Harvard John A. Paulson
School of Engineering
and Applied Sciences

Office of Academic Programs
29 Oxford St, Pierce Hall 110, Cambridge, MA 02138
617 496 5956
jgirash@seas.harvard.edu

John Girash, PhD
Interim Director of Graduate Education

May 11, 2020

In lieu of all dissertation committee members' signatures, I, John Girash, Interim Director of Graduate Education, appointed by the Harvard John A. Paulson School of Engineering and Applied Sciences, **confirm that the Dissertation Committee** has examined a dissertation titled "On the Design and Implementation of High-performance Transaction Processing in Main-memory Databases", presented by Yihe Huang, a candidate for the degree of Doctor of Philosophy in the subject of Computer Science for May 2020 degree conferral, and hereby certify that it is worthy of acceptance as of May 7, 2020.

On the Design and Implementation of High-performance Transaction Processing in Main-memory Databases

A dissertation presented

by

Yihe Huang

to

Harvard John A. Paulson School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

May 2020

Dissertation Advisor:
Professor Eddie Kohler

Author:
Yihe Huang

On the Design and Implementation of High-performance Transaction Processing in Main-memory Databases

Abstract

Main-memory databases are core to many applications, and the performance of modern main-memory database systems is a subject of intense study. It is long understood that the concurrency control algorithm underlying a database system is the deciding factor of how well the system performs under contended workloads. Optimistic concurrency control (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 of 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 many high-contention workloads. 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 4.8× for OCC and 3.8× for MVCC in a TPC-C workload.

Contents

List of Tables

List of Figures

Acknowledgments

Pursuing a doctoral degree in a foreign country is one of the biggest decisions I made in my life so far and also one of the first major life decisions I made on my own. Being able to contribute to science and to advance knowledge and learning is something I have always wanted to do since I was a young kid. It is also something I do not know if I will ever be able to pull off or even can afford to do. At the conclusion of this journey, I am deeply grateful for everyone who offered their help and support all along to make it possible.

I would like to thank numerous professors I worked with over the years for their input and guidance on my work. I have never worked with such great minds before, and all these experiences really make me feel I am a part of the systems research community, not just here in Cambridge, but globally. In particular, I would like to express my gratitude to my advisor Professor Eddie Kohler. I thank them very much for pushing me to try to understand the subtlest differences from previous work or any unexplained results in our experiments. These experiences teach me how to ask important questions and make me a much better researcher. Prof. Kohler has been a great mentor not just academically, but in life. Those conversations we had during uncertain and difficult times for me really meant a lot and helped me adjust to stress, and I am really grateful for them. I would also like to thank Professor Margo Seltzer, who helped me form many important professional connections throughout the years that I am sure I will treasure in my future endeavors. I am grateful that I got to serve alongside both Prof. Kohler and Prof. Seltzer as their teaching fellow to advance our most important mission for our students.

I would also like to thank my numerous collaborators including William Qian, Professors Liuba Shrira and Barbara Liskov for all their help in making my thesis work possible. A very special shout-out to William for building the base version of our MVCC system, and for all the office jokes and tikz tips.

I would also like to thank my parents, my friends, and my very special partner Helen for their emotional and material support throughout the up-and-downs of my PhD endeavor. The journey to pursue a PhD can be stressful at times, and it undoubtedly puts a lot of stress on my life and our relationships, and without their understanding and having them as my support system I mostly likely could not have completed it. To all my loved ones, a very sincere thank you for all your sacrifices and for all what you have done.

Finally, I would like to acknowledge the unprecedented challenges our world faces at this moment. This is the year 2020, and the world is confronting a once-in-a-century pandemic that has overwhelmed health systems and wreaked economic havoc across the world. Most if not all people, myself included, will likely feel the lasting impact and may be forced to change plans. This is also a moment that highlights the importance of pursuing scientific truth. I feel deeply grateful for all the researchers, scientists, and front line medical workers working tirelessly out there saving lives, advancing our understanding of of the disease, and developing cures and vaccines – they are the real heroes of humanity at this moment. Like the rest of the class of 2020 whose identity will probably be forever marked by this global event, we are graduating into an uncertain world. However, dark times call for optimism and light, especially from those of us equipped with the will and knowledge to make a difference. I sincerely hope that, through the next stages of my career, I can find some way to contribute to the mitigation and prevention of future health crises like this. Thank all of you, and we are, and will be, all in this together.

To my parents Ms. Xufang He and Mr. Jingyi Huang.

Chapter 1

Introduction

1.1 Databases in Modern Applications

Database management systems, or simply *databases*, are central to many modern applications. From social media to e-commerce to financial markets, databases power the world's most demanding applications and critical infrastructures.

Databases organize data in some standardized, query-able format (usually tables) so that they can be stored and searched consistently independent of the applications that use them. Modern applications are architected around databases to reach large scale. To serve a large number of clients, applications are typically divided into a client-facing frontend program and a database backend. The frontend program, usually a web page or a smart phone app, is responsible only for rendering user interfaces, presenting data, and communicating with the backend database. Frontend programs are usually stateless and perform tasks such as form validation and data assembly, while all the data processing and storage tasks are handled by the database. This architecture allows these relatively thin frontend programs to be replicated and distributed widely at ease, presenting responsive interactive experiences for the user. It also allows such distributed applications to shift the bulk of

system complexity to its central piece: the database.

Workloads modern databases handle can be divided into two major categories: OLAP and OLTP. OLAP, or *On-Line Analytics Processing* workloads, involve queries over large warehouses of data that are largely static (infrequently updated), but combine and analyze them in user-defined ways to produce insights. Examples of OLAP workloads include astronomy sky surveys, big data analytics, and certain machine learning workloads. OLTP, or *On-Line Transaction Processing* workloads, involve frequent updates to database records, and/or observations of the latest values of such records. Transactions in OLTP workloads are typically short and update-heavy. Examples of OLTP workloads include financial transactions, social media posting and browsing, telecommunication switch table lookups, and many other interactive applications.

OLAP and OLTP workloads are fundamentally different and present different challenges to database designs. **Our work primarily focuses on databases specialized in handling OLTP workloads.**

1.2 Database Transactions

Databases' central role in modern distributed applications subjects them to high concurrency requirements, as a large number of frontend programs can issue requests to the same underlying database system. Additionally, each client request may contain relatively complex *multi-step* operations such as reading from multiple database tables or updating multiple records based on observations made on other database state.

A better solution is to utilize a programming abstraction called *transactions* provided by most modern databases. Transactions are collections of operations that are to be scheduled as a single unit by the database.[1] General database transactions guarantee *ACID* properties:

[1]We follow the "one-shot" transaction model in this work, meaning that we assume a transaction has all

atomicity, consistency, isolation, and durability[2]. In practical terms, it means that programs issuing transactions to the database need not worry about other concurrent operations interfering with the operations within a transaction. The database guarantees that the transaction is executed as a single unit, as if it were the only operation being handled by the system.

The canonical example of a database transaction is a bank account transfer. For example, a bank user wishes to transfer 100 dollars from account A to account B. The transfer is authorized only if account A has enough balance, and once the 100 dollars is withdrawn from account A, it must be deposited into account B, so that no money is lost during the process. Moreover, any observer of the database state must always observe that the total balance of account A and B remains unchanged at any point during the transfer (no intermediate state is observable). The transfer process involves three steps: 1) check that account A has enough balance, 2) withdraw 100 dollars from account A, and 3) deposit 100 dollars to account B. Wrapping these three operations in a database transaction allows the transfer to be handled properly even if multiple such account transfers occurs at the same time.

1.3 Main-memory Databases

Traditional disk-based databases struggle to handle the high transaction throughput demanded by modern large-scale applications. With billions of smart devices online simultaneously issuing database requests, databases today need to handle millions of transactions per second. Persistent storage quickly become a bottleneck in disk-based databases under such a load. Increasingly, high-performance databases trade persistence for performance, because depending on specifics of the workload, not all database operations need

its arguments and parameters available upon start and does not communicate or interact with the caller until it finishes executing.

[2]This work focuses on main-memory databases and we consider durability an orthogonal issue to this study. We still expect transactions to respect all properties other than durability.

to persist immediately. Modern high performance databases keep significant amounts of data in computers' main-memory and process transactions directly in main-memory. Main-memory databases can support persistence and/or high availability by creating periodic persistent snapshots or by replicating state over the network. **This work focuses on the main-memory transaction processing aspect of these systems**, and we consider durability and network communication orthogonal to our work.

1.4 Concurrency Control

Main-memory databases use *concurrency control* mechanisms to implement transactions and to guarantee the *isolation* (the "I" in ACID) property of transactions. When two concurrent transactions access overlapping database records, the two transactions are said to *conflict*. In some cases, such as when both transactions are merely reading the record, no special handling is necessary, but in most cases concurrency control mechanisms are invoked in light of conflicts to ensure they are isolated – no intermediate state generated by one transaction is made visible to the other transaction. (In practice, many transactions do allow such intermediate state to be temporarily visible, but any transactions made such inconsistent observations must eventually abort.)

Two-phase locking is one well-known concurrency control mechanism. In two-phase locking, prior to executing every operation in the transaction, the appropriate locks are acquired for the underlying data record being accessed. This occurs for both reads and writes – read lock are also acquired prior to reading from a database record. As the transaction conclude, all acquired locks are released so that other transactions touching overlapping records can make progress.

Two-phase locking is an example of *pessimistic concurrency control* because it assumes the worst. It anticipates that a concurrent transaction could touch the same records it ac-

cesses, so it proactively locks the records in advance. This, however, results in high overhead for read operations. Reading from a record now involves holding a read lock, which requires expensive atomic compare-and-swap operations in shared memory.

Optimistic concurrency control (OCC) addresses the above problem by assuming that transaction conflicts are unlikely. In a typical OCC implementation, reading from records do not hold locks, but observes a transaction timestamp associated with the record and stores it in the transaction's *read set*. Writes in a transaction do not take effect immediately, but are buffered in the transaction's *write set*. Read and write sets of a transaction are memory private to the transaction and not visible to anyone else. When the transaction finishes executing, the transaction executes a *commit protocol*, which validates the observations in the read set to make sure they are still valid, and make all buffered writes visible while holding the appropriate locks. From a high level, the commit protocol ensures isolation by checking that the transaction's read set does not overlap with any other potentially conflicting transactions' write sets. An OCC transaction aborts if any inconsistencies are detected during the commit protocol or if it fails to acquire all the locks needed to apply the buffered writes. If this happens, the transaction is said to abort, and the caller of the transaction can choose either to retry its execution or to abandon the transaction.

Concurrency control mechanisms can also resolve conflicts in creative ways such as storing multiple versions of the same record. In *multiversion concurrency control* (MVCC), copy-on-write is used when a record is updated, creating a chain of versions sorted by the versions' creation timestamps. Reading from a record now also specifies a *read timestamp*, and it then searches within the version chain of the record to find the version that is visible at the read timestamp. MVCC prevents writes from clobbering reads in a different transaction, while allowing both to proceed at the same time.

1.5 Improving Main-memory Transaction Processing Performance

Due to the crucial role of main-memory databases in enabling responsive applications, improving the performance of contended main-memory transactions is subject of intense study [18,27,29,37,42,43,56–58,64]. Many works focus on addressing known deficiencies in concurrency control mechanisms.

It is understood that while systems based on optimistic concurrency control (OCC) achieve great performance in workloads with low contention. This is because read operations in OCC do not issue any writes to shared memory, meaning that they do not require exclusive ownership of the corresponding cache lines and less cache coherence protocol traffic. However, the validation step in OCC's commit protocol are likely to detect conflicts in heavily contended workloads. Frequent validation failures trigger constant roll-backs and retries of entire transactions, leading to wasted work and even livelock or starvation situations. It is therefore believed that OCC performs poorly in high-contention workloads, leading to performance *collapse* (i.e. transaction throughput crashing to near zero) at very high contention due to a lack of progress guarantees in their optimistic algorithms.

This understanding has prompted numerous recent concurrency control designs alternative to OCC. They include partially pessimistic concurrency control [57], dynamic transaction reordering [64], and MVCC [30, 37]. All these designs claim to achieve superior performance over OCC in high contention workloads, while preserving or even surpassing OCC's performance at low contention.

Careful examination of the methodologies of these studies, however, raises questions. In these studies, when measuring the performance of an existing OCC-based system (often Silo [56]), experiments are often done using the system's original code base. The newly proposed system, however, is measured using its own separate code base. While under-

standable, it raises the possibility that any performance differences between the systems could be due to difference in *implementation details* instead of core concurrency control algorithm differences. More concretely, if one specific implementation of OCC performs badly under high contention, it is inappropriate to conclude that all OCC systems perform badly under high contention, and only a radically different concurrency control mechanism can be effective.

To find out whether such implementation differences exist, and if so how much they impact performance results, we analyzed the code bases of several main-memory database systems, including Silo [56], DBx1000 [63], Cicada [37], ERMIA [30], and MOCC [57]. We found many underappreciated engineering choices – we call them *basis factors* – that dramatically affect these systems' performance in both high and low contention. For instance, some transaction abort and memory allocation mechanisms can exacerbate contention by obtaining a hidden lock in the language runtime, unnecessarily bottlenecking performance at high and low contention, respectively.

To better isolate the impact of concurrency control *algorithms* on performance, we implement and evaluate three concurrency controls – OCC, TicToc [64], and MVCC – in our system, called *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. We show that with good basis factor choices, OCC does not suffer from performance 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 [20] and MVCC performing well at all contention levels [37].

In addition, our insight into transaction performance in main-memory databases prompts us to propose, implement, and evaluate two optimization techniques that improves transac-

tion processing performance for all concurrency control mechanisms we evaluated, across a variety of high-contention workloads. The two techniques, called *commit-time updates* and *timestamp splitting*, effectively eliminate classes of conflicts that are common in our workloads. These techniques have workload-specific parameters, but they are conceptually general, and we applied them without much effort to every workload we investigated. We also developed tools to partially automate their applications in STOv2. 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 inspired by existing work, but we believe we are the first to implement the two techniques' application to TicToc and MVCC. Our insight that they effectively eliminate many common concurrency control conflicts in a variety of workloads is also new.

In summary, our study reveals results that challenge the conventional wisdom of high-contention main-memory transaction performance. We find that OCC performs better than previously reported in a variety of real-world-inspired high-contention workloads. We believe that basis factors unrelated to concurrency control intrinsics contributed the discrepancies between our and prior studies. We also identify two optimizations that help reduce concurrency control conflicts in contended workloads. We demonstrate that these optimizations benefit all concurrency control algorithms we study, are effective in all workloads we measure, and achieve greater performance benefits than concurrency control improvements alone.

The rest of the dissertation is organized as follows. Chapter 2 describes related work in the field of main-memory transaction performance studies. Chapter 3 provides background information on STOv2, the main-memory database system we use for this study, including information on STO [27], a software transactional memory framework upon which STOv2

is based. Chapter 4 describes our experimental setup to facilitate reproduction of our results. Chapter 5 presents a controlled study of the baseline performance of three concurrency control mechanisms we evaluated, where we identify and enumerate the impact of basis factors. Chapter 6 describes commit-time update and timestamp splitting high-contention optimizations and evaluates their effects in our suite of benchmarks. Finally, we conclude by discussing current limitations and promising directions for future work in Chapter 7.

Chapter 2

Related Work

Improving and understanding transaction processing performance is an important and active area of research, and there are numerous exiting work related to our study. Most of them propose new concurrency control algorithms designed for high-contention workloads, while a few of them also demonstrate the impact of non-concurrency-control factors. There are also significant precursors to the two high-contention optimizations we describe in Chapter 6. Many ideas and observations from this work are also related to insights from transactional memory system studies.

2.1 Modern Concurrency Control Research

Concurrency control is a central issue for databases and work goes back many decades [24]. 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" [33]. 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 [56, 65] introduced an OCC protocol that, unlike other implementations [13, 33], 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 § 5.1.

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 [64]. 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 [16] 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 [57] and ACC [53] 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 [6,48] systems, such as ERMIA [30] and Cicada [37], 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 [57]. 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 our OCC system outperforming Cicada at low contention (Figure 5.8b). We believe the explanation involves basis factor choices in Cicada's OCC comparison systems. Our 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 [51] 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 [58] 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.

2.2 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× [19]. A separate study presented a detailed evaluation of implementation and design choices in main-memory database systems, with a heavy focus on MVCC [62]. 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 [23] 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.

2.3 High-Contention Optimizations

Our commit-time update and timestamp splitting optimizations have extensive precursors in other work. They derive from the same intuition as many existing work that *concurrency control conflicts* in database workloads can be alleviated if the system has a more granular understanding of the workload.

Timestamp splitting resembles row splitting, or vertical partitioning [44], which splits records based on workload characteristics to optimize I/O. Taken to an extreme, row splitting leads to column stores [34, 52] or attribute-level locking [38]. Column families [11], which are very similar to timestamp splitting, are already supported by commercial database systems. The goal in these systems are mainly to reduce data movement and copying overhead when accessing large rows. We show that timestamp splitting, especially when combined with commit-time update, achieves significant concurrency control performance benefits that have not been demonstrated before.

Commit-time updates share insight with existing, decade-old techniques such as Fast Path [22] and Escrow Transactions [45] that for certain updates, locks (or observations of concurrency control timestamps) need not be held for the duration of the transaction. While based on similar observations, our application of commit-time updates in the context of modern in-memory database implementations, especially in modern MVCC implementations, is new. Our work also quantitatively demonstrated that the performance benefits to

be gained by revisiting these decade-old ideas are greater than those achieved by recent concurrency control innovations alone. We also discover an interesting synergy between our two optimizations, showing that granularity improvements such as timestamp splitting can expose more opportunities for commit-time updates to take advantage of blind write semantics.

