Opportunities for Optimism in Contended Main-Memory Multicore Transactions

Yihe Huang,[1] William Qian,[1] Eddie Kohler,[1] Barbara Liskov,[2] Liuba Shrira[3]
[1]Harvard University, Cambridge, MA [2]MIT, Cambridge, MA [3]Brandeis University, Waltham, MA
yihehuang@g.harvard.edu, wqian@g.harvard.edu, kohler@seas.harvard.edu,
liskov@piano.csail.mit.edu, liuba@brandeis.edu

ABSTRACT

Optimistic concurrency control, or OCC, can achieve excellent performance on uncontended workloads for main-memory transactional databases. Contention causes OCC's performance to degrade, however, and recent concurrency control designs, such as hybrid OCC/locking systems and variations on multiversion concurrency control (MVCC), have claimed to outperform the best OCC systems. We evaluate several concurrency control designs under varying contention and varying workloads, including TPC-C, and find that implementation choices unrelated to concurrency control may explain much of OCC's previously-reported degradation. When these implementation choices are made sensibly, OCC performance does not collapse on high-contention TPC-C. We also present two optimization techniques, *commit-time updates* and *timestamp splitting*, that can dramatically improve the high-contention performance of both OCC and MVCC. Though these techniques are known, we apply them in a new context and highlight their potency: when combined, they lead to performance gains of 3.4× for MVCC and 3.6× for OCC in a TPC-C workload.

PVLDB Reference Format:
Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *PVLDB*, 13(5): 629–642, 2020.
DOI: https://doi.org/10.14778/3377369.3377373

1. INTRODUCTION

The performance of multicore main-memory transactional systems is a subject of intense study [13, 21, 23, 31, 36, 37, 49–51, 57]. Techniques based on optimistic concurrency control (OCC) perform extremely well on low-contention workloads, thanks to their efficient use of shared memory bandwidth and avoidance of unnecessary memory writes. On high-contention workloads, however, OCC can experience frequent aborts and, in the worst case, *contention collapse*, where performance for a class of transactions crashes to nearly zero due to repeated conflicts.

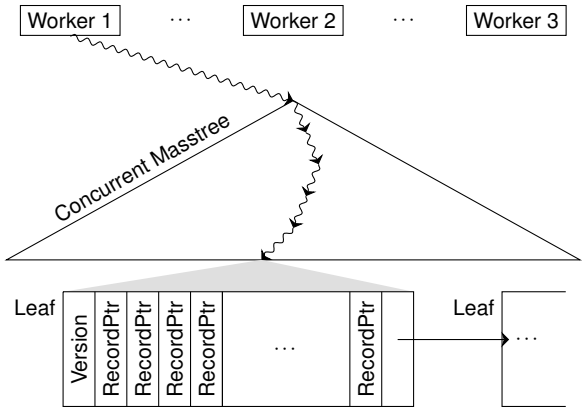
Recent designs targeted at high-contention workloads, including partially-pessimistic concurrency control [50], dynamic transaction reordering [57], and multiversion concurrency control (MVCC) [24, 31], change the transactional concurrency control protocol to better support high-contention transactions. In their evaluations, these designs show dramatic benefits over OCC on high-contention workloads, including TPC-C, and some show benefits over OCC even at low contention [31]. But many of these evaluations compare different code bases, potentially allowing mere implementation differences to influence the results.

We analyzed several main-memory transactional systems, including Silo [49], DBx1000 [56], Cicada [31], ERMIA [24], and MOCC [50], and found underappreciated engineering choices – we call them *basis factors* – that dramatically affect these systems' high-contention performance. For instance, some abort mechanisms exacerbate contention by obtaining a hidden lock in the language runtime.

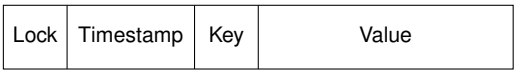
To better isolate the impact of concurrency control (CC) on performance, we implement and evaluate three CC mechanisms – OCC, TicToc [57], and MVCC – in a new system, *STOv2*, that makes good, consistent implementation choices for all basis factors. We show results up to 64 cores and for several benchmarks, including low- and high-contention TPC-C, YCSB, and benchmarks based on Wikipedia and RUBiS. With basis factors controlled, 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 [15] and MVCC performing well at all contention levels [31].

In addition, we introduce, implement, and evaluate two optimization techniques that can improve performance on high-contention workloads for all concurrency control schemes we evaluated (OCC, TicToc, and MVCC). These techniques safely eliminate classes of conflict that were common in our workloads. First, the *commit-time update* technique eliminates conflicts that arise when read-modify-write operations, such as increments, are implemented using plain reads and writes. Second, many records have fields that rarely change; the *timestamp splitting* technique avoids conflicts between transactions that read rarely-changing fields and transactions that write other fields. These techniques have workload-specific parameters, but they are conceptually general, and we applied them without much effort to every workload we investigated. Like MVCC and TicToc, the techniques improve performance on high-contention workloads. However, unlike MVCC, these optimizations have little performance impact at low contention; unlike TicToc and MVCC, they help on every benchmark we evaluate, not just TPC-C; and they benefit TicToc and MVCC as well as OCC. Though the techniques are widely known, our variants are new, and we are the first to report their application to TicToc and MVCC.

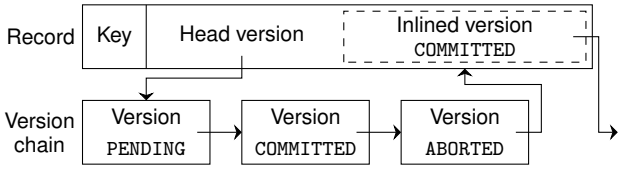
Proceedings of the VLDB Endowment, Vol. 13, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3377369.3377373



(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.*

Figure 1: *STOv2 overview.*

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

2. BACKGROUND

STOv2, or simply STO, is a reimplementation of the STOv1 software transactional memory system [21]. STOv1 supported only OCC and, as we describe later, made questionable implementation choices for some basis factors. STOv2 makes good choices and supports pluggable concurrency control protocols. We focus on three protocols: OSTO, the OCC variant; TSTO, the TicToc OCC variant; and MSTO, the MVCC variant.

Figure 1 provides an overview of the STO system and architecture. STO implements primary and secondary indexes using hash tables and trees. Unordered indexes use hash tables to map keys to records. To support range scans in ordered indexes, STO uses Masstree [33], a highly-concurrent B-tree variant that adopts some aspects of tries. Transactions are written as 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: all transaction parameters are available when a transaction begins, and transactions run without communicating with users. We do not support durability or networking, as they are not primary concerns of this work.

