An Analysis of Concurrency Control Protocols for In-Memory Databases with CCBench

Takayuki Tanabe[1][*], Takashi Hoshino[2], Hideyuki Kawashima[3], Osamu Tatebe[4]

[1]NAUTILUS Technologies, Inc., [2]Cybozu Labs, Inc., [3]Faculty of Environment and Information Studies,
Keio University, [4]Center for Computational Sciences, University of Tsukuba
[1]tanabe@nautilus-technologies.com, [2]hoshino@labs.cybozu.co.jp,
[3]river@sfc.keio.ac.jp, [4]tatebe@cs.tsukuba.ac.jp

ABSTRACT

This paper presents yet another concurrency control analysis platform, CCBench. CCBench supports seven protocols (Silo, TicToc, MOCC, Cicada, SI, SI with latch-free SSN, 2PL) and seven versatile optimization methods and enables the configuration of seven workload parameters. We analyzed the protocols and optimization methods using various workload parameters and a thread count of 224. Previous studies focused on thread scalability and did not explore the space analyzed here. We classified the optimization methods on the basis of three performance factors: CPU cache, delay on conflict, and version lifetime. Analyses using CCBench and 224 threads, produced six insights. (I1) The performance of optimistic concurrency control protocol for a read-only workload rapidly degrades as cardinality increases even without L3 cache misses. (I2) Silo can outperform TicToc for some write-intensive workloads by using invisible reads optimization. (I3) The effectiveness of two approaches to coping with conflict (wait and no-wait) depends on the situation. (I4) OCC reads the same record two or more times if a concurrent transaction interruption occurs, which can improve performance. (I5) Mixing different implementations is inappropriate for deep analysis. (I6) Even a state-of-the-art garbage collection method cannot improve the performance of multi-version protocols if there is a single long transaction mixed into the workload. On the basis of I4, we defined the read phase extension optimization in which an artificial delay is added to the read phase. On the basis of I6, we defined the aggressive garbage collection optimization in which even visible versions are collected. The code for CCBench and all the data in this paper are available online at GitHub.

---

[*]Work done while at University of Tsukuba

1. INTRODUCTION

1.1 Motivation

Transacting processing systems containing thousands of CPU cores in a single server have been emulated [63], implemented [33,57], and analyzed [14]. Along with these systems, a variety of concurrency control protocols for a single server [32,35,53,57,64] have been proposed in recent years for use in many-core architectures. These modern protocols use a variety of optimization methods, and their behaviors depend on the workload characteristics (e.g., thread count, skew, read ratio, cardinality, payload size, transaction size, and read-modify-write or not). Recent new proposal studies have compared these protocols with conventional ones [20,22,23,25,27,29,32,33,35,44,51,53,57,58,61,62,64,65], and recent analytical studies have compared the performance of conventional protocols on a common platform [13,14,60,63]. These studies mostly evaluated protocol scalability. This paper acknowledges that modern protocols are scalable and focuses on other factors that contribute to performance on a many-core environment. This type of analysis is novel to the best of our knowledge.

Fairness is important in comparison. However, some recent studies were unable to perform such analysis fairly because they compared their new protocols with others using different platforms. For example, evaluations of ERMIA [32], mostly-optimistic concurrency control (MOCC) [57], and Cicada [35] used two or three platforms. Experiments using a mix of platforms can produce only approximate results because the performance of protocols on a many-core architecture depends greatly on the exploitation of the underlying hardware. In such conditions, only the macroscopic evaluations (i.e., scalability) can be conducted, and a detailed analysis is difficult. A single unified comparison platform is needed to conduct a detailed analysis.

For a fair comparison, a common analysis platform is necessary. It should provide shared core modules such as access methods (concurrent index) and thread-local data structures (read and write sets) for any protocols. It should also provide a variety of optimization methods to enable obtaining a deep understanding of the protocols. Finally, the platform should be publicly available for reproducing the experiments. Although there are several open-source platforms, including DBx1000 [6], Peloton [8], and Cavalia [4], they do not provide certain modern protocols [35, 56, 57, 64]. Appuswamy et al. [13] evaluated concurrency control protocols in four types of architecture using Trireme, which is not publicly available.

1.2 Contributions

The first contribution of this paper is **CCBench**: a platform for fairly evaluating concurrency control protocols. The protocols are two-phase locking (2PL), Silo, MOCC, TicToc, snapshot isolation (SI), latch-free serial safety net (SSN), and Cicada. CCBench supports seven versatile optimization methods and two identified in this work (Table 1, §3.2). Optimization methods can thus be applied to non-original protocols. For example, the NoWait method [18] can be applied to Silo, the rapid garbage collection (GC) method [35] can be applied to multi-version protocols, and the AdaptiveBackoff optimization method [35] can be applied to all protocols. Fairness in analyzing protocols is achieved through shared modules including a workload generator, access methods (Masstree [7, 36]), local datasets (read/write sets), and a memory manager (mimalloc [11] and AssertiveVersionReuse presented in this paper). Evaluation of protocols under various conditions is enabled by providing of seven workload configuration parameters: skew, payload size, transaction size, cardinality, read ratio, read-modify-write (RMW) or not, and the number of worker threads. CCBench and all of the data in this paper are available online at GitHub [5].

The second contribution is **an analysis of cache, delay, and version lifetime using 224 threads.** We clarified the effects of the protocols on the performance factors by configuring the optimization methods and workload settings. As suggested elsewhere [63] and [33], the era of a thousand cores is just around the corner, so conventional analytical studies focused on evaluating thread scalability [28, 60, 63]. In contrast, we performed all analyses with *224 threads on 224 cores.*

We first investigated the effects of the optimization methods related to **cache**. By determining that a centralized counter increases cache-line conflicts and degrades performance, we gained two insights. **I1:** The performance of optimistic concurrency control (OCC) for a read-only workload rapidly degrades as cardinality increases even without L3 cache misses (§5.2). **I2:** Silo outperforms TicToc for write-intensive workloads by using InvisibleReads (§5.3).

We then investigated the effects of the optimization methods related to **delay**, and gained two more insights. **I3:** The effectiveness of two approaches to cope with conflict (Wait and NoWait) depends on the situation (§6.1). **I4:** OCC reads the same record two or more times due to concurrent transaction interruption. Surprisingly, OCC can improve performance in certain situations with it, and we have defined a new optimization method ReadPhaseExtension based on it (§6.2).

Finally, we investigated the effects of optimization methods related to **version lifetime** for multi-version concurrency control (MVCC) protocols, and gained two final insights. **I5:** Mixing different implementations is inappropriate. Silo outperforms Cicada on the Yahoo! Cloud Serving Benchmark B (YCSB-B) workload in an unskewed environment, which is inconsistent with previously reported testing results on different systems [35]. **I6:** Even a state-of-the-art GC technique cannot improve the performance of MVCC if there is a single long transaction mixed into the workload. To overcome this problem, we defined a new optimization method, AggressiveGC. It requires an unprecedented protocol that weaves GC into MVCC, thus going beyond the current assumption that versions can not be collected if they might be read by transactions (§7.3).

1.3 Organization

The rest of this paper is organized as follows. §2 reviews existing protocols. §3 discusses the fairness condition and presents CCBench. §4 describes the reproduction of experiments. §5 investigates the effect of cache. §6 investigates the effect of delay. §7 investigates the effect of version lifetime management. §8 reviews related work, and §9 concludes this paper and shows future directions.

2. PRELIMINARIES

2.1 Concurrency Control Protocols

The concurrency control protocols we analyzed are classified as (1) pessimistic (2PL [24]), (2) optimistic (Silo [53] and TicToc [64]), (3) multi-version (SI [31] and ERMIA [32]), (4) integration of optimistic and pessimistic (MOCC [57]), and (5) integration of optimistic and multi-version (Cicada [35]).

Silo [53] is an OCC protocol that has influenced subsequent concurrency control protocols. For example, FOE-DUS [33] and MOCC [57] extend the commit protocol of Silo. Silo selects the design of InvisibleReads [37] so that it does not update the metadata during a read operation. The invisible read process avoids cache-line conflicts, so it provides scalability, as shown by Wang and Kimura [57].

TicToc [64] is an OCC protocol based on timestamp ordering with data-driven timestamp management. TicToc has a larger scheduling space than Silo, so it can commit schedules where Silo cannot. TicToc provides three optimization methods. PreemptiveAbort initiates abort processing immediately if an abort is detected before write locking in the validation phase. NoWaitTT does not wait for lock release in the validation phase. It instead releases the locks and retries the validation phase after a fixed duration of a wait without an abort. TimestampHistory expands the scheduling space by recording the write timestamp of an older version so that some of the read operations on the version can be verified after the record is overwritten.

MOCC [57] exhibits high performance for a wide range of workloads by adaptively switching its policy by adding pessimistic locking and temperature management to Silo. MOCC locks high temperature (on highly conflicted) records while keeping the order of records locked to avoid deadlocks in the read phase. The MOCC queuing lock (MQL) integrates an MCS (Mellor-Crummey and Scott) lock, which can time out [46], and a reader/writer fairness MCS lock [38].