Commutativity has long been used to improve concurrency in databases, file systems, and distributed systems [3, 32, 43, 49, 50, 61], 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 [28, §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 [25] 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.

2.4 Transactional Memory

Extensive experience with transactional system implementation is also found in the software transactional memory space [14,17,26]; there are even multiversion STMs [9,21]. Efficient STMs can run main-memory database workloads, and we base our platform on one such system, STO [27]. Some of our baseline choices were inspired by prior STM work, such as SwissTM's contention regulation [17]. 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 [42]. 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) [35, 59, 60]. HTM can also be used to implement efficient deadlock avoidance as an alternative to bounded spinning [59].

Chapter 3

STOv2 System Description

3.1 The STO Software Transactional Memory Framework

STO [27] is a novel software transactional memory framework. Like conventional transactional memory systems, STO uses OCC. Unlike conventional transactional memory systems that track untyped memory reads and writes, STO uses the concept of *transactional datatypes*, which allows objects to interact with the transactional memory system in type-specific ways. STOv2 is built on top of STO's transactional datatype interface to support fast main-memory transactions. While STO has many novel features as a software transactional memory system, we only describe the features relevant to understanding STOv2 here.

Transactions in STO execute in two phases: the *execution phase* and the *commit phase*. During the execution phase, the transaction executes application logic and collects transaction tracking set entries. The commit phase executes a *commit protocol* that validates the transaction tracking set and applies updates.

Transactional datatypes in STO interact with the system in two ways, one for each of the phases described above.

First, each transactional datatype implements a number of *transactional methods*, which are used in the execution phase. Each transactional method calls STO interface methods to register tracking set items with STO. Most datatypes use logical or abstract tracking set items representing the "intent" of the transactional method. For example, in a transactional set implemented as a self-balancing binary search tree, the transactional `find()` method registers only the node containing the matching value, instead of the whole path from the root node to the aforementioned node (as a conventional software transactional memory does), with STO.

Second, each datatype supplies a set of *callbacks* with STO to be used in the commit protocol. STO's commit protocol is further divided into three phases. In Phase 1, STO **locks** all entries in the write set. In Phase 2, STO **validates** entries in the read set, aborting on any validation failure. In Phase 3, STO makes updates visible by **installing** entries in the write set, and release all locks. Callbacks supplied by datatypes allow each datatype to *override* the meanings of "lock", "validate", and "install" operations to suit its unique needs. The overriding of the "install" operation is particularly interesting, as it allows performing complex operations while transaction locks are held for the underlying records being updated. We use this feature to implement commit-time updates, one of our high contention optimization techniques, in STOv2.

3.2 STOv2 Main-memory Database

STOv2 is a main-memory database engine built on top of STO. While STO supports a variety of transactional datatypes such as maps and queues to support generic programming, STOv2 focuses on the most important datatype in database workloads: *database tables*.

Database tables in STOv2 are essentially transactional key-value stores in STO. The keys are just primary keys to the table, and the values are table rows, or records. STOv2 sup-

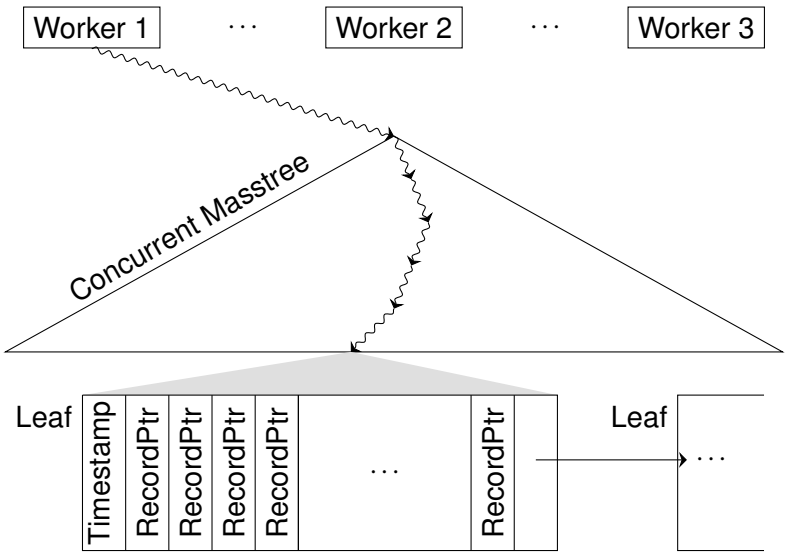


Figure 3.1: *Key-record mapping in STOv2 ordered tables.*

ports ordered and unordered tables. Ordered tables map keys to values using Masstree [39], a highly-concurrent B-tree variant that adopts some aspects of tries. Unordered tables map keys to values using a separate-chaining hash table. Only ordered tables support range scans. Secondary indexes are simply implemented as ordered tables.

STOv2 also re-engineered STO to support a variety of concurrency control algorithms. We focus on three concurrency control variants in this work: OSTO, the OCC variant; TSTO, the TicToc [64] OCC variant; and MSTO, the MVCC variant. Figure 3.1 illustrates the system architecture, showing how ordered tables map keys to database record structures.

Transactions in STOv2 are written as C++ programs that directly operate on objects representing database tables and records.

3.2.1 Phantom protection

Ordered tables in STOv2 support phantom protection when handling scans. Phantom protection guarantees that in additional to values of records visited, any *key gaps* en-

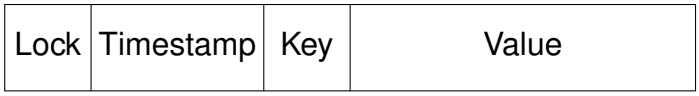


Figure 3.2: *Record structure in OSTO and TSTO.*

countered during scans are also kept transactionally consistent. This avoids "phantoms" – records that do not show up during the first scan but appear during a subsequent scan in the same transaction. STOv2 implements a tree-node-based optimistic phantom protection strategy first introduced in Silo [56]. STOv2 tracks key gaps by observing (i.e. adding to the read set) a timestamp embedded within a Masstree (leaf) node as a proxy for *the set of keys* contained in the node. Every time a key is added to or removed from the tree, the affected leaf nodes' timestamps are updated. At commit time, these node timestamps are automatically validated as part of the read set, preventing transactions having observed phantoms from committing.

This phantom protection strategy is shared by all three concurrency control variants presented in this study: OSTO, TSTO, and MSTO.

3.2.2 OSTO

OSTO, the OCC variant, follows the 3-phase STO commit protocol described in § 3.1. 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. A OSTO record structure contains a full value of the record and a concurrency control timestamp. A copy of the key is also stored so that the full key is readily available within the record itself, without having to reconstruct the key from the Masstree path. Figure 3.2 illustrates OSTO record structures.

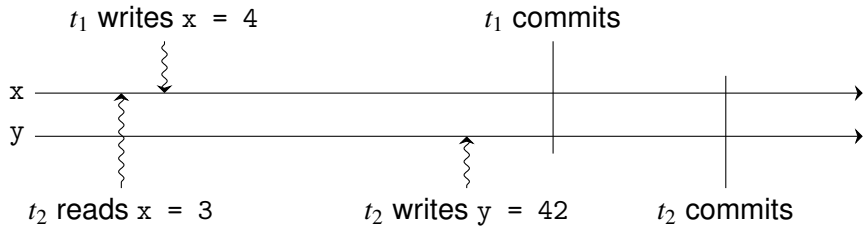


Figure 3.3: *Although t_2 finishes later in time, it can still commit if placed earlier than t_1 in the serial order. OCC will abort t_2; MVCC and TicToc can commit it.*

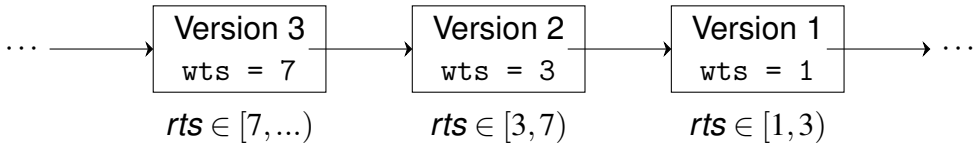
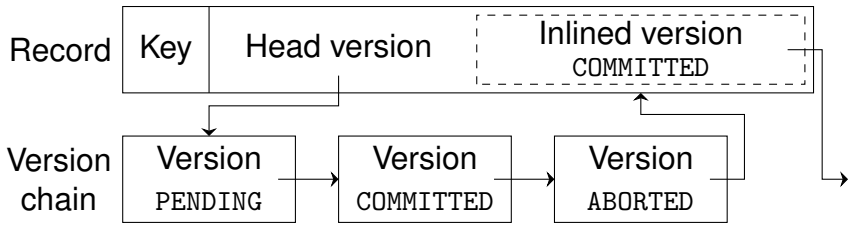


Figure 3.4: *MVCC version chain structure, with the visible range (in terms of rts) shown below each version.*

3.2.3 MSTO

MSTO is an MVCC variant based broadly on Cicada [37], 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 3.3. However, these benefits come at the cost of memory usage, which increases memory allocation and garbage collection overhead and adds pressure on processor caches. MSTO also invokes atomic memory operations 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*. Each version in the version chain is tagged a *write timestamp*, indicating the creation time of the version. When chained together, the write timestamps implicitly convey a visible time range for each version in the version chain. In general, a version is visible from its creation



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

(b) *Version chain element in MSTO.*

Figure 3.5: *Record structure and in MSTO.*

time up to, but not including, the creation time of the next newer version in the chain. See Figure 3.4 for an illustration.

A record in MSTO consists of a key and a pointer to the most recent version, or *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 that points to the previous version in chronological order. The write timestamp is the timestamp of the transaction that created the version; it is analogous to an OSTO record's timestamp. The read timestamp is the timestamp of the most recent transaction that observed the version. It is only safe to commit a new version (i.e. to insert an preceding version to the version chain) if the overwriting version's write timestamp is higher the read timestamp of the overwritten version. This ensures that no committed observations are invalidated after the new version is installed. As in traditional MVCC version chains, the chain is sorted by the write timestamps of versions: a valid 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. Figure 3.5 illustrates MSTO records and their version chain element structure.

Before initiating a transaction, MSTO assigns an execution timestamp ts_{th} used for all

observations during the execution phase. For transactions identified in advance as read-only, $ts_{th} = rts_g$; otherwise, $ts_{th} = wts_g$. rts_g is advanced periodically by aggregating all threads' most recent commit timestamps to ensure that all future read/write transactions will have greater timestamps. This makes read-only transactions read from a recent snapshot of the database, allowing them to skip the commit phase entirely after the execution phase.

MSTO adapts Cicada's commit protocol to fit the three-phase commit protocol in STO. Cicada's commit protocol is slightly different from traditional MVCC. Cicada does not use traditional locks when installing a new version to the version chain. Instead, it inserts a tentative version, called a `PENDING` version, to the version chain in a lock-free way at the beginning of the commit protocol. Since the transaction is not guaranteed to commit by the time the `PENDING` version is inserted, concurrent readers observing this version will wait until the version resolves into a non-pending state. If the transaction aborts, the version resolves to an `ABORTED` state and the concurrent reader would skip the version to continue its search. If the transaction succeeds, the version resolves to a `COMMITTED` state and the reader can return the value stored in the version. The atomic installation of `PENDING` versions make Cicada more efficient than traditional MVCC in write-heavy workloads as it does not need to handle deadlocks.

During commit, 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

```
T1:                    T2:
  read Record A
                         write Record A
  write Record B
```

Figure 3.6: *A transaction interleaving that is allowed in TicToc but not in plain OCC.*

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 skip the commit protocol.

The commit protocol in MSTO/Cicada has the unique property that the atomic insertions of `PENDING` versions, which occurs during the *lock* phase of the 3-phase STO commit protocol, are completely lock-free. This make deadlock avoidance irrelevant in MSTO, and makes MSTO highly efficient in workloads with high degrees of write-write contention, as we will see later in Chapter 6.

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 misses, 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 [37, §3.3]).

3.2.4 TSTO

TSTO is an OSTO variant that uses TicToc [64] in place of plain OCC as the concurrency control mechanism.

TicToc is built on the observation that some transaction OCC decides to abort actually does not need to be aborted. Consider two transactions interleaved as shown in Figure 3.6. Under plain OCC, `T1` has to abort because its commit-time validation will detect record A

has been modified. However, the two transactions can both commit and the end result is still serializable: `T2` can serialize after `T1`.

TicToc captures this by dynamically computing the commit timestamp, which indicates serialization time, using the read and write timestamps of the records a transaction accesses. The commit timestamp is computed to be higher than or equal to any such write timestamps, and strictly higher than read timestamps of items in the write set. In the example in Figure 3.6, TicToc will assign a lower commit timestamp for `T1` between the two transactions. The commit protocol then extends all the read timestamps of observed records to the commit timestamp. The read timestamps function similarly to those in MSTO–they prevent concurrent updates from retroactively invalidating a committed read. TicToc ensures that such concurrent updates do not succeed by involving the read timestamps of write set entries in the computation of the commit timestamp, making sure the commit timestamp must be strictly higher than any such read timestamp. When the computed commit timestamp is higher than the observed read timestamp of a record in the read set, TicToc performs an OCC-style validation on the record to make sure the write timestamp of the record has not changed, and aborts the transaction otherwise.

Compared to plain OCC, TicToc allows for more flexible transaction schedules at the cost of more complex timestamp management. Specifically, during the validation step, TicToc has to atomically extend the read timestamps of the records it has observed to the computed commit timestamp of the transaction, which causes read operations to also write to shared memory. Compared to MVCC, however, it is simpler in that it requires no global timestamp coordination and only stores the most recent version of a record to avoid version chain complexities.

Except for concurrency control, TSTO and OSTO share the identical infrastructure.[1]

[1]TicToc requires special care to support full phantom protection. We discuss the cost of full phantom protection in TicToc in Chapter 7.

TSTO uses the same record structure as OSTO in Figure 3.2, except that the timestamp is now a two-part TicToc timestamp instead of a single OCC timestamp. We do not use the TicToc delta-*rts* encoding [64, §3.6], which we found to lead to false aborts in read-heavy workloads; instead, we use separate, full 64-bit words for the read and write timestamps. The false aborts occur because the delta encoding uses only 15 bits to encode the distance between the read and write timestamps of a record, so that the two timestamps can fit in a single 64-bit word for atomic operations. Because the read timestamp is extended every time a record goes through read set validation, the distance between read and write timestamps for frequently-read records can grow beyond what the 15-bit delta field can represent. TicToc's delta encoding preserves correctness in this case by overflowing the delta field into the write timestamp, inducing (false) validation failures and transaction aborts while preserving correctness. We find that it is not necessary to fit both timestamps in a 64-bit word for correct atomic operation, and using full 64-bit words for read and write timestamps is fine provided that the read timestamp is always accessed before the write timestamp. On x86-TSO machines, this can be easily achieved using a compiler fence.

3.2.5 Garbage collection

STOv2 uses a epoch-based garbage collection mechanism for all three concurrency control variants. The mechanism is similar to the garbage collection algorithm used in read-copy-update (RCU) [40] but with modifications adapting it for use in MVCC systems.

In OSTO and TSTO, deleted objects cannot be immediately deallocated (i.e. returned to the allocator or the operating system) because concurrent transactions could still be accessing them. Each thread therefore marks an object for deallocation at some point in the future and defers the actual deallocation until that time. In MSTO, garbage collection is even more crucial, because without it the version chains would grow indefinitely and quickly exhaust

Epoch	Name	Definition
Global write	we_g	Periodically incremented
Thread-local write	we_{th}	Per-thread snapshot of we_g; used to mark objects for deletion
Global read	re_g	$< \min_{th} we_{th}$
Thread-local read	re_{th}	Per-thread snapshot of re_g
Global GC	gce	$< \min_{th} re_{th}$

Table 3.1: *Garbage-collection-related epochs in STOv2. For any thread th and at all times,* $we_g \geq we_{th} > re_g \geq re_{th} > gce$.

all memory.

We made the observation that the same idea applies to garbage collection in all three concurrency control mechanisms, so the three concurrency control variants in STOv2 can share the same garbage collection mechanism. Despite differences in concurrency control mechanisms, all garbage collection mechanisms achieve the same goal – assigning every object marked for deletion expiration times after which it is safe to physically deallocate the object.

STOv2's common garbage collection mechanism operates on a set of coarse-grained epochs listed in Table 3.1. An epoch advancer thread runs in the background, periodically incrementing the global write epoch (we_g). This thread runs rather infrequently (about only once every millisecond), so the epochs are coarse-grained. Every transaction thread, before transaction start, takes a snapshot of the global write epoch and stores it as a thread-local write epoch (we_{th}). This is also the epoch a transaction uses to mark objects for deletion.

The epoch advancer thread also periodically scans the we_{th} values in all threads and computes a minimum, storing it as the global read epoch (re_g). Upon transaction start every transaction thread also takes a snapshot of the global read epoch, storing it as the thread-local read epoch (re_{th}). The global garbage collection (GC) epoch (gce) is then computed as the minimum of all thread-local global read epochs, also by the epoch advancer thread

by scanning all thread-local read epochs periodically. All these timestamps increase only monotonically every time the epoch advancer thread runs.

Transaction threads mark objects for deletion by putting them onto a per-thread garbage collection queue. As mentioned earlier, every object marked for deletion is tagged with the thread-local write epoch (we_{th}) at the time the object is put on to the garbage collection queue. This marks the expiration time of the object. Between transactions, the transaction thread runs a maintenance function that checks this queue and deallocates any object in the garbage collection queue that is safe to delete. It is safe to deallocate an object only after the global garbage collection epoch advances beyond (strictly greater than) the expiration time with which the object is tagged.