Timestamp	Name	Definition
Global write	wts_g	Periodically incremented
Thread-local write	wts_{th}	Per-transaction snapshot of wts_g; used to mark objects for deletion
Global read	rts_g	$< \min_{th} wts_{th}$
Thread-local read	rts_{th}	Per-transaction snapshot of rts_g
Global GC bound	$gcts$	$< \min_{th} rts_{th}$

Figure 2: *RCU-related timestamps in STOv2. For all threads th and at all times, $wts_g \geq wts_{th} > rts_g \geq rts_{th} > gcts$.*

2.1 OSTO

OSTO, the OCC variant, follows the Silo [49] OCC protocol. During *execution time*, transaction logic generates read and write sets. *Commit time*, which ensures serializability and exposes modifications to other transactions, runs in three phases. In Phase 1, the OSTO library *locks* all records in the write set, aborting if deadlock is suspected. The transaction's timestamp is selected after Phase 1; this marks its serialization point. In Phase 2, the library *validates* that records in the read set have not changed and are not locked by other transactions, aborting on validation failure. In Phase 3, the library *installs* new versions of the records in the write set, updates their timestamps, and releases locks.

OSTO aims to avoid memory contention except as required by workloads. For instance, it chooses transaction timestamps in a scalable way (as in Silo) and avoids references to modifiable global state. Read-copy-update (RCU) techniques [34], a form of garbage collection, are used to recycle memory and reshape data structures. This lets transactions safely access records after their logical deletion and largely eliminates readers-writer locks. RCU requires a mechanism for determining when RCU-deleted objects are safe to free, so STOv2 maintains a set of thread-local variables and several global variables that are periodically updated by a maintenance function. Figure 2 lists these variables. To mark an object for deletion (e.g., a record or tree node), the transaction running on thread *th* stores that object in a list associated with freeing timestamp $fts = wts_{th}$. Since the object might have been accessed by concurrently-running transactions, it is unsafe to free the object until all such transactions have committed or aborted. This is detected using *rts*: any concurrent transaction th' must have $rts_{th'} < fts$, so, since $gcts < rts_{th'}$, it will be safe to free the object once $fts \leq gcts$. An epoch advancer thread periodically increments wts_g and recomputes rts_g and *gcts*; this introduces little contention since it runs just once a millisecond.

2.2 MSTO

MSTO is an MVCC variant based broadly on Cicada [31], though it lacks some of Cicada's advanced features and optimizations. MSTO maintains multiple versions of each record and transactions can access recent-past states as well as present states. Read-only transactions can thus always execute conflict-free, since MSTO effectively maintains consistent database snapshots for all recent timestamps. MVCC can additionally 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 memory usage, which increases memory allocation and garbage collection

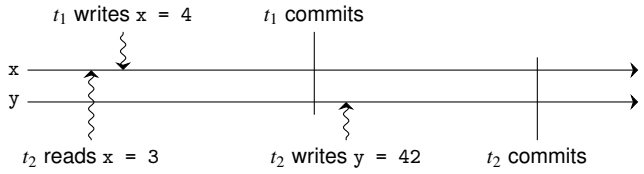


Figure 3: *Although t_2 finishes later in time, it can still commit if placed earlier than t_1 in the serial order. OCC will abort t_2; MVCC and TicToc can commit it.*

overhead and adds pressure on processor caches, and atomic memory operations, which MSTO invokes more frequently than OSTO.

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 the *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 *state*, as well as the record data and a chain pointer. The write timestamp is the timestamp of the transaction that created the version; it thus corresponds to an OSTO record's timestamp. The read timestamp is the timestamp of the latest committed transaction that observed the version. The chain is sorted by write timestamp: a committed chain $v_n, \ldots, v_1$ with latest version v_n will have $rts_i \geq wts_i$, $wts_{i+1} \geq rts_i$, and $wts_{i+1} > wts_i$ for all i.

Before initiating a transaction, MSTO assigns an execution timestamp ts_{th} used for all observations during execution time. For transactions identified in advance as read-only, $ts_{th} = rts_g$; otherwise, $ts_{th} = wts_g$. (A read-only transaction executing at rts_g is guaranteed to experience no conflicts: all read/write transactions at or prior to that timestamp have committed or aborted, and all future read/write transactions will have greater timestamps.) When observing a record, MSTO selects the version visible at ts_{th}. For reads, the version and the record are stored in the read set. For writes, only the record is stored in the write set.

MSTO's commit protocol follows Cicada's. At commit time, MSTO first chooses a commit timestamp with an atomic increment on the global write timestamp, $ts_{thc} := wts_g{+}{+}$. Then, in Phase 1, MSTO atomically inserts a new `PENDING` version with ts_{thc} into each modified record's version chain, ensuring that the chains maintain the prescribed timestamp order. Irreconcilable conflicts detected in Phase 1 cause an abort. (Concurrent transactions that access a `PENDING` version in their execution phase will spin-wait until the state changes.) In Phase 2, MSTO checks the read set: if any version visible at ts_{thc} differs from the version observed at ts_{th}, the transaction aborts; otherwise, MSTO atomically updates the read timestamp on each version v in the read set to $\texttt{v.rts} := \max\{\texttt{v.rts}, ts_{thc}\}$. Finally, in Phase 3, MSTO changes its `PENDING` versions to be `COMMITTED` and enqueues earlier versions for garbage collection. If a transaction is aborted, its `PENDING` versions are changed to `ABORTED` instead. The commit protocol is used only for read/write transactions; read-only transactions always commit.

MSTO incorporates one important Cicada optimization, namely *inlined versions*. One version can be stored inline with the record. This 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 (we do not implement Cicada's promotion optimization [31, §3.3]).

2.3 TSTO

TSTO is an OSTO variant that uses TicToc [57] in place of plain OCC as the CC mechanism. TicToc, like MVCC, uses separate read and write timestamps for each record, but it maintains only the most recent version. It dynamically computes transactions' commit timestamps based on read and write set information. This allows 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 do not use the TicToc delta-*rts* encoding [57, §3.6], which leads to false aborts in read-heavy workloads; instead, we use separate, full 64-bit words for *wts* and *rts*. This change caused no reduction in performance.

3. EXPERIMENT SETUP

We conduct 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 256GB of RAM. Medians of 5 runs 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 thread until they commit.

3.1 Workloads

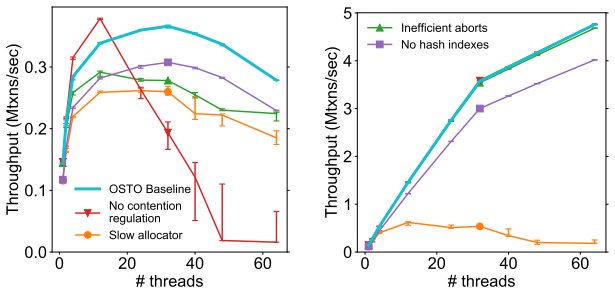
We measure two standard benchmarks, YCSB (A and B) [7] and TPC-C [47], with high and low contention settings. We also measure two additional high-contention workloads modeled after Wikipedia and RUBiS.