Cicada [35] combines the OCC protocol, a multi-version protocol with timestamp ordering, and distributed timestamp generation. The distributed timestamp generation eliminates the need to access the centralized counter that conventional MVCC protocols require, thereby dramatically mitigating cache-line conflicts. Cicada has a variety of optimization methods. EarlyAbort initiates an abort if one is predicted during the read phase. BestEffortInlining embeds an inline version in the record header at the top of the version list to reduce indirect reference cost. PrecheckValidation checks whether read versions are still valid in the early validation phase. RapidGC is a quick and parallel GC optimization method. AdaptiveBackoff dynamically determines how long to wait before retrying. SortWriteSetByContention detects conflicts at an early stage before performing actions unnecessarily due to aborts.
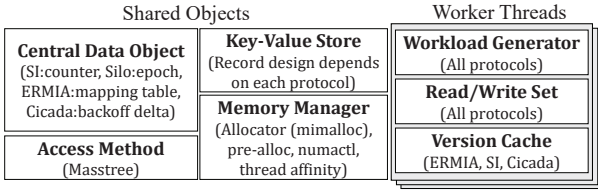
Central Data Object (SI:counter, Silo:epoch, ERMIA:mapping table, Cicada:backoff delta)	Key-Value Store (Record design depends on each protocol)	Workload Generator (All protocols)
	Memory Manager (Allocator (mimalloc), pre-alloc, numactl, thread affinity)	Read/Write Set (All protocols)
Access Method (Masstree)		Version Cache (ERMIA, SI, Cicada)

Figure 1: CCBench Architecture.

SI [16] is an MVCC protocol that can generate write-skew and read-only-transaction anomalies, so it does not produce serializable schedules. Under SI, a transaction reads a snapshot of the latest committed version. The write operations are also reflected in the snapshot. SI requires a monotonic-increasing timestamp assignment for each transaction to provide snapshots. To determine the timestamp, a centralized shared counter is required.

SI with latch-free SSN [32] integrates SI and SSN [55] and exploits the latch-free mechanism [56]. SSN detects and aborts dangerous transactions that could lead to non-serializable schedules. We refer to the integration of SI and SSN as **ERMIA** in this paper.

2PL [24, 59] releases all read/write locks at the end of each transaction. The NoWait method, which immediately aborts the running transaction when a conflict is detected, was originally proposed as a deadlock resolution mechanism [17]. We use it as an optimization method.

2.2　Environment

The evaluation environment consisted of a single server with four Intel (R) Xeon (R) Platinum 8176 CPUs with 2.10GHz processors. Each CPU had 28 physical cores with hyper-threading, and the server had 224 logical cores. Each physical core had 32 KB private L1d cache and 1 MB private L2 cache. The 28 cores in each processor shared a 38.5 MB L3 cache. The total cache size was about 154 MB. Forty-eight DDR4-2666 32 GB DIMMs were connected, so the total size was 1.5 TB.

In all graphs in this paper showing the results of CCBench, each plot point shows the average for five runs, each with more than 3 s. We confirmed that these numbers produce stable CCBench results. The error bars indicate the range between the maximum and minimum values. We counted the number of committed and aborted transactions to calculate average throughput. We used `rdtscp` instruction to measure the latency of the transitions and their portions. Each worker thread was pinned to a specific CPU core in every experiment.

3.　CCBENCH

3.1　Fairness Condition

When analyzing concurrency control protocols, their performance should be evaluated under a fairness condition. The meaning of fairness depends on the situation. In this paper, fairness means that **basic modules are shared in the analysis**. This is because the performance of modern protocols is closely related to the exploitation of the underlying hardware; i.e., they are sensitive to the engineering

details. Therefore, the basic modules of an evaluation platform should be shared for a fair evaluation. Access methods (e.g., Masstree) and local data structures (read and write sets) need to be shared among protocols. The effects of memory allocation should be reduced as much as possible. The workload should be consistent among experiments.

Several analysis studies satisfy our fairness condition. Yu et al. [63] developed DBx1000 [6]. It was used in the evaluation of TicToc paper [64] and in another evaluation with a real 1568 cores machines [14]. DBx1000 currently does not support some modern protocols (Cicada, MOCC, ERMIA). Wu et al. developed Cavalia [4] to compare the hardware transactional memory (HTM)-assisted OCC-style protocol (HTCC) [62] with conventional protocols. Several modern protocols (Cicada, ERMIA, TicToc, MOCC) were beyond their scope. Wu et al. [60] developed Peloton [8] to compare MVCC protocols. Several modern protocols (Cicada, ERMIA, Silo, TicToc, MOCC) were beyond their scope. Appuswamy et al. [13] developed Trireme to compare protocols in four types of architecture, including shared-everything. Several modern protocols (TicToc, Cicada, MOCC, ERMIA) were beyond their scope.

A protocol can include a variety of optimization methods. Even if protocol P does not initially include optimization method O, an analysis platform should provide O to P if users request it because the performance of a protocol greatly depends on the optimization method. For example, RapidGC included in Cicada can also be applied to both SI and ERMIA. NoWait [17] can be applied to Silo to improve performance, as shown in Fig. 10c. Conventional platforms do not support this concept.

Our fairness condition was not satisfied in several studies. The Cicada paper (§4.2) [35] states, "Cicada and existing schemes in DBx1000 share the benchmark code, but have separate data storage and transaction processing engines." The MOCC paper [57] states that the pure OCC, MOCC, and PCC (2PL variants) implementations used the FOE-DUS system, the Dreadlock/WaitDie/BlindDie (2PL variants) implementations used the Orthrus system, and the ERMIA implementation used the ERMIA system, which means three benchmark systems were mixed. The ERMIA paper [32] states that the Silo implementation used the Silo system while the SI/SSI implementation used the ERMIA system, which means two benchmark systems were mixed. In studies using a mix of systems, experts must conduct a huge amount of effort to conduct a fair comparison, which would not be easy, especially in a deep understanding of protocols or optimizations.

3.2　CCBench: A Platform for Fair Analysis

Our CCBench analysis platform for in-memory CC protocols satisfies our fairness condition because it shares the basic modules among protocols. The architecture of CCBench is illustrated in Fig. 1. The code for CCBench is available on GitHub [5].

In CCBench, each thread executes both the client and the server logic. The client generates a transaction with read/write operations using the workload parameters at runtime. The server provides APIs as C++ functions, such as read, write, commit, and abort. The client calls the APIs to run transactions. The server runs the transactions requested from the client inside a worker thread. The client code is separated from the server code, and the both codes

Table 1: Versatile optimization methods in CCBench. —: Irrelevant or Incompatible. Org: Supported by original protocol. CCB: Supported by CCBench. (α) Delay inspired by extra reads in OCC (§6.2). Proposed in this work. (β) CCBench performs read lock for hot records and invisible reads for non-hot records. (γ) NoWait locking optimization detects a write lock conflict and immediately aborts and retries the transaction. NoWaitTT releases all locks and re-locks them. (δ) Lightweight memory management. Proposed in this work (§3.3). (ϵ) PreempiveAbort and TimestampHistory cannot be applied to others. (ζ) SortWriteSetByContention, PrecheckVersionConsistency, EarlyAborts, BestEfforInlining cannot be applied to others. We applied optimization methods as follows. NoWait: Silo in Fig. 10c. RapidGC: ERMIA and SI in all cases. AssertiveVersionReuse: Cicada, ERMIA, and SI in all cases. AdaptiveBackoff: TicToc in Fig. 2c and all protocols in Figs. 4a, 4b, 4c, and 4d.

Performance Factor	Cache		Delay			Version Lifetime	
Optimization Method	Decentralized Ordering	Invisible Reads	NoWait or Wait	Adaptive Backoff	ReadPhase Extension(α)	AssertiveVersion Reuse(δ)	Rapid GC
2PL [59]	Org, CCB	—	Org,CCB	CCB	—	—	—
Silo [53]	Org, CCB	Org, CCB	CCB	CCB	CCB	—	—
MOCC [57]	Org, CCB	Org, CCB (β)	—	CCB	CCB	—	—
TicToc(ϵ) [64]	Org, CCB	—	CCB (γ)	CCB	CCB	—	—
SI [31]	—	—	—	CCB	—	CCB	CCB
ERMIA [32]	—	—	—	CCB	—	CCB	CCB
Cicada(ζ) [35]	Org, CCB	—	—	Org, CCB	CCB	CCB	Org, CCB

are compiled into a single executable binary. The memory allocator, mimalloc [11], allocates interleaved memory among CPU sockets by using the Linux `numactl` command. Memory allocation is avoided as much as possible so that the penalty [19] imposed by the Linux virtual memory system can be ignored and so that the execution performance of protocols is not degraded due to undesirable side-effects of memory allocation. CCBench initializes the database for each experiment, pre-allocates the memory for objects, and reuses the memory. Allocated memory for local data structures (read/write sets) and the generated operations for a transaction are reused for the next transaction. The metadata and record data are carefully aligned to reduce false sharing. A wrapper of Masstree [7] was implemented, and all protocols used it as the access method.

CCBench supports seven versatile optimization methods, as shown in Table 1. (1) DecentralizedOrdering: prevents contended accesses to a single shared counter. (2) InvisibleReads: read operations that do not update memory and cache-line conflicts do not occur. (3) NoWait or Wait: immediate abort upon detecting a conflict followed by retry, or waiting for lock release. (4) ReadPhaseExtension: an artificial delay added to the read phase, inspired by the extra read process that retries the read operation. (5) AdaptiveBackoff: an artificial delay before restarting an aborted transaction. (6) AssertiveVersionReuse: allocates thread-local space, denoted as version cache in Fig. 1, to retain versions so that memory manager access is not needed. (7) RapidGC: frequent updating of timestamp watermark for GC in MVCC protocols. ReadPhaseExtension (4) and AssertiveVersionReuse (6) are presented in this paper.

