the version of the corresponding record, and most aborts concerning those records are eliminated. (Figure 11 demonstrates this for a simple increment transaction.) MSTO goes even further: updaters are added directly to version chains and invoked only when the corresponding values are read (or garbage collected). This avoids validation aborts and additionally supports more flexible transaction ordering than is possible for conventional OCC or even MVCC. Serializability is ensured by restricting the operation of updaters according to a set of isolation requirements, and by updating our commit protocols, especially for MSTO, to prevent interference from concurrent transactions.

To support deferred updates, transactional write sets in STOv2 are generalized to hold updaters as well as values. Each updater encodes the operation it will perform, such as "increment column," and any necessary parameters, such as the amount to increment by. When invoked with a record value as an argument, the updater changes the value according to its encoded operation. Updaters are not limited to simple operations; the updater for the TPC-C new-order transaction's update to the `STOCK` table is shown in Figure 12. However, updaters are limited in terms of the operations they can perform, and limit their containing transactions in terms of abort behavior. These limitations simplify our implementation and guarantee that transactions containing updaters are serializable. Specifically:

1. Each updater applies to a single database record. It can observe its encoded parameters and observe and modify the record's value. It cannot access other database state.
2. If a record associated with a deferred update is deleted or absent, the transaction must abort. The deferred update cannot delete or insert its associated record.
3. The record's observed value may be used to compute the record's new value, but must not otherwise affect the

```
During execution:
    abort if x does not exist
    [tmp, readset[x]] = atomic read of x.value and x.ts
    writeset[x] = tmp + 1
Phase 1 of commit protocol:
    lock x
    tx.ts = atomic increment of cts_g
Phase 2 of commit protocol:
    abort if x.ts != readset[x] or x has been deleted
Phase 3 of commit protocol:
    x.value = writeset[x]
    x.ts = tx.ts
    unlock x
```

(a) *Baseline implementation of a transaction that increments* x*. Concurrent increments can cause aborts in Phase 2.*

```
During execution:
    abort if x does not exist
    writeset[x] = (fn rec => rec.value = rec.value + 1)
Phase 1 of commit protocol:
    lock x
    tx.ts = atomic increment of cts_g
Phase 2 of commit protocol:
    abort if x has been deleted
Phase 3 of commit protocol:
    writeset[x](x) (apply updater)
    x.ts = tx.ts
    unlock x
```

(b) *Implementation of a transaction that increments* x*, including deferred updates. Concurrent increments never cause aborts.*

Fig. 11: *Increment transactions in OSTO, with and without deferred updates.*

transaction's control or data flow. In particular, no record value can cause the transaction to abort.[4]

4. The transaction's return value must be independent of the value produced by the deferred update.
5. A transaction cannot apply both an updater and a conventional write to the same record.

If, for example, some of a particular record's values could cause the transaction to abort, then the corresponding observations must be validated using a conventional optimistic read set.

Deferred updates offer some of the same benefits as transaction chopping [45], which divides a single transaction into multiple pieces that execute partially independently. A deferred update transaction has a main piece that executes according to an optimistic concurrency control protocol with read-set validation, and subsequent pieces – the updaters – that execute on single locked records. However, compared to chopped transaction pieces, updater pieces have limited semantics and execute in a particular phase of the commit protocol. This gives deferred updates serializable results independent of SC-graphs or the contents of concurrently executing transactions, and allow deferred updates to be implemented without a chopping-style analysis of all possible concurrent transactions.

[4] This constrains the use of deferred updates for operations that can overflow. For example, an increment operation can be encoded as a deferred update only if the increment can be performed for any value without error. This is possible for bignums, floating-point numbers, and fixed-size integers with modular or clipped arithmetic, as well as for values that are constrained by external factors, but it is not typically true for fixed-size integers with signaling overflow.

7.1.1 Single-version implementation

In OSTO and TSTO updaters are invoked during the commit protocol. The lock phase (Phase 1) locks all modified records, including records associated with updaters. The validate phase (Phase 2) need not check timestamps on records associated with updaters, but must abort if an associated record has been deleted by a concurrent transaction. Finally, the install phase (Phase 3) executes each updater function object, passing the associated record's current value as an argument.

This implementation is correct because conventional observations are validated optimistically and updater-based observations effectively use a variant of two-phase locking. Just as in conventional single-version OCC, the transaction's serialization point is after Phase 1, when the transaction's unique timestamp is selected. Read-set validation (Phase 2) must check that all optimistic observations are valid through to the serialization point. For deferred updates, though, Phase 2 only needs to validate that the associated records exist. This follows from the isolation requirements, which ensure that the transaction using the updater is unaffected by the computation that the updater performs. The observations associated with deferred updates are performed in Phase 3, but the isolation requirements ensure that these observations are of locked records, and thus equal to the values 2PL would have observed and equal to the values current at the serialization point.