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" [48, §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 [49], which reduces unnecessary conflicts between new-order transactions. We implement contention-aware range indexes (§4.5) 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, while YCSB-B is read-heavy. 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). All YCSB indexes use hash tables.

Our Wikipedia workload is modeled after OLTP-bench [10]. Our RUBiS workload is the core bidding component of the RUBiS benchmark [39], which models an online auction site. Both benchmarks are naturally highly contended. 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.



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

(b) *One warehouse per worker (low contention).*

Figure 4: *OSTO throughput under TPC-C full-mix showing impact of basis factors. Factor optimizations are* individually *turned off from the optimized baseline to demonstrate the capping effect of each factor.*

4. BASIS FACTORS

Main-memory transaction processing systems differ in concurrency control, but also often differ in implementation choices such as memory allocation, index types, and backoff strategy. In years of running experiments on such systems, we have developed a list of *basis factors* where different choices can have significant impact on performance. This section describes the basis factors we have found most impactful. For instance, OCC's contention collapse on TPC-C can stem not from inherent limitations, but from particular basis factor choices. We describe the factors, suggest a specific choice for each factor that performs well, and conduct experiments using both high- and low-contention TPC-C to show their effects 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, which is our focus in this section. The heavy line represents the OSTO baseline 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, 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 memory allocation in MSTO has even larger impact due to multi-version updates; we omit these results for brevity.

4.1 Contention regulation

Contention regulation avoids repeated cache line invalidations by delaying retry after a transaction experiences a conflict. Over-eager retry can cause contention collapse; 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 [35] 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 performance with no backoff. Silo does not enable backoff by default [49]. Lack of contention regulation leads to high performance variations and even performance collapse as contention gets extreme. Silo supports exponential backoff through configuration, but some comparisons using Silo have explicitly disabled that backoff, citing (mild) effects at medium contention [30]. This is an unfortunate choice for evaluations including high-contention experiments.

4.2 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 experienced good results with rpmalloc [40]. A special-purpose allocator could potentially perform even better, and Cicada and other systems implement their own allocators. However, scalable allocators are complex in their own right, and we found bugs in some systems' allocators that hobbled performance at high core counts (§5.3). In our experience scalable general-purpose allocators are now fast enough for use in high-performance transactional software. Some systems, such as DBx1000, reduce allocator overhead to zero by preallocating 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 OSTO performance using the default glibc memory allocator. The default allocator is Silo's default choice [49]. (Silo also supports jemalloc through configuration; this outperforms glibc, but not by much.) OSTO with rpmalloc performs 1.6× better at high contention, and at low contention the glibc allocator becomes a bottleneck and stops the system from scaling altogether.

4.3 Abort mechanism

High-contention workloads stress the *abort mechanism* in transaction systems, since even very fast systems can abort 50% of transaction attempts or more. High abort rates do not necessarily correspond to lower throughput on modern systems, and in particular, reducing abort rates does not always improve performance [31]. 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 *explicitly-checked return values* instead.

The "Inefficient aborts" lines in Figure 4a show OSTO 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.

4.4 Index types

Transaction systems support different *index types* for table indexes. Silo, for instance, uses Masstree [33], 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 [56]. 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 OSTO performance when all indexes use Masstree, whether or not range scans are required. Silo and ERMIA lack hash table support. Hash index sup-

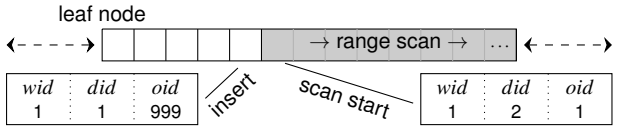


Figure 5: *Example illustrating index contention on the TPC-C* `NEW ORDER` *table. An insert to the end of one district in new-order can conflict with a range scan in delivery on the adjacent district.*

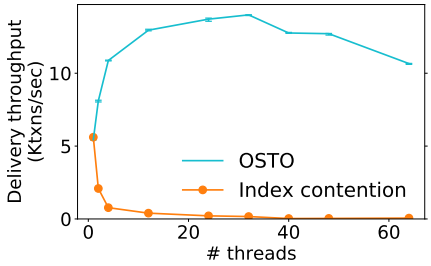


Figure 6: *Throughput of delivery transactions with and without contention-aware indexes. Showing OSTO results under TPC-C full mix, one warehouse.*

port offers 1.2× higher throughput at any contention level; this is less than 2.5× because data structure lookups are not the dominant factor in TPC-C transaction execution.

4.5 Contention-aware indexes

Contention-aware indexes do not greatly affect overall TPC-C performance, but hugely impact the performance of some classes of transaction. A contention-aware index is an index that avoids contention between disjoint ranges. 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. The new-order transaction inserts records at the end of a $\langle wid, did \rangle$ range, while the 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 persistently *within* a B-tree node, then in most systems, phantom protection will cause new-order to the earlier district and delivery to the later district to appear to conflict, inducing aborts in delivery (see Figure 5).

We recommend implementing contention-aware indexing, either automatically or by taking advantage of static workload properties. Our baselines implement contention-aware indexing by leveraging a side effect of Masstree's trie-like structure [33, §4.1]. Certain key ranges in Masstree will never cause 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 therefore reserve eight bytes for each key component in a multi-key index, which maps each key component to distinct layers of B-trees. This technique avoids 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 a TPC-C full-mix in OSTO. When not using 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 [31]. 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.

4.6 Other factors

Transaction internals refers to the 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 we recommend strong transaction internals by default. However, the factors listed above have more performance impact. Replacing STO's highly-engineered internals with Cicada's somewhat simpler versions reduced performance by just 5%.

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 [25,49,57]. 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 can benefit from aborting early. (The lock indicates upcoming changes to the underlying record, and if those changes happen it will often cause the transaction to abort anyway.) Our experience as well as prior study [52, §7.2] finds that write set sorting is expensive and we recommend *bounded spinning* for deadlock avoidance. However, write set sorting generally had relatively low impact ($\approx 10\%$) on TPC-C. The exception was DBx1000 OCC [57], which prevents deadlock using an unusually expensive form of write set sorting: 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. Write-set sorting took close to 30% of the total run time of DBx1000's "Silo" TPC-C under high contention.

4.7 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×).

5. CONCURRENCY CONTROL EVALUATION

Having implemented reasonable choices for the basis factors, we evaluate STOv2's three concurrency control mechanisms on our suite of benchmarks and at different contention levels. Our goal is to separate the performance impacts of concurrency control from those of basis factors.