CCBench enables the use of additional optimization methods for 2PL, Silo, TicToc, SI, ERMIA, and Cicada. It does not support the read-only optimization methods introduced in Silo, FOEDUS, MOCC, and ERMIA because they improve performance only in a specific case (i.e., 99% read-only transactions, 1% write transactions, skew at 0.99), and such a workload is not considered here.

Users of CCBench can easily add new protocols to the platform. After replicating a directory that has an existing protocol, the user rewrites the functions corresponding to the transactions with begin/read/write/commit/abort operations in the transaction executor class. The user can then reuse the basic modules in the platform: workload generator, memory management, and so on. Users can easily attach or detach optimization methods provided by CCBench to their protocols simply by rewriting the preprocessor definition in the CMakeLists.txt. A developer guide for CCBench users is available [1]. One of our team members implemented a simple OCC protocol following this guide [3]; and published a description of the experience for other users [10]. Users can also switch the configuration of the optimization methods, key-value size, and protocol details in the same way as done in the online example. In an experiment, the user can set the workload by specifying the runtime arguments by gflags [9] using the seven workload parameters listed in Table 2. This design makes it easier to conduct experiments than DBx1000 [6], which manages these values as preprocessor definitions in a file.

3.3 Optimization Method Implementations

2PL: For efficiency, we implemented from scratch the reader/writer lock system with the compare-and-swap (CAS) operation used in the protocol.

Silo: We padded the global epoch (64 bits) into the cache-line size to reduce the false sharing that can be caused by a global epoch state change. Calculation of the commit transaction id (TID) is expensive since it requires loops for both the read set and the write set. We reduced these two loops by implementing calculation at write lock acquisition and read verification.

ERMIA: We used latch-free SSN to prevent expensive SSN serial validation. This implementation avoids the validation from being a critical section. We integrated pstamp and sstamp to reduce memory usage as described at Section 5.1 of original paper [56]. We optimized the transaction mapping table. Straightforwardly, the mapping table is designed as a two-dimensional global array with a thread number, TID, cstamp, and last cstamp, which requires a lock manager for updates and read over the rows. This degrades performance due to serialization. Therefore, we designed a data structure that expresses the rows in a one-dimensional array so that it can be latched with a CAS operation. The performance was improved by avoiding seri-

alization and cache-line conflicts for row element access. We used RapidGC [35]. The transaction mapping table objects exploited our AssertiveVersionReuse method, so the memory manager almost did not need to work during experiments.

TicToc: We implemented all three optimization methods (NoWaitTT, PreemptiveAbort, TimestampHistory). We also improved algorithm 5 in the original paper [64] by removing the redundant read timestamp updates. This is effective because if the recorded read timestamp is the same as the previous one, it does not need to be updated.

MOCC: MOCC periodically resets temperature information [57], which switches many cache-lines to the invalid state simultaneously and thereby degrades performance. In contrast, our implementation stores the epoch ID of the latest reset timing and temperature together and thereby avoids multiple resets in the same epoch. This reduces the cost of cache-line conflicts and the number of reset operations, thereby maintaining fine control of the temperature. We did not implement MQL since our environment had only four NUMA (non-uniform memory access) nodes, so the effect would be limited. We instead used the reader/writer lock that was used in our 2PL implementation.

Cicada: We implemented all six optimization methods: SortWriteSetByContention, PrecheckVersionConsistency, AdaptiveBackoff, EarlyAborts, RapidGC, and BestEffortInlining. Moreover, we fixed a logical bug, i.e., the incomplete version consistency check in the validation phase. In the original paper [35], only the read version is confirmed to still be the visible version. However, whether the latest version is still the visible version needs to be confirmed as well. The existence of a newer version in the observable range means that it should have been read and the transaction view will thus be broken. We turned off the one-sided synchronization to improve throughput.

CCBench: In the MVCC protocols (SI, ERMIA, and Cicada), creating new versions and deleting old versions put a load on the memory manager. We therefore developed a new optimization method dubbed, AssertiveVersionReuse that avoids overloading the memory manager in MVCC protocols. This method enables each worker thread to maintain a container for future version reuse. When GC begins, the versions collected are stored in this container. A new version is taken from this container except if it is empty. If it is empty, a request for space is sent to the memory manager as usual. The space needed for GC is estimated before the experiment, and memory space is allocated in advance. This optimization minimizes the burden on the memory manager for the MVCC protocols. Moreover, we introduce the use of ReadPhaseExtension to delay execution. It was inspired by the extra read process described in §6.2. We also introduce AggressiveGC optimization, which collects even visible versions, described in §7.3.

3.4 Configurable Workloads

CCBench supports the seven parameters shown in Table 2. Skew is an access pattern that follows a Zipf distribution. Cardinality is the number of records in the database. The payload is the size of a record (key plus value). Transaction size is the number of operations in a transaction. Read ratio is the ratio of reads in a transaction. Write model is whether to perform RMW or not. Thread count is the number of worker threads (fixed to 224 here except for reproduction). To determine the skew, CCBench uses a fast

Table 2: Analyzed parameter sets. (α) Cardinality (from 10^3 to 10^9 records); (β) Cardinality (10^3 or 10^6 records); (γ) Read ratio (from 0% to 100%); (δ) Transaction size (from 10 to 100 operations); (ϵ) Payload size (from 4 to 1000 bytes); (ζ) Skew (from 0.6 to 0.99).

|	Cache		Delay		Version
Figure Number	F5	F7	F9	F10	F11
Skew	0	0	0.8	0.9	ζ
Cardinality	α	β	10^8	10^8	10^8
Payload (byte)	4	4	4	ϵ	4
Xact size	10	10	δ	10	10
Read ratio (%)	0,5	γ	50,95	50	50,95
Thread count	Always 224 except from reproduction				
Read modify write	Always off except from reproduction				

approximation method [26]. We analyzed CC protocols by using 224 threads using CCBench. Most of the benchmarks were the YCSB workload, and some were variants of YCSB. The analyzed parameter sets are summarized in Table 2. We varied the skew, cardinality, payload, transaction size, and read ratio and fixed the thread count at 224. The caption in the table describes the variance in parameters using $\alpha \ldots \zeta$[1].

This paper focuses on identifying factors that significantly affect performance in a highly parallel environment. We have chosen YCSB like benchmarks because they can generate various workloads despite the simpleness of its data model which offers only read and update operations for a single table with the primary index. It is preferred that a benchmark also supports realistic workloads, which often contain insertion, deletion, range search, and secondary indexes for multiple tables. Difallah et al. [21] summarized such workloads, which included an industry-standard benchmark, TPC-C [12]. CCBench currently supports only a subset of TPC-C including New-Order and Payment transactions, and we obtained the following results: (1) CCBench exhibited scalability in both the thread count and the warehouse count; (2) The majority of the execution time was for the index traversal, and its acceleration was important for high performance; (3) Different analysis platforms exhibited different behavior even for the same workload, depending on the design and the implementation. The details are described in the extended version [49]. CCBench will support TPC-C full-mix in future work, which will be available at [1].

4. REPRODUCTION OF PRIOR WORK

Before presenting our analysis of reproducing experiments performed in prior work, we explain how we validated the correctness of the CCBench implementation, so that it successfully reproduced the results of experiments performed in prior work. We did this by evaluating DBx1000 [6], a standard analysis platform, as used by Bang et al. [14], to validate the behavior of CCBench. For all graphs showing DBx1000 results, we set CPU_FREQ to 2.095 after measuring the real clock rate used for rdtscp instruction; each plot point shows the average of ten runs. The duration of each run was 4.5 to 10 s. As the access method, we used a hash index for DBx1000 and Masstree for CCBench. DBx1000 does not support RMW operations in the YCSB workload, so their plots are not presented in the reproduction results for MOCC and Cicada.
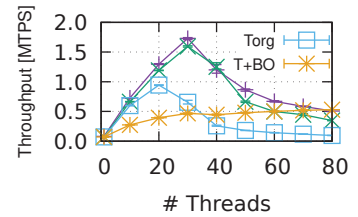
---

[1]More spaces were explored, and they are found online [2].
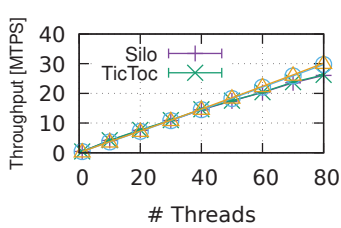
(a) YCSB (Read-Only). 2 queries per transaction, skew 0.
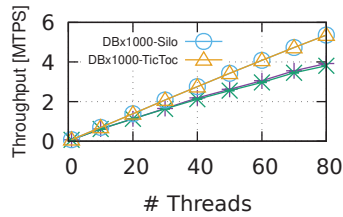
(b) YCSB (Medium Contention). 16 queries per transaction, 90% reads, 10% writes, skew 0.8.
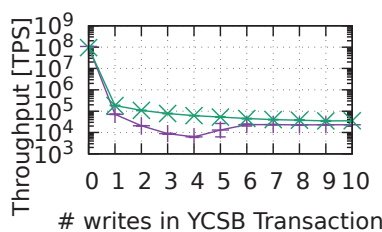
(c) YCSB (High Contention). 16 queries per transaction, 50% reads, 50% writes, skew 0.9.
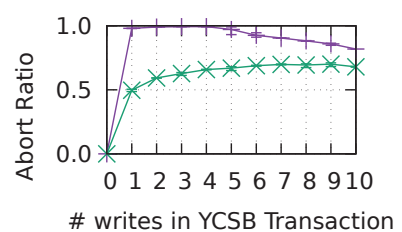
Figure 2: Reproduction of TicToc experiment. 10 M records, payload 1 K bytes. Different access methods (hash for DBx1000, Masstree for CCBench). CCBench without index outperformed (39.6 Mtps) DBx1000 (30 Mtps) at 80 threads.
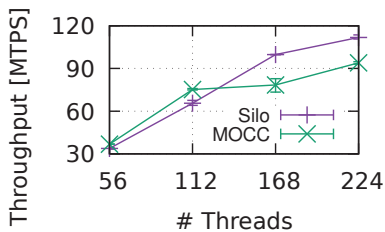
(a) Read Only Workload.

