Toward Coordination-free and Reconfigurable Mixed Concurrency Control

Dixin Tang and Aaron J. Elmore, ***University of Chicago***

https://www.usenix.org/conference/atc18/presentation/tang

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Toward Coordination-free and Reconfigurable Mixed Concurrency Control

Dixin Tang
University of Chicago

Aaron J. Elmore
University of Chicago

Abstract

Recent studies show that mixing concurrency control protocols within a single database can significantly outperform a single protocol. However, prior projects to mix concurrency control either are limited to specific pairs of protocols (e.g mixing two-phase locking (2PL) and optimistic concurrency control (OCC)) or introduce extra concurrency control overhead to guarantee their general applicability, which can be a performance bottleneck. In addition, due to unknown and shifting access patterns within a workload, candidate protocols should be chosen dynamically in response to workload changes. This requires changing candidate protocols online without having to stop the whole system, which prior work does not fully address. To resolve these two issues, we present CormCC, a general mixed concurrency control framework with no coordination overhead across candidate protocols while supporting the ability to change a protocol online with minimal overhead. Based on this framework, we build a prototype main-memory multicore database to dynamically mix three popular protocols. Our experiments show CormCC has significantly higher throughput compared with single protocols and state-of-the-art mixed concurrency control approaches.

1 Introduction

With an increase in CPU core counts and main-memory capacity, concurrency control has become a new bottleneck in multicore main-memory databases due to the elimination of disk stalls [8, 34]. New concurrency control protocols and architectures focus on enabling high throughput by fully leveraging available computation capacity, while supporting ACID transactions. Some protocols try to minimize the overhead of concurrency control [10], while other protocols strive to avoid single contention points across many cores [27, 35]. However, these single protocols are typically designed for specific workloads, may only exhibit high performance under their optimized scenarios, and have poor performance in others. Consider H-Store's concurrency control protocol that uses coarse-grained exclusive partition locks and a simple single-threaded executor per partition [10]. This approach is ideal for partitionable workloads that have tuples partitioned such that a transaction is highly likely to access only one partition. But this approach suffers decreasing throughput with increasing cross-partition transactions [34, 27, 20]. Here, an optimistic protocol may be preferred if the workload mainly consists of read operations or a pessimistic protocol may be ideal if the workload exhibits high conflicts. An appealing solution to this tension is to combine different protocols such that each protocol can be used to process a part of workload that they are optimized for, and avoid being brittle to scenarios where single protocols suffer.

Efficiently mixing multiple concurrency control protocols is challenging in several ways. First, it should not be limited to a specific set of protocols (e.g. OCC and 2PL [22, 28]), but be able to extend to new protocols with reasonable assumptions. Second, the overhead of mixing multiple protocols should be minimized such that the overhead is not a performance bottleneck in any scenario. A robust design should ensure that in any case the mixed execution does not perform worse than any single candidate protocol involved. Finally, as many transactional databases back user-facing applications, dynamically switching protocols online in response to workload changes is necessary to maintain performance.

While several recent studies focus on mixed concurrency control, they only address a part of the above challenges. For example, MOCC [28] and HSync [22] are designed to mix OCC and 2PL with minimal mixing overhead but fails to extend to other protocols; Callas [31] and Tebaldi [23] provide a general framework that can cover a large number of protocols, but their overhead of mixing candidate protocols is non-trivial in some scenarios and do not support online protocol switch thus failing

to address the workload changes. Our prior work [25] envisions an adaptive database that mixes concurrency control, but does not includes a general framework to address these challenges.

In this paper, we present a general mixed concurrency control scheme **CormCC** (*Coordination-free and Reconfigurable Mixed Concurrency Control*) to systematically address all the aforementioned challenges. CormCC decomposes a database into partitions according to workload access patterns and assigns a specific protocol to each partition, such that a protocol can be used to process the parts of a workload they are optimized for. We then develop several criteria to regulate the mixed execution of multiple forms of concurrency control to maintain ACID properties. We show that under reasonable assumptions, this method allows correct mixed execution without coordination across different protocols, which minimizes the mixing overhead. In addition, we develop a general protocol switching method (i.e. reconfiguration) to support changing protocols online with multiple protocols running together; the key idea is to compose a *mediated* protocol compatible with both the old and new protocol such that switch process does not have to stop all transaction workers while minimizing the impact on throughput and latency. To validate the efficiency and effectiveness of CormCC, we develop a prototype main-memory database on multi-core systems that supports mixed execution and dynamic switching of three widely-used protocols: a single-threaded partition based concurrency control (*PartCC*) from H-Store [10], an optimistic concurrency control (*OCC*) based on Silo [27], and a two-phase locking based on VLL (*2PL*) [21]. [1]

The main contributions of this paper over our vision paper [25] are the following:

- A detailed analysis of mixed concurrency control design space, a general framework that is not limited to a specific set of protocols to mix multiple forms of concurrency control without introducing extra overhead of coordinating conflicts across protocols.

- A general protocol switching method to reconfigure a protocol for parts of a workload without stopping the system or introducing significant overhead.

- A thorough evaluation of state-of-the-art mixed concurrency control approaches, CormCC's end-to-end performance over varied workloads, and the performance benefits and overhead of CormCC's mixed execution and online reconfiguration.

[1] Note that we use strong 2PL (SS2PL), but refer to it as 2PL

2 Related Work

With the increase of main memory capacity and the number of cores for a single node, recent research focuses on improving traditional concurrency control protocols on modern hardware. We classify these works into optimizing single protocols, mixing multiple protocols, and adaptable concurrency control.

Optimizing Single Protocols Many recent projects consider optimizing a single protocol, which we believe are orthogonal to this project. H-Store [8, 10] developed a partitioned based concurrency control (PartCC) to minimize concurrency control overhead. The basic idea is to divide databases into disjoint partitions, where each partition is protected by a lock and managed by a single thread. A transaction can execute when it has acquired the locks of all partitions it needs to access. Exploiting data partitioning and the single-threaded model is also adopted by several research systems [8, 10, 32, 18, 19, 11]. Other projects propose new concurrency controls optimized for multi-core main-memory databases [6, 14, 27, 17, 35, 20, 13, 29, 30, 36, 12, 37, 16] or new data structures to remove the bottlenecks of traditional concurrency control [34, 21, 15, 9].

Mixing Multiple Protocols Callas [31] and its successive work Tebaldi [23] provide a modular concurrency control mechanism to group stored procedures and provide an optimized concurrency control protocol for each group based on offline workload analysis; for transaction conflicts across groups, Callas and Tebaldi introduce one or multiple protocols in addition to protocols for each group to resolve them. While stored procedure oriented protocol assignment can process conflicts within the same group more efficiently, the additional concurrency control overhead from executing both in-group and cross-group protocols can become the performance bottleneck for a main-memory database on a multi-core server. Section 3 shows a detailed discussion. In addition, the grouping based on offline analysis assumes workload conflicts are known upfront, which may not be true in real applications. Our work differs from Callas and Tebaldi in that our mixed concurrent control execution does not introduce any extra concurrency control overhead, which greatly reduces the mixing overhead. Additionally, we do not assume a static workload or require any knowledge about workload conflicts in advance, but allow protocols are chosen and reconfigured online in response to dynamic workloads. Some other projects exploit the mixed execution of 2PL or OCC [33, 6, 28, 22]. CormCC differs in that our framework is more general and can be extended to more protocols.