Correctness analysis for OSTO and TSTO

We now demonstrate the correctness of the garbage collection mechanism in OSTO and TSTO using the following logical analysis. In OSTO and TSTO, an object is deleted by a transaction only after it is physically "unlinked" from a shared transactional data structure. This means after the transactional deletion occurs, future transactions can no longer reach the object being deleted. Garbage collection then just needs to make sure that all concurrent transactions that could potentially hold references to the deleted object finish before physically deleting the object. By the time the global garbage collection epoch advances beyond the thread-local write epoch (*we*) at the time of deletion, all threads must have thread-local write epochs greater than *we* (see the epoch invariant in Table 3.1). This that means all transactions currently active in the system are now running after the deleted object was physically unlinked from shared data structure, so it is safe to physically delete the object.

Note that in OSTO and TSTO, we could have simply used the global read epoch (re_g) as the garbage collection epoch, making the mechanism identical to the one in RCU. The current mechanism is therefore more conservative than necessary. We need the global read

epoch to GC epoch indirection for correct garbage collection in MSTO, and we decided to keep the same mechanism for OSTO and TSTO.

Correctness analysis for MSTO

Garbage collection is slightly more complicated in MVCC, because old versions are never explicitly deleted from MVCC version chains. Instead of being explicitly deleted, an old versions quietly *expires* once no future transactions will read that version. In practice, it means that all reads from future transactions will be *caught* by a newer version.

To achieve this, we mark old versions for deletion when a new, committed version is installed. The global write epoch (we_g) at the time the *new* version is installed is used to mark the expiration time of the old version. Unlike in OSTO and TSTO, we must use an extra indirection involving aggregating all thread-local read epochs to compute the global garbage collection epoch (*gce*). This is due to the fact that read-only transactions in MSTO run in the recent past at timestamp rts_g (see § 3.2.3).

Once *gce* advances beyond the expiration time of an old version, all active transactions in the system, including read-only ones, must be running with re_{th} values greater than the expiration time of the old version, which is the we_{th} value when the newer version is installed. This in turn means that all transactions are now executing with read timestamps greater than the write timestamp of the new version, therefore all reads will be caught by the new version.

This setup allows the version chain to contain dangling pointers (i.e. pointers pointing to expired old versions). As a result there is no need to scrub pointers when garbage-collecting old versions. Since all transactional reads will be caught by newer versions, the dangling pointers are guaranteed to never be dereferenced.

3.2.6 Deletes in MSTO

Deletes in MSTO are tricky because in a multi-version system, a delete should not physically unlink a record from the index until no transactions are executing at a timestamp before the record has been deleted. One way to achieve this is to never physically unlink any records – any records that ever existed in the system will be preserved. While this works, it creates the problem that a range scan may need to skip through a potentially unbounded number of logically deleted records.

MSTO addresses this "multi-version deletion" problem by lazily cleaning up unneeded records. This cleanup is executed by the garbage collection mechanism. Installation of a version with a "deleted" status enqueues a garbage collection callback that is invoked when the version immediately older than the deleted version expires. When the callback executes, it 1) searches the index structure to check whether the deleted version is the *only* unexpired version in the version chain of the record, and 2) if true, physically unlinks the record from the index structure. The two steps above are executed as a critical section using Masstree's internal locking to ensure their correctness with respect to concurrent Masstree inserts, deletes, and lookups.

Chapter 4

Experiment Setup

We demonstrate our findings by measuring the throughput of our database system using a suite of benchmarks. These benchmarks are shared in the evaluation of both our baseline system in Chapter 5 and the optimized system in Chapter 6. We use this brief chapter to describe the experimental platform and benchmarks we used to conduct these experiments.

4.1 Experiment Setup

We conduct our experiments on Amazon EC2 m4.16xlarge dedicated instances, each powered by a pair of Intel Xeon E5-2686 v4 CPUs clocked at 2.30 GHz. Each machine is equipped with 32 CPU cores (64 hyperthreads) and 256GB of RAM, evenly distributed across two NUMA nodes. Medians of 5 runs are reported with minimum and maximum values shown as error bars. Some results show little variation, so error bars are not always visible. In all experiments, aborted transactions are automatically retried on the same thread until they commit successfully.

4.2 Workloads

We measure two standard benchmarks, YCSB (A and B) [12] and TPC-C [54], 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" [55, §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 worker thread, contention is relatively rare (cross-warehouse transactions still introduce some conflicts); when many threads access one warehouse, many transactions conflict. We enable Silo's fast order-ID optimization [56], which reduces unnecessary conflicts between new-order transactions. For tables that are insert-only in the workload (e.g. the `HISTORY` table), we generate unique primary keys for each new row locally on each thread to reduce contention on the right-most leaf node of the table. We implement contention-aware range indexes (§ 5.1.5) 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 bench-

mark 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 [15]. Our RUBiS workload is the core bidding component of the RUBiS benchmark [46], which models an online auction site. Both benchmarks are naturally highly contended. Whenever necessary, indexes use Masstree to support range queries.

We also evaluate the TPC-C benchmarks in other implementations, 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.

Chapter 5

Understanding Baseline Performance

5.1 Basis Factors

Main-memory transaction processing systems differ in concurrency control algorithms, 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*, which are design choices that 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 stems from particular basis factor choices, but not inherent limitations of the concurrency control algorithm. 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 5.1 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

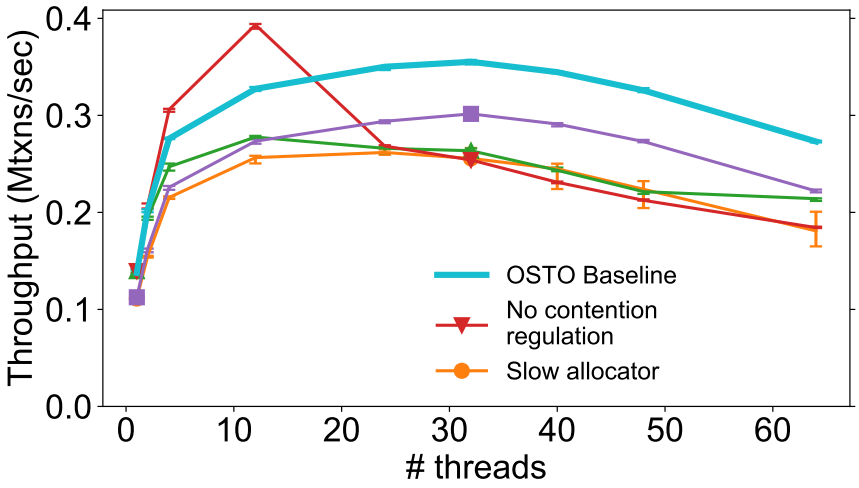
high-contention TPC-C, four factors have 20% or more impact on performance, and one factor can cause collapse. We will show in Chapter 7 that the lack of contention regulation combined with an additional point of contention in the benchmark also leads to surprising performance collapses. In TSTO and MSTO, the basis factors have similar impact, except that memory allocation in MSTO has larger impact due to multi-version updates. We omit these results to avoid repetition.

5.1.1 Contention regulation

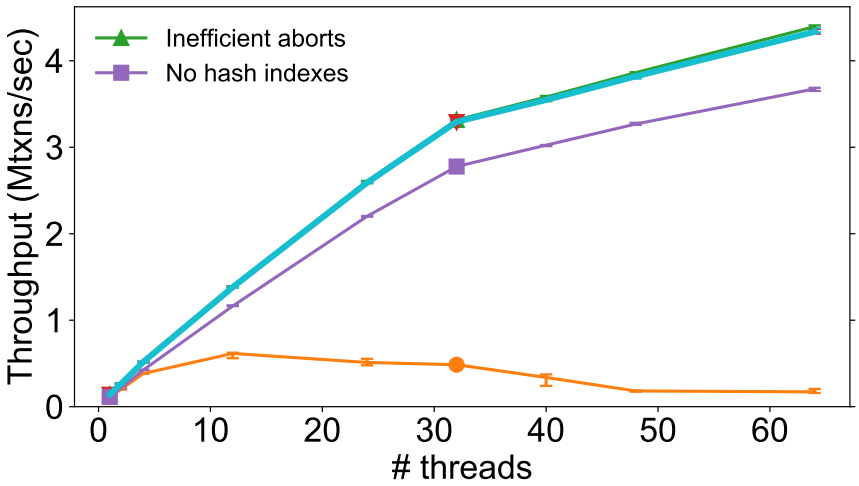
Contention regulation mechanisms in transaction processing systems, especially those with OCC-like concurrency control algorithms, increase the chance of forward progress in highly contended workloads. They help prevent live lock and starvation situations due to repeated conflicts and retries. Contention regulation can be implemented at different levels in a transaction-processing system. At a low level, locks may implement contention regulation by backing off after a failed compare-and-swap to avoid repeated cache line invalidations. At a higher level, contention regulation mechanisms may inject a delay before retrying a transaction that repeatedly aborts.

We focus on contention regulation at the transaction level. Figuring out the right amount of delay before retrying the aborted transaction (due to conflict) is critical. Over-eager retry leads to excessive cache line bouncing and even contention collapse; over-delayed retry wastes system resources by leaving 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 [41] 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 5.1 show OSTO performance with no



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



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

Figure 5.1: *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.*

backoff before retrying aborted transactions. Lack of contention regulation leads to performance collapse as contention gets extreme, as demonstrated by the *sharp* drop in performance from 12 to 24 threads in Figure 5.1a. It is worth noting that Silo [56], a popular system frequently regarded as a state-of-the-art OCC implementation in performance stud-

ies, disables backoff by default. Silo supports exponential backoff through configuration, but some comparisons using Silo have explicitly disabled that backoff, citing (mild) overheads at medium contention [36]. This is an unfortunate choice for evaluations including high-contention experiments.

5.1.2 Memory allocation

Transactional systems stress *memory allocation* by allocating and freeing many records and index structures. This is particularly relevant for MVCC systems, where every update allocates memory 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 [47]. 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 (§ 5.2.3). 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 are never inserted into – and should be avoided.

The "Slow allocator" lines in Figure 5.1 show OSTO performance using the default glibc memory allocator. Silo uses the system default allocator [56], though it can be configured to use other general-purpose allocators that are `malloc`-compatible. OSTO with rpmalloc performs 1.5× better at high contention, and at low contention the glibc allocator

becomes a bottleneck and stops the system from scaling altogether.

5.1.3 Abort mechanism

High-contention workloads stress *abort mechanisms* in transaction processing systems due to the high rate of aborts occurring in the system. High abort rates do not necessarily correspond to lower throughput on modern systems, and in particular, reducing abort rates does not always improve performance. We find this to be true both in our experience (see Chapter 7) and in prior work [37]. However, some abort mechanisms impose surprising overheads. C++ exceptions – a tempting abort mechanism for programmability reasons – can acquire a global lock in certain language runtime implementations (the default C++ runtime on many GNU Linux operating systems has this problem). This protects exception-handling data structures from concurrent modification by the dynamic linker. When using C++ exceptions to handle aborts, this lock becomes a central point of contention for all aborted transactions.

We are not implying that certain operating systems have bad language runtime implementations. This is more of a misuse of language features by transaction processing system designers. Exceptions, by definition, are supposed to be rare occurrences, therefore the language runtime focuses on making the non-exception common path fast at the cost of a slower exception path. To avoid such hidden overheads imposed by the language runtime, we recommend implementing aborts using *explicitly-checked return values*. In our implementation, all transactional operations that can abort return a boolean value indicating whether an abort needs to occur.

The "Inefficient aborts" lines in Figure 5.1a show OSTO performance using C++ exceptions for aborts. Original STO, Silo, and ERMIA all abort using exceptions. Fast abort support offers 1.2–1.5× higher throughput at high contention.

5.1.4 Index types

Index types refer to the types of data structures used to implement database tables or indexes. Systems typically choose from two types of data structures, ordered and unordered, to implement database tables based on workload needs. Ordered data structures are usually tree-like and support efficient range scans, while unordered data structures rely on pseudo-random hashes to support fast point queries.

In terms of functionality, ordered data structures form a superset of unordered data structures, and many databases simply use ordered data structures for all tables and indexes to simplify implementation. Silo, for instance, uses Masstree [39], a B-tree-like structure, to implement all tables. Most TPC-C implementations we have examined use unordered data structures, or hash tables more specifically, for indexes unused in range queries. Some implementations use hash tables for *all* indexes and implement complex workarounds for range queries [63]. Hash tables offer $O(1)$ access time where ordered 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* to implement database indexes when the workload allows it, and ordered (B-tree) indexes elsewhere.

The "No hash index" lines in Figure 5.1 show OSTO performance when all indexes use Masstree, whether or not range scans are required. Silo and ERMIA lack hash table support. Hash index support 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.

5.1.5 Contention-aware indexes

We discovered an interesting instance of *index contention* in TPC-C that does not greatly affect the overall throughput, but can cause starvation of certain classes of trans-

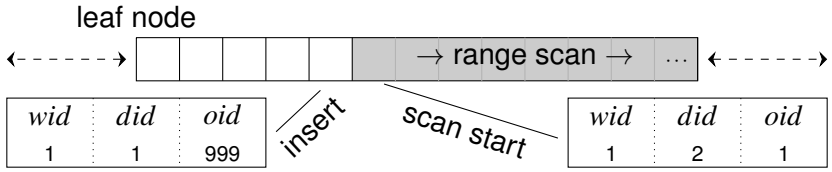


Figure 5.2: *Example illustrating index contention on the TPC-C* `NEW ORDER` *table. An insert to the end of one district in new-order can conflict with a range scan in delivery on the adjacent district.*

actions. The contention stems from disjoint ranges false-sharing of B-tree leaf nodes in secondary indexes.

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. This workload appears naturally partitionable: new-order and delivery transactions operating on distinct $\langle wid, did \rangle$ pairs need not conflict.

This is, however, not the case in some index implementations. An index may choose to treat the whole $\langle wid, did, oid \rangle$ tuple as an opaque unit, and as a result a district boundary is likely to fall *within* a B-tree leaf node. For example, $\langle wid{=}1, did{=}1, oid{=}999 \rangle$ and $\langle wid{=}1, did{=}2, oid{=}1 \rangle$ are consecutive in the key space, and they are likely to reside in the same B-tree leaf node. If this occurs, the tree-node-timestamp-based phantom protection (described in § 3.2.1) will cause new-order transactions on the earlier district and delivery transactions on the later district to appear to conflict, inducing aborts in delivery and even starving it. See Figure 5.2 for an illustration of this scenario.

We recommend implementing *contention-aware indexing*: indexes that are not susceptible to the aforementioned false sharing problem. This can be done 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 [39, §4.1]. Certain key ranges in Masstree will never cause phantom-protection conflicts. As illustrated in

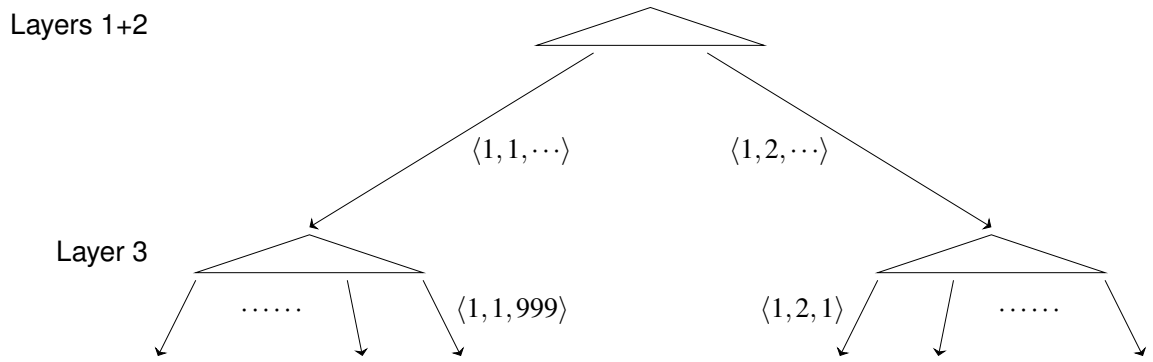
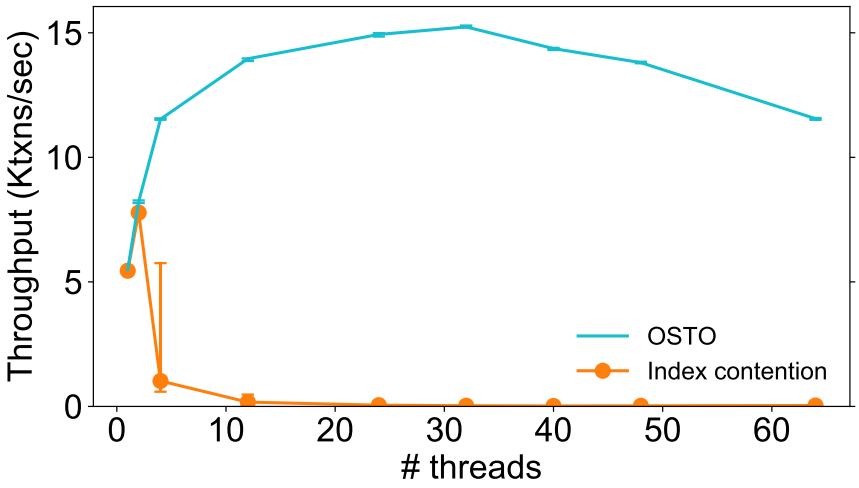


Figure 5.3: *Contention-aware index routes keys with distinct prefixes to different leaves.*

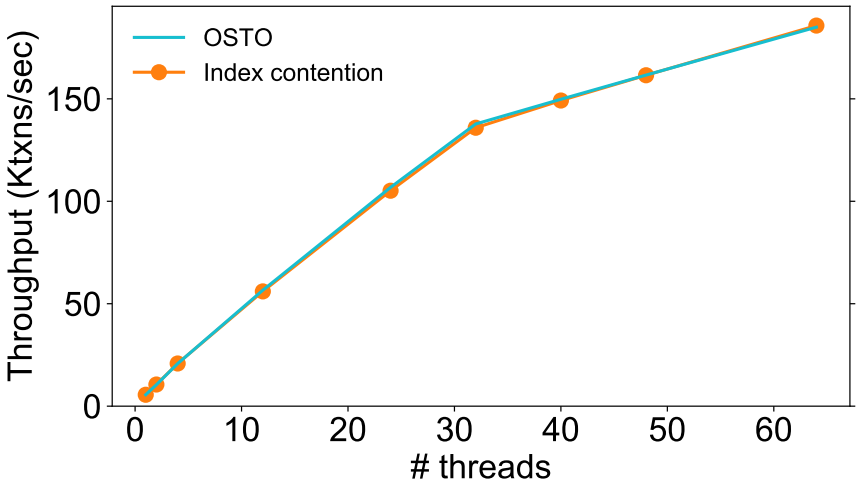
Figure 5.3, by representing a $\langle wid, did \rangle$ pair using a multiple of 8 bytes, keys with different $\langle wid, did \rangle$ prefixes are routed to different B-trees in the bottom layer, guaranteeing that these keys never share leaf nodes. Scans of all orders within a $\langle wid, did \rangle$ range thus never conflict with inserts or deletes into any other range. To implement contention-aware indexing, we 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 the index contention at the cost of larger key size (24 bytes instead of 8 bytes).