Prior work showed OCC performance collapsing at high contention on TPC-C, but our findings are quite different. OSTO's high-contention TPC-C throughput is approximately 0.6× that of

System	Contention regulation	Memory allocation	Aborts	Index types	Transaction internals	Deadlock avoidance	Contention-aware index
Silo [49]	−−	−−	−−	−	−	+	+
STO [21]	−−	−−	−−	+	+	+	+
DBx1000 OCC [56]	+	N/A	+	+	−	−−	−−
DBx1000 TicToc [57]	+	N/A	+	+	−	+	−−
MOCC [50]	N/A	+	+	+	+	+	−−
ERMIA [24]	+	+	−−	−	+	+	+
Cicada [31]	+	+	+	+	+	N/A	N/A
STOv2 (this work)	+	+	+	+	+	+	+

Figure 7: *How comparison systems implement the basis factors described in §4. 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×.*

MSTO, even at 64 threads. Neither system either scales or collapses. At low contention, however, OSTO throughput is approximately 2× that of MSTO. These results hold broadly for our other benchmarks.

5.1 Overview

Figure 8 shows the transaction throughput of all three STOv2 variants on all our benchmarks, and 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 quite reach 4% of the mix when 63 other threads are performing transactions on the same warehouse; we observe 3.2%.) Perfect scalability would show as a diagonal line through the origin and the data point at 1 thread.

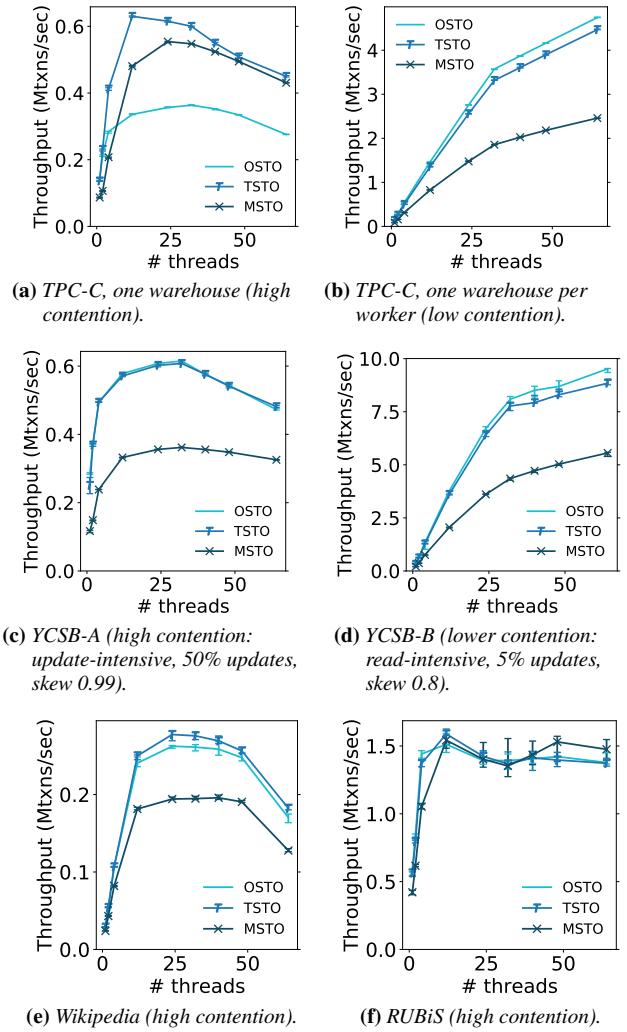
Only low-contention benchmarks (TPC-C with one warehouse per worker, Figure 8b, and YCSB-B, Figure 8d) approach perfect scalability. (The change in slope at 32 threads is due to our machine having 2 hyperthreads per core.) On high-contention benchmarks, each mechanism scales up to 4 or 8 threads, then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

When scalability is good, performance differences are due primarily to the inherent overhead of each mechanism. In Figure 8b, 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 8c). 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. We do not claim that OCC will *never* collapse. It is easy to cause OCC contention collapse for some transaction classes in a workload, such as by combining fast transaction processing ("modify single value") with analytics ("read entire database"), where MVCC could avoid collapse by executing read-only queries in the recent past. However, we did find it striking that these important, real-world-inspired benchmarks did not collapse, and that some of these benchmarks showed MVCC having similar scaling behavior as OCC under contention.

Some differences from prior results are worth mentioning. Our YCSB-A results are lower than those reported previously [31]. This



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

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

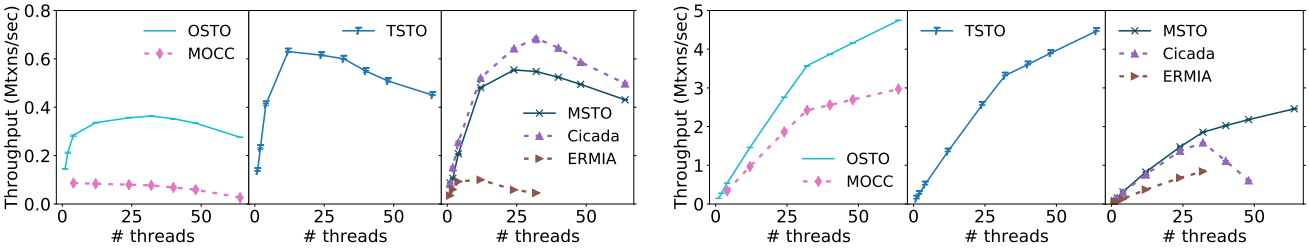
(c) *YCSB-A (high contention: update-intensive, 50% updates, skew 0.99).*

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

(e) *Wikipedia (high contention).*

(f) *RUBiS (high contention).*

Figure 8: *STOv2 performance on Wikipedia and RUBiS workloads.*



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

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

Figure 9: *Cross-system comparisons: STOv2 baselines and other state-of-the-art systems, TPC-C full mix.*

can be attributed to 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.

5.2 Benefits of reordering

Figure 8a (high-contention TPC-C) shows that TSTO, which implements TicToc concurrency control, has an advantage even over MSTO (MVCC). 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 1.3×. However, 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. Note that this particular conflict is a false conflict: the transactions actually access distinct columns in the warehouse table. Both TicToc and MVCC can reduce aborts due to this conflict by rescheduling the new-order transaction to commit with an earlier commit timestamp. This reduces aborts and improves performance, but it generalizes poorly. 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 §7 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.