Adaptable Concurrency Control Adaptable concurrency control has been studied in several research works.

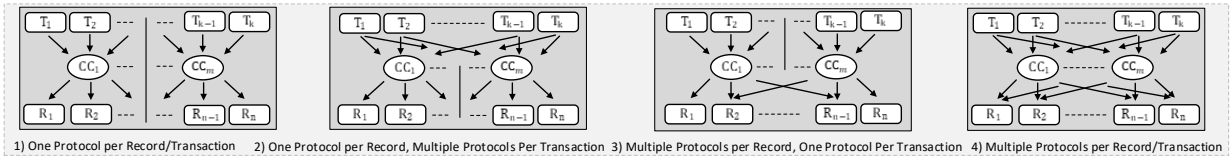

Figure 1: Design choices of mixed concurrency control

At the hardware level, ProteusTM [7] is proposed to adaptively switch across multiple transaction memory algorithms for different workloads, but cannot mix them to process different parts of a workload. Tai et al. [24] shows the benefits of adaptively switching between OCC and 2PL. RAID [4, 3] proposes a general way to change a single protocol online. CormCC differs in that it allows multiple protocols running in the same system and supports to reconfigure a protocol for parts of workload. In this scenario, the presence of multiple protocols during the reconfiguration presents new challenges we are addressing.

3 Design Choices

While combining multiple protocols into a single system can potentially allow more concurrency to improve the overall throughput, it comes with the cost of higher concurrency control overhead. In this section, we discuss two key design choices to explore this trade-off and show how this trade-off motivates the design of CormCC. Specifically, we consider whether a record can be accessed (i.e. read/write) by multiple protocols and whether a single transaction involves multiple protocols. Based on the design choices, the overhead can be defined as *the cost of executing more than one protocol for each transaction plus the cost of synchronizing the concurrent read/write operations across different protocols for each record in the database*. Note that in this paper, we assume all the transaction logic and the corresponding concurrency control logic of a transaction are executed by a single thread (i.e. transaction worker), which is a common model in the design of mixed concurrency control [31, 23, 28, 22, 6]. We now discuss the four possible designs that are shown in Figure 1.

One Protocol per Record and per Transaction This is the simplest case, which is the left most part of Figure 1. We see that each transaction (denoted by `T`) can only choose one protocol (i.e. `CC` in Figure 1) and each record (denoted by `R`) can be accessed via one protocol. While the mixing overhead is minimal (based on our overhead definition), this design has very limited applicability since each transaction can only access the partition of records managed by a specific protocol. To the best of our knowledge, no previous work adopts it.

One Protocol per Record, Multiple Protocols per Transaction This design further allows that one transaction executes multiple protocols (shown in the second design in Figure 1); it provides the flexibility that transactions can access any record and allows a specific protocol to process all access to each record according to their access patterns. On the other hand, the execution of a single transaction may involve a larger set of instructions from multiple protocols, which makes CPU instruction cache less efficient. However, according to our experiment in Section 6.5 this mixing overhead is very low. MOCC [28] adopts this design to mix OCC and 2PL, but does not have a general framework.

Multiple Protocols per Record, One Protocol per Transaction An alternative design (the third one in Figure 1) is that each record can be accessed via multiple protocols and each transaction executes one protocol. This design is useful when the semantics of a subset of transactions (e.g. from stored procedures) can be leveraged by an optimized protocol. It raises a problem, however, that co-existence of multiple protocols on the same set of records should be carefully synchronized. One solution is to let all protocols share the same set of concurrency control metadata and carefully design each protocol such that all concurrent access to the same records can be synchronized without introducing additional coordination. This solution requires specialized design for all protocols and thus is limited in its applicability. Prior works Hekaton [6] and HSync [22] adopt this design, but can only combine 2PL and OCC.

Multiple Protocols per Record and per Transaction This design (the fourth one in Figure 1) provides the most fine-grained and flexible mixed concurrency control; each transaction can mix multiple protocols and each record can be accessed via different protocols. To process the concurrent access from different protocols over the same records, additional protocols are introduced. For example, Callas [31] and its successive work Tebaldi [23] organized protocols into a tree, where the protocols in leaf nodes process conflicts of the assigned transactions and the protocols in interior nodes process conflicts across its children. The overhead here is that for each record access, multiple concurrency control logic should be executed to resolve the conflicts across different protocols over that record. Such overhead, as we show in Section 6.3, can become a performance bottleneck in the multi-core main-memory database.

Summary The above discussion (and our experiments) show that the second design, which lets each pro-

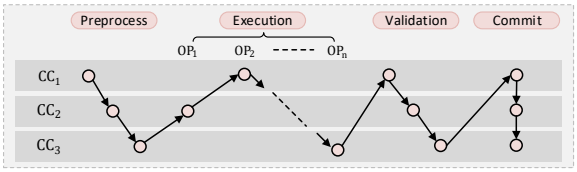

Figure 2: An Example of CormCC execution

tocol exclusively process the access of a subset of records to minimize synchronization cost and allows protocols are mixed within a transaction to provide mixing flexibility, strikes a good trade-off between leveraging the performance benefits of single protocols and minimizing mixing overhead. CormCC draws its spirit and builds a general and coordination-free framework.

4 CormCC Design

We consider a main-memory database with multiple forms of concurrency control on a multi-core machine. Each table includes one primary index and zero or more secondary indices. A transaction can be composed into read, write (i.e. update), delete, and insert operations to access the database via either primary or secondary indices in a key-value way. The database is (logically) partitioned with respect to candidate protocols within the system, that is, each partition is assigned a single protocol. Each protocol maintains an independent set of metadata for all records. For all operations on the records of a partition, the associated protocol executes its own concurrency control logic to process these operations (e.g. preprocess, reading/writing, commit), which we denote as *a protocol managing this partition*. We use a concurrency control lookup table to store the mapping from primary keys to the protocol. The lookup table maintains the mapping for the whole key space and is shared by all transaction workers. Prior work [26] shows that such a lookup table can be implemented in a memory-efficient and fast way, and thus will not be the performance bottleneck of CormCC. CormCC regards secondary indices as logically additional tables storing entries to primary keys (not pointers to records); it adopts a dedicated protocol (e.g. OCC) to process all concurrent operations over secondary indices. Transactions are routed to a global pool of transaction workers, each of which is a thread or a process occupying one physical core. This worker executes the transaction to the end (i.e. commit or abort) without interruption. We additionally use a coordinator to manage the online protocol reconfiguration for all transaction workers. It collects statistical information periodically from all workers, builds a new lookup table accordingly, and finally lets all workers use it. We first outline the CormCC protocol and then show the correctness of CormCC. After that, we discuss online protocol reconfiguration within CormCC.

