THE UNIVERSITY OF CHICAGO

TOWARD COORDINATION-FREE AND RECONFIGURABLE MIXED CONCURRENCY CONTROL

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
DIXIN TANG

CHICAGO, ILLINOIS
2017
To my parents
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 RELATED WORK</td>
<td>4</td>
</tr>
<tr>
<td>3 MIXED CONCURRENCY CONTROL DESIGN CHOICES</td>
<td>8</td>
</tr>
<tr>
<td>4 CORMCC DESIGN</td>
<td>11</td>
</tr>
<tr>
<td>4.1 CormCC Protocol</td>
<td>12</td>
</tr>
<tr>
<td>4.2 Correctness</td>
<td>13</td>
</tr>
<tr>
<td>4.3 Online Reconfiguration</td>
<td>16</td>
</tr>
<tr>
<td>4.4 Range Queries and Phantoms</td>
<td>19</td>
</tr>
<tr>
<td>4.5 Logging and Recovery</td>
<td>21</td>
</tr>
<tr>
<td>4.6 Extending CormCC</td>
<td>22</td>
</tr>
<tr>
<td>5 PROTOTYPE DESIGN</td>
<td>23</td>
</tr>
<tr>
<td>5.1 Mixed Execution</td>
<td>25</td>
</tr>
<tr>
<td>5.1.1 Normal Execution</td>
<td>25</td>
</tr>
<tr>
<td>5.1.2 Online Reconfiguration</td>
<td>26</td>
</tr>
<tr>
<td>5.2 Building Predictive Model</td>
<td>26</td>
</tr>
<tr>
<td>5.2.1 Modeling and Feature Engineering</td>
<td>26</td>
</tr>
<tr>
<td>5.2.2 Feature Extraction</td>
<td>27</td>
</tr>
<tr>
<td>5.2.3 Building a Predictive Model</td>
<td>29</td>
</tr>
<tr>
<td>5.3 Overhead and Optimizations</td>
<td>30</td>
</tr>
<tr>
<td>6 EXPERIMENTS</td>
<td>31</td>
</tr>
<tr>
<td>6.1 Prototype Implementation</td>
<td>31</td>
</tr>
<tr>
<td>6.2 Benchmarks &amp; Experiment Settings</td>
<td>32</td>
</tr>
<tr>
<td>6.3 Comparison with Tebaldi</td>
<td>34</td>
</tr>
<tr>
<td>6.4 Tests on Varied Workloads</td>
<td>36</td>
</tr>
<tr>
<td>6.5 Test Mixed Execution</td>
<td>38</td>
</tr>
<tr>
<td>6.6 Test Mediated Switching</td>
<td>40</td>
</tr>
<tr>
<td>7 CONCLUSION</td>
<td>42</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>43</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Design choices of mixed concurrency control</td>
<td>8</td>
</tr>
<tr>
<td>4.1</td>
<td>An Example of CormCC execution</td>
<td>12</td>
</tr>
<tr>
<td>4.2</td>
<td>Examples of deadlock across protocols</td>
<td>14</td>
</tr>
<tr>
<td>4.3</td>
<td>Problems during protocol reconfiguration</td>
<td>17</td>
</tr>
<tr>
<td>4.4</td>
<td>An Example of mediated switching</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>CormCC Prototype</td>
<td>23</td>
</tr>
<tr>
<td>5.2</td>
<td>Mixed execution of PartCC, OCC, and 2PL</td>
<td>25</td>
</tr>
<tr>
<td>5.3</td>
<td>Workers estimating record contention</td>
<td>28</td>
</tr>
<tr>
<td>6.1</td>
<td>A three-layer configuration of Tebaldi</td>
<td>34</td>
</tr>
<tr>
<td>6.2</td>
<td>Comparison under different partitionability</td>
<td>34</td>
</tr>
<tr>
<td>6.3</td>
<td>Comparison under different conflicts</td>
<td>34</td>
</tr>
<tr>
<td>6.4</td>
<td>Holistic test for CormCC under YCSB varied workloads over time</td>
<td>36</td>
</tr>
<tr>
<td>6.5</td>
<td>Holistic test for CormCC under TPC-C varied workloads over time</td>
<td>36</td>
</tr>
<tr>
<td>6.6</td>
<td>CormCC throughput ratio to single protocols (YCSB)</td>
<td>36</td>
</tr>
<tr>
<td>6.7</td>
<td>CormCC throughput ratio to single protocols (TPC-C)</td>
<td>36</td>
</tr>
<tr>
<td>6.8</td>
<td>Measuring partitionability</td>
<td>38</td>
</tr>
<tr>
<td>6.9</td>
<td>Measuring read/write ratio</td>
<td>38</td>
</tr>
<tr>
<td>6.10</td>
<td>Testing the overhead of mixed execution</td>
<td>38</td>
</tr>
<tr>
<td>6.11</td>
<td>Testing mediated switching</td>
<td>40</td>
</tr>
</tbody>
</table>
LIST OF TABLES