7.1.2 Multi-version implementation

Multi-version concurrency control can commit transactions in serializable orders impossible for single-version systems, but observations, such as read-modify-writes, constrain this reordering. When a record is observed at some timestamp, that prevents any modification of the record from committing at any prior timestamp. The MSTO implementation of deferred updates loosens this restriction, allowing transactions containing read-modify-write operations to commit out of order by adding updaters directly to version chains. The updaters are executed lazily when the full underlying record values are observed. A read-modify-write executed using an updater does not observe the underlying record's value until the updater executes (before updater execution, the transaction only observes whether the record is present). Therefore, the transaction does not need to update any previous

```
class NewOrderStockUpdater {
public:
    NewOrderStockUpdater(int32_t qty, bool remote)
        : update_qty(qty), is_remote(remote) {}

    void operate(stock_value& sv) const {
        if ((sv.s_quantity - 10) >= update_qty)
            sv.s_quantity -= update_qty;
        else
            sv.s_quantity += (91 - update_qty);
        sv.s_ytd += update_qty;
        sv.s_order_cnt += 1;
        if (is_remote)
            sv.s_remote_cnt += 1;
    }

private:
    int32_t update_qty;
    bool is_remote;
};
```

Fig. 12: *Updater for* STOCK *table records in TPC-C's new-order transaction. The* operate *method encodes the operation (stock deduction and replenishment).*

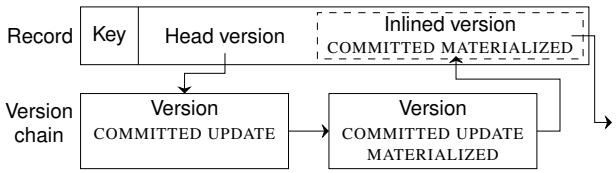


Fig. 13: *Record structure in MSTO with deferred updates. Each version can contain an updater, a materialized version (a full version), or both; the data in an updater or materialized version is read-only once it is assigned. Concurrent transactions can insert more updaters either above or below the* COMMITTED UPDATE*, but not below any materialized version (including the* COMMITTED MATERIALIZED UPDATE*). Like materialized versions, updater versions can be pending or aborted.*

versions' *rts* values until the updater executes. This allows future transactions to commit changes before the updater.

Deferred updates required the following changes to MSTO. In addition to the *rts*, *wts*, and *value* slots present in conventional MSTO versions, deferred updates adds an *updater* slot that can hold an updater and an *mstatus* word holding materialization status flags (Figure 13). A version v added by a conventional write will have a full record value in its $v.value$ slot, an empty $v.updater$ slot, and $v.mstatus = $ MATERIALIZED. A version w added by a deferred update will, in contrast, have an updater in its $w.updater$ slot, an empty $w.value$ slot, and $w.mstatus = $ UPDATE. If and when w is observed by a conventional read, MSTO must run a *materialize* procedure that computes the corresponding full record value, fills in the $w.value$ slot, and changes $w.mstatus$ to MATERIALIZED UPDATE.[5] The materialization procedure requires careful engineering to ensure safety and serializability. It must monitor the version chain for concurrent updates, and it must adjust previous versions' *rts* values to prevent future updates from being inserted during the computation. Furthermore, materialization interacts with garbage collection: the procedure for garbage collecting version chains must materialize values (it is unsafe to recycle an updater until a later materialized version exists). Algorithm 2 shows a version of MSTO's commit protocol Phase 1 that supports deferred updates. This algorithm must validate that updaters are not applied to deleted versions or absent records, requiring a new check that the newly inserted version does not invalidate a later updater (lines 7–12). It also must check that no previously inserted version would invalidate a newly inserted updater (lines 13–20). These checks must be performed *after* the version is inserted into the version chain, ensuring that if conflicting versions are inserted concurrently, at least one of those versions will view the other version and abort.

Algorithm 2 MSTO commit protocol with deferred updates. Compare Algorithm 1.

```
ALLOW-PRECEDE(v, newerv)
1  // Test if version v may precede version newerv in a chain, where
2  // newerv has a greater timestamp, considering only materialization.
3  // Aborted versions are ignored; pending or committed deleted
4  // versions cannot precede pending or committed updates.
5  return v.status == ABORTED or newerv.status == ABORTED
6       or v.mstatus ≠ DELETED or newerv.mstatus ≠ UPDATE

MSTO-PHASE1-WITH-UPDATES(tx)
 1  for each wv ∈ tx.writeset:
 2      wv.wts = tx.ts
 3      wv.status = PENDING
 4      if ¬TRY-INSERT-WRITE-VERSION(wv):
 5          ABORT(tx); return
 6      // Now wv is in the version chain at the proper location.
 7      // Check for updater invalidation
 8      v = wv.record.head
 9      while v ≠ wv:
10          if ¬ALLOW-PRECEDE(wv, v):
11              ABORT(tx); return
12          v = v.prev
13      // Check for concurrent reads and updater invalidation
14      v = wv.prev
15      while v.status ≠ COMMITTED:
16          if ¬ALLOW-PRECEDE(v, wv):
17              ABORT(tx); return
18          v = v.prev
19      if v.rts > wv.wts or ¬ALLOW-PRECEDE(v, wv):
20          ABORT(tx); return
```

Algorithm 3 shows the algorithm used to materialize an updater version. MSTO must traverse the version chain back-

[5] It is important to preserve $w.updater$ in case of concurrent access by other transactions.