Figure 5.4a shows the impact of contention-aware indexes on *delivery* transactions in high contention TPC-C. 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 [37]. 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.

The larger key size in our contention-aware indexes adds negligible performance overhead under low contention, as shown in the results in Figure 5.4b.



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



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

Figure 5.4: *Throughput of delivery transactions with and without contention-aware indexes. Showing OSTO results with TPC-C full mix.*

5.1.6 Other factors

Other basis factors like transaction internals and deadlock avoidance have visible but more limited impact.

Transaction internals refers to the mechanisms for maintaining transaction tracking sets. These mechanisms are private to each transaction thread so they do not need to be

thread-safe, but they still need to be carefully engineered to ensure that they introduce minimal overhead in serial execution because they sit on the critical path of every transactional operation. Since optimistic transactions do not apply updates at execution time but instead buffer them in the write sets, a later read from the same transaction may need to search within the tracking set to return the buffered (or "correct") result. To facilitate this lookup, transaction tracking sets typically maintain internal state that maps the physical in-memory location of a record to the corresponding tracking set entry. Good transaction internals implement this as a fast and well-tuned hash table. We recommend strong transaction internals by default, although the factors listed before have more performance impact. Replacing STOv2's highly-engineered internals with Cicada's simpler internals reduced performance by just 5%.

Transaction processing systems also need *deadlock avoidance or detection strategies* as they can acquire more than one lock at the same time during the lock phase of the commit protocol. Transactions can access records in arbitrary orders, but the locks must be acquired according to the same order across all transactions to avoid deadlocks. Early OCC database implementations achieve this by sorting their write sets according to some global order [31,56,64]. In the context of main-memory databases, this sorting can be done quickly by using the memory addresses of the corresponding records or data structure nodes as sort keys. An alternative technique called *bounded spinning* is popular in transactional memory systems. This technique attempts to acquire locks without sorting, but instead of waiting indefinitely for the lock to become available, and then aborts the transaction at the lock phase if it takes too long to acquire a lock in the write set, assuming that deadlock is occurring. Our experience, as well as a prior study [59, §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%) in OLTP workloads as the write sets are typically small. The exception was DBx1000 OCC [64], which prevents deadlock

System	Contention regulation	Memory allocation	Aborts	Index types	Transaction internals	Deadlock avoidance	Contention-aware index
Silo [56]	−−	−−	−−	−	−	+	+
STO [27]	−−	−−	−−	+	+	+	+
DBx1000 OCC [63]	+	N/A	+	+	−	−−	−−
DBx1000 TicToc [64]	+	N/A	+	+	−	+	−−
MOCC [57]	N/A	+	+	+	+	+	−−
ERMIA [30]	+	+	−−	−	+	+	+
Cicada [37]	+	+	+	+	+	N/A	N/A
STOv2 (this work)	+	+	+	+	+	+	+

Table 5.1: *How comparison systems implement the basis factors described in § 5.1. 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×.*

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.

5.1.7 Summary

Table 5.1 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× STOv2's performance), poor ("−", 0.7–0.9×), or very poor ("−−", less than 0.7×).

5.2 Baseline 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 conten-

tion 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 $0.6\times$ that of MSTO, even at 64 threads. Neither system scales nor collapses. At low contention, however, OSTO throughput is $1.9\times$ that of MSTO. These results hold broadly for our other benchmarks.

5.2.1 Overview

Figures 5.5 – 5.7 shows the transaction throughput of all three concurrency control variants in STOv2 on our suite of benchmarks, and with thread counts varying from 1 to 64. The committed mix of transactions conforms to the TPC-C specification except in the one-warehouse, high core count settings. (In OSTO at 64 threads, 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 5.5b, and YCSB-B, Figure 5.6b) approach perfect scalability. The change in slope in these scalability graphs at 32 threads is due to hyperthreading (the machine has only 32 physical cores, and beyond 32 threads it begins to utilize hyperthreads). On high-contention benchmarks, each mechanism scales up to 4 or 8 threads, then levels off. Performance declines a bit as thread counts further increase, but does not collapse.

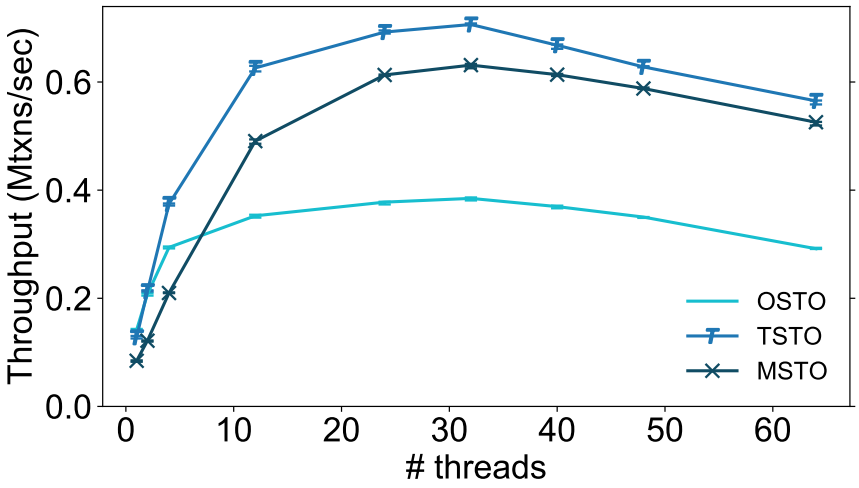
When scalability is good, performance differences are due primarily to the inherent overhead of each mechanism. In Figure 5.5b, for example, TSTO's more complex timestamp management causes it to slightly underperform low-overhead OSTO, while MSTO's

considerably more complex version chain limits its throughput to 0.52× that of OSTO.

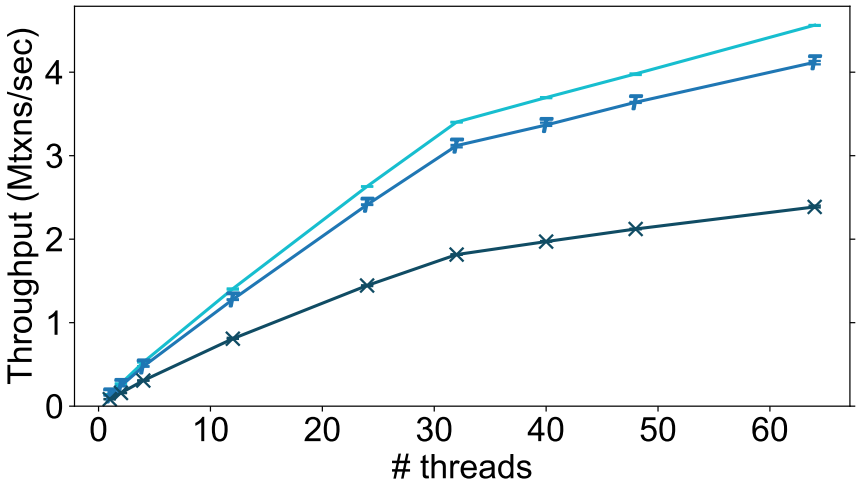
Some of the high-contention benchmarks impose conflicts that affect all mechanisms equally. For example, YCSB-A has fewer than 0.1% read-only transactions and high key skew (many transactions touch the same keys). This prevents TicToc and MVCC from discovering safe commit orders, so OSTO, TSTO, and MSTO all scale similarly, and OSTO outperforms MSTO by 1.5–1.7× due to MSTO overhead (Figure 5.6a). On other benchmarks, the mechanisms scale differently. For example, in high-contention TPC-C (Figure 5.5a), OSTO levels off after 4 threads, while MSTO and TSTO scale well to 12 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 in any concurrency control variant, and our MVCC implementation has significant overhead relative to 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 OLTP workloads with OLAP ones, where MVCC could avoid collapse by executing analytical (often 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 scaling behavior similar to OCC under contention.

Some differences from prior results are worth mentioning. Our YCSB-A results are lower than those reported previously [37]. This can be attributed to our use of the YCSB-mandated 1000-byte records; DBx1000 uses 100-byte records. Cicada's reported results for Silo and "Silo′" (DBx1000 Silo) show total or near performance collapse at high contention, but our OCC measurements show no such collapse. We attribute this difference to



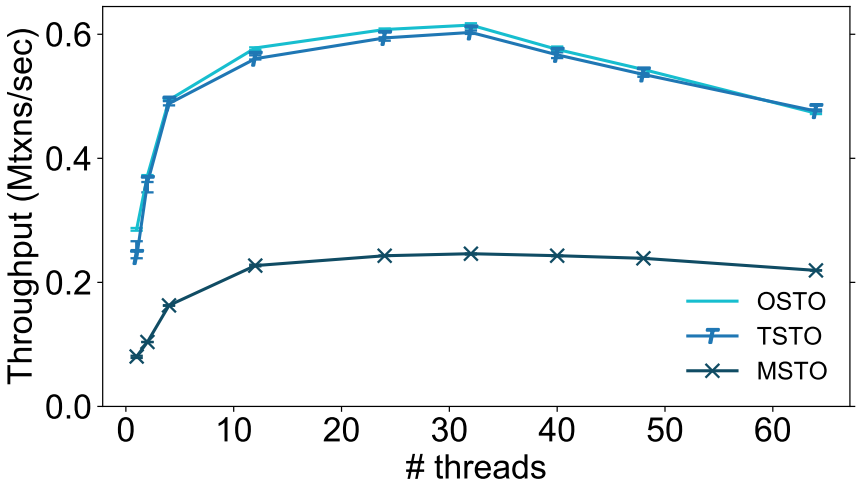
(a) *One warehouse (high contention).*



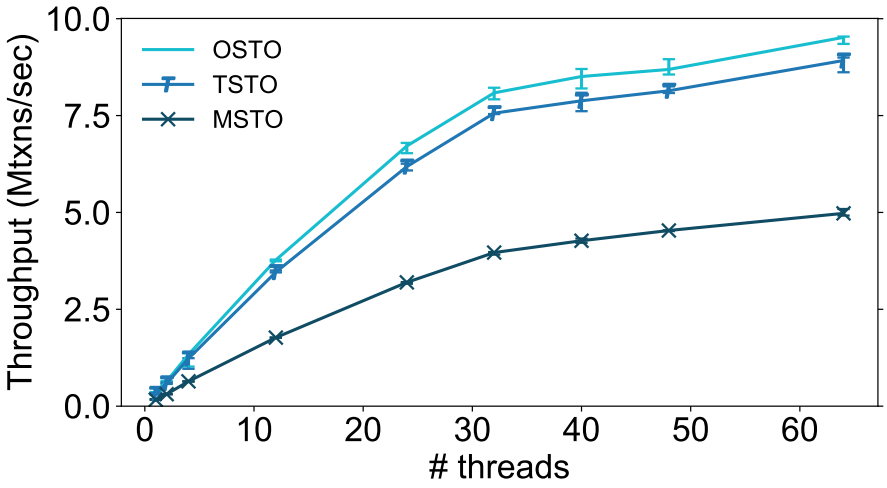
(b) *One warehouse per worker (low contention).*

Figure 5.5: *STOv2 baseline systems performance on TPC-C workloads.*

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.



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

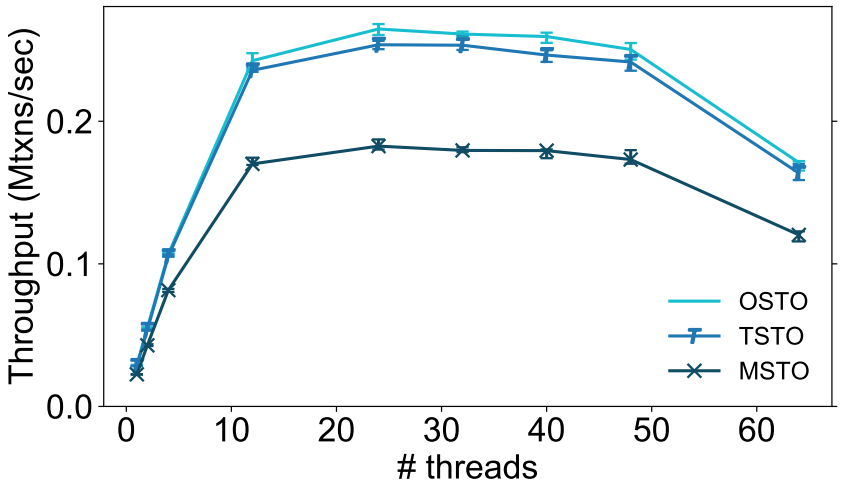


(b) *YCSB-B (lower contention: read-intensive, 5% updates, skew 0.8).*

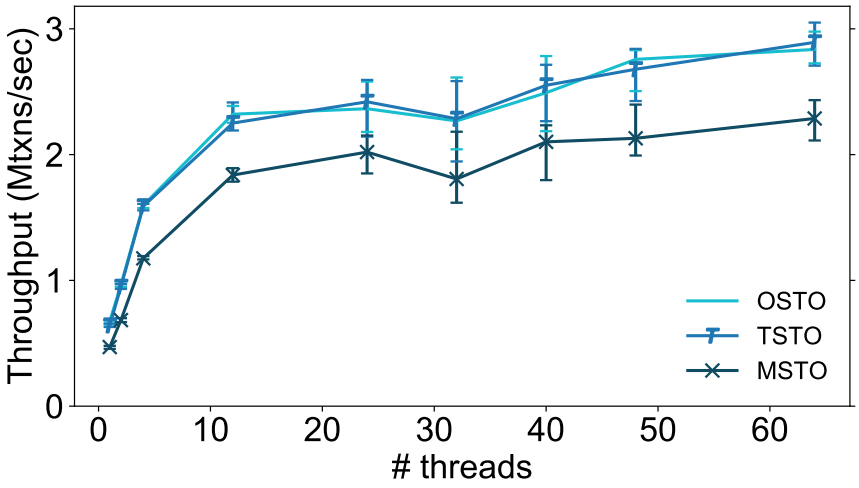
Figure 5.6: *STOv2 baseline systems performance on YCSB workloads.*

5.2.2 Benefits of reordering

Figure 5.5a (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 the version chain and garbage collection overheads and outperforms MSTO by up to 1.3×. However, this effect



(a) *Wikipedia (high contention).*

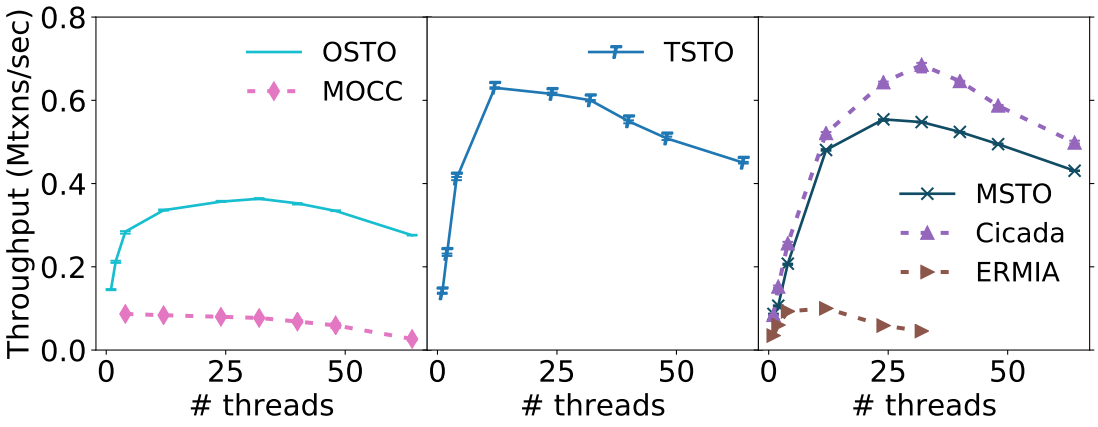


(b) *RUBiS (high contention).*

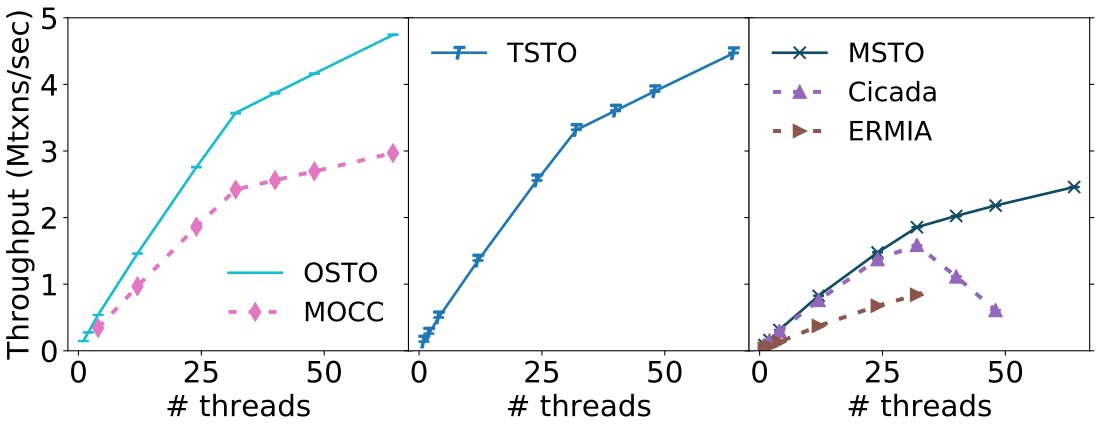
Figure 5.7: *STOv2 performance on Wikipedia and RUBiS workloads.*

is limited to TPC-C. We observed no significant benefit of TSTO over OSTO in any other workload.