4.1 CormCC Protocol

CormCC divides a transaction's life cycle into four phases: Preprocess, Execute transaction logic (Execution), Validation, and Commit. We adopt this four-phase model because most concurrency control protocols can fit into it. We use a transaction execution example in Figure 2 to explain the four phases.

Preprocess The preprocess phase executes concurrency control logic that should be executed before the transaction logic. Figure 2 shows that CormCC iterates over all candidate protocols (denoted as `CC`) and executes their preprocess phases respectively. Typical preprocess phase includes initializing protocol-specific metadata. For example, 2PL Wait-die needs to acquire a transaction timestamp to determine the relative order to concurrent transactions, or partition-based single-thread protocol (e.g. PartCC) acquires locks in a predefined order for partitions the transaction needs to access.

Execution Transaction logic is executed in this phase. As shown in Figure 2, for each operation (denoted as `OP`) issued from the transaction, CormCC first finds the protocol managing the record using the concurrency control lookup table. Then, CormCC utilizes the protocol's concurrency control logic to process this operation. For example, if the transaction reads an attribute of a record managed by 2PL, it acquires its read lock and returns the attribute's value. Note that insert operations can find the corresponding protocol in the lookup table even though the record to be inserted is not in the database because the table stores the mapping of the whole key space.

Validation Each validation phase of all protocols are executed sequentially in this phase. If the transaction passes all validation, it enters the Commit phase; otherwise, CormCC aborts it. For example, if OCC is involved, CormCC executes its validation to verify whether records read by OCC during Execution have been modified by other transactions. If yes, the transaction is aborted; otherwise, it passes this validation.

Commit Finally, CormCC begins an atomic Commit phase by executing commit phases of all protocols. For example, one typical commit phase, like OCC, will apply all writes to the database and make them visible to other transactions via releasing all locks. Note that an abort can happen in the Execution or Validation phase, in which case CormCC calls the abort functions of all protocols respectively.

4.2 Correctness

We first show the criteria for CormCC to generate serializable schedule for a set of concurrent transactions (i.e. its result is equivalent to some serial history of these transactions) to guarantee the consistency of databases.

We then discuss how CormCC avoids deadlock, and show that CormCC is recoverable (i.e. any committed transaction has not read data written by an aborted transaction).

Serializability To guarantee that CormCC is serializable, we require all candidate protocols are commit ordering conflict serializable (COCSR). It means that if two transactions t_i and t_j have conflicts on a record r (i.e. t_i and t_j reads/writes r and at least one of them is a write) and t_j depends on t_i (i.e. t_i accesses r before t_j), then t_i must be committed earlier than t_j.

Given that all candidate protocols are COCSR, we now show that CormCC is also COCSR. Suppose that two transactions t_i and t_j have conflicts on a record set R, we consider two cases. First, if for any conflicted record $r \in R$ the conflicted operation of t_i accesses r before t_j, then we have t_j depends on t_i. Since any candidate protocol is COCSR, for each conflicted record r they ensure that t_i is committed before t_j. Therefore in this case, CormCC can maintain COCSR property. In the second case, there exist two records r_1 and $r_2 \in R$, and t_i accesses r_1 before t_j and t_j accesses r_2 before t_i. We have t_j depends on t_i and t_i depends on t_j. In this case, r_1 and r_2 must be managed by two separate protocols p_1 and p_2 since each single protocol is COCSR and it is not possible to form a conflict-cycle in one protocol. Consider p_1, which manages r_1. Because it is COCSR, it enforces that t_i is committed earlier than t_j since t_j depends on t_i based on their conflicts on r_1. On the other hand, p_2 enforces that t_i is committed earlier than t_j. Therefore, there is no valid commit time for both t_i and t_j to suffice the above two constraints. Thus, in this case committing both transactions is impossible and the COCSR property of CormCC is also maintained. Finally, since COCSR is a sufficient condition for conflict serializable [2], CormCC is conflict serializable.

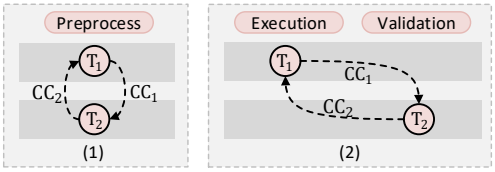

Figure 3: Examples of deadlock across protocols

Deadlock Avoidance While each individual protocol can provide mechanisms to avoid or detect deadlocks, mixing them using CormCC without extra regulation may not make the system deadlock-free. One potential solution can be using a global deadlock detection mechanism. However, this contradicts our spirit of not coordinating candidate protocols.

Alternatively, we examine the causes of deadlock and find that under reasonable assumptions, CormCC can mix single protocols without coordination. Specifically, the deadlock can happen within a single phase or across phases when two transactions wait for each other due to their conflicts on records that are managed by separate protocols. Figure 3 shows two such cases. The first case shows that the single-phase deadlock happens because both protocols CC_1 and CC_2 can make transactions T_1 and T_2 wait within one phase. For the second case, we see that T_1 waits for T_2 in the Execution phase due to CC_1 and additionally introduce conflicts to make T_2 wait for itself based on CC_2 in the Validation phase. Such deadlock is possible because a protocol (e.g. CC_2) can introduce conflicts across phases, that is, the conflicts introduced by T_1 in Execution are detected by T_2 in Validation.

CormCC avoids the two cases by requiring each protocol make transactions wait due to conflicts in no more than one phase, and in each phase only one protocol make transactions wait because of conflicts. If CormCC can meet the two criteria and each protocol can avoid or detect deadlocks, then CormCC is deadlock-free.

Recoverable To guarantee CormCC recoverable, we require that each candidate protocol is strict, which means that a record modified by a transaction is not visible to concurrent transactions until the transaction commits. This ensures transactions never read dirty writes (i.e. uncommited writes) and thus are recoverable [2]. Since each record written by a transaction is managed by a single strict protocol, the execution of CormCC will never read dirty writes and is recoverable.

Supported Protocols A candidate protocol incorporated in CormCC should be COCSR and strict. We find a wide range of protocols can meet these criteria including traditional 2PL and OCC [2], VLL [21], Orthrus [20], PartCC from H-Store [10], and Silo [27].

To enforce that CormCC meets the criteria of deadlock-free, CormCC requires each protocol specifies the phases where it makes transactions wait. Based on these specification, it is easy to detect whether the above criteria hold for a given set of protocols.

4.3 Online Reconfiguration

Online reconfiguration is to switch a protocol for a subset of records without stopping all transaction workers. CormCC uses a coordinator to manage this process. At first, the coordinator collects statistical information from all workers periodically and generates a new concurrency control lookup table. When a worker completes a transaction, it adopts the new lookup table while workers that have not finished their current transactions still use the old one. After all workers use the new table, the old one is deleted. We now introduce the challenge of supporting such online protocol switch.