5.3 Cross-system comparisons

Figure 9 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 9a shows that both MOCC and ERMIA struggle at high contention (the reason is locking overhead). Cicada outperforms both MSTO and OSTO, and matches TSTO's performance at high contention. Cicada implements more optimizations than MSTO. For instance, where other MVCC systems, including MSTO, use a shared, and possibly contended, global variable to assign execution timestamps, Cicada uses "loosely synchronized software clocks", a scalable distributed algorithm based on timestamp counters; and Cicada's "early version consistency check" and "write set sorting by contention" optimizations attempt to abort doomed transactions as soon as possible, thereby reducing wasted work. Nevertheless, Cicada outperforms MSTO by at most 1.25× at all contention levels, and MSTO slightly ouperforms Cicada at low contention (Figure 9b). This contrasts with Cicada's own evaluation, which compared systems with different basis factor choices, and in which Cicada outperformed all other systems, even on low contention benchmarks, by up to 3×. At low contention, however, Cicada's performance collapses at high core counts due to memory exhaustion (Figure 9b). This appears to be an issue with Cicada's special-purpose memory allocator, since there is no exhaustion when that allocator is replaced with jemalloc, the default allocator in DBx1000.

6. HIGH-CONTENTION OPTIMIZATIONS

Our *commit-time update* (CU) and *timestamp splitting* (TS) optimization techniques can eliminate whole classes of conflict from transactional workloads. They improve performance, sometimes significantly, for all of our workloads on OCC, TicToc, and MVCC, as we show in the next section. They also exhibit synergy: on some workloads, TS makes CU far more effective. Our prototype implementations of the these techniques require effort from transaction programmers, as the instantiation of each technique depends on workload. This differs from CC mechanisms such as TicToc and MVCC, which require no effort from transaction programmers. However, the techniques are conceptually general, and applying them to a given workload is not difficult. CU and TS also eliminate classes of conflict that CC protocols cannot, resulting in bigger improvements than can be achieved by CC alone.

6.1 Commit-time updates

The read and write sets central to OCC and MVCC systems represent read-modify-write operations as pairs of reads and writes. An increment, for example, becomes a read of the old value followed by a write of the new value. Unfortunately, this representation can cause conflicts beyond those required by operation semantics. If the result of an increment is not otherwise observed by its transaction, the increment could be represented in terms of concurrency control as a kind of blind write, where the write set contained a notation indicating the value should be incremented when the transaction commits. Since blind writes are subject to fewer conflicts than reads, this design can eliminate many conflicts.

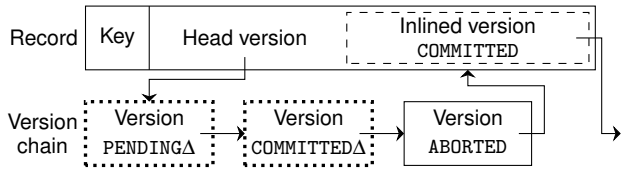
The *commit-time update* feature in STOv2 represents certain read-modify-write operations as varieties of blind write, thus allowing many transactions to avoid semantically unnecessary conflicts. Commit-time updates work for OCC, TicToc, and MVCC. The implementation centers on function objects called *updaters* that act as operations on a record type. A write set component can either

```
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;
};
```

(a) *Commit-time updater for `STOCK` table records in TPC-C's new-order transaction. The `operate` method encodes the operation (stock deduction and replenishment).*



(b) *Record structure in MSTO with commit-time updates. The `COMMITTED`Δ version encodes an updater. Concurrent transactions can insert more delta versions either before or after the `COMMITTED`Δ.*

Figure 10: *Commit-time updates.*

be a value, as in conventional CC, or an updater indicating a read-modify-write. Each updater encodes the operation to be performed on a record and any parameters to that operation. When invoked, it modifies the record according to its encoded parameters. Its execution is isolated to a single record: it may access the record and its encoded parameters, but not any other state. A single transaction may invoke many updaters, however.

Updaters have benefit when they eliminate transactional reads. A transaction can use an updater when a record is updated by read-modify-write, the record is not otherwise observed, and the record does not further affect the transaction's execution, either in terms of control flow or data flow. Here, for example, `T1` could use an updater to modify `x` (the updater would perform the boxed operations), but `T2` should not (part of `x` is returned from the transaction so `x` must be observed):

```
T1:
  tmp = y.col1;
  x.col2 += 1;
  x.col3 = max(tmp, x.col1);
  return tmp;
```

```
T2:
  tmp = y.col1;
  x.col1 += tmp;
  return x.col1;
```

OCC and TicToc updaters are invoked at commit time, in the install phase (see §2.1, Phase 3), while the relevant record is locked. In MVCC, however, updaters are added to the version chain as independent entities called *delta versions*. This preserves transaction ordering flexibility: just as MVCC blind writes can commit out of order (when allowed by read timestamps), delta versions can be added out of order to a version chain. Delta versions contain an updater rather than a materialized record value (Figure 10b). Reading a delta version thus requires a *flattening* procedure, which computes a materialized value by invoking updaters in order, oldest to newest, on a copy of the preceding full (non-delta) version. The resulting materialized value is copied into the delta version, creating a full version ready for reading. Multiple threads can flatten concurrently; delta versions are locked only during the final copy stage. Transactions must also be prevented from inserting delta versions into chains that are concurrently being flattened.

Delta versions impact MSTO's garbage collection, since a version may be marked for deletion only if a newer *full* version exists (a newer delta version does not suffice). MSTO ensures that whenever a full version is created – either directly, through a conventional write, or indirectly, when a read flattens a delta version – all older versions are enqueued for RCU garbage collection. Flattening is periodically applied to infrequently-read records to ensure version chains do not grow without bound.

We evaluate many classes of commit-time updates, such as 64-bit integer addition, integer max, blind writes, and updates specialized for specific TPC-C transactions.

Commit-time updates relate to commutativity, which has long been used to reduce conflicts and increase concurrency in transactional systems [3, 6, 21, 54]. Though they can represent both commutative and non-commutative read-modify-write operations, they do not support some optimizations possible only for commutative operations [37].

Our current MSTO commit-time update implementation is conservative and could be improved. When commit-time updates are enabled, to facilitate safe interactions between updates and reads, MSTO transactions initially execute at timestamp $ts_{th} = rts_g$, the global *read* timestamp, rather than wts_g, the global write timestamp. This forces all reads (which may trigger flattening) to happen before all commit-time modifications (the only operations that insert delta versions), leading to more commit-time invalidations for these reads.

6.2 Timestamp splitting

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. Schema transformations such as row splitting and vertical partitioning [38] use these patterns to reduce database I/O overhead by, for example, only keeping frequently-accessed record fragments in a memory cache. The *timestamp splitting* optimization uses these patterns to avoid classes of contention.