6.1 Parameters for generating training cases for predictive model . . . . . . . . . . 33
ACKNOWLEDGMENTS

Firstly, I would like to express my sincere gratitude to my advisor Prof. Aaron Elmore for the continuous support on this project, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

I would like to thank all people, who have given me advice and feedback on this project, including Michael Franklin, Zehchao Shang, Hao Jiang, and Adam Dziedzic. I also thank Neha Narula and Mike Stonebraker whose insights and discussions with lead to the initial ideas of CormCC.

My sincere thanks also goes to my parents for supporting me spiritually throughout my Ph.D study of the first two years.
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 multi-core database to dynamically mix three popular protocols: partition-based single-thread concurrency control protocol (PartCC), OCC, and 2PL, and design a predictive model based on a set of simple features to predict the ideal protocols for parts of workload. Our experiments show CormCC has significantly higher throughput compared with single protocols and state-of-the-art mixed concurrency control approaches.
CHAPTER 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, 36]. 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 [28, 37]. 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 [36, 28, 22]. 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 way to address this tension is to mix different protocols in a database 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 [24, 30]), 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 protocols 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 [30] and HSync [24] are designed to mix OCC and 2PL with minimal mixing overhead but fails to extend to other protocols; Callas [33] and Tebadi [25] 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.

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 workload they are optimized for. We then develop several theorems to regulate the mixed execution of multiple forms of concurrency control to maintain ACID properties, avoid or detect deadlocks, and be recoverable. 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 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 [28], and a two-phase locking based on VLL (*2PL*) [23]. ¹ We further build a predictive model based on a set of simple but efficient features to select the protocol for parts of a workload under varied workloads.

The main contributions of this paper are the following:

¹. Note that we use strong 2PL (SS2PL), but refer to it as 2PL
• 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, and formal proofs of guaranteeing its correctness.

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

• A prototype system supporting dynamically mixing PartCC, 2PL, and OCC.

• 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.
CHAPTER 2
RELATED WORK

A concurrency control protocol manages interleaved execution of simultaneous transactions to guarantee that effects of concurrent transactions are isolated from each other and maintains the consistency of a database. Two-phase locking (2PL) and optimistic concurrency control (OCC) are two classical protocols, which ensures correct execution of concurrent transactions [2]. 2PL includes two phases: expanding phase and shrinking phase. In the expanding phase of 2PL, to access a database object (e.g. a record or a page), a transaction needs to acquire the lock associated with it. After this transaction completes acquiring all locks, it enters the shrinking phase, where the transaction should release all locks and is not allowed to acquire new locks. This method is based on a pessimistic assumption that potential conflicts are common and locks should be acquired in advance to block conflicted operations. As opposed to 2PL, OCC assumes that conflicts are rare and does not block transactions’ normal execution. OCC tracks read/write operations of a transaction and stores them in its local read/write set and validates whether the read set overlaps with the write sets of any concurrent transactions. If not, OCC commits the write operations to databases. Otherwise, OCC aborts this transaction and restarts it. 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 are considering optimizing a single protocol, which we believe are orthogonal to this project. Harizopoulos et al. [8] studies the detailed instruction breakdown consumed by major components of a conventional DBMS and finds that when OLTP databases are kept in main memory, majority of the instructions are spent on components related to concurrency control and recovery. The authors develop a main memory database, H-Store, that uses a partitioned based concurrency con-
PartCC) to minimize locking and latch overhead [10]. In PartCC, databases are divided 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. LADS [34] extends this idea by dynamically partitioning workloads and assigning transactions to workers to ensure higher CPU utilization and load balance. Exploiting data partitioning and the single-threaded model is also adopted by several research systems [20, 21, 11].