Algorithm 3 Materializing deferred updates.

```
MATERIALIZE(readv)
 1  trace = empty stack of versions
 2  v = readv
 3  while v.status ≠ COMMITTED or v.mstatus == UPDATE:
 4      trace.PUSH(v); v = v.prev
 5  vbuf = copy of v.value
 6  v.rts = atomic max{v.rts, readv.wts}      // prevent later commit
 7  while trace is not empty:
 8      vbase = v
 9      v = trace.POP()
10      // Invariant: The COMMITTED version at or before vbase
11      // has rts ≥ readv.wts, so no new committed versions can
12      // appear between vbase and v after this point.
13      // Usually vbase == v.prev, but not always.
14      while vbase.wts < v.prev.wts:
15          trace.PUSH(v); v = v.prev
16      wait until v.status ≠ PENDING
17      if v.status == COMMITTED:
18          v.rts = atomic max{v.rts, readv.wts}
19          if v.mstatus == UPDATE:
20              apply v.updater to vbuf
21          else
22              vbuf = copy of v.value
23  // Now v == readv; versions before readv are fixed;
24  // and vbuf is the correct materialized value for readv.
25  if TRY-LOCK(readv) is successful:
26      readv.value = vbuf
27      readv.mstatus = MATERIALIZED UPDATE
28      UNLOCK(readv)
29      enqueue readv for garbage collection of its predecessors
```

ward to the previous COMMITTED MATERIALIZED version, then traverse forward, applying the intervening updaters in timestamp order. This is done while acquiring minimal locks to improve concurrency. Care is required to prevent concurrent updates: when the materialization process applies an updater, that fixes the version chain before that point, preventing all transactions with earlier timestamps from committing. (The updates to *v.rts* on lines 6 and 18 accomplish this; they synchronize with the *rts* check on line 19 of Algorithm 2.) Care is also required to detect versions added during the traversal. (Lines 13–15 accomplish this.) Each thread computes the materialized record on a thread-local copy, allowing multiple threads to materialize the same version chain concurrently. Once the materialized record has been computed, the version that initiated the materialization is locked while the record data is copied over.

Updater versions also impact MSTO's garbage collection. MSTO ensures that whenever a materialized version is created – either directly, through a conventional write, or indirectly, through a explicit materialization – all older versions are enqueued for RCU garbage collection. Additionally, MSTO periodically materializes infrequently-read records so that older versions can be marked for garbage collection and version chains do not grow without bound.

Our implementation doesn't bother to fill in intermediate deferred updates as we walk up the chain. This is a performance tradeoff: we believe it is rare that such versions will be observed. This means we assume updaters are deterministic. It is also worth noting that the *rts* updates on lines 6 and 18 may create version chains where a version's read timestamp is larger than the write timestamp of its successor. This is intentional and safe. Attempts to assign precise read timestamps would be vulnerable to bugs caused by concurrent updates, and our other algorithms understand that a version's true read timestamp is bounded above by the next committed version's write timestamp.

The materialization procedure guarantees correctness because it processes the versions in timestamp order as it moves up the stack, including any concurrent committed versions that were added to the chain. Progress is ensured because after each iteration of the main loop, a new version has been flattened; only a limited number of new versions can be added to the chain as we move up because of the way we modify the read timestamp of each version when we process it.

7.2 Timestamp splitting

The deferred update optimization reduces aborts by splitting transaction computation into pieces, and by moving some computation into the commit protocol's install phase where validation cannot cause aborts. The complementary *timestamp splitting* optimization, described here, reduces aborts by splitting *records* into pieces with independent timestamps; concurrent transactions that modify different pieces cannot cause aborts. Timestamp splitting draws inspiration from schema transformations such as row splitting and vertical partitioning [39], which use sub-record access patterns to reduce database I/O overhead (for example, they might only keep frequently-accessed record fragments in a memory cache), but instead uses sub-record access patterns to reduce contention and aborts.

Many database records comprise multiple pieces of state subject to different access patterns. For instance, records in a relational database may have many columns, some of which are accessed more often or in different ways. Timestamp splitting divides a record's columns into disjoint subsets and assigns one timestamp per subset. Transactions that read or modify such a record observe just those timestamps sufficient to cover the columns they observed or modified. In a typical example, shown in Figure 14, one timestamp covers infrequently-modified columns while another timestamp covers the rest of the record. Simple splitting like this is frequently useful. In TPC-C's `CUSTOMER` table, the columns with the customer's name and ID are often observed but never modified, whereas other columns, such as those containing the customer's balance, change frequently; using

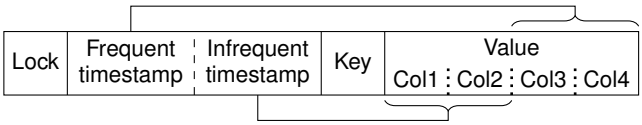


Fig. 14: *Record structure in OSTO with timestamp splitting. The frequent timestamp protects the frequently-updated columns, while the infrequent timestamp only updates if col1 or col2 change. This allows transactions that only read col1 and col2 to avoid conflicts with those that only write col3 and col4.*

a separate timestamp for name and ID allows observations that only access name and ID to proceed without conflict even as balance-related columns change.