Timestamp splitting divides a record's columns into subsets and assigns one timestamp per subset. When modifying a record, the system updates all timestamps that overlap the modified columns, but when *observing* a record, the system can observe just those timestamps sufficient to cover the observed columns. In a typical example, shown in Figure 11, 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 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 and TSTO implement timestamp splitting by changing records to contain one *or more* timestamps, rather than exactly one timestamp, as shown in Figure 11. MSTO currently implements timestamp splitting in a more heavyweight manner, with vertical partitioning: each record is split into multiple tables with the same primary key. This is more expensive than OSTO's implementation,

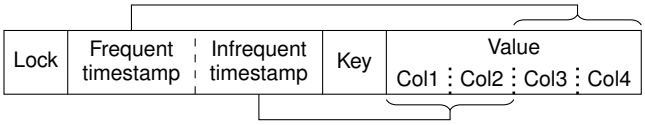


Figure 11: *Record structure in OSTO with timestamp splitting. The frequent timestamp protects the frequently-updated columns, while the infrequent timestamp only updates if col1 or col2 change. This allows transactions that only read col1 and col2 to avoid conflicts with those that only write col3 and col4.*

and we plan to investigate a lighter-weight implementation strategy in future work. Although all systems 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. The implementation also currently requires record subsets to be disjoint.

Timestamp splitting can expose additional commit-time update opportunities. For example, this transaction appears not to benefit from commit-time 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 – `x.col1` is not otherwise observed.

6.3 Implementation in workloads

To implement CU and TS, we manually inspected our workloads. For TS, we generally assign records' frequently-updated columns to a separate timestamp. In YCSB, column access is random and we partition columns evenly into two disjoint timestamps. Transaction programs identify the columns they access, but the column-to-timestamp assignment is handled automatically by our library. For CU, we create an updater type per operation. Some examples: in RUBiS, an updater changes an item's `max-bid` and `quantity` columns; in TPC-C, an updater on warehouse increments its `ytd` (orders year-to-date) field, and one on customer 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.

Commit-time 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.

7. OPTIMIZATIONS EVALUATION

We now evaluate the commit-time 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 CC techniques. We conduct a series of experiments on STOv2 with these optimizations, using all three CC mechanisms, and measure them against TPC-C, YCSB, Wikipedia, and RUBiS workloads.

7.1 Combined effects

Figure 12 shows the effects of applying commit-time updates (CU) and timestamp splitting (TS) together under high and low contention, and on TPC-C and YCSB workloads.

In high-contention TPC-C (Figure 12a), CU+TS greatly improves throughput of all three CC mechanisms, with gains ranging from 2× (TSTO) to 3.9× (OSTO). These gains are larger than those of the CC algorithms alone.

High-contention TPC-C and YCSB-A did not scale to 64 threads under any of our three CC algorithms, but once CU+TS are added, MSTO does scale at least that far (though not perfectly). OSTO and TSTO schemes cannot sustain throughput at high contention due to the inherent limitations of single-version CCs for handling read-only transactions. However, optimized MSTO does not outperform its OSTO or TSTO counterparts until extremely high contention (1-warehouse TPC-C at more than 20 cores, skewed write-heavy YCSB-A at more than 12 cores), and optimized OSTO and TSTO always outperform unoptimized MSTO.

For low-contention TPC-C, CU+TS adds about 10% performance overhead to OSTO and TSTO. This is roughly comparable to the difference between unoptimized TSTO and OSTO at low contention, and is significantly less than the difference between unoptimized MSTO and OSTO. However, CU+TS adds close to 30% overhead for low-contention TPC-C in MSTO. The cause is garbage collection of long delta version chains, specifically warehouse `ytd` values. Garbage collector improvements could potentially reduce this overhead; alternately, perhaps MVCC systems should consider switching off CU when contention is low. In all cases, the added overhead of CU+TS does not affect scalability.

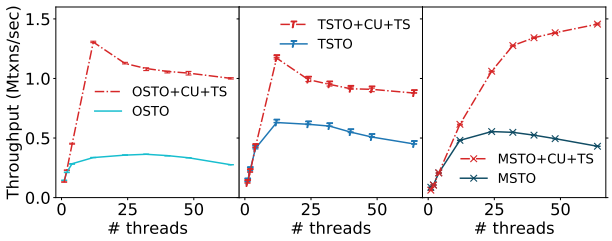
Although YCSB-B is a relatively low-contention benchmark, CU+TS improves the performance of both OSTO and TSTO. Upon investigation, we discovered that CU+TS reduces the amount of data retrieved from and written to the database by accessing only the columns specified. CU+TS still incurs a small overhead for MSTO because (1) TS is implemented using vertical partitioning, which results in additional index lookups, and (2) CU for MSTO introduces delta versions for updates, which incurs allocation and flattening overhead.

CU+TS also benefits all three CCs under the high-contention Wikipedia and RUBiS workloads, as shown in Figures 12e and 12f. In Wikipedia, CU+TS improves performance by 2.6–3.5× at high core counts, while in RUBiS the gains vary from 1.2–1.4×, depending on the underlying CC used.

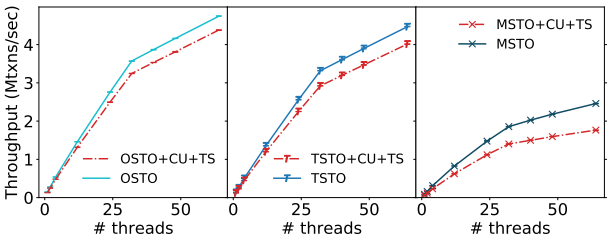
In summary, CU+TS benefits all three CCs measured, and can be applied to benefit many different workloads.

7.2 Separate effects

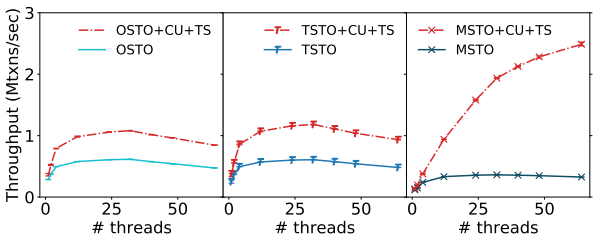
Figure 12g shows the distinct effects of CU and TS on our high-contention benchmarks for OCC and MVCC. In some workloads, such as TPC-C, CU and TS produce greater benefits together than would be expected from their individual performance. This is especially clear for MSTO: CU and TS *reduce* performance when applied individually, but improve performance by 3.38× at 64 threads when applied in combination. This is because many frequently-updated columns can be updated using CU, but only if the infrequently-updated column values use a separate timestamp. Of the two optimizations, CU is more frequently useful on its own. For instance, the highest overall performance for Wikipedia is obtained by applying CU to OSTO. This is an indication that write-write conflicts are predominant in these workloads, since CU reduces the impact of write-write conflicts while TS reduces the impact of read-write false sharing.