Beyond designing new protocols, recent studies focus on identifying bottlenecks of traditional concurrency control components. One study performs an evaluation of seven concurrency control protocols on main-memory databases using a simulated 1024 cores [36]. The authors find that no one protocol can scale to this large number of cores for various reasons, such as lock thrashing and timestamp allocation limitations. Other studies identify a traditional centralized lock manager as 2PL’s performance bottleneck [23, 9]. VLL [23] removes this lock manager and co-locates locks with records to eliminate the overhead of latch contention, cache misses, and linked list operations. Jung et al. [9] adopts bulk lock allocation and de-allocation to reduce latching overhead in lock managers. Orthrus [22] shows metadata synchronization is another bottleneck that prevents scalability under high contention. It adopts staged and partitioned lock acquisition to minimize this cost. FOEDUS [13] exploits NVRAM and extends the decentralized design of Silo to scale up to hundreds of cores.

**Mixing Multiple Protocols** Several projects explore mixing multiple protocols in one database system. Callas [33] and its successive work Tebaldi [25] 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 base protocols 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. § 3 shows the 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 a priori knowledge about workload conflicts, but allow protocols are chosen and reconfigured online in response to dynamic workloads. Ziv et al. [40] studies the theory of correctly composing (mixed) concurrency controls. Philip et al. [35] studies reducing OCC aborting overhead using 2PL to rerun aborted transactions. Hekaton [6], MOCC [30], and HSync [24] adopt a hybrid of 2PL and OCC concurrency control. 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. 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. [26] 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.

Several projects explore mixing different isolation levels to explore trade-off between consistency and cost [14] or to support OLTP and OLAP applications simultaneously [11, 1], which is a different problem from what we study in this paper. Here, we focus on maintaining a single isolation level and do not target long-running analytic transactions.
CHAPTER 3
MIXED CONCURRENCY CONTROL 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 chapter, 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 is executed by a single thread (i.e. transaction worker), which is a common model in the design of mixed concurrency control [33, 25, 30, 24, 6]. We now discuss the four possible designs that are shown in Figure 3.1.

One Protocol per Record and per Transaction This is the simplest case, which is the left most part of Figure 3.1. We see that each transaction (denoted by T) can only choose one protocol (i.e. CC in Figure 3.1) 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 best of our knowledge, none of previous works adopts this design.
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 3.1); 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 § 6.5 this mixing overhead is very low. MOCC [30] 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 3.1) 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 a specially 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 any additional coordination overhead. This solution requires specialized design for all protocols and thus is limited in its applicability. Prior works Hekaton [6] and HSync [24] adopt this design but can only support 2PL and OCC.

Multiple Protocols per Record and per Transaction This design (the fourth one in Figure 3.1) 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 [33] and its successive work Tebaldi [25] 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 § 6.3, can become a performance bottleneck in the multi-core main-memory database.

Summary The above discussion (and the corresponding experiments) shows that the second design, which lets each protocol 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, therefore, draws its spirit and builds a general and coordination-free framework over it.
CHAPTER 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. We show how to reduce that overhead in our prototype design (§ 5.3). 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. Prior work [27] 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 replaces the old table atomically. The details of deciding the criteria of reconfiguration is protocol-specific; we discuss them in our prototype design (§ 5). We first outline the CormCC protocol and then provide the criteria of guaranteeing that CormCC is serializable, can avoid or detect deadlock, and is recoverable. Next, we discuss online protocol reconfiguration within
CormCC. Finally, we discuss range queries, logging and recovery, and extending 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 4.1 to explain the four phases. Note that if a protocol does not include a phase, CormCC just skips this phase for that protocol.

![Figure 4.1: An Example of CormCC execution](image)

**Preprocess** The preprocess phase executes concurrency control logic that should be executed before the transaction logic. Figure 4.1 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 ordering transactions. 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 4.1, 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 the read lock of this record 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** In this phase, each validation phase of all protocols are executed sequentially. If the transaction passes all validation, it enters the Commit phase; otherwise, CormCC aborts it. For example, if OCC is involved in this transaction, CormCC will executes its validation phase to verify whether records read by OCC during Execution have been modified by other transactions or not. 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

In this section, we first provide criteria for candidate protocols to guarantee serializability of CormCC, discuss how CormCC avoids deadlock, and prove that CormCC is recoverable.