OSTO, TSTO, and MSTO implement timestamp splitting by changing records to contain one *or more* timestamps, rather than exactly one timestamp, as shown in Figure 14. Although we support arbitrary numbers of timestamps per record, our evaluation only shows results for two timestamps. Additional timestamps have costs as well as benefits – for instance, read and write sets as well as record layouts take more memory – and on all of our benchmarks, three timestamps performed worse than two.

Timestamp splitting can expose additional deferred update opportunities. For example, this transaction appears not to benefit from deferred updates, since it observes `x.col1` and `x.col2`:

```
tmp = y.col1;
x.col1 += tmp;
return x.col2;
```

However, if `x.col1` and `x.col2` are covered by different timestamps, the modification to `x.col1` can be implemented via an updater since `x.col1` is not otherwise observed.

Our implementation of timestamp splitting effectively stores several "sub-records" within the record, as shown in Figure 14. In MSTO we maintain a separate chain for each sub-record. In both cases, deferred updates apply to sub-records, not the record as a whole.

7.3 Complementary effects

Deferred updates and timestamp splitting have complementary effects. In the common case, timestamp splitting separates a frequently-updated part of a record from an infrequently-updated, frequently-read part. We often found that the infrequently-updated part of a record contained record IDs and other data used in transaction control and data flow, whereas the frequently-updated part of the record was treated in a more isolated manner from the rest of the transaction. This meant that transactions originally unsuitable for deferred updates (because part of a record's value was used in control flow) became suitable after timestamp splitting (because after splitting, the updated parts were not used in control flow).

MSTO deferred updates particularly benefit from timestamp splitting because splitting makes individual versions smaller and reduces calls to the materialization procedure. After splitting, reads of the infrequently-updated part of a record no longer cause materialization of the frequently-updated part, so materialization occurs at a lower rate. This means that there are more opportunities for concurrent writers to reorder their changes.

7.4 Implementation in workloads

To implement these optimizations, we manually inspected our workloads. For timestamp splitting, we inspected each record type to determine whether and how it could be split. For record types with no frequently-updated columns, such as TPC-C's `HISTORY` table, we just use one timestamp as usual. If the record type has both frequently-updated and infrequently-updated columns – as is the case in many of TPC-C's tables, including `ORDER` and `STOCK` – the frequently-updated columns are assigned a separate timestamp. If all columns of a record type are frequently-updated, as is the case in YCSB, half of the columns are arbitrarily chosen to be assigned a separate timestamp. (As we'll see, this choice has limited benefit for OSTO and TSTO, but good benefits on MSTO.) Transaction programs identify the columns they access when making point and range queries, but the column-to-timestamp assignment is handled automatically by STOv2.

For deferred updates, we create an updater implementation for each relevant record type. Our workloads make use of many different deferred updates. Some examples: in RUBiS, an updater changes an item's `max-bid` and `quantity` columns; in TPC-C, an updater on the `WAREHOUSE` table increments its `ytd` (orders year-to-date) field, and one on the `CUSTOMER` table updates several of its fields for orders and payments. The shortest updater takes about 10 lines of code, including boilerplate; the longest, on TPC-C's `CUSTOMER` table, takes about 30 lines. Using deferred updates changes some full validations in the read set to existence validations. For example, the number of full read validations in read sets for TPC-C new-order transactions shrink by 30% on average, and for payment transactions by 50%. Fewer full read validations means fewer read-write dependency edges between transactions and fewer conflicts. Since deletions can prevent deferred updates from being reordered, workloads with frequent inserts, deferred updates, and deletions on the same key may find that DU is less effective at reducing conflicts between transactions.

Deferred updates reduce transaction read set sizes. For example, the read sets for TPC-C new-order transactions shrink by 30% on average, and payment transactions by 50%.

Smaller read sets mean fewer read-write dependency edges between transactions and fewer conflicts.

The implementation of these optimizations was facilitated by the STO platform, which allows application programmers to participate in some aspects of concurrency control through its transaction-aware datatypes.

8 Evaluation of high-contention optimizations

We now evaluate the deferred update and timestamp splitting optimizations to better understand their benefits at high contention, their overheads at low contention, and their applicability to different workloads and concurrency control protocols. We observe significant benefits from these optimizations on high-contention workloads on all concurrency control protocols, with little degradation of low-contention performance. The performance benefits achievable from high-contention optimizations outstrip those achievable by switching the underlying concurrency control protocol. Figure 15 shows the effects of applying deferred updates (DU) and timestamp splitting (TS), both separately and together, for all three concurrency control protocols, and on TPC-C, YCSB, Wikipedia, and RUBiS workloads with different amounts of contention.

The most dramatic improvement is observed in high-contention TPC-C (Figure 15a). DU+TS greatly improves throughput of all three CCs, with gains ranging from 2× (TSTO) to 5× (OSTO); each optimized CC performs better than any unoptimized systems. After optimization, MSTO even scales to 64 threads, though imperfectly, on this high-contention TPC-C workload. Overall, however, optimized MSTO only outperforms optimized OSTO or TSTO under extremely high contention (20 cores running high-contention TPC-C); at lower contention levels (e.g., Figure 15b), MSTO's multi-version overhead limits its performance.