(b) Read-Write YCSB. 224 threads.

(c) Read-Write YCSB. 224 threads.

Figure 3: Reproduction of MOCC experiment. YCSB, 50 records, 10 ops/trans, Skew 0, Payload 4 bytes, reads were pure read, writes were read-modify-write.

4.1 TicToc

Our first experiment was aimed at reproducing TicToc results. We experimented with the settings for Figs. 6, 7, and 8 in the original paper [64]. We set the epoch interval longer than 40 ms so that cache-line conflicts for the global epoch rarely occurred in Silo. The experimental results are shown in Figs. 2a, 2b, and 2c. TicToc uses NoWait optimization instead of NoWaitTT. Note that hereafter, **TicToc** means to attach NoWait instead of NoWaitTT. Torg is the original TicToc with all the optimizations, including NoWaitTT. T+BO is TicToc using AdaptiveBackoff.

Fig. 2a shows that Silo and TicToc had a comparable performance for read-only workloads, consistent with the results shown in Fig. 6 in the original paper. Fig. 2b shows that Silo and TicToc both scale in performance. This is also consistent with the results shown in Fig. 7 in the original paper. The results for DBx1000 are also shown in the figures, and their behaviors were similar to those for CCBench. DBx1000 outperformed CCBench, which we attribute to the difference in access methods. A hash index was used for DBx1000 while Masstree was used for CCBench, and hash indexes are typically faster than tree indexes. We determined that CCBench without indexing had 30% higher performance (39.6 Mtps) than DBx1000 (30 Mtps). As we have already stated, the effect of the access method must be considered for a fair comparison.

One finding is particularly noteworthy. In Fig. 2c, the behaviors of Silo and TicToc differ from those in Fig. 8 in the original paper [64]. The performance of TicToc improved as the number of threads was increased, but when the number exceeded a certain value, the performance started to deteriorate due to an increase in read/write conflicts. There-

fore, we presumed that some sort of backoff was applied to TicToc in the original paper, and attached AdaptiveBackoff to TicToc. This is shown as T+BO in Fig. 2c. The performance curve for T+BO is similar to that in Fig. 9 in the original paper. The lower performance of the original TicToc protocol shown as Torg, is attributed to excessive delays and read validation failures. NoWaitTT inserts fixed delays into the validation phase, which tends to result in read validation failure. Therefore, Torg shows low performance in this graph, and TicToc shows better performance due to the shorter read phase. The results for DBx1000 are also plotted in the figures, and their behaviors are similar to our results.

Fig. 2a shows that Silo and TicToc exhibited linear scalability for a read-only workload, and Fig. 6 in the original paper shows almost linear scalability. Fig. 2b above and Fig. 7(a) in the original paper show almost the same results. In summary, CCBench tended to closely reproduce the results for TicToc, and the results of CCBench are consistent with those of DBx1000.

4.2 MOCC

The second experiment was aimed at reproducing MOCC results [57]. Our MOCC implementation differs from the original one in terms of temperature and MQL, as described in §3.3. To reproduce the results in the original paper, we experimented with the settings shown in Figs. 6 and 7 in the original paper. The details of the workload are described in graph captions.

The results in Fig. 3a appropriately reproduce those in Fig. 6 in the original paper. The performance scales with the number of threads in the read-only workloads. The underperformance of MOCC with more than 168 threads compared with Silo is attributed to the overhead of MOCC.
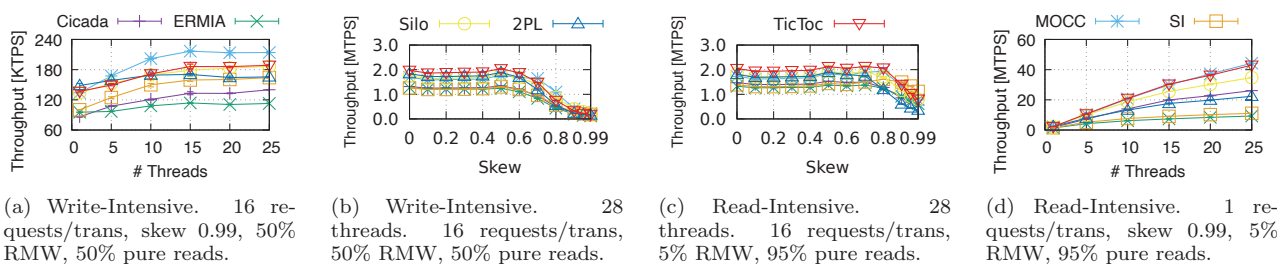
3536

Figure 4: Reproduction of Cicada experiment. 10 M records, payload 100 bytes.

(a) Write-Intensive. 16 requests/trans, skew 0.99, 50% RMW, 50% pure reads.

(b) Write-Intensive. 28 threads. 16 requests/trans, 50% RMW, 50% pure reads.

(c) Read-Intensive. 28 threads. 16 requests/trans, 5% RMW, 95% pure reads.

(d) Read-Intensive. 1 requests/trans, skew 0.99, 5% RMW, 95% pure reads.

When MOCC reads a record, it accesses the corresponding temperature information and checks whether the temperature is high.

The results in Figs. 3b and 3c closely reproduce those in Fig. 7 in the original paper. When the write ratio increased from 0% to 10%, the abort rate rapidly increased, and the performance deteriorated. This is because the write operation produces a large number of read validation failures. As shown in Fig. 3b, the performance of Silo deteriorated at a write rate of 0-40%, while it improved at a write rate of 40-60%. There is thus a trade-off between reduced record-access contention due to reduced write operations and frequent read validation failures due to increased read operations. In contrast, MOCC showed stable performance, apparently because abort is less likely to occur by temperature control.

MOCC and Silo exhibited scalability, as shown in Fig. 3a above, while MOCC and FOEDUS exhibited scalability in the original paper [57] (in Fig. 6). Fig. 3b shows the throughputs of MOCC and Silo for various write ratios. They are almost the same as those in Fig. 7 in the original paper. The abort ratios of Silo and MOCC shown in Fig. 3c is consistent with that in Fig. 7 in the original paper. In summary, CCBench closely reproduced the results for MOCC.

4.3 Cicada

Our third experiment was aimed at reproducing Cicada results. The workload parameters were set almost the same as those in the original paper [35]. The details of the workload are described in the graph caption. The differences between this paper and the original paper are as follows: (1) we redesigned and reimplemented all protocols; (2) Adaptive-Backoff was implemented in all protocols because it is easily attachable and effective for performance improvement.

Figs. 4a, 4b, 4c, and 4d present the experimental results reproducing those in Figs. 6a, 6b, 6c, and 7 in the original paper, respectively. In Figs. 4b, 4c, and 4d, the tendencies in our results for Silo, TicToc, MOCC, ERMIA, and 2PL are the same as those in the corresponding figures in the original paper. In contrast, the results for Cicada differ. Fig. 4d shows that Cicada had worse performance than Silo and TicToc for a read-intensive workload, and this is inconsistent with the results shown in Fig. 7 in the original paper. We discuss this inconsistency in **Insight 5** (§7.1).

5. ANALYSIS OF CACHE

Here, we discuss two effects of the cache. **Cache-line conflict** occurs when many worker threads access to the same cache-line with some writes (e.g., accessing a single

shared counter). The cache-line becomes occupied by a single thread, and the other threads are internally blocked. **Cache-line replacement** occurs when transactions access a large amount of data (e.g., high cardinality, low skew, or large payload). The data accesses tend to replace the contents of the L1/L2/L3 cache.

5.1 Cache-Line Conflict

Some protocols use centralized ordering (e.g., a shared counter) [31, 32] while others use decentralized ordering [35, 53, 57, 64]. We first analyzed the negative effect of centralized ordering. The results are shown in Fig. 5. Since the results for YCSB-A have the same tendency as those for YCSB-B, they are omitted. Fig. 5 shows that the throughputs of ERMIA and SI did not scale although their scheduling spaces are wider than those of single-version concurrency control (SVCC) serializable protocols. This is because both protocols depend on a centralized data structure, which causes cache-line conflicts, which in turn degrade performance. This is consistent with previous findings [53, 57].