Careful manual inspection of the workload reveals that 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



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



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

Figure 5.8: *Cross-system comparisons: STOv2 baselines and other state-of-the-art systems, TPC-C full mix.*

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 § 6.5

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.

5.2.3 Cross-system comparisons

Figure 5.8 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 5.8a 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 5.8b). 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 5.8b). 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.

Chapter 6

High Contention Optimizations

6.1 Overview

The baseline evaluation in the previous chapter shows that all three concurrency control mechanisms perform badly under high contention. Although none of them collapse, they all struggle to maintain performance as the concurrency level increases. This is unsurprising as the high contention workloads, by definition, contain high degrees of conflicts. However, our analyses show that many conflicts registered by concurrency control mechanisms are not necessarily conflicts mandated by workload semantics. The true level of conflicts in the workload can be greatly reduced, improving performance.

We show that by applying two optimization techniques, *commit-time update* (CU) and *timestamp splitting* (TS), we can eliminate whole classes of conflicts. They improve performance, sometimes significantly, for all of our workloads on OCC, TicToc, and MVCC. They also exhibit synergy: on some workloads, TS makes CU far more effective, and together they can achieve greater benefit than the sum of their individual benefits. Our current implementations of the these techniques require effort from transaction programmers, as the instantiation of each technique depends on the workload. This differs from concur-

rency control changes 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, especially with the help of tools that automate some of the more tedious tasks. CU and TS also eliminate classes of conflicts that concurrency control algorithms cannot, resulting in bigger improvements than those achieved by changing concurrency control alone.

6.1.1 Commit-time updates (CU)

In many implementations, transactional read-modify-write operations are represented as multiple entries in the tracking set. A read-modify-write is often broken into a read followed by a write, leading to a read set entry and a write set entry in the tracking set. This representation is general and intuitively correct, but it also means the read set entry associated with the read-modify-write operation now needs to be validated at commit time, as if the read really observes transactional state. Many classes of read-modify-write operations, however, do not actually observe, or more precisely *externalize*, transactional state.

Take the increment operation as an example. In many cases, an increment operation used in a transaction does not export the value of the record being incremented. It only cares about the increment actually takes place. This is common in transactions that maintain various counters in the database. From the perspective of workload semantics, this type of increment operations are blind writes. However, as mentioned earlier, these increment operations are represented as pairs of read and write set entries in many systems. This adds unnecessary read-write dependencies between increment operations, resulting in conflicts that are not required by operation semantics.

Commit-time updates avoid these conflicts entirely by using a special write set entry that indicates that a value should be incremented instead of overwritten, and stores only

the increment amount in the write set. The actual increment is then applied at commit time during the install phase, when the appropriate locks are already held.

We implement commit-time updates for OCC, TicToc, and MVCC. The implementation centers on function objects called *updaters* that encode operations on a datatype. A write set component can either be a full value used to overwrite the old value, as in conventional transaction processing systems, or an updater indicating a read-modify-write that does not externalize the concrete value of the record. The 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, however, invoke many updaters. Updaters currently do not support read-modify-write operations that spread across multiple records.

An updater can be used if a read-modify-write operation within a transaction does not externalize, by either data flow or control flow, the value of the underlying record. In addition to the increment operation mentioned before, many other operations also fall into this category, such as maintaining a running maximum. 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;
```

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

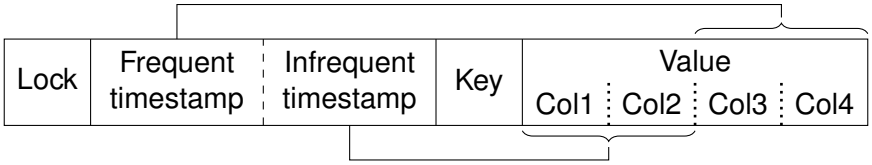
Commit-time updates relate to commutativity, which has long been used to reduce conflicts and increase concurrency in transactional systems [3, 10, 27, 61]. Though they can represent both commutative and non-commutative read-modify-write operations, they do not support some optimizations possible only for commutative operations [43].

6.1.2 Timestamp splitting (TS)

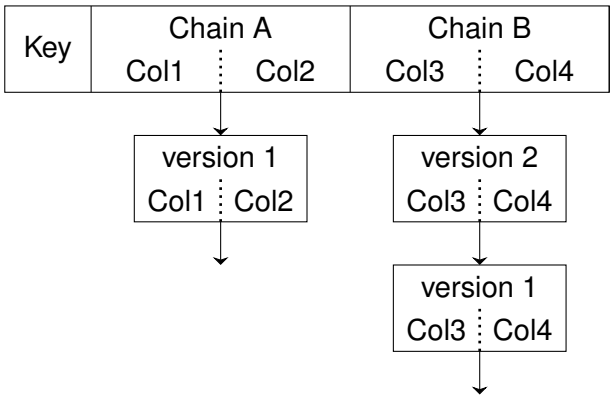
Many database records comprise multiple pieces of state subject to different access patterns. For instance, records in a relational database may have many columns, some of which are accessed more often or in different ways. Schema transformations such as row splitting and vertical partitioning [44] use 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 concurrent control conflicts.

We observe that in the database table schema used by many workloads, columns that are frequently updated and columns that are infrequently or even never updated are often kept in the same row. In conventional database systems, all columns in the same row are subject to concurrency control as a monolithic unit. This means that read accesses to the infrequently updated part of the row will seemingly conflict with an update to the frequently updated part of the row.

This occurs in a number of workloads, including the widely-used TPC-C benchmark. For example, new-order transactions in the benchmark read the "tax rate" column from the `DISTRICT` table, and payment transactions update the "year-to-date" amount in that table. In most systems new-order and payment transactions will thus conflict on the `DISTRICT` table row, although by workload semantics there isn't any read-write dependencies between the two transactions on this table row. We identified similar instances of false conflicts in



(a) *Record structure in OSTO and TSTO with timestamp splitting.*



(b) *Record structure in MSTO with timestamp splitting.*

Figure 6.1: *Record structures with timestamp splitting. Assume the record has four columns, where Col1 and Col2 are infrequently updated, and Col3 and Col4 are frequently updated.*

many other tables in TPC-C, as well as in all other benchmarks we measured. Timestamp splitting divides a record's columns into disjoint subsets, called *column groups*, and makes concurrency control operate independently for each column group. Figure 6.1 shows examples of record structures with timestamp splitting in STOv2 concurrency control variants.

In OSTO and TSTO, we simply let each record contain one *or more* concurrency control timestamps, one for each column group. When modifying a record, the system updates all timestamps that overlap the modified columns, but when observing a record, the system observes only those timestamps sufficient to cover the observed columns. In a typical example, shown in Figure 6.1a, one timestamp covers infrequently updated columns while another timestamp covers the rest of the record. Simple two-part 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 contain-

ing the customer's balance, change frequently; using a separate timestamp for name and ID allows observations that access only name and ID to proceed without conflict even as balance-related columns change.

MSTO implements timestamp splitting by including one or more version chains per-record, one for each column group, as illustrated in Figure 6.1b. The version chains operate independently as if they are different records, but they are stored under the same primary key. At execution time, the transaction computes the desired chain(s) to perform MVCC operations on based on the column access information supplied.

Although our system supports splitting records into an arbitrary number of column groups, in our evaluation we only show results with two column groups. Additional column groups come at higher costs – for instance, read and write sets as well as record layouts take more memory – and on all of our benchmarks, splitting records into more than two column groups negatively impacts performance.

Potential synergy with CU

Timestamp splitting can expose additional commit-time update opportunities. For example, this transaction appears not to benefit from commit-time updates, since an observation of `x` is externalized (by returning `x.col2`).

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

However, if `x.col1` and `x.col2` are assigned to separate column groups by timestamp splitting and treated as independent units by concurrency control, the modification to `x.col1` can be implemented via an updater – the observation of `x.col1` is not externalized.

6.2 Implementation of CU

In concurrency control mechanisms where only the latest version of each record is kept, like OCC and TicToc, updaters are invoked during the install phase (§ 3.2.2, Phase 3) of the commit protocol. They apply modifications in-place on the records directly while the corresponding transaction locks are held. This approach no longer works for MVCC because records are never updated in-place – old versions of a record are immutable, and a new version has to be created and inserted into the version chain for every update.

6.2.1 CU implementation in MSTO

We implement commit-time updates in MSTO using special version chain elements called *delta versions*. Delta versions simply contain updaters and do not represent the full value of the version, as illustrated in Figure 6.2. Delta versions are inserted into the version chain as blind writes.

Since delta versions do not contain full values, reading a delta version requires a *flattening* procedure. This procedure computes the full, materialized value of the record at the read timestamp of the transaction. It starts from the most recent full version of the record and applies all updater operations in delta versions in write timestamp order. The resulting full value of the version is then copied into delta version, turning the delta version into a full version ready for normal MVCC reads.

6.2.2 Concurrent flattening in MSTO CU

The flattening procedure is thread-safe. Each thread that flattens 1) creates a private copy of the full version used as the "basis" of the flattening procedure, 2) applies updater operations in delta versions to the private copy to build the materialized value, and 3) copies the final materialized value to the target delta version while holding a lock. If a thread

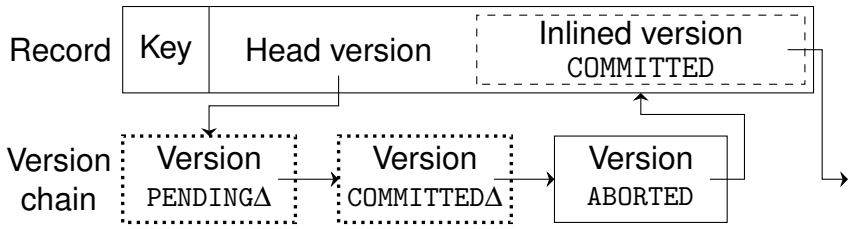


Figure 6.2: *Record structure in MSTO with commit-time updates. The* COMMITTEDΔ *version encodes an updater. Concurrent transactions can insert more delta versions either before or after the* COMMITTEDΔ*.*

detects that the delta version is locked by another thread's flattening process at any time, it can simply abandon its own flattening process and wait for the lock to become free.

This, however, works only if concurrent flattening procedures initiated at the same delta version are guaranteed to produce the same materialized value of the record. It means transactions must prevent new versions, delta or full, from being inserted to the segment of the version chain that is being flattened.

Currently we use a *flatten-freeze* technique to ensure correct concurrent flattening. The flattening procedure works in two phases, because MSTO version chains are singly-linked lists. The first phase walks down the version chain from the delta version being "read", pushing delta versions into a stack until it reaches a full version that can be used as a basis for flattening. The second phase then pops delta versions from the stack and apply updaters in them to the full version in the correct order. During the first phase that walks down the chain, the version chain is *frozen* to prevent new version insertions by extending the read timestamps of every version visited to the write timestamp of the delta version where the flattening started. This guarantees that any new versions older than the delta version (which are the only versions that could have interfered with the flattening process) cannot be inserted and committed in the version chain segment where flattening occurs. (Due to the read timestamp consistency check mandated by the MVCC commit protocol: the inserted version's write timestamp must be greater than the read timestamp of the version it over-

writes.) The first flattening procedure that finishes the first-phase version chain walk freezes the chain segment, and all subsequent flattening procedures on the same delta version are guaranteed to reach the same result.

6.2.3 Impact on MSTO garbage collection

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 marked for garbage collection. To prevent an old version from being enqueued for deletion by multiple threads, each old version also contains an atomic flag indicating whether it has already been put on the garbage collection queue of some thread. The maintenance function responsible for garbage collection also periodically performs flattening on infrequently-read records so that version chains do not grow indefinitely.

6.3 Implementation of TS

We implement timestamp splitting in STOv2 using both compile-time and runtime components.

Static workload properties are analyzed to create a plan for splitting records into column groups, expressed as a series of C++ template specializations. The template specializations map columns to the timestamp or version chain of the column group it belongs to, as well as implementing specialized copy functions that write only to affected column groups while installing new versions of column groups.

In single-version concurrency control variants (OSTO and TSTO), we do not need to alter the underlying storage of the row, but need only to re-associate timestamps to column

```
<?xml version="1.0"?>
<ts-split-definition>

    <record name="district_value">
        <split name="district_value_infreq">
            <field type="vchar" width="10" name="d_name"/>
            <field type="vchar" width="20" name="d_street_1"/>
            <field type="vchar" width="20" name="d_street_2"/>
            <field type="vchar" width="20" name="d_city"/>
            <field type="fchar" width="2" name="d_state"/>
            <field type="fchar" width="9" name="d_zip"/>
            <field type="int" width="64" name="d_tax"/>
        </split>
        <split name="district_value_freq">
            <field type="int" width="64" name="d_ytd"/>
        </split>
    </record>

    <record name=...
        ...
    </record>

    ...

</ts-split-definition>
```

Figure 6.3: *An example of the human-readable XML expression of TS policy, showing column groups for TPC-C* `DISTRICT` *table records.*

groups. In MSTO, however, different column groups have to be stored in separate objects for the version chains to operate independently from one another. This complicates the TS implementation in MSTO a little, as now a transaction that reads from multiple column groups will need to access multiple MVCC old versions. To unify the interface presented to the application, STOv2 adds an indirection called *record accessors* that internally handle this translation from a named column to the actual memory address of the column value. Record accessors are also statically expressed as C++ template specializations. Due to the large amount of C++ boilerplate code involved, we developed an automated tool that

generates all C++ template specialization code required to implement TS for all three concurrency control variants. The tool generates code from an XML document that expresses column group assignment for each record in a more human-readable format demonstrated in Figure 6.3. The analysis of workload properties currently still has to be done manually.

At runtime, each transaction now needs to supply column access information when accessing a record, specifying the columns and the type of operation (read, write, or update) it wishes to perform on each column. STOv2 then computes from the column access information the column groups being accessed, and automatically performs the appropriate operations with the corresponding concurrency control timestamps or MVCC version chains. This computation is performed dynamically at runtime to allow maximum flexibility.

6.4 Workload integration

We currently manually inspect and analyze our workloads to identify opportunities for CU and TS. Based on analysis results, we implement optimization policies in the form of C++ updater template specializations (for CU) and XML column group definitions (for TS). These definitions are then fed through the appropriate code generation tools to become part of the source code for STOv2 and applied at compile time.

We now provide some additional examples of how we exploit CU and TS opportunities in the workloads we measure. We use TS to divide records into frequently and infrequently updated column groups. For records where such access pattern is unclear, like in YCSB where column selection is random, we partition each record into two evenly divided parts: one for odd-numbered columns and the other for even-numbered ones. Update operations in YCSB are blind writes making them ideal candidates for commit-time updates. In RUBiS, an updater is used to modify an item's `max-bid` and `quantity` columns. In TPC-C, an updater on warehouse increments its `ytd` (year-to-date order amount) field, and one on