Challenge The potential problem is that some transaction workers may use the new lookup table while others

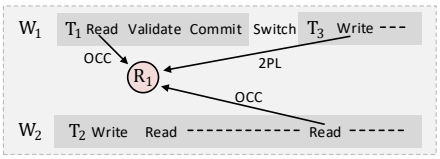

Figure 4: Problems during protocol reconfiguration

are using the old one. Figure 4 shows an example of this problem. Consider two workers W_1 and W_2, which execute transactions T_1 and T_2 respectively. Assume that both T_1 and T_2 access a record (i.e. R_1) that is managed by OCC. During their execution, the coordinator informs that the protocol managing R_1 should be switched to 2PL. Therefore, after T_1 finishes W_1 checks this message, performs the switch (i.e. using 2PL to access R_1), and starts a new transaction (T_3). Since OCC and 2PL maintain a different set of metadata, T_3 and T_2 are not aware of the conflict of T_2 reading R_1 and T_3 writing R_1, which may make database result in an inconsistent state.

Mediated Switching To address this issue, we propose mediated switching; it adopts a *mediated protocol* that is compatible with both old and new protocols. During reconfiguration, the coordinator lets all workers asynchronously change the old protocol to the mediated one; After all workers use the mediated protocol, the coordinator then informs them to adopt the new protocol.

We compose a mediated protocol that can execute concurrency control logic of both old and new protocols. Specifically, a mediated protocol first executes the Preprocess logic of both old and new protocols; then, it enters the Execution phase, where for each record access the mediated protocol executes the Execution logic of both protocols. The mediated protocol's Validation, Commit, and Abort also executes the corresponding logic of both protocols. While it is easy to compose a protocol that executes the logic of both protocols, one problem is how to unify different ways of applying modifications (i.e. insert/delete/write) of different protocols. We find there are two ways to apply modifications: in-place modification during execution and lazy modification during commit phase. In the mediated protocol, we always opt for lazy modification, which means storing the modification in a local buffer during execution and applying them when the transaction is committed. Since all protocols are strict, deferring the actual modification to the commit phase does not violate correctness of protocols. For example, 2PL performs in-place write, while OCC writes the new value into a local buffer and applies it in the commit phase. In our mediated protocol, we choose the OCC approach of applying writes.

We use the example in Figure 5 to illustrate this process, where we need to switch the protocol managing R_1 from OCC to 2PL. The mediated protocol here will execute the logic of both OCC and 2PL (denoted as OPCC). OPCC adopts the following logic:

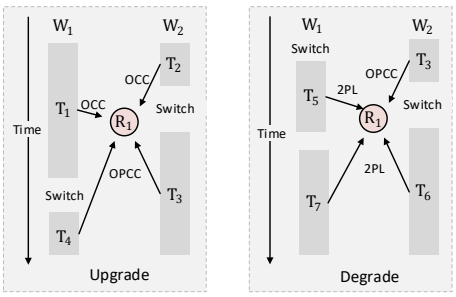

Figure 5: An Example of mediated switching

- If OPCC reads a record, it applies a read lock (2PL) and reads its value and timestamp into the read set (OCC).
- If OPCC writes a record, it applies a write lock (2PL) and stores the record along with new data into the write set (OCC).
- In the validation phase, it locks all records in write set (e.g. for critical section of Silo OCC)[2] and then validate the read set using OCC logic.
- In the commit phase, it applies all writes and release locks acquired by OCC and 2PL respectively.

The switch via mediated protocol is composed of two phases: *upgrade* and *degrade*. The protocol switch is initiated when the coordinator finds that the protocol for a record set *RS* should be changed. It starts the *upgrade phase* by issuing a message to all workers to let them switch the protocol for *RS* to the mediated protocol. Each transaction worker checks for this message between transactions and acknowledges the message to the coordinator. During this asynchronous process, workers that have received the message access *RS* using the mediated protocol, while other workers may access *RS* using the old protocol, which happens when they are running transactions that started before the switch. The left part of Figure 5 shows an example of upgrade phase. We see that worker W_2 finished T_2 first; thus, it begins to access R_1 using OPCC (i.e. in transaction T_3). At the same time, T_1 still accesses R_1 using OCC. According to the above OPCC description, we see that the conflicts on R_1 from W_1 and W_2 can be serialized because OPCC runs the full logic of OCC. After all workers acknowledge the switch for *RS*, the degrade phase begins with the coordinator messaging workers about the degrade to the new protocol. Therefore, workers are using either the mediated protocol or the new protocol, and serializability is guaranteed by the new protocol logic (e.g. 2PL in our example) used in both execution modes. The right part of Figure 5 shows an example of degrade phase, where the conflicts over R_1 can be serialized because OPCC also executes the full logic of 2PL.

[2]Note that OCC and 2PL use different sets of metadata; no deadlock can exist between them

5 Prototype Design

We build a prototype main-memory multi-core database that can dynamically mix PartCC from H-Store [10], OCC from Silo [27], and 2PL from VLL [21] using CormCC. PartCC partitions the database and associates each partition an exclusive lock. Every transaction first acquires all locks for the partitions it needs to read/write in a predefined order before the transaction logic is executed. Then, the transaction is executed by a single thread to the end without additional coordination. In Silo OCC, each record is assigned a timestamp. During transaction execution, Silo OCC tracks read/write operations, and stores the records read by the transactions along with their associated timestamps into a local read set and all writes into a local write set. In the validation phase, Silo OCC locks the write set and validates whether records in read set are changed using their timestamps. If the validation succeeds, it commits; otherwise, it aborts. VLL is an optimized 2PL by co-locating each lock with each record to remove the contention of the centralized lock manager.

Our prototype partitions primary indices and corresponding records using an existing partitioning algorithm [5]. Each partition is assigned with a transaction worker (i.e. thread or process) and each worker is only assigned transactions that will access some data in its partition (the *base partition*), but may also access data in other partitions (the *remote partition*). CormCC selects a protocol for each partition according to its access pattern. A partition routing table maintains the mapping from primary keys to partition numbers and also corresponding protocols. We use stored procedures as the primary interface, which is a set of predefined and parameterized SQL statements. Stored procedures can provide a quick mapping from database operations to the corresponding partitions by annotating the parameters that can be used to identify a base partition to execute the transaction and other involved partitions [10].