To investigate the cause of the inefficiency, we measured the throughput of `fetch_add`. The results are shown in Figs. 6. The L3 cache miss rate was measured using the Linux perf tool. Fig. 6a shows that one thread exhibited the best performance. This is because frequent reads/writes to the same memory address by multiple threads cause many cache-line conflicts. Fig. 6b shows that the L3 cache miss rate increased with the number of threads. Cache-line conflicts often result in L3 cache misses and longer communication delays because the CPU core uses a cache-line, and the other cores running in different CPU sockets also require the line. In the experiment setting, 56 threads used two sockets, and 112 or more threads used four sockets. Thus, a higher L3 cache miss rate ($\geq 30\%$) indicates communication among sockets, which greatly degrades the performance of `fetch_add`. This means that centralized ordering should be avoided.

5.2 Cache-Line Replacement

We discuss the results for YCSB-B. In Figs. 5a and 5c, as cardinality increased, (1) the throughput first improved and then deteriorated, and (2) the abort ratio monotonically improved. When cardinality was quite low, the footprint of accessible records in database was small so that all of the records fit inside the L3 cache. The contention is more likely with L1 or L2 cache. As cardinality increases, such contention is alleviated since the number of accessible records increases, and the performance improves. It is likely that with more cardinality, the total number of records starts to
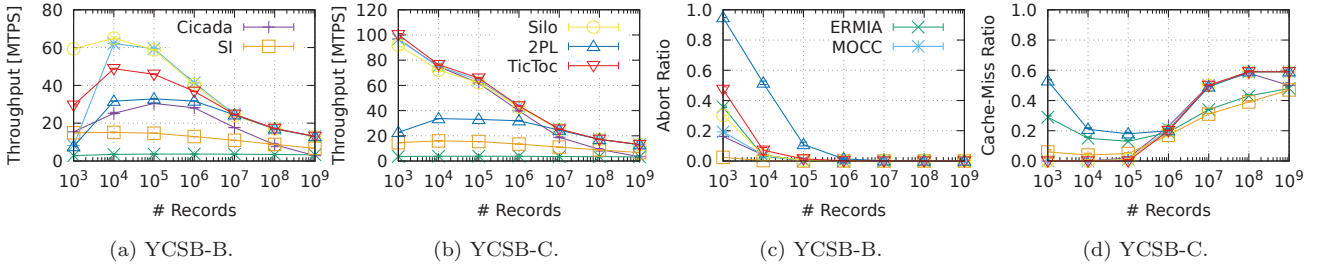
(a) YCSB-B.   (b) YCSB-C.   (c) YCSB-B.   (d) YCSB-C.

Figure 5: Varying cardinality: payload 4 bytes, 224 threads, skew 0, 10 ops/trans.

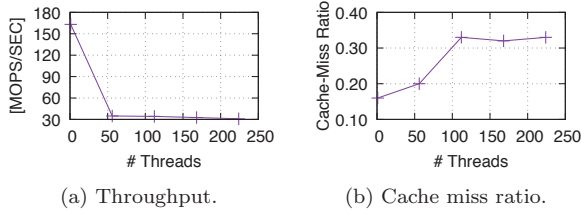(a) Throughput.   (b) Cache miss ratio.

Figure 6: Scalability of fetch_add.

overflow L3 cache. This results in L3 cache misses, and consecutive accesses to remote socket cache or DRAM degrades the performance.

We investigated the strong effect of the L3 cache miss ratio. As shown in Fig. 5d, when cardinality was no more than 10^5, the L3 cache miss ratios for Silo, TicToc, and Cicada were almost zero. This is because the L3 cache size was 38.5 MB and the size of a record including the header was 64 bytes ($64 \times 10^5 < 38.5 \times 10^6$). However, as cardinality increased, their throughputs decreased linearly, as shown in Fig. 5b. This is attributed to L1/L2 cache misses. When cardinality was more than 10^5, we observed that L3 cache misses affected performance even for read-only workloads.

The negative effect of read contention is shown in Figs. 1 and 6 in the original paper [57]. The cache-line conflicts that occurred due to read lock contention degraded performance, and the difference in performance between OCC and 2PL rapidly increased as the number of threads increased. However, there is no mention in the paper that the greater the number of L3 cache misses for a read-only workload, the smaller the performance difference between OCC and 2PL. As shown in Fig. 5b, when cardinality was no less than 10^7, Silo, TicToc, and 2PL had almost the same throughput. Moreover, their L3 cache miss ratios were almost the same, as shown in Fig. 5d. For read-only workloads, the differences in performance among protocols converged due to the L3 cache misses when the database size was large. This has not been reported so far.

Insight 1: Even if the entire database is within L3 cache, as cardinality increases, (1) OCC for read-intensive workloads improves due to a decrease in L1/L2 cache-line conflicts, and (2) OCC for read-only workloads deteriorates due to an increase in L1/L2 cache-line replacements. If the entire database slightly overflows the L3 cache, the performances of the protocols diverge. If the size of the entire database is much larger than the L3 cache, the performances of Silo, TicToc, MOCC, and 2PL converge due to frequent L3 cache-line replacements.

5.3 Effect of Invisible Reads

Updating shared data among threads running in different CPU sockets trigger L3 cache misses. It is mainly due to the cache-line conflicts, leading to performance degradation. The read operation in Silo does not update the corresponding metadata, which can prevent cache-line conflicts. This read method is referred to as InvisibleReads [37]. It effectively improves performance of protocols for read-intensive workloads; however, its behaviors have not been explored yet. The read operations of TicToc typically update the corresponding metadata (i.e., read timestamp), so they are not InvisibleReads.

To investigate the details of InvisibleReads, we measured the performance of Silo and TicToc for various read ratios and cardinalities. The results of experiments are shown in Figs. 7a-7c. Fig. 7a shows that TicToc-1 K (where 1 K means 1000 records) outperformed Silo-1 K for low cardinality and write-intensive workloads. The abort ratio of TicToc-1 K was much worse than that of Silo-1 K, as shown in Fig. 7b. As shown in Fig. 7a, in the read-most (90%) case, Silo-1 K outperformed TicToc-1 K, as evidenced by the rapid increase in the abort ratio (Fig. 7b). The InvisibleReads method contributed to the performance improvement in this case. For high cardinality workloads, Silo-1 M (where 1 M means 1 million records) always outperformed TicToc-1 M. The abort ratios of Silo-1 M and TicToc-1 M were almost zero, as shown in Fig. 7b. This result seems mysterious. The abort ratios for both Silo and TicToc were almost the same, and their L3 cache miss ratios were almost the same, as shown in Fig. 7c. However, Silo-1 M outperformed TicToc-1 M. This is because dealing with contention resolution processing in TicToc requires three functions: timestamp history management, early abort judgment, and read timestamp update. These functions require executing additional instructions and thus degrade performance. If InvisibleReads is woven into the process, a protocol has difficulty exploiting them.

Insight 2: For a detailed analysis of protocols, contention should be carefully considered. The occurrence of contention depends not only on the read ratio but also on cardinality and skew. We observed that Silo outperformed TicToc not only for read-only workloads but also for write-intensive workloads. This fact has not been reported in previous studies [13, 35, 64]. These results are due to the use of InvisibleReads method, which prevents cache misses in low contention cases. The efficiency of InvisibleReads is provided by its simple design that does not require computations for conflict resolution or anti-dependency tracking.
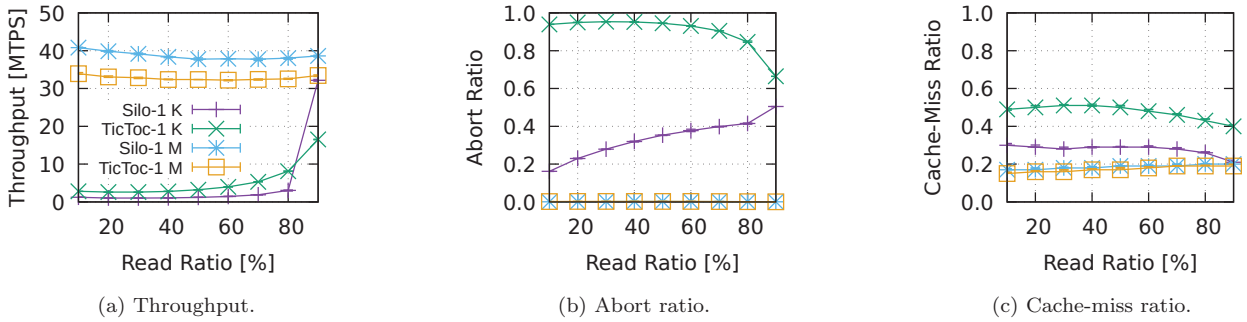
(a) Throughput.

(b) Abort ratio.

(c) Cache-miss ratio.

Figure 7: Analysis of InvisibleReads. 224 threads, payload 4 bytes, skew 0, 10 ops/trans. cardinality 10^3 or 10^6 records.

(a) Throughput.
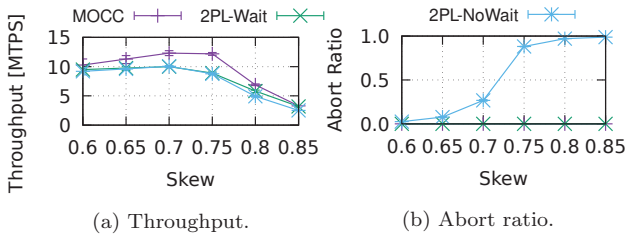
(b) Abort ratio.