```
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;
};
```

Figure 6.4: *Updater for* `STOCK` *table records, used by TPC-C's new-order transactions. The* `operate()` *method encodes the commit-time operation.*

customer updates several of its fields for orders and payments. Currently we create updaters that are datatype-specific, meaning that only one updater can be allowed per record. An updater can still be parameterized to support different operations on a given record type.

Currently, updater code is not auto-generated, but is typically short enough to implement by hand. Auto-generation of updater code is left for future work. The shortest updater takes about 10 lines of code, including boilerplate; the longest, on TPC-C's `CUSTOMER` table, takes about 30 lines. Figure 6.4 shows a C++ definition of an updater for TPC-C `STOCK` table records that handles stock deduction and replenishment in new-order transactions.

Commit-time updates reduce transaction read set sizes significantly. For example, in TPC-C, all operations on the `STOCK` table can be implemented via commit-time updates. This reduces the read set size of each new-order transaction from 33 to 23 (30% reduction)

on average. In a TPC-C payment transaction, all warehouse and district YTD updates are commutative, so are customer YTD payment and payment count updates. Commutative updates reduce the read set size by by half, from 6 to 3, in this transaction, even without discounting the reading of constant data such as warehouse tax rate, customer name, credit, etc. Smaller read sets mean fewer read-write dependency edges between transactions and fewer conflicts.

6.5 Evaluation

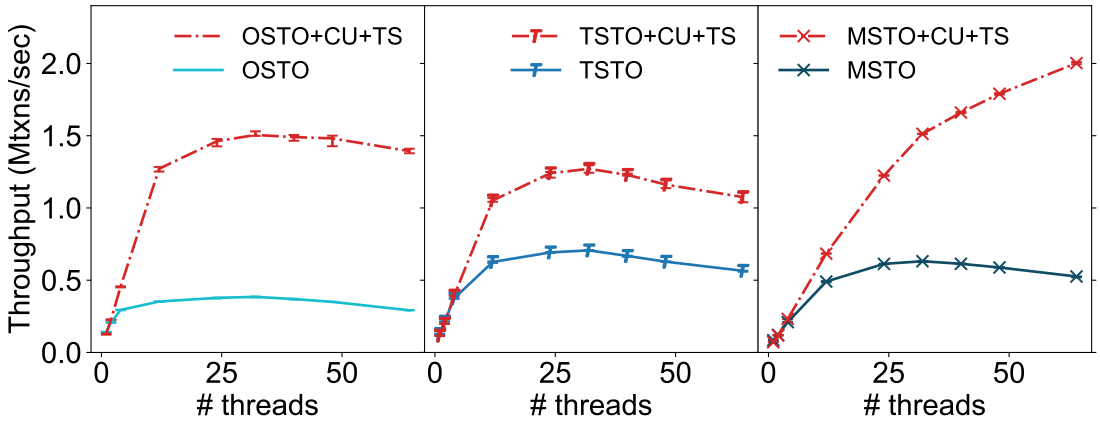
We now evaluate the commit-time update and timestamp splitting optimizations to better understand their benefits at high contention, their overheads at low contention, and their applicability to different workloads and CC techniques. We conduct a series of experiments on STOv2 with these optimizations, using all three CC mechanisms, and measure them against TPC-C, YCSB, Wikipedia, and RUBiS workloads.

6.5.1 Combined effects

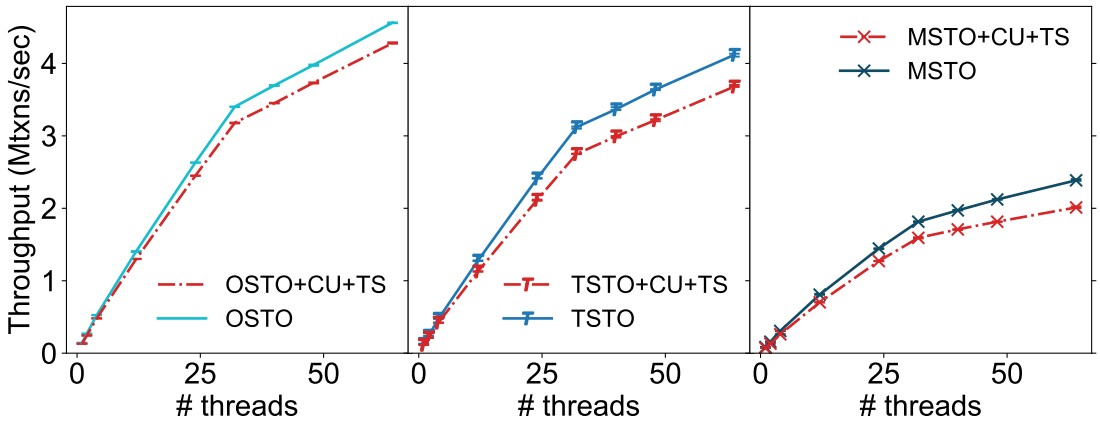
Figures 6.5–6.7 shows the effects of applying commit-time updates (CU) and timestamp splitting (TS) together under our suite of workloads.

In high-contention TPC-C (Figure 6.5a), CU+TS greatly improves throughput for all three concurrency control variants, with gains ranging from 2× (TSTO) to 4.8× (OSTO). In comparison, the best-performing concurrency control variant measured for this workload, TSTO, outperforms the lowest-performing concurrency control variant OSTO by just 1.8× (Figure 5.5a). This shows that CU and TS optimizations achieve much higher gains than concurrency control algorithm changes.

High-contention TPC-C and YCSB-A did not scale to 64 threads under any of our three concurrency control variants. However, with combined CU and TS optimizations



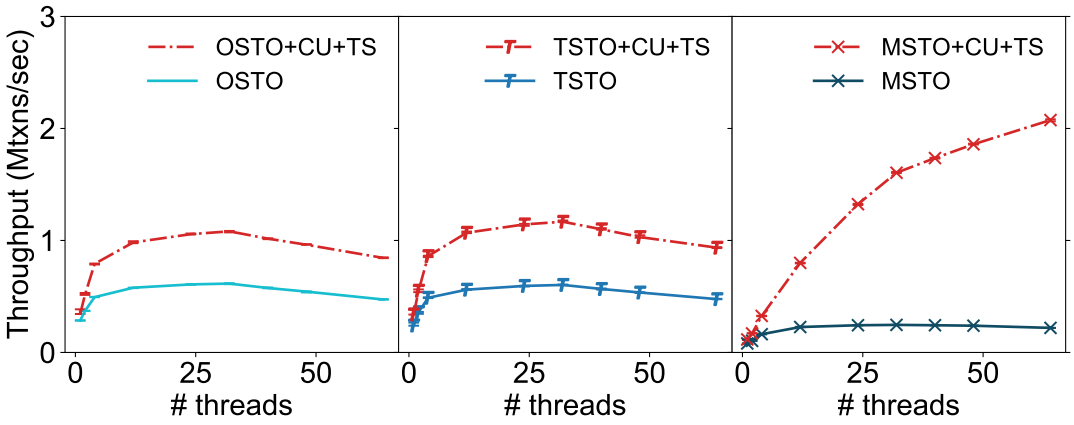
(a) *TPC-C, one warehouse (high contention).*



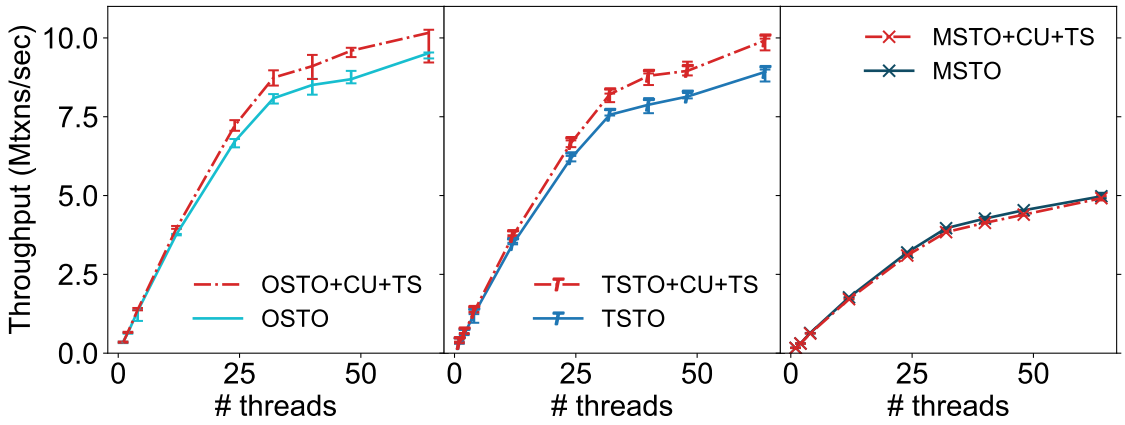
(b) *TPC-C, one warehouse per worker (low contention).*

Figure 6.5: *TPC-C results with high contention optimizations.*

(+CU+TS), MSTO becomes the only scalable variant. This is thanks to MSTO's lock-free pending version insertion technique, which makes the lock phase of the commit protocol deadlock-free even without deadlock detection. OSTO and TSTO cannot sustain throughput at high core counts due to the inherent limitations of single-version concurrency controls when handling read-only transactions. However, the absolute throughput of optimized MSTO does not outpace its OSTO or TSTO counterparts until extremely high contention (1-warehouse TPC-C at more than 20 cores), and optimized OSTO and TSTO always out-



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



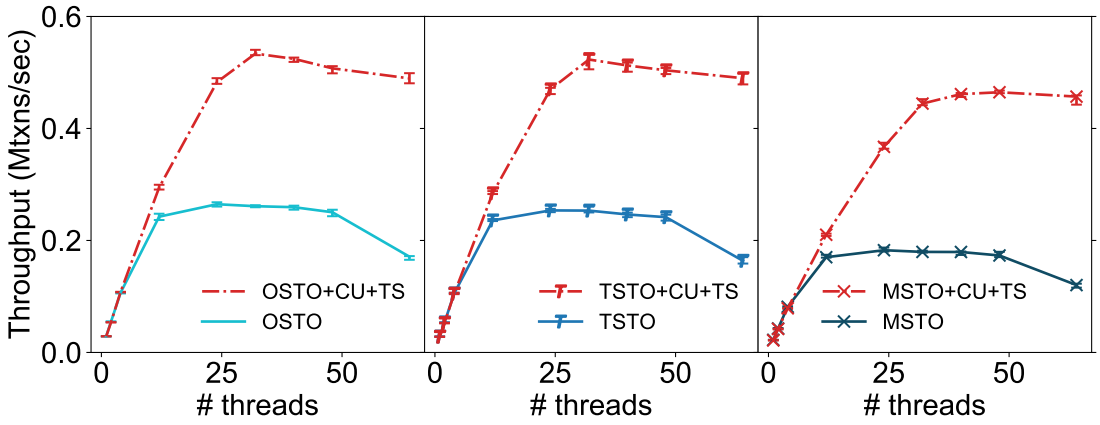
(b) *YCSB-B (lower contention: read-intensive, 5% updates, skew 0.8).*

Figure 6.6: *YCSB results with high contention optimizations.*

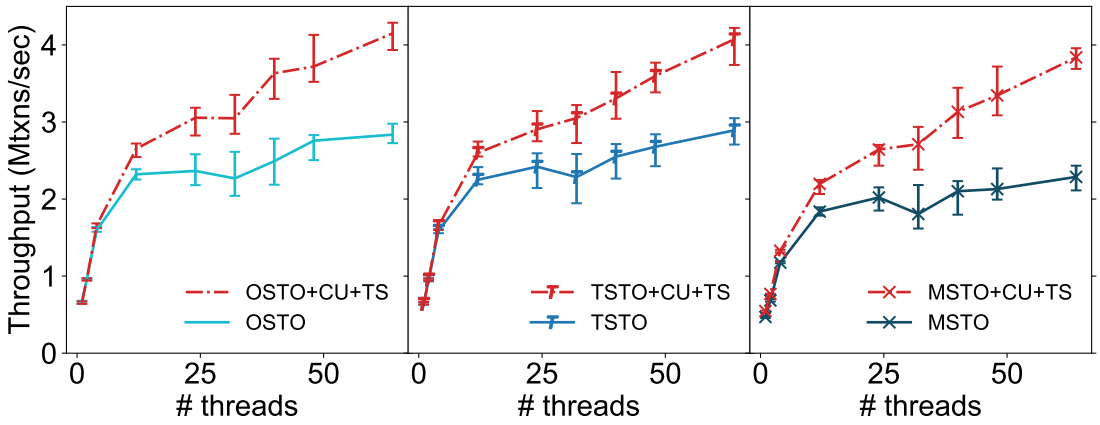
perform baseline MSTO.

Similar trends are observed in the high-contention, update intensive YCSB-A workload (Figure 6.6a). MSTO with CU+TS is again the only scalable variant we measured. It also manages to achieve much higher gains (as high as 9.5×) in this workload, because the workload becomes essentially conflict-free with commit-time updates turning updates to blind writes.

For low-contention TPC-C (Figure 6.5b), where high contention optimizations do not



(a) *Wikipedia (high contention).*



(b) *RUBiS (high contention).*

Figure 6.7: *Wikipedia and RUBiS results with high contention optimizations.*

help reduce conflicts, CU+TS adds performance overhead ranging from 6% (in OSTO) to 18% (in MSTO). This is roughly comparable to the difference between baseline TSTO and OSTO at low contention, while significantly less than the difference between baseline MSTO and OSTO. Low-contention TPC-C results in Figure 5.5b show that OSTO is faster than than TSTO by roughly 10%, and OSTO is 1.93× faster than MSTO.

CU+TS adds slightly more overhead at low-contention TPC-C in MSTO due to its version chain complexity and its garbage collection overhead. The high absolute throughput

in low contention workloads can lead to long version chains containing delta versions. As a specific example, updates to warehouse `ytd` values, which are only written to and never read from, can lead to long version chains that needs to be cleaned up by periodic flattening at garbage collection time. Garbage collection improvements can potentially reduce this overhead and may be an interesting direction for future work. In all cases, the added overhead of CU+TS does not affect scalability. We will discuss in Chapter 7 that alternative CU and TS implementations in MSTO can lead to surprisingly worse performance overhead at low-contention.

We find that CU+TS also improves OSTO and TSTO performance in YCSB-B, despite it being a low-contention benchmark. Upon investigation, we discovered that CU+TS reduces the amount of data retrieved from and written to the database because CU updaters move data within *only* the columns specified. CU+TS incurs a negligible overhead for MSTO in this benchmark, showing that the multiple version chain organization and the additional garbage collection and flattening mechanisms do not add overhead for a read-mostly workload.

CU+TS also benefits all three concurrency control variants under the high-contention Wikipedia and RUBiS workloads, as shown in Figure 6.7. In Wikipedia, CU+TS improves performance by 2.8–3.8× at high core counts, while in RUBiS the gains vary from 1.4–1.7×, depending on the underlying concurrency control used.

In summary, CU+TS benefits all three concurrency control algorithms we studied, and can be applied to benefit many different workloads.

6.5.2 Separate effects

Table 6.1 shows a breakdown of the effects of CU and TS, when applied individually, on our high-contention benchmarks for OCC and MVCC. In some workloads, such as TPC-C,

Benchmark	OSTO	OSTO+CU	OSTO+TS	OSTO+CU+TS
TPC-C	292	307 (1.05×)	577 (1.98×)	1393 (4.78×)
YCSB	473	855 (1.81×)	466 (0.99×)	844 (1.78×)
Wikipedia	171	498 (2.91×)	178 (1.04×)	489 (2.86×)
RUBiS	2836	4148 (1.46×)	2829 (1.00×)	4150 (1.46×)

Benchmark	MSTO	MSTO+CU	MSTO+TS	MSTO+CU+TS
TPC-C	526	474 (0.90×)	442 (0.84×)	2002 (3.81×)
YCSB	219	1324 (6.04×)	591 (2.70×)	2075 (9.46×)
Wikipedia	120	458 (3.81×)	121 (1.00×)	457 (3.80×)
RUBiS	2287	3732 (1.63×)	2488 (1.09×)	3838 (1.68×)

Table 6.1: *Throughput in Ktxns/sec at 64 threads in high-contention benchmarks, with improvements over respective baselines in parentheses.*

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 3.81× at 64 threads when applied in combination. This is because although many frequently-updated columns can be updated using CU, it is only effective if the infrequently-updated columns are assigned to a separate column group that is independently concurrency-controlled (as explained in § 6.1.2).

Of the two optimizations, CU is more frequently useful on its own. For instance, the highest overall performance in the Wikipedia benchmark is obtained by applying CU only to OSTO. In the RUBiS benchmark, CU alone is responsible for most of the gains in CU+TS for both OSTO and MSTO. This is an indication that write-write conflicts are predominant in these workloads. CU reduces the impact of write-write conflicts while TS reduces the impact of read-write false sharing.

Chapter 7

Discussion

As in all complex systems, the devil is in the details. Different design choices of the same mechanisms we describe in this work can have significant correctness and performance impact. We discuss some of the instances here.

7.1 Phantom Protection in TSTO

STOv2 uses the phantom protection strategy described in § 3.2.1 to ensure transactional consistency of key gaps. Masstree maintains OCC-style timestamps in leaf nodes already for its concurrent operations. For OSTO and MSTO, since the commit timestamps of all transactions monotonically increase[1], a simple OCC-style validation of Masstree's leaf node timestamps at commit time is enough to guarantee serializability.

TSTO is different in that it allows more flexible commit schedules than does OCC. One may mistake this flexibility for some kind of "backwards compatibility" with OCC and conclude that the OCC-style Masstree node timestamp validation still works for phantom

[1]Technically, read-only transactions in MSTO commit at timestamps in the recent past. But since they read from immutable snapshots, phantom protection is irrelevant for this class of transactions.

protection purposes, but we demonstrate that this is not true.

Consider two transactions T1 and T2 that execute concurrently and interleave as shown below. Suppose Record A has wts=100 and rts=100 initially. K is a key in an index accessed by both T1 and T2.