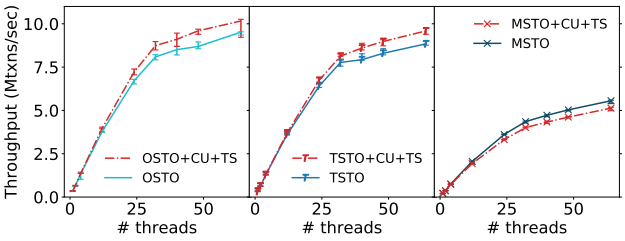
(a) *TPC-C, one warehouse (high contention).*



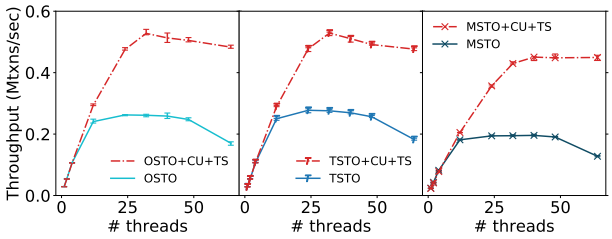
(b) *TPC-C, one warehouse per worker (low contention).*



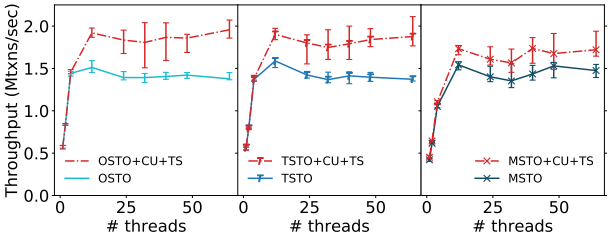
(c) *YCSB-A (high contention: update-intensive, 50% updates, skew 0.99).*



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



(e) *Wikipedia (high contention).*



(f) *RUBiS (high contention).*

Benchmark	OSTO	OSTO+CU	OSTO+TS	OSTO+CU+TS	MSTO	MSTO+CU	MSTO+TS	MSTO+CU+TS
TPC-C	276	286 (1.04×)	432 (1.57×)	1001 (3.63×)	431	269 (0.62×)	410 (0.95×)	1456 (3.38×)
YCSB	473	855 (1.81×)	466 (0.99×)	844 (1.78×)	326	1851 (5.68×)	687 (2.11×)	2487 (7.64×)
Wikipedia	170	487 (2.86×)	167 (0.98×)	483 (2.84×)	128	311 (2.43×)	128 (1.01×)	449 (3.52×)
RUBiS	1378	1924 (1.40×)	1368 (0.99×)	1957 (1.42×)	1475	1692 (1.15×)	1505 (1.02×)	1721 (1.17×)

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

Figure 12: *STOv2 performance with commit-time updates and timestamp splitting (CU+TS).*

CU alone performs badly on some MSTO benchmarks; on high-contention TPC-C, MSTO+CU achieves 0.62× the throughput of MSTO alone. This is due to our current conservative implementation: when CU is enabled MVCC transactions execute in the recent past and experience more conflicts and more aborts. A prototype CU implementation that executes MVCC+CU transactions at the current timestamp, rather than the recent past, performs better, achieving 0.95× the throughput of MSTO alone. CU+TS performs equally well in both systems, since TS allows even the conservative CU to avoid all relevant conflicts.

8. FUTURE WORK

In future work, we hope to investigate the remaining bottlenecks in STOv2's performance. We plan to focus on the MSTO implementation of both timestamp splitting and commit-time updates; in addition MSTO might benefit from garbage collection improvements and additional Cicada optimizations. In OSTO the transaction processing machinery accounts for just 4.2% of the total runtime in low-contention TPC-C, leaving little room for further improvement.

Additionally, we believe that static analysis could help identify potential hotspots for false sharing in indexes and database records. This could lead to tools that more fully automate the application of commit-time updates and timestamp splitting.

9. RELATED WORK

9.1 Modern concurrency control research

Concurrency control is a central issue for databases and work goes back many decades [18]. As with many database properties, the best concurrency control algorithm can depend on workload, and OCC has long been understood to work best for workloads "where transaction conflict is highly unlikely" [27]. Since OCC transactions cannot prevent other transactions from executing, OCC workloads can experience starvation of whole classes of transactions. Locking approaches, such as two-phase locking (2PL), lack this flaw, but write more frequently to shared memory. Performance tradeoffs between OCC and locking depend on technology characteristics as well as workload characteristics, however, and on multicore main-memory systems, with their high penalty 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 [49,58] introduced an OCC protocol that, unlike other implementations [8, 27], lacked any per-transaction contention point, such as a shared timestamp counter. Though Silo addressed some starvation issues by introducing snapshots for read-only transactions, and showed some reasonable results on a high-contention workload, 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 §4.

Since Silo, many new concurrency control techniques have been introduced. We concentrate on those that aim to preserve OCC's low-contention advantages and mitigate its high-contention flaws.

TicToc's additional read timestamp allows it to commit some apparently-conflicting transactions by reordering them [57]. Timestamp maintenance becomes more expensive than OCC, but reordering has benefits for high-contention workloads. We present results for our implementation of TicToc.

Transaction batching and reordering [11] 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. We consider our workload-specific optimizations orthogonal to these techniques as our optimizations eliminate unnecessary dependency edges altogether instead of working around them.

Hybrid concurrency control in MOCC [50] and ACC [46] uses online conflict measurements and statistics to switch between OCC-like and locking protocols dynamically. Locking can be expensive (it handicaps MOCC in our evaluation), but prevents starvation.

MVCC [4, 41] systems, such as ERMIA [24] and Cicada [31], keep multiple versions of each record. The multiple versions allow more transactions to commit through reordering, and read-only transactions can *always* commit. 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 and less conservative than OCC-style read set validations, and to allow more transaction schedules to commit. The SSN mechanisms in ERMIA, however, involve expensive global thread registration and deregistration operations that limited its scalability [50]. In our experiments, ERMIA's locking overhead – a kind of basis factor – 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 in both low- and high- contention situations. This disagrees with our results, which show OSTO outperforming Cicada at low contention (Figure 9b). We believe the explanation involves basis factor choices in Cicada's OCC comparison systems. Our MSTO MVCC system is based on Cicada, though we omit several of its optimizations.

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 commit-time updates and timestamp splitting can alleviate these conflicts.

Static analysis can improve the performance of high-contention workloads, since given an entire workload, a system can discover equivalent alternative executions that generate many fewer conflicts. Transaction chopping [44] uses global static analysis of all possible transactions to break up long-running transactions such that subsequent pieces in the transaction can be executed conflict-free. More recent systems like IC3 [51] combine static analysis with dynamic admission control to support more workloads. Static analysis techniques are complementary to our work, and we hope eventually to use static analysis to identify and address false sharing in secondary indexes and database records, and to automate the application of commit-time updates and timestamp splitting.