The mixed execution of the three protocols start with Preprocess phase, where PartCC acquires all partition locks in a predefined order. Transactions may wait in this phase because of partition lock requests. Next, transactions enter Execution, where CormCC uses PartCC, OCC, and 2PL to process record access operations according to which partition the record belongs to. Note that only 2PL will make transactions wait in this phase, so transactions in the Execution phase will not wait for those blocked in the Preprocess, which indicates deadlocks across PartCC and 2PL is impossible. After the Execution phase, Validation begins and OCC acquires write locks and validates the read set [27]. Only OCC requires transactions to wait; thus, they will not be blocked by previous phases. Finally, there is no wait in commit phase; all protocols apply writes and release all locks. We show that such mixed execution is correct. First, for PartCC and 2PL if a transaction t_i conflicts with another transaction t_j, t_i cannot proceed until t_j commits or vice versa, so PartCC and 2PL are COCSR. Shang et al. [22] have proven that Silo OCC is also COCSR. Therefore, their mixed execution using CormCC maintains COCSR. In addition, each of PartCC, 2PL, and OCC can independently either avoid or detect deadlocks and make transactions conflict-wait in only one mutually exclusive phase among Preprocess, Execution, or Validation. Thus, their mixed execution can also prevent or detect deadlocks. Finally, all protocols are strict, so is their mixed execution.

To enable dynamic protocol reconfiguration, we build two binary classifiers to predict the ideal protocol for each partition. The detailed discussion of classifiers are presented in a technical report [1].

6 Experiments

We now evaluate the effectiveness of mixed execution and online reconfiguration of CormCC. Our experiments answer four questions: 1) How does CormCC perform compared to state-of-the-art mixed concurrency control approaches (Section 6.3)? 2) How does CormCC adaptively mix protocols under varied workloads over time (Section 6.4)? 3) What is the performance benefit and overhead of mixed execution (Section 6.5)? 4) What is the performance benefit and overhead of online reconfiguration (Section 6.6)?

All experiments are run on a single server with four NUMA nodes, each of which has a 8-core Intel Xeon E7-4830 processor (2.13 GHz), 64 GB of DRAM and 24 MB of shared L3 cache, yielding 32 physical cores and 256 GB of DRAM in total. Each core has a private 32 KB of L1 cache and 256 KB of L2 cache. We disable hyperthreading such that each worker occupies a physical core. To eliminate network client latency, each worker combines a client transaction generator.

6.1 Prototype Implementation

We develop a prototype based on Doppel [17], an open-source multi-core main-memory transactional database. Clients issue transaction requests using pre-defined stored procedures, where all parameters are provided when a transaction begins, and transactions are executed to the completion without interacting with clients. Stored procedures issue read/write operations using interfaces provided by the prototype. Each transaction is dispatched to a worker that runs this transaction to the end (commit or abort).

Workers access records via key-value hash tables. Each worker thread occupies a physical core and main-

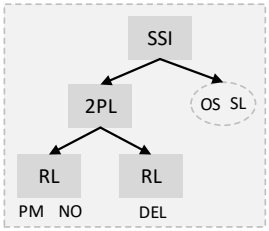

Figure 6: A three-layer configuration of Tebaldi

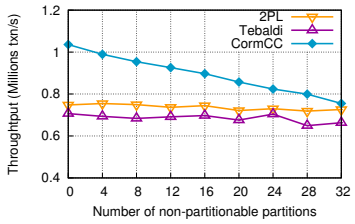

Figure 7: Comparison under different partitionability

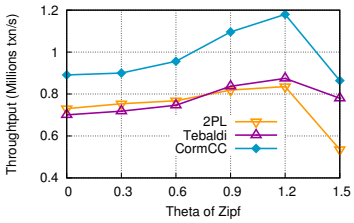

Figure 8: Comparison unber different conflicts

tains its own memory pool to avoid memory allocation contention across many cores [34]. A coordinator thread is used for extracting features for the prediction classifiers from statistics collected by workers and predicting the ideal protocol to be used. Our prototype supports automatically selecting PartCC [10], Silo OCC [27], or No-Wait VLL [21] for each partition. Note that we have compared 2PL variants No-Wait and Wait-Die, and find that 2PL No-Wait performs best in most cases because of lower synchronization overhead of lock management [20]. We do not implement logging in CormCC since prior work shows that logging is not a performance bottleneck [17, 38]. Our comparison additionally includes a general mixed concurrency control framework based on Tebaldi [23] (denoted as Tebaldi) and a hybrid approach of OCC and 2PL [22, 28] (denoted as Hybrid), that adopts locks to protect highly conflicted records but uses validation for the rest. We statically tune the set of highly conflicted records to make Hybrid have the highest throughput. Specifically, for our highly conflicted workload we protect 1000 mostly-conflicted records for each partition and for our lowly conflicted workload no records will be locked and we use OCC for them.

6.2 Benchmarks & Experiment Settings

We use YCSB and TPC-C in our experiments. We generate one table for YCSB that includes 10 million records, each with 25 columns and 20 bytes for each column. Transactions are composed of mixed read and read-modify-write operations. The partitioning of YCSB is based on hashing its primary keys. TPC-C simulates an order processing application. We generate 32 warehouses and partition the store according to warehouse IDs except the Item table, which is shared by all workers. We use the full mix of five procedures.

To generate varied workloads, we tune three parameters. The first is the percentage of cross-partition transactions ranging from 0 to 100. We set the number of partitions a cross-partition transaction will access as 2; The second is the mix of stored procedures for a workload. Note that YCSB only has one stored procedure, and we tune the number of operations per transaction and the ratio of read operations. Finally, we vary data access skewness. We use Zipf to generate record access distribution within a partition. *Theta* of Zipf can be varied from 0 to 1.5. This means for TPC-C within a partition determined by WarehouseID, we skew record access for related tables. We also vary these parameters to train our classifiers for protocol prediction. The detailed configuration of training classifiers is illustrated in [1].

6.3 Comparison with Tebaldi

Tebaldi [23] is a general mixed concurrency control framework that groups stored procedures according to their conflicts and build a hierarchical concurrency control protocols to address in-group and cross-group conflicts. Figure 6 shows a three-layer configuration for TPC-C. We see that NewOrder (`NO`) and Payment (`PM`) are in the same group and their conflicts are managed by runtime pipeline (`RL`). Runtime pipeline is an optimized 2PL that can leverage the semantics of stored procedures to pipeline conflicted transactions. Delivery (`DEL`) alone is in another group also run by runtime pipeline. Conflicts between the two groups are processed by 2PL. OrderStatus (`OS`) and StockLevel (`SL`) are executed in a separate group. Their conflicts with the rest of the groups are processed by Serializable Snapshot Isolation (`SSI`). For every operation issued by a transaction, it needs to execute all concurrency control logic from the root node to the leaf node to delegate the conflicts to specific protocols. For example, any operation issued by NewOrder needs to go through the logic of SSI, 2PL, and RL. While this approach can process conflicts in a more fine-grained way, the overhead of multiple protocols for all operations can limit the performance. Another restriction of Tebaldi is that it relies on the semantics of stored procedures to assign protocols. For workloads including data dependent behaviour (e.g. hot keys or affinity between keys) and not having stored procedures with rich semantics (i.e. YCSB in our test), Tebaldi can only use one protocol to process the whole workload. In our test, we use this 3-layer configuration for Tebaldi, which has the best performance for TPC-C [23]. Note that we use an optimized Runtime Pipeline reported in [31], which can eliminate the conflicts between Payment and NewOrder.