```
T1:                                     T2:
  read Record A
                                          (read node timestamp)
                                          observe key gap containing K

                                          write Record A

                                          commit protocol:
                                            commit timestamp <- 101
                                            validate node timestamp - OK

  insert K
  (increment node timestamp)

  commit protocol:
    commit timestamp <- 100
    validate Record A - OK
```

According to the TicToc commit protocol, both T1 and T2 will be able to commit, and T1 commits before T2 in serialization order. However, if we execute the two transaction sequentially, it's clear that T2 should observe that key K exists, instead of seeing a key gap.

The problem is that the node timestamps in Masstree, which are used for phantom protection, by default are not compatible with TicToc. These timestamps by default operate in an OCC manner–a changed timestamp indicates that the content of the node has been modified, prompting concurrent readers to retry. All of this occurs within Masstree routines but not during the transaction commit protocol, so the node timestamps do not participate in TicToc's commit timestamp computation, leading to incorrect transaction schedules. We thus add TicToc timestamps to Masstree leaf nodes and observe these timestamps instead

for phantom protection. The original OCC-like timestamps are preserved in Masstree nodes for Masstree's internal synchronization.

Using TicToc timestamps for phantom protection comes with certain overhead. These timestamps are in addition to Masstree's native node timestamps, affecting the carefully engineered cache line alignment of Masstree's leaf nodes. The timestamps now also participate in the commit protocol, leading to more atomic operations in shared memory.

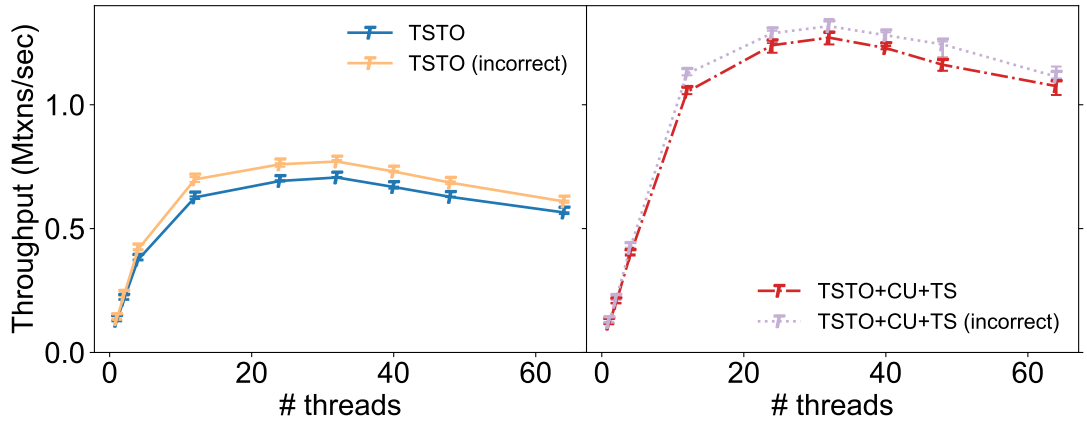
The original TicToc algorithm and its implementation in DBx1000 [63, 64] do not address the issue of phantom protection. Prior work, including Cicada [37], opted to used the *incorrect* phantom protection implementation, which simply uses Masstree's native node timestamps, as an "upper bound" of TicToc's performance. We evaluate both the correct (full) and incorrect TicToc phantom protection implementations using the TPC-C benchmark to quantify how much this upper bound overestimates TicToc's performance.

Figure 7.1 shows the results. We find that full phantom protection implementation adds about 7-10% overhead in both high and low contention TPC-C. The overhead is consistent across core counts and does not appear to impact performance dynamics. Since the incorrect TicToc phantom protection implementation appears to perform reasonably close to the full implementation, we consider it suitable for estimation or comparison purposes, though more studies on more workloads are needed.

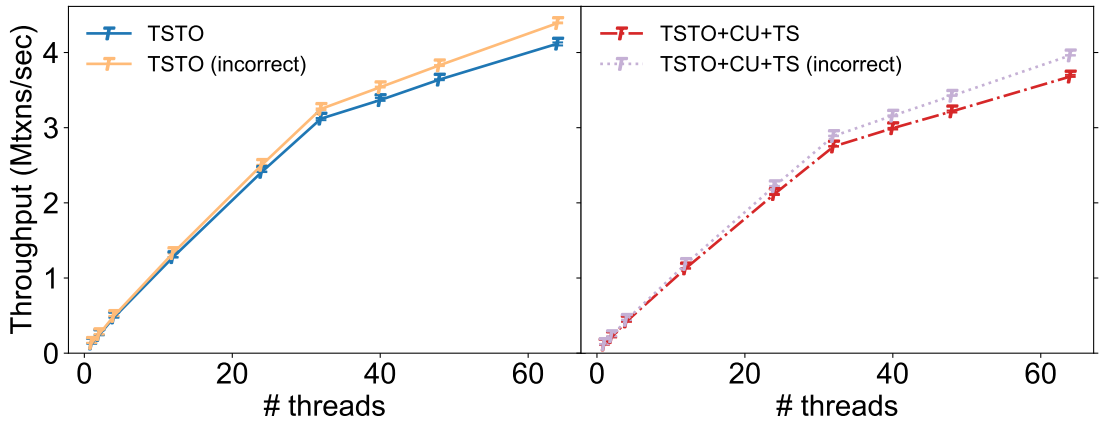
7.2 Conflicts due to Sequential Insertion

A common pattern in database workloads is inserting rows into tables without explicitly specifying the primary key. This almost always occurs when creating new database records. In many databases, primary keys are generated via auto-increment. This leads to *sequential insertions* into the database table when creating records.

Silo first discovered that the strict order ID generation policy in TPC-C induces unnec-



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



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

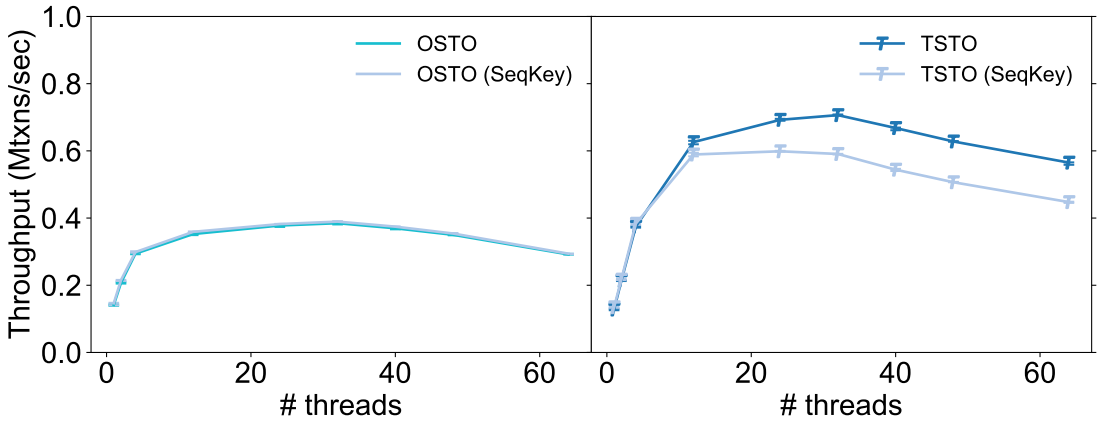
Figure 7.1: *TSTO full vs. incorrect phantom protection comparison using TPC-C.*

essary conflicts between new-order transactions [56]. The TPC-C spec requires that all new orders within the same district have consecutive, monotonically increasing order IDs. This makes insertions to new-order tables non-blind writes, as it must validate at commit time the generated order ID hasn't been taken by another committed transaction. Silo notices that this aspect of the TPC-C spec makes the workload inherently non-scalable. It thus proposes fast order ID generation, which allows gaps in order IDs within the same district, drastically improving performance and scalability.

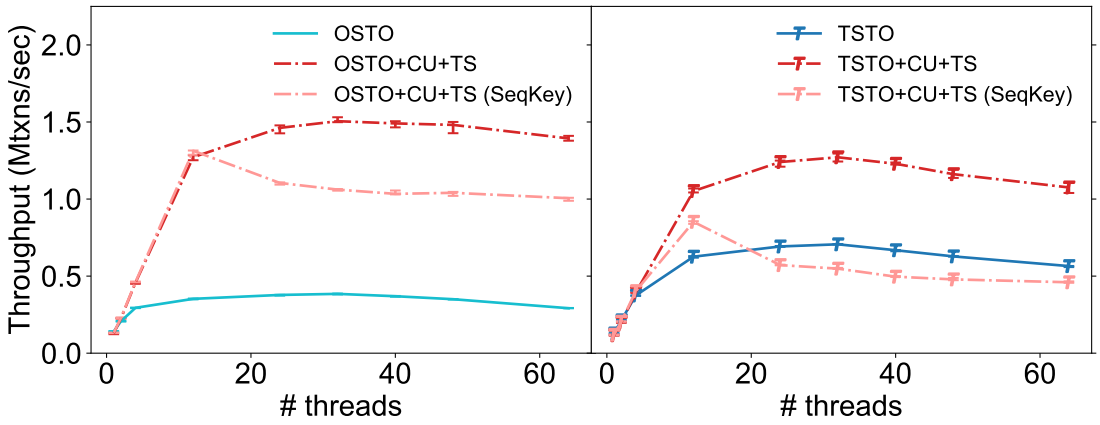
We further observe that even by making table row insertions blind writes from a concurrency control perspective, the relative closeness and/or strict monotonicity of generated primary keys in the key space still negatively impact performance, and in certain cases can even lead to situations that resemble a contention collapse. For example, if a newly generated primary key must be larger than all existing keys in the key space, then all insertions to an ordered table will contend on the right-most leaf node of the B-tree. Although this need not be a concurrency control conflict, it still leads to cache line bouncing and causes sharp drops in performance.

We demonstrate this effect using the key-generation policy for TPC-C's `HISTORY` table. The TPC-C specification [55, §1.3.1, page 15] says that this table does not require a primary key, as "within the context of the benchmark, there is no need to uniquely identify a row within this table". The `HISTORY` table is a table that is only inserted to and never queried from, likely serving only record-keeping purposes. Our TPC-C implementation uses Masstree to implement this table. We compare two implementations, one with a simple per-table atomic counter to generate the Masstree keys used to insert new records into the table, and the other using a multi-part key that includes an atomic counter component for uniqueness, but also includes the customer and district IDs to *spread out* the key distribution and to the avoid concurrency hot spot in the right-most leaf node. Note that both implementations conform to the TPC-C specification.

Figure 7.2 shows the results comparing the two `HISTORY` table implementations under a high-contention TPC-C workload with OSTO and TSTO, we find that the difference in `HISTORY` table key choices, which is something the TPC-C specification doesn't even care about, can have significant performance implications. Although Figure 7.2a suggests that baseline performance is not affected much, Figure 7.2b shows that once we eliminate some conflicts at the concurrency control level using high contention optimizations, insertion to the `HISTORY` table becomes a contention hot spot. With sequential insertion, both op-



(a) *Impact on OSTO and TSTO baselines.*

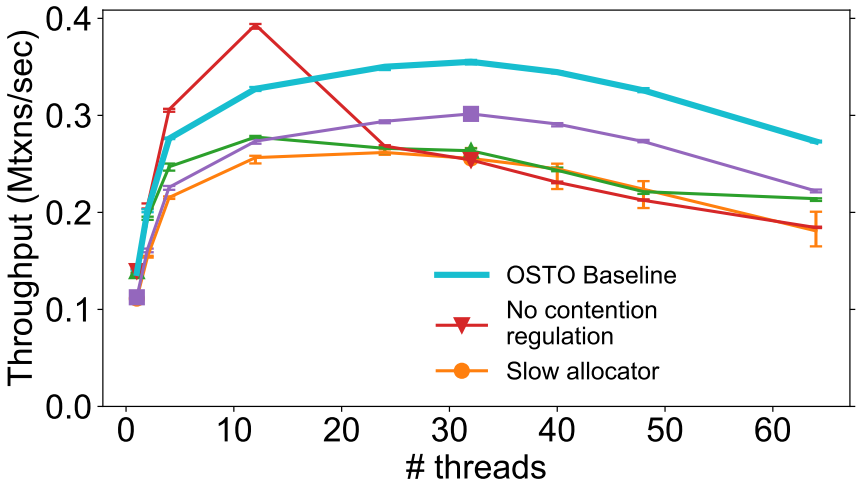


(b) *Impact on optimized OSTO and TSTO.*

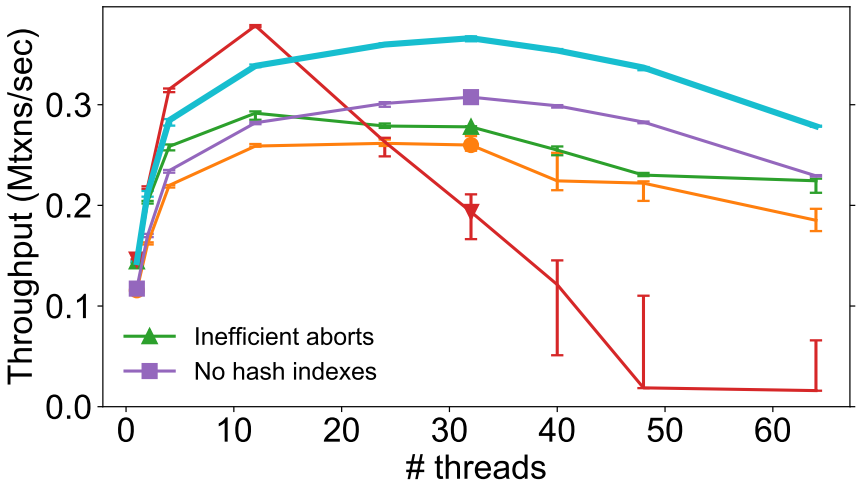
Figure 7.2: *Performance impact of sequential* `HISTORY` *table insertions (SeqKey) in high contention TPC-C.*

timized OSTO and optimized TSTO see a sharp performance drop from 12 threads to 24 threads, similar to the contention collapse we observed due to lack of contention regulation in § 5.1.1, The drop is even more significant in TSTO due to the additional contention on the node TicToc timestamp used for phantom protection. The contention collapse is so bad that it almost renders the high contention optimizations ineffective at high core counts.

In certain high-contention scenarios, this additional conflict can have extra devastating performance effects. When the sequential insertion conflict in TPC-C, for example, oc-



(a) *Without sequential* `HISTORY` *table insertions.*



(b) *With sequential* `HISTORY` *table insertions.*

Figure 7.3: *Impact of sequential* `HISTORY` *table insertion on TPC-C basis factor experiments.*

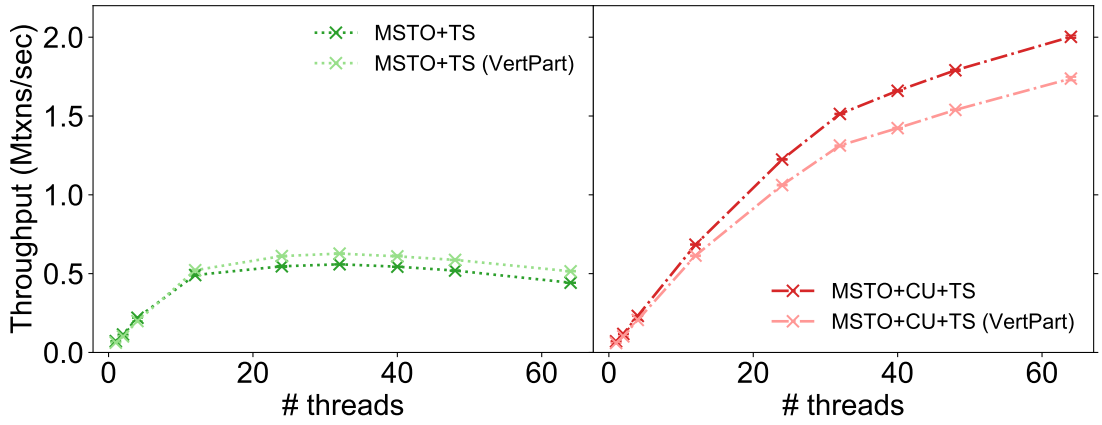
curs without contention regulation, the performance completely collapses to close to zero. This is demonstrated in Figure 7.3, using the same factor analysis experiment in § 5.1. Figure 7.3a shows the results of the experiment without sequential insertion, and hence conflict on the right-most leaf node of the tree, in the `HISTORY` table. Figure 7.3b shows the

results with sequential insertion. The red "No contention regulation" lines in the two graphs shows stark differences. Without contention regulation and without the sequential key insertion conflict, although performance does drop sharply from 12 to 24 threads, it never experiences a full collapse that sees performance crashing to near zero. When the lack of contention regulation is compounded with the additional right-most leaf node conflict due to sequential key insertions, the performance collapsed to near zero. This is evidence that sometimes an additional, single point of contention can have unexpected and disproportional performance consequences.

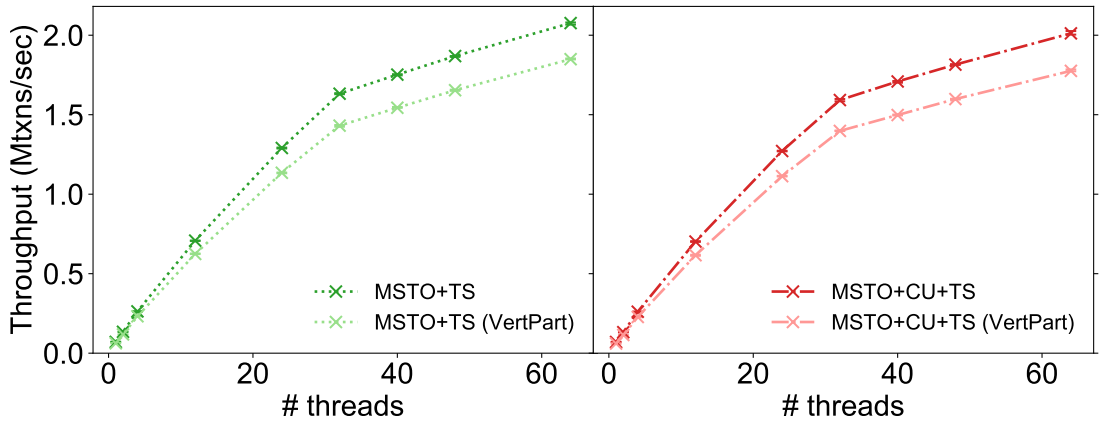
7.3 MSTO Timestamp Splitting Implementation Choices

Timestamp splitting (TS) in MSTO is implemented as multiple version chains per record. We choose this implementation over another popular approach, vertical partitioning, because it does not increase the total number of tables in the database. It also requires no additional table lookups when multiple column groups within the same record are accessed. Vertical partitioning, however, is much easier to implement in an existing system, often requiring mere configuration changes. We measure both vertical partitioning and our multi-chain TS implementations in MSTO and compare their performance.

Experimental results in Figure 7.4 show that vertical partitioning incurs an $\sim 13\%$ overhead in most cases. The exception is in high contention and without CU (TS-only, the left graph in Figure 7.4a), which shows vertical partitioning has a slight advantage ($1.16\times$) over multi-chain TS. We believe this is due to 1) the overhead in computing column-to-column-group mappings during transaction execution, a step our vertical partitioning implementation currently skips, and 2) TS alone does not significantly reduce conflicts in this workload for MSTO. The column-to-column-group mapping computation overhead, however, is mainly an artifact of us implementing transactions directly as C++ programs. In a



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



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

Figure 7.4: *Performance comparison of different MSTO TS implementations: multi-chain (default) and vertical partitioning (VertPart).*

real system, this mapping would still have to computed even with vertical partitioning.

Another TS implementation we considered for MSTO was to only use a single chain per record, but in each old version, instead of storing a full value, we store only the updated portion of the value, and some encoded information about which columns are updated. This becomes similar to the delta version structure in commit-time updates. We ended up not pursuing this approach because accessing an infrequently updated column in this design requires searching *deep* into the version chain, skipping through many new versions created

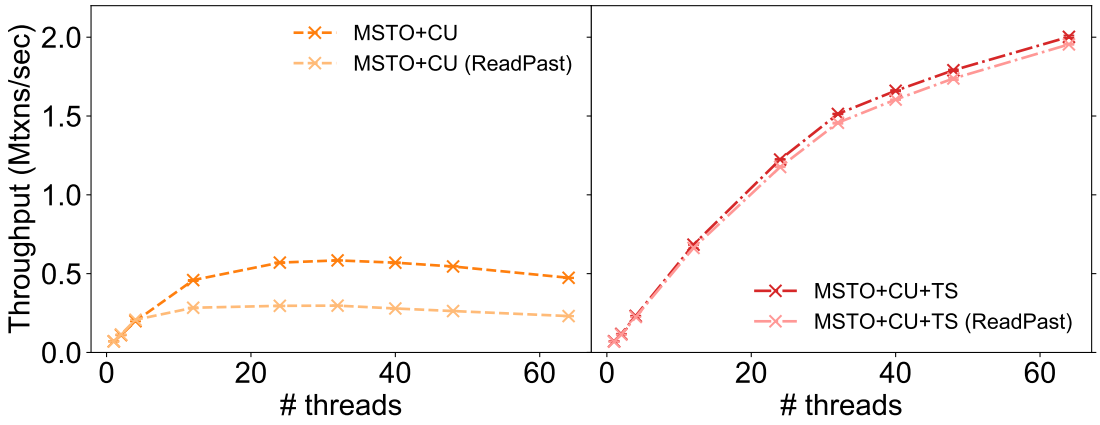
by updates to (other) column groups that are frequently updated.

7.4 MSTO Commit-time Update Implementation Choices

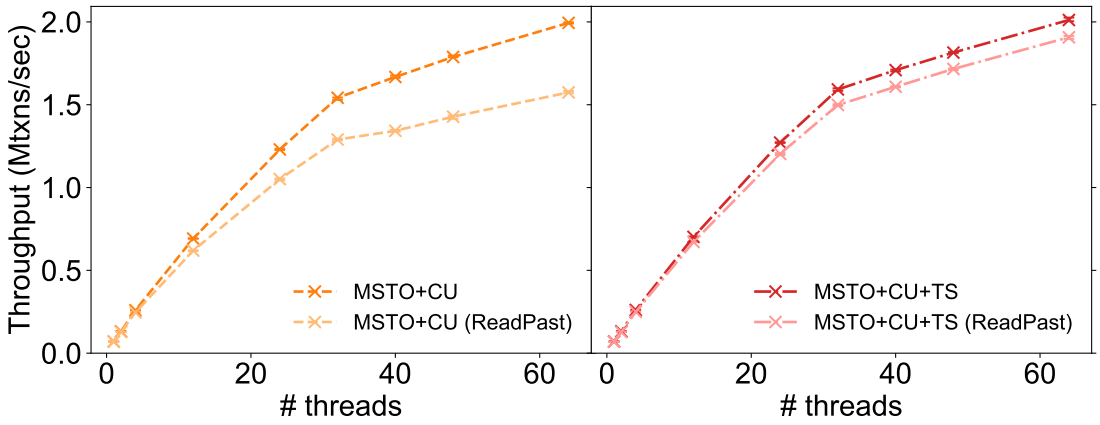
Much of the complexity of commit-time updates (CU) in MSTO stems from their interaction with reads. The flattening of delta versions requires that no new versions are inserted down the version chain, beginning from the version where the flattening process first initiates. Our flatten-freeze CU implementation extends the read timestamp of each version visited during flattening, "freezing" them in place and to prevent new version insertions.

An alternative approach, which does not involve extending the read timestamps of multiple versions at execution time, is to make all transactions observe from the recent snapshot used by read-only transactions at execution time, when CU is enabled. This simplifies flattening, which can simply traverse down the version chain without worrying about new version insertions. The potential cost of this approach may be two-fold: 1) transactions reading from a snapshot in the past will still need to commit at present (if it's a read/write transaction), making it more likely for the execution-time observations to fail commit-time validations; 2) a version in the past is relatively deep down in the version chain, reading from which could incur additional pointer chases and cache misses.

To evaluate which approach works better, we compare these two implementations again using high and low contention TPC-C benchmarks. Figure 7.5 shows the results. The default flatten-freeze implementation performs better than the alternative reading-in-the-past implementation across the board. The overhead of reading-in-the-past is exacerbated when only CU is enabled (left-hand-side graphs), where the default flatten-freeze implementation outperforms it by up to 2×. Without TS, the version chain gets longer, and reading from the snapshot means having to search deeper down the version chain. With both CU and TS enabled, reading-from-the-past does not actually incur significant overhead com-



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



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

Figure 7.5: *Performance comparison of different MSTO CU implementations: flatten-freeze (default) and reading in the past (ReadPast).*

pared to the flatten-freeze approach. This is an indication that CU and TS applied together very effectively separate column groups into read-frequent and update-frequent ones. We find that commit-time read validation failures is the dominant factor explaining the performance difference. Using the read-in-the-past CU implementation, payment transactions sees a concentrated abort rate of 72%, while in flatten-freeze the aborts are more evenly distributed between payment and new-order transactions at about 40% (1 warehouse and 24 threads).

7.5 Future work

This work investigated and answered many important questions related to transaction processing performance in main-memory database systems, but still there are many unanswered questions that could be interesting directions for future studies.