Figure 8: Wait or NoWait: YCSB-B, 100 M records, 224 threads, payload 4 bytes, 10 ops/trans.
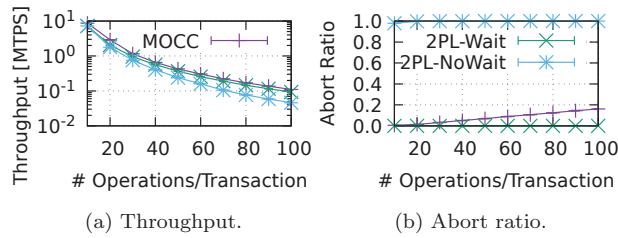

(a) Throughput.

(b) Abort ratio.

Figure 9: Impact of transaction size: YCSB-B, 100 M records, 224 threads, payload 4 bytes, skew 0.8.

6. ANALYSIS OF DELAY

6.1 Effect of NoWait

A method for mitigating performance degradation caused by conflicts is to place an additional artificial delay before a retry, such as with NoWaitTT [64] and AdaptiveBackoff [35]. In contrast to waiting for a lock release, the NoWait method immediately releases all of acquired locks when a conflict is detected and immediately retries. The question then in dealing with contention is whether to choose NoWait or Wait.

To answer this question, we compared the performance of three methods for various degrees of skew: 2PL-Wait, 2PL-NoWait, and MOCC (mix of OCC and 2PL-Wait). All operations of each transaction for 2PL-Wait were sorted in advance to avoid deadlock. As shown in Fig. 8a, the performance of all three protocols started to degrade at a skew of 0.7 due to a large number of conflicts. MOCC demonstrated excellent performance, as shown in Fig. 8a. This is because MOCC manages fewer lock objects for workloads with less skew. 2PL-Wait and 2PL-NoWait showed similar performances at skews of 0.60-0.85, so the cost of abort-retry and

the cost of waiting for lock were similar. So, we need more contention to answer the question.

To provide more contention, we increased the transaction size. The cost of aborts should be smaller for short transactions and larger for long ones. Therefore, with long transactions, the abort-retry cost is higher and may exceed the lock waiting cost. Fig. 8 shows that the lock waiting cost of Wait and the abort-retry cost of NoWait were similar. Fig. 9a shows that Wait began to outperform NoWait as the transaction size increased. In contrast, Silo-NoWait outperformed Silo(-Wait), as shown in Fig. 10 (described in §6.2). The answer then is that it depends on the situations. Therefore, the control of waiting time is important for efficiency.

Insight 3: One cannot give a general statement about which is better, NoWait or Wait since it depends on the situations. As shown in Fig. 9a, Wait begins to outperform NoWait as the transaction size increases. On the other hand, as shown in Fig. 10c, Silo with NoWait comes to outperform the original Silo. This suggests the importance of adaptive behavior. Although adaptive concurrency controls have been studied [27, 43, 47, 57], their importance has not been discussed in depth regarding optimizations except for backoff [35]. Studies on adaptive optimization methods remain on the frontier.

6.2 Effect of Payload Size

We analyzed the effect of payload size, which had not previously been done, to the best of our knowledge. Fig. 10a shows the relationship between throughput and payload size for Silo and TicToc. Throughput initially increased with payload size, and then started to decrease at a certain point. An increase in payload size would likely degrade OCC performance, which is consistent with the throughput decrease shown in the right half of the graph. However, the throughput increase shown in the left half of the graph is counter-intuitive.

We hypothesized that this increase was due to the delay caused by the extra reads process, where two or more reads were performed if there was an interruption in the concurrent update transactions. An increase in payload size lengthens the time to read the payload. This increases the probability of an interruption due to an update transaction, which increases the number of extra reads. This behavior extends the read phase and reduces the number of transactions that run through the validation phase in parallel. As the number of committed update transactions decreases, the number of read validation failures also decreases, which leads
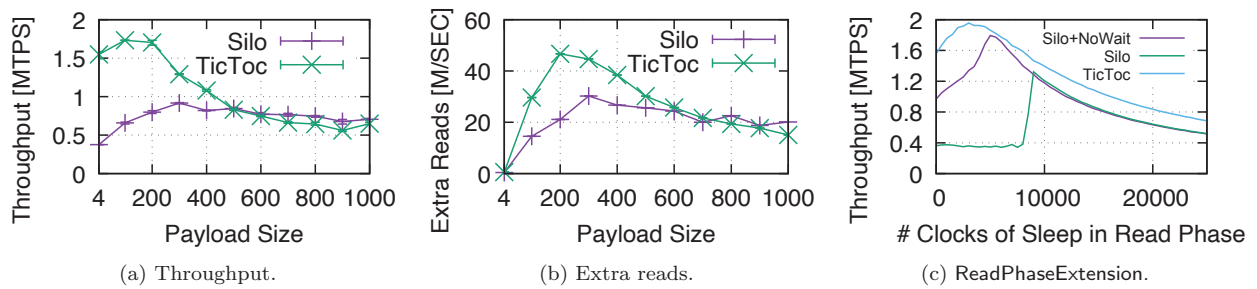
(a) Throughput. (b) Extra reads. (c) ReadPhaseExtension.

Figure 10: Effect of delay provided by extra read: YCSB-A, 100 M records, 224 threads, skew 0.9, 10 ops/trans.

to throughput improvement. Delaying the retry of atomic reads reduces the number of concurrent worker threads involved in the validation phase for the entire system, which also reduces contention.

Besides Wait, NoWait, and AdaptiveBackoff, we present a novel delay-related optimization method, ReadPhaseExtension, which was inspired by the positive effect of extra reads. Comparing Figs. 10a and 10b reveal that the curves are similar. This indicates that throughput and the number of extra reads are correlated. If this is correct, adding an artificial delay into the read phase should produce similar results. We conducted such an experiment for a payload size of 4 bytes. The result in Fig. 10c shows that a certain amount of additional delay (less than 9000 clocks) improved performance. This effect is similar to that of the extra read process. We refer to this artificial delay as ReadPhaseExtension and define it as a new optimization method. ReadPhaseExtension is configured by setting the delay on the basis of local conflict information. This optimization method can exploit information on the access conflicts for each record during the read phase whereas AdaptiveBackoff uses only global information across all worker threads.

Insight 4: The extra read process plays a key role in the performance of OCC protocols. It is known that the contention regulation caused by delay can contribute to performance improvement depending on the situation. A remarkable finding here is that the ReadPhaseExtension inspired by the extra read process can also improve performance. ReadPhaseExtension differs from NoWaitTT since it can exploits information on conflicting records inside transactions to adjust the delay whereas the delay in NoWaitTT is fixed. Such additional delay in the read phase and the use of thread-local conflict information combine to create a unique optimization method.

7. ANALYSIS OF VERSION LIFETIME

7.1 Determining Version Overhead

The life of a version begins when a corresponding write operation creates it. The version state is called **visible** during the period when other transactions can read it. Otherwise, the version state is called **non-visible**. Recent CC protocols typically make the version visible at the pre-commit phase [30, 53]. After a certain period, the life of the version ends, and it is made non-visible. SVCC protocols typically make a version by overwriting its previous version, with the former becoming visible and the latter becoming non-visible at the same time. MVCC protocols typically make a version

on a different memory fragment from other visible versions of the same record. Therefore, their life does not end unless GC conducts its work.

The performance of MVCC is thought to suffer from the existence of many visible versions [34,42]. They lead to a significant burden due to memory exhaustion or an increase in cache-line replacements for traversal or installation. MVCC protocols can provide higher performance than SVCC ones due to their larger scheduling spaces. It should be noted that architectural factors sometimes negate the benefits, as mentioned in §5.1. We analyzed this situation in depth. Note that the cost for allocating new pages is negligible in the results below since our AssertiveVersionReuse (§3.2) conducts the space allocation beforehand.

MVCC protocols often exhibit high performance compared with SVCC protocols for high contention workloads. To begin, we measured the throughput for various degrees of skew (0 to 0.99), as shown in Fig. 11. Here we focus on the difference between Cicada (MVCC) and Silo (SVCC). In Figs. 11a and 11b for YCSB-A, Cicada outperformed Silo when the skew was higher than 0.8. The larger the payload, the greater the superiority of Cicada. This is attributed to Cicada having larger scheduling spaces than Silo; nevertheless, these results confirm that the performance of MVCC suffers from having many versions [34, 42]. In Figs. 11c and 11d, Cicada underperformed Silo, TicToc, and MOCC on YCSB-B.

To clarify the reason, we analyzed the characteristics of the versions. Larger scheduling spaces are supported by many versions, and they can be an overhead. Such overhead in modern protocols remains to be analyzed. We show the latency breakdown with a skew of 0, where MVCC is at a disadvantage, in Fig. 12. In Figs. 12a and 12b, we can see that the major overhead of Cicada lies in the read and write operations rather than validation or GC. To clarify the reasons for the overhead, we designed an SVCC version of Cicada, Cicada-SV, that overwrites the inline versions of Cicada. This eliminates the need to allocate new areas, so Cicada-SV performs like an SVCC protocol. We prepared a special workload in which transactions were completely partitioned to avoid contention because Cicada-SV does not cope well with contention. The results are shown in Fig. 13. The latencies of Cicada-SV and Silo were almost the same. We attribute the overhead shown by Cicada in Figs. 12a and 12b to the cost of version chain traversals.