We first compare CormCC with Tebaldi, implemented in our prototype, using TPC-C over mixing well-partitionable and non-partitionable workloads. We par-

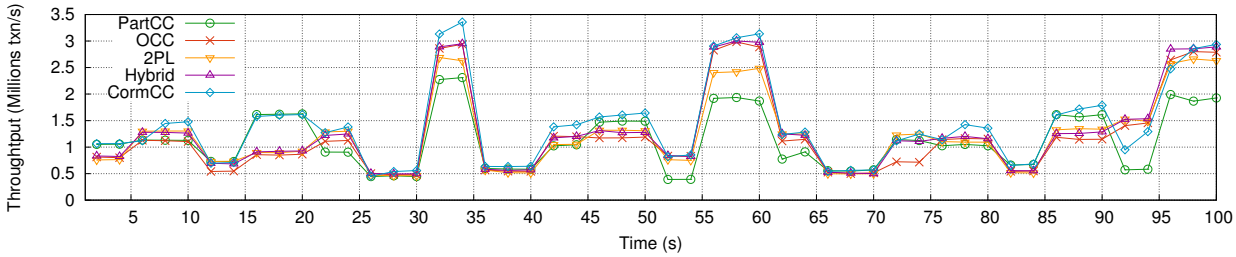

Figure 9: Holistic test for CormCC under YCSB varied workloads over time

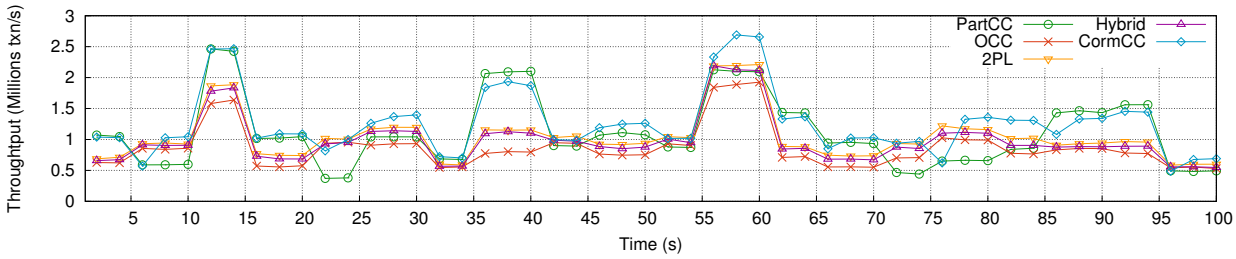

Figure 10: Holistic test for CormCC under TPC-C varied workloads over time

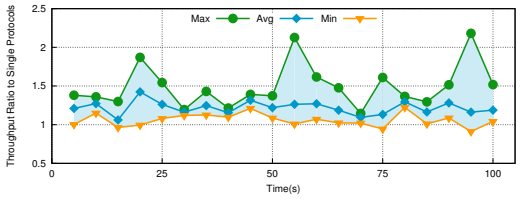

Figure 11: CormCC throughput ratio to single protocols for YCSB

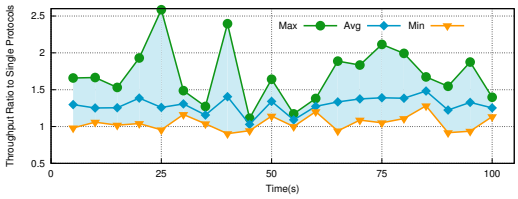

Figure 12: CormCC throughput ratio to single protocols for TPC-C

tition the database into 32 warehouses and start our test with a well-partitionable workload (i.e. each partition receives 100% single-partition transactions), and then increase the number of non-partitionable warehouses (i.e. each receives 100% cross-partition transactions) by an interval of 4. Throughout this test, we use default transaction mix of TPC-C. Since both Tebaldi and CormCC use 2PL as their candidate protocol, our test additionally includes the results of 2PL.

Figure 7 shows the performance results of three protocols. CormCC first adopts PartCC and then proceeds to mix PartCC and 2PL for workloads with mixed partitionability. When the workload becomes non-partitionable, 2PL is used by CormCC. We see that CormCC always performs better than Tabaldi and 2PL because it can leverage the partitionable workloads. Tebaldi always performs slightly worse than 2PL because of its concurrency control overhead from multiple protocols. While such overhead is not substantial in a distributed environment as shown in the original paper [23], it can become a bottleneck in a main-memory multi-core database due to the elimination of network I/O operations.

To highlight Tebaldi's performance benefits of efficiently processing conflicts, we increase the access skewness within each warehouse by varying the *theta* of Zipf distribution from 0 to 1.5 with an interval 0.3. Here, we choose 16 warehouses as partitionable and the rest as non-partitionable. Figure 8 shows that the throughput of all protocols increases at first, because more access skewness introduces better access locality and improves CPU cache efficiency. Then, high conflicts dominate the performance and the throughput decreases for all protocols. We see that with higher conflicts Tebaldi gradually outperforms 2PL, and suffers less throughput loss in the workload with very high conflicts.

These tests show that while Tebaldi can efficiently process conflicts, it comes with a non-trivial concurrency control overhead. In addition, Tebaldi needs to know the conflicts of a workload a priori such that it can utilize the static analysis [23] to make an efficient configuration offline. In contrast, CormCC mixes protocols with minimal overhead, requires no knowledge of conflicts beforehand, and can dynamically choose protocols online.

6.4 Tests on Varied Workloads

We evaluate the holistic benefits of CormCC by running YCSB with randomizing benchmark parameters every 5 seconds. We compare the same randomized run (e.g. same parameters at each interval) with fixed protocols and Hybrid. CormCC collects features every second and concurrently selects the ideal protocol for each partition.

We randomly vary five parameters: i) read rate chosen in 50%, 80%, and 100%; ii) number of operations per transaction, selected between 15 and 25; iii) theta of Zipf for data access distribution; chosen from 0, 0.5, 1, and

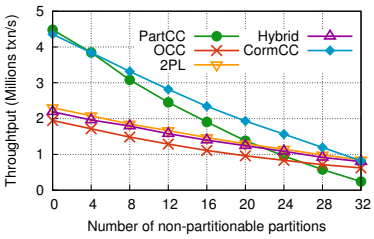

Figure 13: Measuring partitionability

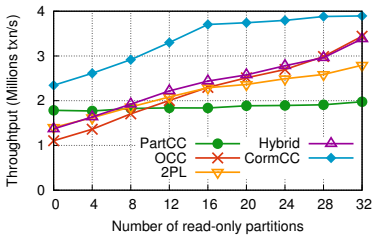

Figure 14: Measuring read/write ratio

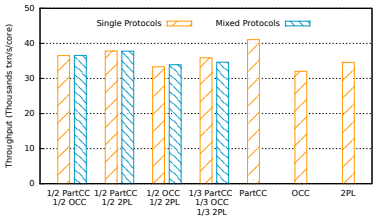

Figure 15: Testing the overhead of mixed execution

1.5; iv) the number of partitions that have cross-partition transactions (with the remaining partitions as well partitionable): we randomly choose the number from 0 to 32 with the interval 4; v) the percentage of cross-partition transactions for partitions in (iv), randomly selected between 50% and 100%.