**Serializability** We show that CormCC is conflict serializable if all candidate protocols are commit ordering conflict serializable (COCSR) in Theorem 1. For all proofs in this section, we assume the candidate protocol set is $P$, which has $N$ protocols $p_1 \ldots p_N$.

**Theorem 1.** Given that each protocol $p_i \in P$ manages a partition of database records, and each partition is only managed by a single protocol. If all protocols $p_i \in P$ are commit ordering conflict serializable (COCSR), then CormCC is also COCSR and thus conflict serializable.

**Proof.** By the definition of COCSR, we need to prove that for any two committed transaction $t(i)$ and $t(j)$ by CormCC that have conflicts, their commit time follows $c(i) < c(j)$ if $t(j)$ depends on $t(i)$ or vice versa. Commit time here means the time a transaction enters the
Commit phase.

Suppose that \( t(i) \) and \( t(j) \) have conflicts on a record set \( R \), we consider two cases:

- if for any conflicted record \( r \in R \) the conflicted operation of \( t(i) \) accesses \( r \) before \( t(j) \), we have \( t(j) \) depends on \( t(i) \). Since any protocol \( p_i \in P \) is COCSR, for each conflicted record \( r \), they ensure that \( c(i) < c(j) \). This also applies to the symmetric case where \( t(j) \) accesses any \( r \in R \) before \( t(i) \).

- otherwise, 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 \( t(j) \). \( r_1 \) 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 \( c(i) < c(j) \) since \( t(j) \) depends on \( t(i) \) based on their conflicts on \( r_1 \). On the other hand, \( p_2 \) enforces that \( c(j) < c(i) \) because \( t(i) \) depends on \( t(j) \) based on their conflicts on \( r_2 \). Therefore, there is no valid commit time for both \( t(i) \) and \( t(j) \) to suffice the above two constraints. Thus, we have that either \( t(i) \) or \( t(j) \) is not committed, this violates our assumption that \( t(i) \) and \( t(j) \) are committed transactions.

Therefore, we prove that CormCC is COCSR. Since COCSR is a sufficient condition for conflict serializable, CormCC can generate conflict serializable schedules.

![Figure 4.2: Examples of deadlock across protocols](image)

**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 phases or across phases when two transactions wait for each other due to their conflicts on records that are managed by separate protocols. Figure 4.2 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. Based on this analysis, we develop Theorem 2 that provides the conditions to make CormCC deadlock-free.

**Definition 1.** For a protocol $p$ generating conflict serializable schedules, if transactions $t(i)$ and $t(j)$ have conflicts and $p$ requires $t(i)$ not proceed until $t(j)$ finishes, then we say $t(i)$ conflict-waits $t(j)$ in $p$.

**Theorem 2.** If i) each protocol $p \in P$ can avoid or detect deadlocks, ii) each protocol $p \in P$ introduces a conflict-wait in no more than one phase, and iii) in each phase only one protocol can make transactions conflict-wait, then CormCC can also avoid or detect deadlocks.

**Proof.** For two transactions $t(i)$ and $t(j)$, if $t(i)$ and $t(j)$ have no conflicts or the two transactions have conflicts but neither of them waits for the other, there is no deadlock. Otherwise, the two transactions have conflict-waits. Suppose that the first conflict-wait occurs in the phase $s$ and is introduced by protocol $p$. We consider two cases:

- $t(i)$ conflict-waits for $t(j)$ and $t(j)$ proceeds into a successive phase $t$. Since $t(i)$ is waiting in the phase $s$, it cannot introduce conflicts to make $t(j)$ wait in phase $t$ (by
condition ii)). Therefore, there is no waiting cycle in this case. This also applies to the symmetric case where \( t(j) \) conflict-waits for \( t(i) \) and \( t(i) \) continues its execution.

- otherwise, \( t(i) \) and \( t(j) \) conflict-wait for each other in \( s \) and form a cycle. For a given phase only one candidate protocol can make transactions conflict-wait (by condition iii)); therefore, this formation of this cycle is from a single protocol (i.e. \( p \)). Since any protocol can detect deadlocks (by condition i)), CormCC can also detect deadlocks.