Similar effects are visible on other benchmarks. The high-contention YCSB-A (Figure 15d), Wikipedia (Figure 15g), and RUBiS (Figure 15h) workloads benefit from the techniques, especially at high core counts. Deferred updates generally have more impact than timestamp splitting (TPC-C's schema has more natural split points than the other schemas). The techniques can reduce performance slightly, especially on MSTO and on low-contention benchmarks; for instance, in Figure 15c, unoptimized MSTO performs better than all optimized versions. However, we were surprised to find that even in some low-contention benchmarks, they can improve performance slightly: consider, for example, the OSTO and TSTO graphs in Figures 15e and 15f. This is because TS can reduce the amount of data retrieved from and written to the database by accessing subsets of columns, and DU can reduce read set validation costs.

Figure 15i shows the distinct effects of DU and TS on our high-contention benchmarks for OCC and MVCC. In some workloads, such as TPC-C, DU and TS produce greater benefits together than would be expected from their individual performance. This is especially clear for MSTO: DU and TS *reduce* performance when applied individually, but improve performance by 4.74× at 64 threads when applied in combination. This is because many frequently-updated TPC-C fields can be updated using DU, but only after the infrequently-updated column values use a separate timestamp. Of the two optimizations, DU is more frequently useful on its own. For instance, the highest overall performance for Wikipedia is obtained by applying DU to OSTO. This is an indication that write-write conflicts are predominant in these workloads, since DU reduces the impact of write-write conflicts while TS reduces the impact of read-write false sharing.

9 Conclusion

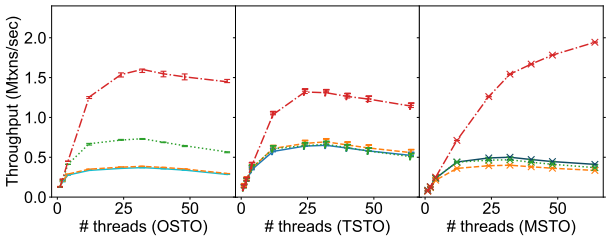
We investigated three approaches to improving the throughput of main-memory transaction processing systems under high contention, namely basis factor improvements, concurrency control algorithms, and high-contention optimizations. Poor basis factor choices can cause damage up to and including performance collapse: we urge future researchers to consider basis factors when implementing systems, and especially when evaluating older systems with questionable choices. Given good choices for basis factors, we believe that high-contention optimizations – deferred updates and timestamp splitting – are more powerful than choice of concurrency control algorithm. DU+TS can improve performance by up to 5× over base concurrency control for TPC-C, while the difference between unoptimized CC algorithms is at most 2×.

It is possible that a future workload-agnostic concurrency control algorithm with no visibility into record semantics might capture the opportunities exposed by DU+TS, but we are not optimistic. We believe that the improvement shown by TicToc and MVCC on high-contention TPC-C is more likely to be the exception than the rule. The best way to improve high-contention main-memory transaction performance is to eliminate classes of conflict, as CU+TS explicitly do. Though in our work these mechanisms require some manual intervention to apply, we hope future work will apply them automatically, for instance by using static analysis to identify potential instances of false sharing.

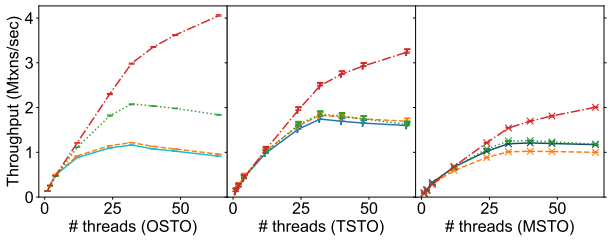
Finally, we are struck by the overall high performance of OCC on both low and high contention workloads, although MVCC and other CC mechanisms may have determinative advantages in workloads unlike those we tried.

Our code and benchmarks are available online at this repository, under the `vldbj20` tag:

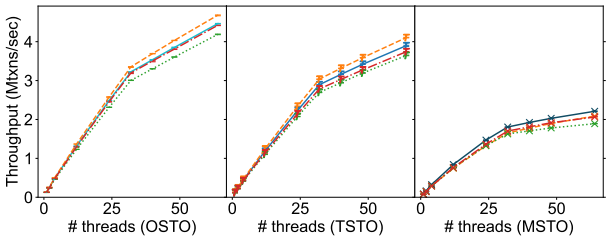
`https://readablesystems.github.io/sto`



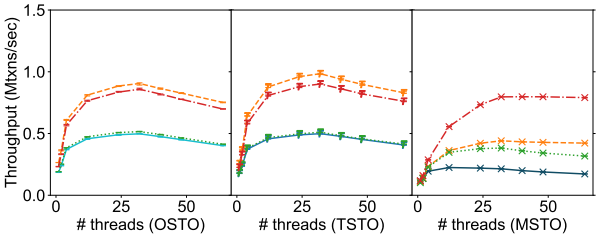
(a) *TPC-C, one warehouse (high contention).*