The test starts with a well-partitionable workload of 80% read rate, 15 operations per transaction, and a uniform access distribution (i.e. $theta = 0$). Figure 9 shows the test results of every 2 seconds for 100 seconds in total. We see that in almost all cases CormCC can either choose the best protocol or find a mixed execution to outperform any candidate protocols and Hybrid approach, while not experiencing long periods of throughput degradation due to switching. CormCC can achieve at most 2.5x, 1.9x, 1.8x, and 1.7x throughput of PartCC, OCC, 2PL, and Hybrid respectively.

We additionally test the performance variations of CormCC under randomized varied workloads of TPC-C. We partition the database into 32 warehouses, and have each worker collect features every second and select the ideal protocol for each warehouse at runtime. We permute four parameters to generate varied workloads. First, we randomly select a transaction mix in 10 candidates, where one is default transaction mix of TPC-C and other nine are randomly generated. Then, we vary the three parameters: record skew for related tables, the number of warehouses that have cross-partition transactions, and the percentage of cross-partition transactions in the same way as YCSB test. We report the results of every 2 seconds in Figure 10. The test starts with well-partitionable default transaction mix of TPC-C and varies workload every 5 seconds. We see that it has similar behaviours of Figure 9, where CormCC can almost always perform the best. In this test, CormCC can achieve at most 2.8x, 2.4x, 1.7x, and 1.8x throughput of PartCC, OCC, 2PL, and Hybrid respectively.

We then report the ratios of the mean throughput of CormCC (after protocol switching) to that of the worst and best single protocols (labeled by `max` and `min` respectively) for each varied workload (i.e. every 5s) of both benchmarks in Figure 11 and Figure 12. We additionally report the ratio of the mean throughput of CormCC to the average throughput of three single fixed protocols (labelled by `avg`) in each varied workload. We see that the highest ratio CormCC can achieve for YCSB and TPC-C is 2.2x and 2.6x respectively. For 55% workloads of YCSB and 85% workloads of TPC-C, the average ratio is at least 1.2x. The lowest ratio in YCSB and TPC-C test is 0.91x and 0.94x respectively, which means that for the two benchmarks CormCC can achieve at least 91% and 94% throughput of the best protocol due to wrong protocol selection for some partitions. These results show that CormCC can achieve significant performance gains when a wrong protocol is selected for a workload, can improve the throughput over single protocols for a wide range of varied workloads, and is robust to dynamic workloads.

6.5 Evaluating Mixed Execution

In this subsection, we first evaluate the performance benefits of CormCC over single protocols and Hybrid approaches, and then test the overhead of CormCC.

We first show how mixed well-partitionable and non-partitionable workloads based on YCSB benchmark influence the relative performance of CormCC to other protocols. We partition the database into 32 partitions, and start our test with a well-partitionable workload and then increase the number of non-partitionable partitions by an interval of 4. In this test, each transaction includes 80% read operations. For these tests, we use transactions consisting 20 operations and skew record access within each partition using Zipf distribution with $theta = 1.5$.

Figure 13 shows that CormCC always performs best because it starts with PartCC and then adaptively mixes PartCC for partitionable workloads and 2PL for highly conflicted non-partitionable counterpart. Compared to CormCC, the performance of PartCC degrades rapidly due to high partition conflicts and other protocols cannot take advantage of partitionable workloads.

We then test workloads with the increasing percentage of read operations. Initially, operations accessing each partition include 80% read operations; we increase the number of partitions receiving 100% read operations by an interval of 4. In this test, our workload includes 16 partitions having 100% cross-partition transactions among them, while the others are only accessed by single-partition transactions.

Figure 14 shows that CormCC has remarkable

throughput improvement over other protocols by combining the benefits of PartCC, OCC, and 2PL. Specifically, CormCC first mixes PartCC and 2PL, and then applies OCC for non-partitionable and read-only partitions. While Hybrid can adaptively mix OCC and 2PL, it is sub-optimal due to failing to leverage the benefits of PartCC. In these tests, the speed-ups of CormCC over PartCC, OCC, 2PL, and Hybrid can be up to 3.4x, 2.2x, 1.9x, and 2.0x respectively.

Next, we test the overhead of CormCC. We first execute transactions using CormCC and track the percentage of each transaction's operations executed on records owned by a specific protocol (e.g. 1/2 of the transaction's records use OCC and 1/2 of the transaction's records use 2PL). With the percentages collected, we execute a mix of transactions where a corresponding percentage of the transactions are executed exclusively on a single protocol (e.g. 1/2 of the transactions are only OCC and 1/2 are only 2PL), and compare the throughputs of the two approaches. Note that to test the overhead without involving the performance advantages of CormCC over single protocols, we use a single core to execute all transactions.

Figure 15 shows a micro-benchmark to evaluate mixed execution overhead. We execute 50,000 transactions, each having 20 operations with 50% read operations, with the rest as read-modify-write operations. Key access distribution is uniform. The dataset is partitioned into 32 partitions; 10 of them are managed by OCC, 10 of them are for 2PL, and 12 are PartCC. The "mixed protocols" shows the average throughput of CormCC with different percentage of operations executed by different protocols; the "single protocol" results show the average throughput of a single protocol (e.g. 100% of transactions use OCC), or using single protocols to exclusively execute a corresponding percentage of transactions. We find that our method has roughly the same throughput as a mix of "single protocols", which shows that the overhead of mixed concurrency control is minimal in CormCC. This is largely due to the fact that we do not add extra meta-data operations to synchronize conflicts across protocols.

6.6 Evaluating Mediated Switching

To evaluate the performance benefits and overhead of mediated switching (denoted as Mediated), we compare it with a method of stopping all protocol execution and applying the new protocol (denoted as StopAll). In our test, we perform a protocol switch from OCC to 2PL using five YCSB workloads with uniform key access distribution. The first workload only includes short-lived transactions with each having 10 read and 10 read-modify-write operations. The other workloads include a mix of short and long-running transactions. We generate long-running transactions by introducing client think/wait time to short transactions. The long transactions last 0.5s, 1s, 2s, and 4s for the four workloads respectively and are dedicated to one worker. We collect throughput every second and report the average throughput during protocol switching. We ensure that switch happens at the start, end, and middle of a long running transaction, which represent that switch waits for little, whole, and half of the transaction respectively, and report three test cases for each mixed workload.