9.2 Basis factors

Several prior studies have measured the effects of various basis factors on database performance. A recent study found that a good memory allocator alone can improve analytical query processing performance by 2.7× [14]. A separate study presented a detailed evaluation of implementation and design choices in main-memory database systems, with a heavy focus on MVCC [55]. 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 [17] 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, such as how a transaction restarts and system resource considerations, are analogous to basis factors we identified 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.

9.3 High-contention optimizations

Our commit-time update and timestamp splitting optimizations have extensive precursors in other work. Timestamp splitting resembles row splitting, or vertical partitioning [38], which splits records based on workload characteristics to optimize I/O. Taken to an extreme, row splitting leads to column stores [28, 45] or attribute-level locking [32]. 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.

Commutativity has long been used to improve concurrency in databases, file systems, and distributed systems [3,26,37,42,43,54], with similar effects on concurrency control as commit-time updates. We know of no other work that applies commutativity or commit-time updates to MVCC records, though many systems reason about the commutativity properties of modifications to MVCC indexes. Upserts in BetrFS [22, §2.2] resemble how we encode commit-time 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 [19] involve techniques and data structures that resemble commit-time updates, though their goal is to reduce I/O bandwidth usage in an read-mostly OLAP system.

9.4 Transactional memory

Extensive experience with transactional system implementation is also found in the software transactional memory space [9,12,20]; there are even multiversion STMs [5, 16]. Efficient STMs can run main-memory database workloads, and we base our platform on one such system, STO [21]. Some of our baseline choices were inspired by prior STM work, such as SwissTM's contention regulation [12]. STO's type-aware concurrency control included preliminary support for commit-time updates and timestamp splitting, but only for OCC.

STO has also been used as a baseline for other systems that address OCC's problems on high-contention workloads, such as DRP [36]. DRP effectively changes large portions of OCC transactions into commit-time updates by using lazy evaluation, automatically implemented by C++ operator overloading, to move most computation into OCC's commit phase. This 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 (HTM) [29, 52, 53]. HTM can also be used to implement efficient deadlock avoidance as an alternative to bounded spinning [52].

10. 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 – commit-time updates and timestamp splitting – are arguably more powerful than concurrency control algorithms. CU+TS can improve performance by up to 3.6× 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 CU+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.

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 `vldb20` tag:

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

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.

11. REFERENCES

[1] N. Abramson. The Aloha system: Another alternative for computer communications. In *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference*, AFIPS '70 (Fall), pages 281–285. ACM, 1970.

[2] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems (TODS)*, 12(4):609–654, 1987.

[3] B. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems (TODS)*, 17(1):163–199, 1992.

[4] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[5] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[6] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17. ACM, 2013.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SOCC '10, pages 143–154. ACM, 2010.

[8] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proceedings of the 2013 International Conference on Management of Data*, SIGMOD '13, pages 1243–1254. ACM, 2013.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, DISC '06, pages 194–208. Springer, 2006.

[10] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[11] B. Ding, L. Kot, and J. Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *PVLDB*, 12(2):169–182, 2018.

[12] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 155–165. ACM, 2009.

[13] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, pages 401–414. ACM, 2014.

[14] D. Durner, V. Leis, and T. Neumann. 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.

[15] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11):1190–1201, 2015.

[16] S. Fernandes and J. Cachopo. A scalable and efficient commit algorithm for the JVSTM. In *Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing*, Apr. 2010.

[17] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 258–264. ACM, 2005.

[18] G. Held, M. Stonebraker, and E. Wong. INGRES: a relational data base system. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 409–416. ACM, 1975.

[19] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 543–554. ACM, 2010.

[20] M. Herlihy and E. Koskinen. 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, pages 207–216. ACM, 2008.

[21] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys '16. ACM, 2016.

[22] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, et al. BetrFS: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies*, FAST '15, pages 301–315. ACM, 2015.

[23] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, Aug. 2008.

[24] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1675–1687. ACM, 2016.

[25] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 International Conference on Management of Data*, SIGMOD '15, pages 691–706. ACM, 2015.

[26] H. F. Korth. Locking primitives in a database system. *Journal of the ACM (JACM)*, 30(1):55–79, 1983.

[27] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[28] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica analytic database: C-Store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

[29] V. Leis, A. Kemper, and T. Neumann. 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*, pages 580–591, 2014.

[30] H. Lim. Line comment in experiment script (run_exp.py). Available at `https://github.com/efficient/cicada-exp-sigmod2017/blob/5a4db37750d1dc787f71f22b425ace82a18f6011/run_exp.py#L859`, Jun 2017.

[31] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 International Conference on Management of Data*, SIGMOD '17, pages 21–35. ACM, 2017.

[32] K. S. Maabreh and A. Al-Hamami. Increasing database concurrency control based on attribute level locking. In *2008 International Conference on Electronic Design*, pages 1–4. IEEE, 2008.

[33] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th European Conference on Computer Systems*, EuroSys '12, pages 183–196. ACM, 2012.

[34] P. E. McKenney and S. Boyd-Wickizer. RCU usage in the Linux kernel: One decade later. Technical report, 2012.

[35] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[36] S. Mu, S. Angel, and D. Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of the 14th European Conference on Computer Systems*, EuroSys '19, pages 40:1–40:16. ACM, 2019.

[37] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 511–524. ACM, 2014.

[38] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)*, 9(4):680–710, 1984.

[39] OW2 Consortium. RUBiS. Available at `https://rubis.ow2.org/`.

[40] Rampant Pixels. rpmalloc - rampant pixels memory allocator. Available at `https://github.com/rampantpixels/rpmalloc`, Apr 2019.

[41] D. P. Reed. *Naming and synchronization in a decentralized computer system.* PhD thesis, Massachusetts Institute of Technology, 1978.

[42] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract data types. 1983.

[43] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[44] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)*, 20(3):325–363, 1995.

[45] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-Store: a column-oriented DBMS. *PVLDB*, pages 553–564, 2005.

[46] D. Tang, H. Jiang, and A. J. Elmore. 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.

[47] Transaction Processing Performance Council. TPC benchmark C. Available at `http://www.tpc.org/tpcc/`.

[48] 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`, Feb 2010.

[49] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32. ACM, 2013.

[50] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2):49–60, 2016.

[51] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1643–1658. ACM, 2016.

[52] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys '14, pages 26:1–26:15. ACM, 2014.

[53] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104. ACM, 2015.

[54] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.

[55] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *PVLDB*, 10(7):781–792, 2017.

[56] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.

[57] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. TicToc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642. ACM, 2016.

[58] W. Zheng, S. Tu, E. Kohler, and B. Liskov. 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, pages 465–477. ACM, 2014.