Since having many visible versions degrades performance, reducing the number should improve performance. RapidGC optimization effectively mitigates this problem. To evaluate its efficiency, we varied the interval setting and measured the
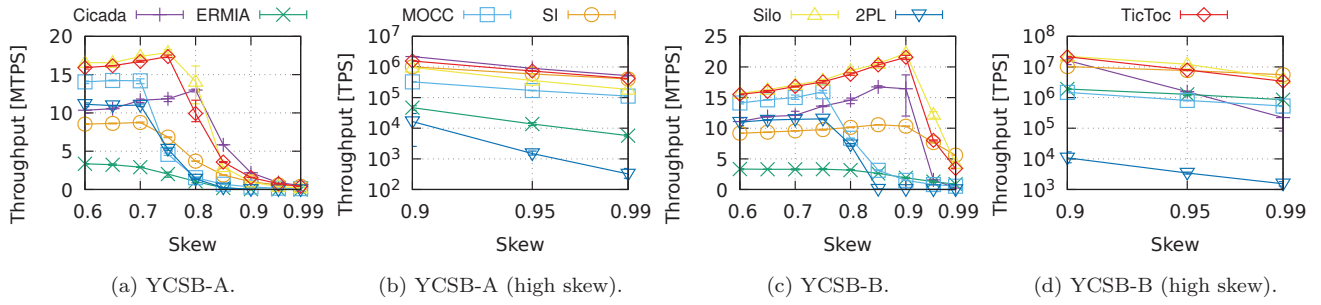
(a) YCSB-A. (b) YCSB-A (high skew). (c) YCSB-B. (d) YCSB-B (high skew).

Figure 11: Effect of workload skew: 100 M records, 224 threads, payload 4 bytes, 10 ops/trans.

(a) YCSB-A. (b) YCSB-B.
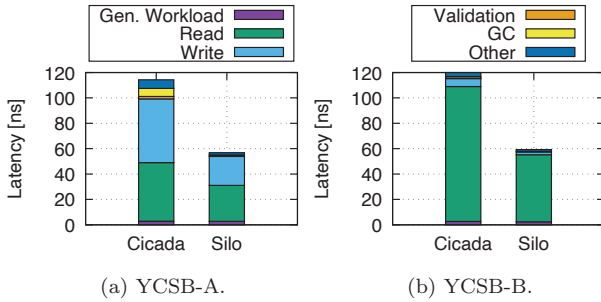
Figure 12: Latency breakdown: skew 0, 100 M records, 224 threads, payload 4 bytes, 10 ops/trans.
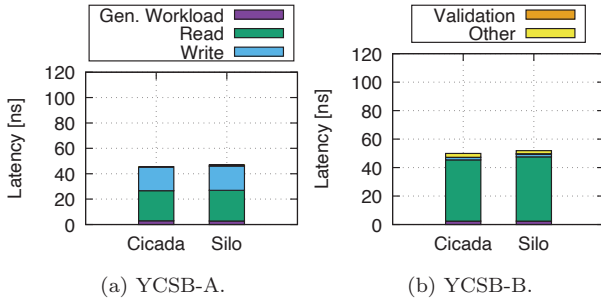

(a) YCSB-A. (b) YCSB-B.

Figure 13: Latency breakdown of Cicada-SV and Silo for partitioned workload. Other settings were the same as for Fig. 12.

Figure 14: Analysis of RapidGC: YCSB-A, skew 0, 224 threads, 1 M records, payload 4 bytes, 10 ops/trans. Inserted delays for long transactions: 0 to 10 ms.

the difference and found that the version management cost of Cicada is not negligible even for a low contention case, as shown in Figs. 12a and 12b. It is difficult to obtain precise knowledge using different reference implementations or platforms, and the deep analysis of CC protocols must be done on the same platform.

7.2 Limit of Current Approach

We investigated the behavior of RapidGC using the same workloads as in our third experiment with a long transaction. Even state-of-the-art GC does not sufficiently reduce the number of visible versions if there is only a single long transaction. This is because modern protocols assume that transaction execution is one-shot, and that the transactions are relatively short (e.g., 16 operations for original YCSB). Long transactions except read-only ones have been ignored in modern transaction studies.

To generate a long transaction, we added an artificial delay at the end of the read phase. Both long and short transactions used the same number of operations with the same read-write ratio. One worker thread executed the long transaction, and the remaining worker threads executed the short transactions. The skew was set to 0 so contention in record accesses rarely occurred and thus did not affect performance, even though there was a long transaction.

We measured the performance of Cicada under the workload described above, varying the RapidGC interval settings and the delay added to the long transaction. As shown in Fig. 14, performance saturated when a delay was inserted.

throughput of Cicada. The legend 0 and 100 us in Fig. 14 show that high throughputs were obtained when the interval was no more than 100 us, as reported in prior studies [35,60]. This is because the number of visible versions was small, and most of them were kept in cache. So, can RapidGC with its short intervals perform efficiently for any workload? The answer is no. We explain the reason why in §7.2.

Insight 5: The overhead of multi-version management is not negligible. Silo and TicToc outperformed Cicada in high-skew (0.99), read-intensive (YCSB-B), non-RMW, high cardinality (100 M records) cases. A previous study [35] found that all three exhibited a similar performance for a similar workload (skew 0.99, read 95%, RMW, 10 M records, payload 100 byte), as shown in Fig. 7 in the original paper. This inconsistency is due to the use of different platforms in the previous study. Using a single platform, we observed
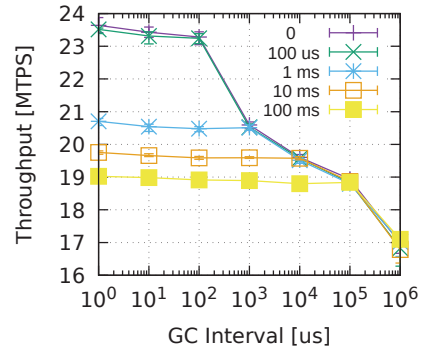
Saturation occurred when the GC interval was the same as the added delay. For example, the light blue line includes a long transaction with a 1 ms delay, and performance saturated when the GC interval was 1 ms. Similar behaviors were observed with longer delays. This is because current GC methods do not collect visible versions that may be read by active long transactions. The current GC scheme does not collect the versions until the transaction finishes. We consider that this limitation is the primary hurdle to improving MVCC performance by reducing the number of visible versions. We could not obtain results for a 1 s delay because such a delay requires a huge amount of memory, which causes the Linux OOM (out-of-memory) killer to kill the process.

7.3 Aggressive Garbage Collection

From the limitation of the current GC scheme described above, we suggest a novel GC scheme, AggressiveGC, that aggressively collects versions beyond the current ones to deal with long transactions. For example, the multi-version timestamp ordering (MVTO) protocol could be integrated with a GC method that aggressively collects visible versions. It could make some versions non-visible even though active or future transactions need to read them. Such a protocol might incur read operation failures unlike conventional MVTO, which could be handled by aborting the transaction and retrying it with a new timestamp. Restricting the number of versions in kVSR [39] and 2V2PL [15, 48] has been discussed. However, only the latest contiguous versions are kept there, so this approach is less flexible than our suggested scheme. We claim that the visible versions do not need to be contiguous and the number of versions can be flexible depending on the context. An interesting topic in our proposed scheme is the risk of starvation. One way to mitigate this problem is to manage the priorities among transactions such as wait-die [45], which has not yet been discussed in modern MVCC studies.

We suggest two optimizations for write operations in terms of aggressive protocols. The first is version overwriting, i.e., creating a new version by overwriting the memory segment of the previous version, which becomes non-visible at the same time, as is done in SVCC protocols. Version overwriting is efficient because two operations are combined into one operation. The second is non-visible write, i.e., making versions non-visible from the beginning of their life. The idea of non-visible write was originally proposed as the Thomas write rule [52] and recently generalized as the non-visible write rule (NWR) [41] to deal with blind writes. Novel version lifetime management is a promising way to improve MVCC protocols.

Insight 6: Even state-of-the-art GC cannot hold down the number of versions in MVCC protocols if a single long transaction is mixed into the workload. An aggressive approach can solve this problem by aggressively changing the version state to non-visible for both reads and writes, even if transactions still require the state.

8. RELATED WORK

Yu et al. [63] evaluated CC protocols using DBx1000 [6], which is open source. They evaluated the scalability of CC protocols using a real machine and an emulator. Three recent protocols [32,35,57] supported in CCBench, are not included in DBx1000 [6]. Wu et al. [60] empirically evaluated

MVCC protocols using Peloton [8], which is open source. They evaluated not only scalability but also the contention effect, read ratio, attributes, memory allocation, and index. They did not evaluate SVCC and the modern MVCC protocols evaluated in this paper [35, 53, 57, 64]. Appuswamy et al. [13] evaluated CC protocols in four types of architecture using Trireme, which is not open source. They determined that the shared-everything architecture is still the best option for contention-tolerant in-memory transaction engines. CC protocols for distributed systems have been evaluated elsewhere [28, 54]; this paper focuses on a single many-core architecture.