Therefore, given the three conditions CormCC can avoid or detect deadlocks.

\[ \square \]

**Recoverable** To guarantee CormCC recoverable, we require that candidate protocols are strict. The following theorem shows its correctness.

**Theorem 3.** Given that each partition of database records is managed by a single protocol and each protocol \( p \in P \) is strict, then CormCC is strict.

**Proof.** For a record \( r \) that is written by a transaction \( t(i) \), \( r \) will not be read or overwritten by other transactions until \( t(i) \) commits or aborts as \( r \) is managed by a single strict protocol \( p \) (by the definition of strictness). Therefore, CormCC is strict. Since strictness is a sufficient condition for being recoverable, CormCC is also recoverable. \[ \square \]

### 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. It collects statistical information from all workers periodically, generates a new concurrency control lookup table, and lets each worker use the new table without stopping the whole system. In this section, we first introduce the challenge of supporting an online protocol switch and then introduce our technique called *mediated switching* to address the challenge. The proof for the correctness of mediated switching is given in Appendix A.
Challenge Simply letting each worker switch the protocol independently may not ensure serializability, that is, during a protocol switch for some records, some workers use the new table (accessing these records using the new protocol), while the remaining workers still use the old table to process the current transactions. Here, figure 4.3 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 $W_1$ it 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 protocol reconfiguration, the coordinator lets all workers asynchronously change the old protocol to the mediated protocol; 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 the 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 potential
problem is how to unify different ways of applying modifications (i.e. insert/delete/write) of different protocols. Generally speaking, 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 candidate 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 4.4 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:

- 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)\(^1\) 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 4.4 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 4.4 shows an example of degrade phase, where the conflicts over \(R_1\) can be serialized because OPCC also executes the full logic of 2PL.

### 4.4 Range Queries and Phantoms

Phantom problems can occur when a transaction scans a range of records and the membership of this range is changed due to insert or delete of another transaction, which violates

---

1. Note that OCC and 2PL use different sets of metadata; no deadlock can exist between them
serializability. To support range queries and avoid these problems, CormCC requires that a primary index is partitioned into ranges that form a full key space (i.e. no empty ranges between partitions) and that each partition is managed by a single candidate protocol. In addition, we assume that each candidate protocol implements a customized range query protocol to prevent phantom problems (e.g. next-key locking [17] for 2PL or version-based validation for OCC [28]). To ensure the correctness of mixed range query protocols, we further assume that each range query protocol is COCSR. Based on these assumptions, we now describe how CormCC processes range queries.

If the range belongs to a single partition, this query can be processed by the protocol managing the partition; otherwise, we partition the range query into subrange queries such that each subrange belongs to a single partition; for each subrange query, we apply the owning protocol to process them and combine results from the sub-queries. During this process, any insert or delete operations conflicting with a range query must conflict with one of the sub-queries because we partition the key space into continuous ranges and the protocol processing the sub-query must also process the insert/delete operations. Therefore, the conflicts between range query and delete/insert operations can be serialized. Note that phantom problems are also possible for read or delete operations that fail because of missing keys. Such problems can be avoided since single read and delete operations access a single partition using the same protocol. Note that if multiple lock granularities were to be supported they must align with primary keys partitioning and cannot span protocols in our current system.

We now informally show that our method of mixing range query protocols maintains conflict serializability, which is proven similarly to Theorem 1. Consider two transactions $t(i)$ and $t(j)$ that span two protocols $p_1$ and $p_2$. If $t(i)$ depends on $t(j)$ in $p_1$ and $t(j)$ depends on $t(i)$ in $p_2$ (i.e. they have conflict cycle), we show that it’s impossible that both $t(i)$ and $t(j)$ are committed. Recall that $p_1$ is COCSR, which requires the commit time of $t(i)$, $c(i)$, is larger than that of $t(j)$, $c(j)$. Conversely, the conflict on $p_2$ requires $c(i)$ less than $c(j)$. Therefore, no valid commit time exists for both transactions and committing both
transactions is impossible. This shows that our mixed method is COCSR, which is sufficient to maintain conflict serializability.