7.5.1 Better locks

STOv2 does not rely on state-of-the-art locking mechanisms to achieve its high performance. Locks in STOv2 are just basic compare-and-swap (CAS) spin locks. Threads attempt to acquire a lock by performing a CAS on the lock word, and then *pause* the processor if the CAS fails before retrying. Currently threads do not backoff between lock attempts – we find that inserting randomized backoffs between CAS retries adds too much overhead (due to the randomness computation) and lead to idleness.

The timestamp splitting optimization reduces read-write dependencies between transactions and also reduces cache line bouncing of the lock word. This reduces STOv2's reliance on high-performance locking mechanisms that perform well under high lock contention.

Commit-time updates, however, can still greatly benefit from better locking mechanisms. Although commit-time updates reduce write-write conflicts at the concurrency control level, lock contention is still high when many threads tries to update the same record. STOv2 currently avoids excessive cache line bouncing by aborting transactions that fail to acquire locks within a certain time limit (see bounded spinning in § 5.1.6), but finding the optimal bound can be difficult – a longer spin bound may lead to more cache line bouncing, while a shorter spin bound causes more transaction aborts and wasted work. It would be interesting to investigate alternative lock implementations that more efficiently regulate lock contention.

7.5.2 Multi-version indexes

In MSTO, the index mapping structure is simply a concurrent B-tree and is not multi-versioned. Instead, the index keeps all keys that could still be accessed (including deleted ones) and lazily cleans up deleted rows via garbage collection. This is in contrast to the approach used in Cicada [37], where each node of the B-tree forming the index structure is multi-versioned. Multi-version indexes are more complicated and use more memory, because every update to the index also requires copy-on-write updates of index nodes. It also requires reference counting to garbage-collect unreachable index nodes. It also comes with the drawback that concurrent insertions to the index with different keys but within the same node would appear to conflict at the concurrency control level.

However, as Cicada results show, multi-version indexes appear to be another solution to the secondary index contention problem discussed in § 5.1.5. Multi-version indexes also have the advantage that read-only transactions now truly never abort (in MSTO read-only transactions can still abort for phantom protection reasons, although we observe this very rarely), and the commit protocol can be skipped completely for these transactions. Multi-version indexes also simplify the interaction between deletes and scans in that scans no longer need to skip through deleted keys. Although our results suggest that single-version concurrent indexes can perform just as well as multi-version indexes in the benchmarks we measure, it would be interesting to more thoroughly evaluate the costs and benefits of using multi-version indexes in these workloads.

7.5.3 Better contention-aware indexes

Contention-aware indexes (§ 5.1.5) are critical in avoiding a surprising class of false conflicts. The current design, which relies on the trie-like properties of Masstree and requires larger key sizes, is very specific to Masstree and requires some manual effort to

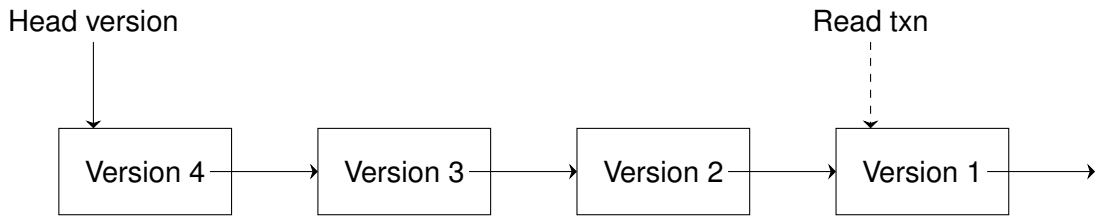


Figure 7.6: *Versions must be kept alive due to an active transaction under the current epoch-based garbage collection mechanism.*

specify component boundaries in a multi-part key.

There are ways to implement contention-aware indexes more generally. One observation is that whenever it crosses a component boundary, the change in the raw value of the key tends to spike. For example, when representing the `NEW-ORDER` table multi-part key using 32 bits for the combined warehouse/district IDs, and another 32 bits for the order ID, keys $\langle wid = 1, did = 1, oid = 999 \rangle$ and $\langle wid = 1, did = 2, oid = 1 \rangle$ are separated by more than four billion in raw key values. Consecutive keys with the same $\langle wid, did \rangle$ prefix are much closer, often separated only by one in raw value. It is possible to detect spikes like this during B-tree node splits to make sure that keys on different sides of a spike do not end up in the same leaf node. Such contention-aware indexes could be more general and do not rely on Masstree's trie layers.

7.5.4 Garbage collection improvements

Garbage collection in MSTO currently requires atomic operations. Every version contains an atomic flag indicating whether the version has already been put onto the garbage collection queue of a different thread, and every thread marks old versions for garbage collection by atomically test-and-seting this flag. The cost of requiring one CAS operation per version marked for garbage collection might become a bottleneck in large-scale systems. This is needed for correct concurrent operation under the current design, but alternative designs that do not require such atomic operations might be possible.

The epoch-based garbage collection mechanism also doesn't free memory at the optimal rate. Long-running transactions can hold up the global GC epoch and prevent garbage collection from making progress. This can lead to unnecessary old versions accumulating in the middle of the version chain for records that are frequently updated while also accessed by a long-running transaction. As illustrated in Figure 7.6, an active transaction, potentially long-running, with a relatively old read timestamp can hold up garbage collection of the entire version chain up to the version being read. In Figure 7.6, Versions 1 through 4 all have to be kept alive under the current garbage collection mechanism. However, if the read transaction (potentially accessing Version 1, shown by the dotted arrow in the figure) is the *only* active transaction in the system that can access versions older than Version 4, there is no need to preserve Versions 2 and 3. The current epoch-based garbage collection mechanism is not optimal when it comes to identifying unneeded old versions to garbage collect.

Solutions exist to more eagerly garbage-collect such old versions. They require potentially more sophisticated garbage collection mechanisms, with the benefit of less memory consumption. Systems like this have already been proposed [8]. More studies on mechanisms like this and more detailed evaluation of their performance impact would be a promising direction of future research.

7.5.5 Automated analysis of workload properties

High contention optimizations presented in this work currently still require manual effort to inspect and analyze static workload properties. These optimizations would be more useful if such analysis can be automated.

Timestamp splitting opportunities may be discovered by analyzing column access patterns in transactions using algorithmic techniques like frequent item set mining [7]. It might

be more difficult to identify commit-time updates opportunities, but common operations such as increments and blind writes should be relatively easy to identify. Developing tools that can process a workload, preferably in the form of an industry standard query language such as SQL, and automatically extract TS and CU policies for the workload is an exciting opportunity for future work, as we have already demonstrated the effectiveness of these optimizations in a number of widely-used benchmarks.

7.5.6 More workloads

Our suite of benchmarks represent a broader range of OLTP workloads than most existing studies, but it is still by no means exhaustive. There are workloads where more complex concurrency control algorithms may show definitive advantages, such as in hybrid OLTP-OLAP workloads. In future work it would be helpful to study more OLTP workloads to further investigate the effectiveness and applicability of the high-contention optimizations.

7.5.7 Relaxed consistency models

Our work only investigates systems that follow *strict serializability*, the strictest consistency model for database transactions. In fact, a lot of the complexities in STOv2, especially those in the implementation of commit-time updates in MSTO, are due to the need to support strictly serializable transactions. It is shown an overwhelming majority of database usage in modern applications preserves functional integrity of the applications without adhering to the strictest consistency models [4]. Relaxed consistency models such as *eventual consistency* [5] have seen their popularity rise in applications that require high throughput but can tolerate certain anomalies. It would be interesting to investigate whether incorporating relaxed consistency would help reduce the complexity and improve performance of STOv2, or whether STOv2 can achieve consistently high performance without compromis-

ing consistency guarantees.

7.5.8 Persistence

Our work addresses only the in-memory transaction processing component of main-memory databases. In many applications, effects of transactions also need to be persisted to prevent data loss due to system shutdowns or failures. In future studies it would be useful to explore options to extend STOv2 to support persistent transactions.

Persistence is considered expensive because writing to durable storage is orders of magnitude slower than in-memory operations. However, there are many known techniques to mitigate this overhead. Systems like Silo-R [65] tried to achieve this by moving persistence off the critical path of transaction processing. New technologies also open up new avenues for fast persistence. For example, emerging nonvolatile RAM technology has the potential to support durable in-memory operations with minimal modifications to the applications. With ultra-fast inter- and intra-datacenter networking, high-availability and durability can also be achieved by replicating transactional operations or the effects of transactions on remote backup machines. Future work may investigate how to integrate these techniques to STOv2 and evaluate their trade-offs.

Chapter 8

Conclusion

Main-memory databases are central to many modern applications. Scalable applications require databases to process potentially contended transactional workloads with high throughput. We investigated three approaches to improving transaction processing throughput in main-memory database systems under high contention: basis factor improvements, concurrency control algorithms, and high-contention optimizations. We found that 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 4.8× over base concurrency control for TPC-C, while the difference between unoptimized concurrency control 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 so far we have not observed any. 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 conflicts, as CU+TS explicitly do. Though in our work these mechanisms require some manual effort to apply, we hope future work will apply them more automatically.

Finally, we are struck by the overall high performance of OCC on both low and high contention workloads, although MVCC and other concurrency control mechanisms may have determinative advantages in certain workloads, such as hybrid OLTP/OLAP workloads. We show that the implementation details and design choices of phantom protection and our high contention optimizations can have nontrivial correctness and performance consequences, especially when in use with more complex concurrency control algorithms. These factors make it more difficult to predict or understand performance. While we acknowledge the novelty and sophistication of concurrency control algorithms that specifically address OCC's known limitations, our results point to the potential for OCC to take on a more expanded role in modern main-memory database systems geared towards OLTP workloads.

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. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 International Conference on Management of Data*, SIGMOD '15, pages 1327–1342. ACM, 2015.

[5] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3):20–32, 2013.

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

[7] C. Borgelt. Frequent item set mining. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 2(6):437–456, 2012.

[8] J. Böttcher, V. Leis, T. Neumann, and A. Kemper. Scalable garbage collection for in-memory mvcc systems. *PVLDB*, 13(2):128–141, 2019.

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

[10] 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.

[11] Cockroach Labs. Column families – CockroachDB Docs. Available at `https://www.cockroachlabs.com/docs/stable/column-families.html`.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] D. Gawlick. Processing "hot spots" in high performance systems. In *Proc. Spring COMPCON 85, 30th IEEE Computer Society International Conference*, pages 249–251, 1985.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems (TODS)*, 11(4):405–430, 1986.

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

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

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

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

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

[55] 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.

[56] 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.

[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 *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1643–1658. ACM, 2016.

[59] 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.

[60] 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.

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

[62] 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.

[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 *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642. ACM, 2016.

[65] 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.