Whereas previous studies mostly evaluated scalability and did not explore the behavior of protocols when thread parallelism was set to a high degree [20,22,23,25,27,29,32,33,35, 44,51,53,57,58,61,62,64,65], we fixed the thread parallelism at 224 and analyzed protocols for various settings. We classified a variety of methods on the basis of three performance factors: cache, delay, and version lifetime. This analysis let us identify three new optimization methods.

9. CONCLUSION

Using CCBench, we analyzed concurrency control protocols and optimization methods for various settings of the workload parameters with the number of threads fixed at 224, whereas previous studies mostly focused on thread scalability, and none of them explored the space we analyzed. We classified versatile optimization methods on the basis of three performance factors: cache, delay, and version lifetime. Through the analysis of protocols with CCBench, we gained six insights. **I1:** The performance of optimistic concurrency control for a read-only workload rapidly degrades as cardinality increases even without L3 cache misses. **I2:** Silo outperforms TicToc for write-intensive workloads, which is attributed to InvisibleReads for unskewed high cardinality cases. **I3:** The effectiveness of two approaches to coping with conflict (Wait and NoWait) depends on the situation. **I4:** Extra reads can regulate contention. **I5:** Results produced from mixed implementations may be inconsistent with the theory. **I6:** Even a state-of-the-art garbage collection method RapidGC cannot improve the performance of multi-version concurrency control if there is a single long transaction mixed into the workload On the basis of **I4**, we defined the ReadPhaseExtension optimization in which an artificial delay is added to the read phase. On the basis of **I6**, we defined the AggressiveGC optimization in which even visible versions are collected.

In future work, we plan to support TPC-C full-mix [49], and to include logging and recovery modules based on our preliminary studies [40, 50]. The code for CCBench and all the data in this paper are available online at GitHub [5]. We expect that CCBench will help to advance transaction processing research.

10. REFERENCES

[1] CCBench Developer Guide. https://github.com/thawk105/ccbench/tree/master/cc_format.

[2] CCBench Experimental Data. https://github.com/thawk105/ccdata.

[3] CCBench OCC. https://github.com/thawk105/ccbench/tree/master/occ.

[4] Code of Cavalia. https://github.com/Cavalia.

[5] Code of CCBench. https://github.com/thawk105/ccbench.

[6] Code of DBx1000. https://github.com/yxymit/DBx1000.

[7] Code of Masstree. https://github.com/kohler/masstree-beta.

[8] Code of Peloton. https://pelotondb.io.

[9] gflags. https://github.com/gflags/gflags.

[10] How to Extend CCBench. https://medium.com/@jnmt.

[11] mimalloc. https://github.com/microsoft/mimalloc.

[12] The Transaction Processing Council. TPC-C Benchmark (Revision 5.11), February 2011.

[13] R. Appuswamy, A. G. Anadiotis, D. Porobic, M. Iman, and A. Ailamaki. Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-Contention Workloads. *PVLDB*, 11(2):121–134, 2017.

[14] T. Bang, N. May, I. Petrov, and C. Binnig. The Tale of 1000 Cores: An Evaluation of Concurrency Control on Real(Ly) Large Multi-Socket Hardware. In *DaMoN*, 2020.

[15] R. Bayer, H. Heller, and A. Reiser. Parallelism and Recovery in Database Systems. *ACM TODS*, 5(2):139–156, 1980.

[16] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *SIGMOD Record*, volume 24, pages 1–10, 1995.

[17] P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[18] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[19] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *EuroSys*, pages 211–224, 2013.

[20] M. Dashti, S. Basil John, A. Shaikhha, and C. Koch. Transaction Repair for Multi-Version Concurrency Control. In *SIGMOD Conf.*, pages 235–250, 2017.

[21] 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.

[22] B. Ding, L. Kot, and J. Gehrke. Improving Optimistic Concurrency Control through Transaction Batching and Operation Reordering. *PVLDB*, 12(2):169–182, 2018.

[23] D. Durner and T. Neumann. No False Negatives: Accepting All Useful Schedules in a Fast Serializable Many-Core System. In *ICDE*, pages 734–745, 2019.

[24] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Comm. ACM*, 19(11):624–633, 1976.

[25] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein. High Performance Transactions via Early Write Visibility. *PVLDB*, 10(5):613–624, 2017.

[26] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD Record*, volume 23, pages 243–252, 1994.

[27] J. Guo, P. Cai, J. Wang, W. Qian, and A. Zhou. Adaptive Optimistic Concurrency Control for Heterogeneous Workloads. *PVLDB*, 12(5):584–596, 2019.

[28] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An Evaluation of Distributed Concurrency Control. *PVLDB*, 10(5):553–564, 2017.

[29] Y. Huang, W. Qian, E. Kohler, B. Liskov, and L. Shrira. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *PVLDB*, 13(5):629–642, 2020.

[30] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a Scalable Approach to Logging. *PVLDB*, 3(1-2):681–692, 2010.

[31] H. Jung, H. Han, A. Fekete, U. Röhm, and H. Y. Yeom. Performance of Serializable Snapshot Isolation on Multicore Servers. In *DASFAA*, pages 416–430, 2013.

[32] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD Conf.*, pages 1675–1687, 2016.

[33] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD Conf.*, pages 691–706, 2015.

[34] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB*, 5(4):298–309, 2011.

[35] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *SIGMOD Conf.*, pages 21–35, 2017.

[36] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*, pages 183–196, 2012.

[37] V. J. Marathe, W. N. Scherer, and M. L. Scott. Design Tradeoffs in Modern Software Transactional Memory Systems. In *LCR*, pages 1–7, 2004.

[38] J. M. Mellor-Crummey and M. L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *SIGPLAN Notices*, volume 26, pages 106–113, 1991.

[39] T. Morzy. The Correctness of Concurrency Control for Multiversion Database Systems with Limited Number of Versions. In *ICDE*, pages 595–604, 1993.

[40] Y. Nakamura, H. Kawashima, and O. Tatebe. Integration of TicToc Concurrency Control Protocol with Parallel Write Ahead Logging Protocol. *Journal of Network Computing*, 9(2):339–353, 2019.

[41] S. Nakazono, H. Uchiyama, Y. Fujiwara, Y. Nakamura, and H. Kawashima. NWR: Rethinking

Thomas Write Rule for Omittable Write Operations. http://arxiv.org/abs/1904.08119, 2020.

[42] T. Neumann, T. Mühlbauer, and A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD Conf.*, pages 677–689, 2015.

[43] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-Driving Database Management Systems. In *CIDR*, 2017.

[44] G. Prasaad, A. Cheung, and D. Suciu. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *SIGMOD Conf.*, pages 527–542, 2020.

[45] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. System Level Concurrency Control for Distributed Database Systems. *ACM TODS*, 3(2):178–198, 1978.

[46] M. L. Scott and W. N. Scherer. Scalable Queue-based Spin Locks with Timeout. In *SIGPLAN Notices*, volume 36, pages 44–52, 2001.

[47] Y. Sheng, A. Tomasic, T. Zhang, and A. Pavlo. Scheduling OLTP Transactions via Learned Abort Prediction. In *aiDM*, 2019.

[48] R. E. Stearns and D. J. Rosenkrantz. Distributed Database Concurrency Controls Using Before-Values. In *SIGMOD Conf.*, pages 74–83, 1981.

[49] T. Tanabe, T. Hoshino, H. Kawashima, and O. Tatebe. An Analysis of Concurrency Control Protocols for In-Memory Databases with CCBench (Extended Version). http://arxiv.org/abs/2009.11558, 2020.

[50] T. Tanabe, H. Kawashima, and O. Tatebe. Integration of Parallel Write Ahead Logging and Cicada Concurrency Control Method. In *BITS*, pages 291–296, 2018.

[51] D. Tang, H. Jiang, and A. J. Elmore. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All. In *CIDR*, 2017.

[52] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM TODS*, 4(2):180–209, 1979.

[53] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore in-Memory Databases. In *SOSP*, pages 18–32, 2013.

[54] C. Wang, K. Huang, and X. Qian. A Comprehensive Evaluation of RDMA-enabled Concurrency Control Protocols. http://arxiv.org/abs/2002.12664, 2020.

[55] T. Wang, R. Johnson, A. Fekete, and I. Pandis. The Serial Safety Net: Efficient Concurrency Control on Modern Hardware. In *DaMoN*, 2015.

[56] T. Wang, R. Johnson, A. Fekete, and I. Pandis. Efficiently Making (Almost) Any Concurrency Control Mechanism Serializable. *VLDB Journal*, 26(4):537–562, 2017.

[57] T. Wang and H. Kimura. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *PVLDB*, 10(2):49–60, 2016.

[58] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling Multicore Databases via Constrained Parallel Execution. In *SIGMOD Conf.*, pages 1643–1658, 2016.

[59] G. Weikum and G. Vossen. *Transactional Information Systems*. Elsevier, 2001.

[60] 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.

[61] Y. Wu, C.-Y. Chan, and K.-L. Tan. Transaction Healing: Scaling Optimistic Concurrency Control on Multicores. In *SIGMOD Conf.*, pages 1689–1704, 2016.

[62] Y. Wu and K.-L. Tan. Scalable In-Memory Transaction Processing with HTM. In *ATC*, pages 365–377, 2016.

[63] 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.

[64] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time Traveling Optimistic Concurrency Control. In *SIGMOD Conf.*, pages 1629–1642, 2016.

[65] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, and X. Zhang. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for in-Memory Databases. *PVLDB*, 9(6):504–515, 2016.