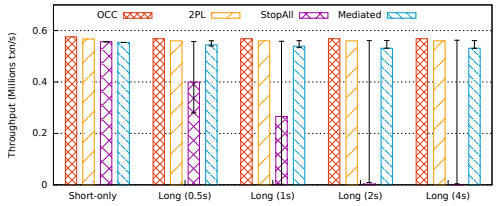

Figure 16: Testing mediated switching

As shown in Figure 16, we see that Mediated and StopAll have a minimal throughput drop compared to 2PL when the workload only includes short transactions. When long-running transactions are introduced, StopAll suffers due to waiting for the completion of long-running transactions, while Mediated can still maintain high throughput during the switch because Mediated does not stop all workers, but let them adopt both 2PL and OCC (i.e. upgrade phase); then, the coordinator notifies all workers to adopt 2PL (i.e. degrade phase) after the long transaction ends. Mediated protocol can achieve at least 93% throughput of OCC or 2PL due to the overhead of executing the logic of two protocols.

In addition, we perform the same test for all other pairwise protocol switching. We find that the overhead is minimal under short-only workload. When long-running transactions are introduced, the maximum throughput drop is about 20% during protocol switching from PartCC to OCC. This is acceptable compared to StopAll, which cannot process new transactions in the switch process. These experiments show that Mediated can maintain reasonable throughput during a protocol switch, even in the presence of long transactions.

7 Conclusion

By exploring the design space of mixed concurrency control, CormCC presents a new approach to generally mixing multiple concurrency control protocols, while not introducing coordination overhead. In addition, CormCC proposes a novel way to reconfigure a protocol for parts of a workload online with multiple protocols running. Our experiments show that CormCC can greatly outperform static protocols, and state-of-the-art mixed concurrency control approaches in various workloads.

References

[1] Toward Coordination-free and Reconfigurable Mixed Concurrency Control (Technical Report). https://newtraell.cs.uchicago.edu/files/tr_authentic/TR-2018-06.pdf.

[2] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing, 1986.

[3] BHARGAVA, B. K., HELAL, A., FRIESEN, K., AND RIEDL, J. Adaptility experiments in the RAID distributed data base system. In *Ninth Symposium on Reliable Distributed Systems, SRDS 1990, Huntsville, Alabama, USA, October 9-11, 1990, Proceedings* (1990), pp. 76–85.

[4] BHARGAVA, B. K., AND RIEDL, J. A model for adaptable systems for transaction processing. *IEEE Trans. Knowl. Data Eng. 1*, 4 (1989), 433–449.

[5] CURINO, C., ZHANG, Y., JONES, E. P. C., AND MADDEN, S. Schism: a workload-driven approach to database replication and partitioning. *PVLDB 3*, 1 (2010), 48–57.

[6] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), pp. 1243–1254.

[7] DIDONA, D., DIEGUES, N., KERMARREC, A., GUERRAOUI, R., NEVES, R., AND ROMANO, P. Proteustm: Abstraction meets performance in transactional memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016* (2016), pp. 757–771.

[8] HARIZOPOULOS, S., ABADI, D. J., MADDEN, S., AND STONEBRAKER, M. OLTP through the looking glass, and what we found there. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008* (2008), pp. 981–992.

[9] JUNG, H., HAN, H., FEKETE, A. D., HEISER, G., AND YEOM, H. Y. A scalable lock manager for multicores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), pp. 73–84.

[10] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S. B., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB 1*, 2 (2008), 1496–1499.

[11] KEMPER, A., AND NEUMANN, T. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany* (2011), pp. 195–206.

[12] KIM, K., WANG, T., JOHNSON, R., AND PANDIS, I. ERMIA: fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 1675–1687.

[13] KIMURA, H. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (2015), pp. 691–706.

[14] LARSON, P., BLANAS, S., DIACONU, C., FREEDMAN, C., PATEL, J. M., AND ZWILLING, M. High-performance concurrency control mechanisms for main-memory databases. *PVLDB 5*, 4 (2011), 298–309.

[15] LEVANDOSKI, J. J., LOMET, D. B., AND SENGUPTA, S. The bw-tree: A b-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013* (2013), pp. 302–313.

[16] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* (2014), pp. 479–494.

[17] NARULA, N., CUTLER, C., KOHLER, E., AND MORRIS, R. Phase reconciliation for contended in-memory transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* (2014), pp. 511–524.

[18] PANDIS, I., JOHNSON, R., HARDAVELLAS, N., AND AILAMAKI, A. Data-oriented transaction execution. *PVLDB 3*, 1 (2010), 928–939.

[19] PANDIS, I., TÖZÜN, P., JOHNSON, R., AND AILAMAKI, A. PLP: page latch-free shared-everything OLTP. *PVLDB 4*, 10 (2011), 610–621.

[20] REN, K., FALEIRO, J. M., AND ABADI, D. J. Design principles for scaling multi-core OLTP under high contention. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 1583–1598.

[21] REN, K., THOMSON, A., AND ABADI, D. J. Lightweight locking for main memory database systems. vol. 6, pp. 145–156.

[22] SHANG, Z., LI, F., YU, J. X., ZHANG, Z., AND CHENG, H. Graph analytics through fine-grained parallelism. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 463–478.

[23] SU, C., CROOKS, N., DING, C., ALVISI, L., AND XIE, C. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017* (2017), pp. 283–297.

[24] TAI, A. T., AND MEYER, J. F. Performability management in distributed database systems: An adaptive concurrency control protocol. In *MASCOTS '96, Proceedings of the Fourth International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems, February 1-3, 1996, San Jose, California, USA* (1996), pp. 212–216.

[25] TANG, D., JIANG, H., AND ELMORE, A. J. Adaptive concurrency control: Despite the looking glass, one concurrency control does not fit all. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings* (2017).

[26] TATAROWICZ, A., CURINO, C., JONES, E. P. C., AND MADDEN, S. Lookup tables: Fine-grained partitioning for distributed databases. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012* (2012), pp. 102–113.

[27] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 18–32.

[28] WANG, T., AND KIMURA, H. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB 10*, 2 (2016), 49–60.

[29] WANG, Z., MU, S., CUI, Y., YI, H., CHEN, H., AND LI, J. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 1643–1658.

[30] WU, Y., CHAN, C. Y., AND TAN, K. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 1689–1704.

[31] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRITSOS, M., AND WANG, Y. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), pp. 279–294.

[32] YAO, C., AGRAWAL, D., CHEN, G., LIN, Q., OOI, B. C., WONG, W., AND ZHANG, M. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Trans. Knowl. Data Eng. 28*, 10 (2016), 2635–2650.

[33] YU, P. S., AND DIAS, D. M. Analysis of hybrid concurrency control schemes for a high data contention environment. *IEEE Trans. Software Eng. 18*, 2 (1992), 118–129.

[34] YU, X., BEZERRA, G., PAVLO, A., DEVADAS, S., AND STONEBRAKER, M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB 8*, 3 (2014), 209–220.

[35] YU, X., PAVLO, A., SANCHEZ, D., AND DEVADAS, S. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 1629–1642.

[36] YUAN, Y., WANG, K., LEE, R., DING, X., XING, J., BLANAS, S., AND ZHANG, X. BCC: reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *PVLDB 9*, 6 (2016), 504–515.

[37] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 276–291.

[38] ZHENG, W., TU, S., KOHLER, E., AND LISKOV, B. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* (2014), pp. 465–477.