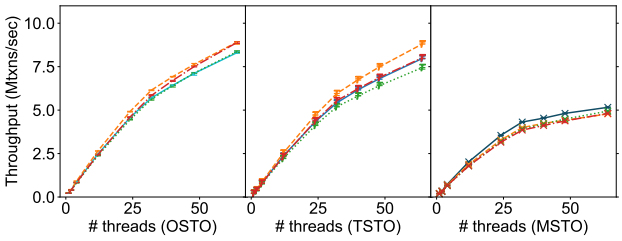
(b) *TPC-C, four warehouses (medium contention).*



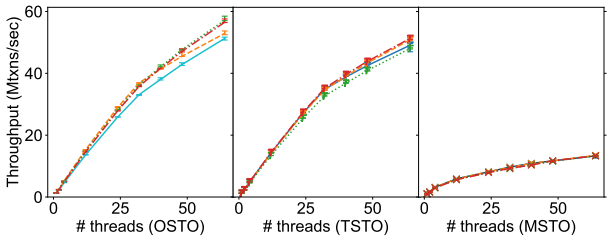
(c) *TPC-C, one warehouse per worker (low contention).*



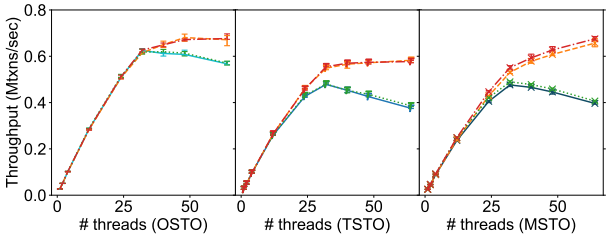
(d) *YCSB-A (high contention: update-intensive, 50% updates, skew 0.99).*



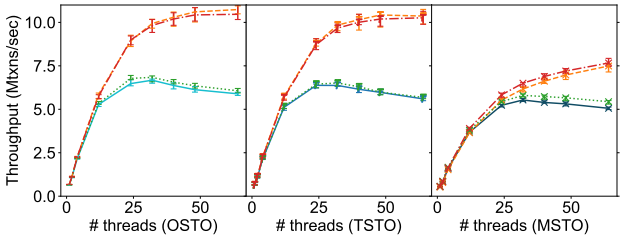
(e) *YCSB-B (lower contention: read-intensive, 5% updates, skew 0.8).*



(f) *YCSB-C (read-only workload).*



(g) *Wikipedia (high contention).*



(h) *RUBiS (high contention).*

Benchmark	OSTO	OSTO+DU	OSTO+TS	OSTO+DU+TS	MSTO	MSTO+DU	MSTO+TS	MSTO+DU+TS
TPC-C	279	292 (1.05×)	556 (1.99×)	1397 (5.01×)	413	332 (0.80×)	368 (0.89×)	1957 (4.74×)
YCSB-A	405	751 (1.85×)	413 (1.02×)	707 (1.75×)	174	423 (2.43×)	319 (1.83×)	796 (4.57×)
Wikipedia	565	679 (1.20×)	572 (1.01×)	677 (1.20×)	403	663 (1.65×)	405 (1.00×)	662 (1.64×)
RUBiS	5924	10633 (1.79×)	6098 (1.03×)	10516 (1.78×)	5171	7624 (1.71×)	5452 (1.05×)	7827 (1.51×)

(i) *Throughput in Ktxns/sec at 64 threads in high-contention benchmarks, with improvements over respective baselines in parentheses.*

Fig. 15: *STOv2 performance with deferred updates and timestamp splitting (DU+TS).*

Acknowledgements Part of the work on basis factors was presented by Yihe Huang at the Student Research Competition at the 27th ACM Symposium on Operating Systems Principles (SRC @ SOSP 2019). We also thank the the AWS Cloud Credits for Research Program for providing us compute infrastructure. This work was funded through NSF awards CNS-1704376, CNS-1513416, CNS-1513447, and CNS-1513471. We're grateful to Stratos Idreos, Andy Pavlo, and Peter Alvaro for thoughtful comments on earlier drafts. Thanks also to anonymous reviewers of the work.