4.5 Logging and Recovery

We use a uniform logging format and recovery system for all candidate protocols in CormCC. Since CormCC guarantees conflict serializability, ARIES [18] is compatible. However, we focus on supporting a fast and parallel logging and recovery protocol designed for multi-core main-memory databases, SiloR [39]. SiloR contains redo-only logging and relies on a global and periodically advanced counter (epoch), whose value represents a short time period. Global epochs enable multiple logs that are appended in parallel and out-of-order, and provides a partial order of log segments to enable easy parallel recovery. Specifically, each epoch is a logging and recovery unit: the results of a transaction that logs in epoch $e$ are released to clients (i.e. committed) only when the logs of all transactions that belong to $e$ are persisted to disk (i.e. batch logging). SiloR adopts physical logging with each log entry consisting of a log sequence number (i.e. LSN) for the committed transaction plus the table, key, and value information for all records modified by that transaction. Note that since logs are written in parallel and potentially out-of-order, LSN is not based on log position but indicates the version of values associated with it. The LSN is composed of an epoch number and a version number representing the write order within the epoch, where the larger a log's LSN is, the more recent the update. In the recovery phase, SiloR partitions the database records into disjoint subsets and assign each subset to a single worker; each worker starts with the latest checkpoint (if present) and replays each epoch that has been persisted by applying the latest modification (i.e. the record with the largest LSNs) to records assigned to it. Our extensions do not change the recovery for SiloR.

We extend SiloR in two ways: i) we add undo logging to SiloR to enable protocols that apply writes to databases in the execution phase (e.g. 2PL and PartCC) and ii) we let candidate protocols generate LSNs that track write dependencies between transactions,
which is critical for recovery.

Undo logging serves for crash recovery and for transaction roll back. SiloR’s recovery does not require undo logging even when protocols apply writes before commit, as each epoch of logs does not contain log entries from uncommitted transactions. Therefore we only require an undo log to support transaction rollback due to aborts. Specifically, we let each worker maintain a log buffer tracking writes and undos of the transaction it is executing. If the transaction aborts, we simply undo its writes according to the undo log. When the transaction enters commit phase, the undo log buffer is cleaned for the next transaction.

Since SiloR is designed for OCC, we only discuss computing LSNs for other protocols (e.g. 2PL and PartCC). The LSN is obtained in the commit phase by combining the current epoch number and monotonically increasing atomic counters. Each protocol maintains their own counter that is incremented on transaction commit. Such LSNs reflect serializability and thus can track write dependencies. If a transaction spans multiple protocols, we have multiple LSNs each computed by a single protocol. For each LSN, we output one log entry, which includes the LSN and all record values modified by the owning protocol. Since each protocol manages a disjoint subset of records, for a specific record the owning protocol’s LSN reflects the write dependency for this record and determines the most recent value for recovery.

4.6 Extending CormCC

Adding a new protocol to CormCC must meet and not break any conditions in Theorem 1 to Theorem 3. To support range queries, the new protocol should customize its own range query protocol that must maintain COCSR. The new protocol should maintain its own separate set of concurrency control metadata (e.g. locks or timestamps), which can be either stored within records or shared in a global data structure. Mediated protocols of the new protocol and each old protocol should also be added to the system. Note that CormCC currently only supports single-version records and we plan to explore MVCC in the future.
CHAPTER 5

PROTOTYPE DESIGN

We build a prototype main-memory multi-core database that can dynamically mix PartCC from H-Store [10], OCC from Silo [28], and 2PL from VLL [23] 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 concurrency control. 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, Silo OCC commits; otherwise, it aborts. VLL optimizes 2PL by co-locating each lock with its corresponding record to reduce contention of the centralized lock manager.

The right part of Figure 5.1 shows how our prototype mixes PartCC, OCC, and 2PL. It 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 concurrency control protocol for each partition according to its access pattern. An 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 left part of Figure 5.1 shows the runtime architecture of our prototype that adopts CormCC to support the mixed execution of PartCC, OCC, and 2PL, and a predictive model to select the optimal protocol. Not shown in the figure, is the central coordinator thread that manages these components and receives extracted features from workers.

Given an OLTP application, we use synthesized workloads to train the predictive model offline based on simple classifiers (e.g. Decision Tree). We synthesize various workloads using application information (e.g. stored procedures). For example, the stored procedures can be mixed to explore the effects of the percentage of read operations on candidate protocols. For each synthesized workload, we run all candidate protocols, extracts features for each protocol, and outputs a training case composed of their average feature values, and a class label that represents the protocol(s) having the highest throughput for a given workload. Note that although we use synthesized workloads, it is possible to use online workloads (e.g. logs) to train the model.