References

1. Abramson, N.: The Aloha system: Another alternative for computer communications. In: Proceedings of the November 17-19, 1970, Fall Joint Computer Conference, AFIPS '70 (Fall), pp. 281–285. ACM (1970)
2. Agrawal, R., Carey, M.J., Livny, M.: Concurrency control performance modeling: Alternatives and implications. ACM Transactions on Database Systems (TODS) **12**(4), 609–654 (1987)
3. Badrinath, B., Ramamritham, K.: Semantics-based concurrency control: Beyond commutativity. ACM Transactions on Database Systems (TODS) **17**(1), 163–199 (1992)
4. Bernstein, P.A., Goodman, N.: Multiversion concurrency control—theory and algorithms. ACM Transactions on Database Systems (TODS) **8**(4), 465–483 (1983)
5. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Science of Computer Programming **63**(2), 172–185 (2006)
6. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, SOCC '10, pp. 143–154. ACM (2010)
7. Diaconu, C., Freedman, C., Ismert, E., Larson, P.A., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: SQL Server's memory-optimized OLTP engine. In: Proceedings of the 2013 International Conference on Management of Data, SIGMOD '13, pp. 1243–1254. ACM (2013)
8. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Proceedings of the 20th International Symposium on Distributed Computing, DISC '06, pp. 194–208. Springer (2006)
9. Difallah, D.E., Pavlo, A., Curino, C., Cudre-Mauroux, P.: OLTP-bench: An extensible testbed for benchmarking relational databases. PVLDB **7**(4), 277–288 (2013)
10. Ding, B., Kot, L., Gehrke, J.: Improving optimistic concurrency control through transaction batching and operation reordering. PVLDB **12**(2), 169–182 (2018)
11. Dragojević, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, pp. 155–165. ACM (2009)
12. Dragojević, A., Narayanan, D., Hodson, O., Castro, M.: FaRM: Fast remote memory. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI '14, pp. 401–414. ACM (2014)
13. Durner, D., Leis, V., Neumann, T.: On the impact of memory allocation on high-performance query processing. In: Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN '19. ACM (2019). DOI 10.1145/3329785.3329918. URL http://dx.doi.org/10.1145/3329785.3329918
14. Faleiro, J.M., Abadi, D.J.: Rethinking serializable multiversion concurrency control. PVLDB **8**(11), 1190–1201 (2015)
15. Fernandes, S., Cachopo, J.: A scalable and efficient commit algorithm for the JVSTM. In: Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing (2010)
16. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: Proceedings of the 24th annual ACM Symposium on Principles of Distributed Computing, PODC '05, pp. 258–264. ACM (2005)
17. Held, G., Stonebraker, M., Wong, E.: INGRES: a relational data base system. In: Proceedings of the May 19-22, 1975, national computer conference and exposition, pp. 409–416. ACM (1975)
18. Héman, S., Zukowski, M., Nes, N.J., Sidirourgos, L., Boncz, P.: Positional update handling in column stores. In: Proceedings of the 2010 International Conference on Management of Data, SIGMOD '10, pp. 543–554. ACM (2010)
19. Herlihy, M., Koskinen, E.: Transactional boosting: A methodology for highly-concurrent transactional objects. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '08, pp. 207–216. ACM (2008)
20. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA '93, pp. 289–300. ACM, New York, NY, USA (1993). DOI 10.1145/165123.165164. URL http://doi.acm.org/10.1145/165123.165164
21. Herman, N., Inala, J.P., Huang, Y., Tsai, L., Kohler, E., Liskov, B., Shrira, L.: Type-aware transactions for faster concurrent code. In: Proceedings of the 11th European Conference on Computer Systems, EuroSys '16. ACM (2016)
22. Huang, Y., Qian, W., Kohler, E., Liskov, B., Shrira, L.: Opportunities for optimism in contended main-memory multicore transactions. PVLDB **13**(5), 629–642 (2020). DOI 10.14778/3377369.3377373. URL http://www.vldb.org/pvldb/vol13/p629-huang.pdf
23. Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., et al.: BetrFS: A right-optimized write-optimized file system. In: 13th USENIX Conference on File and Storage Technologies, FAST '15, pp. 301–315. ACM (2015)
24. Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E.P.C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., Abadi, D.J.: H-Store: A high-performance, distributed main memory transaction processing system. PVLDB **1**(2), 1496–1499 (2008). DOI 10.14778/1454159.1454211. URL http://dx.doi.org/10.14778/1454159.1454211
25. Kim, K., Wang, T., Johnson, R., Pandis, I.: ERMIA: Fast memory-optimized database system for heterogeneous workloads. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, pp. 1675–1687. ACM (2016)
26. Kimura, H.: FOEDUS: OLTP engine for a thousand cores and NVRAM. In: Proceedings of the 2015 International Conference on Management of Data, SIGMOD '15, pp. 691–706. ACM (2015)
27. Korth, H.F.: Locking primitives in a database system. Journal of the ACM (JACM) **30**(1), 55–79 (1983)
28. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Transactions on Database Systems (TODS) **6**(2), 213–226 (1981)
29. Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., Bear, C.: The Vertica analytic database: C-Store 7 years later. PVLDB **5**(12), 1790–1801 (2012)
30. Leis, V., Kemper, A., Neumann, T.: Exploiting hardware transactional memory in main-memory databases. In: IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014, pp. 580–591 (2014). DOI 10.1109/ICDE.2014.6816683. URL https://doi.org/10.1109/ICDE.2014.6816683
31. Lim, H.: Line comment in experiment script (run_exp.py). Available at https://github.com/efficient/cicada-exp-sigmod2017/blob/5a4db37750d1dc787f71f22b425ace82a18f6011/run_exp.py#L859 (2017). URL https://github.com/efficient/cicada-exp-sigmod2017/blob/5a4db37750d1dc787f71f22b425ace82a18f6011/run_exp.py#L859
32. Lim, H., Kaminsky, M., Andersen, D.G.: Cicada: Dependably fast multi-core in-memory transactions. In: Proceedings of the 2017 International Conference on Management of Data, SIGMOD '17, pp. 21–35. ACM (2017)
33. Maabreh, K.S., Al-Hamami, A.: Increasing database concurrency control based on attribute level locking. In: 2008 International Conference on Electronic Design, pp. 1–4. IEEE (2008)

34. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: Proceedings of the 7th European Conference on Computer Systems, EuroSys '12, pp. 183–196. ACM (2012)
35. McKenney, P.E., Boyd-Wickizer, S.: RCU usage in the Linux kernel: One decade later. Tech. rep. (2012)
36. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems (TOCS) **9**(1), 21–65 (1991)
37. Mu, S., Angel, S., Shasha, D.: Deferred runtime pipelining for contentious multicore software transactions. In: Proceedings of the 14th European Conference on Computer Systems, EuroSys '19, pp. 40:1–40:16. ACM (2019)
38. Narula, N., Cutler, C., Kohler, E., Morris, R.: Phase reconciliation for contended in-memory transactions. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, pp. 511–524. ACM (2014)
39. Navathe, S., Ceri, S., Wiederhold, G., Dou, J.: Vertical partitioning algorithms for database design. ACM Transactions on Database Systems (TODS) **9**(4), 680–710 (1984)
40. OW2 Consortium: RUBiS. Available at `https://rubis.ow2.org/`. URL `https://rubis.ow2.org/`
41. Rampant Pixels: rpmalloc - rampant pixels memory allocator. Available at `https://github.com/rampantpixels/rpmalloc` (2019). URL `https://github.com/rampantpixels/rpmalloc`
42. Reed, D.P.: Naming and synchronization in a decentralized computer system. Ph.D. thesis, Massachusetts Institute of Technology (1978)
43. Schwarz, P.M., Spector, A.Z.: Synchronizing shared abstract types. ACM Transactions on Computer Systems (TOCS) **2**(3), 223–250 (1984)
44. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Symposium on Self-Stabilizing Systems, pp. 386–400. Springer (2011)
45. Shasha, D., Llirbat, F., Simon, E., Valduriez, P.: Transaction chopping: Algorithms and performance studies. ACM Transactions on Database Systems (TODS) **20**(3), 325–363 (1995)
46. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995, pp. 204–213. ACM (1995). DOI 10.1145/224964.224987. URL `https://doi.org/10.1145/224964.224987`
47. Spiegelman, A., Golan-Gueta, G., Keidar, I.: Transactional data structure libraries. In: Proceedings the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16. ACM (2016)
48. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., et al.: C-Store: a column-oriented DBMS. PVLDB pp. 553–564 (2005)
49. Tang, D., Jiang, H., Elmore, A.J.: Adaptive concurrency control: Despite the looking glass, one concurrency control does not fit all. In: The 8th Biennial Conference on Innovative Data Systems Research, CIDR '17 (2017)
50. Transaction Processing Performance Council: TPC benchmark C. Available at `http://www.tpc.org/tpcc/`. URL `http://www.tpc.org/tpcc/`
51. Transaction Processing Performance Council: TPC benchmark C standard specification, revision 5.11. Available at `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf` (2010). URL `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf`
52. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: Speedy transactions in multicore in-memory databases. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles, SOSP '13, pp. 18–32. ACM (2013)
53. Wang, T., Kimura, H.: Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. PVLDB **10**(2), 49–60 (2016)
54. Wang, Z., Mu, S., Cui, Y., Yi, H., Chen, H., Li, J.: Scaling multicore databases via constrained parallel execution. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, pp. 1643–1658. ACM (2016)
55. Wang, Z., Qian, H., Li, J., Chen, H.: Using restricted transactional memory to build a scalable in-memory database. In: Proceedings of the 9th European Conference on Computer Systems, EuroSys '14, pp. 26:1–26:15. ACM (2014)
56. Wei, X., Shi, J., Chen, Y., Chen, R., Chen, H.: Fast in-memory transaction processing using RDMA and HTM. In: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, pp. 87–104. ACM (2015). DOI 10.1145/2815400.2815419. URL `http://doi.acm.org/10.1145/2815400.2815419`
57. Weihl, W.E.: Commutativity-based concurrency control for abstract data types. IEEE Transactions on Computers **37**(12), 1488–1505 (1988)
58. Wu, Y., Arulraj, J., Lin, J., Xian, R., Pavlo, A.: An empirical evaluation of in-memory multi-version concurrency control. PVLDB **10**(7), 781–792 (2017)
59. Yu, X., Bezerra, G., Pavlo, A., Devadas, S., Stonebraker, M.: Staring into the abyss: An evaluation of concurrency control with one thousand cores. PVLDB **8**(3), 209–220 (2014)
60. Yu, X., Pavlo, A., Sanchez, D., Devadas, S.: TicToc: Time traveling optimistic concurrency control. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, pp. 1629–1642. ACM (2016)
61. Zheng, W., Tu, S., Kohler, E., Liskov, B.: Fast databases with fast durability and recovery through multicore parallelism. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, pp. 465–477. ACM (2014)