CormCC uses the trained model online to predict the ideal concurrency control protocol under workload variation. The key to enabling this model is monitoring and extracting features from active workloads. We design two strategies to trigger feature extraction: (i) always monitor using sampling or (ii) monitor the throughput of a database and perform extraction if throughput changes more than a threshold (e.g. 20%) or some defined time period elapses. The features of a partition are passed to the predictive model, which outputs an ideal protocol for this partition. If a protocol switch is triggered for a partition, the coordinator informs their workers to change execution engines for the new protocol.
5.1 Mixed Execution

In this section, we introduce how CormCC mixes PartCC, OCC, and 2PL during normal execution and protocol switch.

![Diagram of mixed execution of PartCC, OCC, and 2PL]

**Figure 5.2:** Mixed execution of PartCC, OCC, and 2PL

5.1.1 Normal Execution

Figure 5.2 gives an overview of the execution of CormCC with PartCC, OCC, and 2PL. 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 [28]. 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.

Such mixed execution meets all conditions of Theorem 1 to Theorem 3. First, PartCC and 2PL are rigorous, and therefore are COCSR; Shang et al. [24] have proven that Silo OCC is also COCSR. By Theorem 1, 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 (by Theorem 2). Finally, all candidate protocols are strict, so is their mixed execution (by Theorem 3).

### 5.1.2 Online Reconfiguration

The key to enabling online reconfiguration is to compose the mediated protocol, which we have discussed for OCC and 2PL in § 4.3. Here, we first describe the mediated protocol between PartCC and OCC (denoted as PartOCC). If we need to switch the protocol for a partition from PartCC to OCC, transaction workers using PartOCC first acquires the partition lock and executes OCC logic for all access (e.g. read/write) to this partition. During the upgrade phase, conflicts on these partition can be synchronized by the partition lock; in the degrade phase, conflicts can be processed by OCC logic. The mediated protocol between PartCC and 2PL is similar to PartOCC.

### 5.2 Building Predictive Model

In this section, we first present our features that support our predictive model for PartCC, 2PL, and OCC. Then, we discuss methods of efficiently extracting these features. Finally, we introduce training this model. Note that we use record access to represent transactions reading or writing a record and partition access to represent transactions accessing records within a partition.

#### 5.2.1 Modeling and Feature Engineering

While many workload characteristics can impact the comparative performance of candidate protocols, careful feature selection and engineering can improve performance modelling and reduce the amount of data collected by workers. We group features into partition-level features that impact the performance of PartCC and record-level features that impact the
We track partition access conflicts (PartConf) that reveal the duration PartCC transactions would spend on requesting partition locks. This feature is based on the number of cross-partition transactions and the mean number of partitions involved in cross-partition transactions. It determines the applicability of PartCC, as high PartConf increases the likelihood that PartCC will stall due to waiting for partition locks.

We then track three record-level features, which model the impact of the mix of stored procedures and the distribution of record access (e.g. Zipf). First, we detect the possibility of concurrent transactions reading or writing the same records (RecCont). Then, we track the ratio of read operations (ReadRatio) and the average number of records accessed by transactions (TransLen). These three features together determine record conflicts across concurrent transactions, which is a critical factor influencing the performance of PartCC, OCC, and 2PL.

5.2.2 Feature Extraction

The basic process of feature extraction involves concurrency control workers collecting statistics, reporting them to a centralized coordinator, and the coordinator extracting features from these statistics. We first describe feature extraction of ReadRatio and TransLen since they are trivial to extract. Then, we discuss the challenging feature extraction of RecCont. We use the same method of extracting RecCont to obtain PartConf.

ReadRatio. The number of read and write operations are sampled from all concurrency control workers and the ratio is the number of reads divided by total operations.

TransLen. We use a weighted number of records per transaction access to represent TransLen, as we observe that table sizes can affect the number of conflicts (e.g. in TPC-C a write operation on the Warehouse table results in more conflicts than on the Stock table). We give different weights to record access to different tables in inverse proportion to table sizes. To calculate TransLen transaction workers track the number of transactions sampled
and records accessed for each table. The coordinator takes the sum of the weighted access number for each table and divides it by the total number of sampled transactions.

RecCont To detect record contention, a simple but unscalable approach is to have workers collect most frequently records and corresponding frequency and aggregate them at a centralized coordinator to determine contention. Instead, we develop a method that distributes the centralized feature computation into workers to reduce the feature extraction time.

Embedding independent and parallel feature extraction into workers is challenging as different protocols demonstrate record contention with different behaviours (e.g. OCC exhibits contention using aborts, while 2PL reacts to contention by waiting for requesting locks and/or aborting). To extract the same record contention among all candidate protocols, we build a separate process that emulates concurrent record access to estimate contention such that dependence on behaviour of protocols is minimized. This process includes two lightweight repetitive phases: a mark phase and a detection phase, which are executed independently by each worker. In the mark phase, one worker tracks a fixed number of record accesses and marks these records as accessed. After that, the worker moves into the detection phase, where it samples a configured number of record accesses and checks how many other workers have marked these records. Afterwards, it clears all marks and repeats the process. Intuitively, as Figure 5.3 shows, the approach has workers mark a fixed number of records, followed by a wide window to detect sampled conflicts with other workers, without using heavyweight synchronization techniques.

To enable our algorithm, we let each record incorporate a counter recording the number of marks received, which is accessible by all workers.\footnote{Note that it is possible to associate each record with a shared counter (e.g. ranges) to reduce overhead.} In addition, each worker maintains a local structure for managing detection metadata and recording contention. It includes a
record set to track records that have been marked, and for each table, it contains a contention value along with the number of sampled record access.

Each worker invokes the algorithm for each record access. It starts with the mark phase. Workers check whether the input record already exists in the record set. If so, the algorithm returns as we do not mark a record twice for one worker. Otherwise, this worker increments the record’s counter and puts the record’s reference into the private record set. If a fixed number of records has been marked, this algorithm moves into the detection phase. The detection phase samples record access according to the configured sample rate. For each sampled record, the worker copies the counter’s value of the record and checks whether this record is marked by itself – and if so it decrements the local value. The local value represents the number of other workers having marked this record. It is added to the amount of contention detected and the number of sampled access is increased correspondingly. Finally, after one worker has sampled record access for a configured number of times, it releases its marked records and repeats the mark phase. The coordinator collects the amount of contention and the number of sampled access from each worker and sums the ratio of the two values for all tables as the final \texttt{RecCont} score. For \texttt{PartConf}, we sampled the transactions, and use the above method to mark and detect conflicts for each partition access.

5.2.3 Building a Predictive Model

We use the above features to build a multi-tier predictive model by cascading binary classifiers. We build each classifier using a subset of critical features. The first classifier, C1, determines whether to use PartCC or not. It is based on partition access characteristics (\texttt{PartConf}) and record access characteristics (\texttt{RecCont}, \texttt{TransLen}, and \texttt{ReadRatio}). If PartCC is not used, C2 decides to use OCC or 2PL. This classifier is only based on record conflicts (\texttt{RecCont}, \texttt{TransLen}, and \texttt{ReadRatio}).

We train these classifiers by synthesizing a set of training workloads, which can effectively explore feature space such that trained models can accurately predict the ideal concurrency

29
control for future workloads. In addition to one-time offline training, it is also possible to replay online workloads (i.e. using online workload trace) offline to generate training cases and add them to our model incrementally.

We explore PartConf by varying the ratio of cross-partition transactions and fix number of partitions cross-partition transactions access. We set parameters to make stored procedures span multiple partitions (e.g. for TPC-C we can increase the ratio of remote warehouse access). Specifically, we binary search the ratio of cross-partition transactions to find the cutting point whether to use PartCC or other protocols.

We search the feature space of RecCont, TransLen, and ReadRatio by varying record access distribution and stored procedure mix. We use a Zipf distribution and evenly search its theta value with an interval (e.g. 0.1). Note that theta 0 generates a uniform distribution. For TransLen and ReadRatio we mix stored procedures in pairs. For example, for any pair of two stored procedures A and B from an application, the mix includes 100% A 0% B, 66% A 37% B, 37% A 63% B and 0% A 100% B. We find pairwise mixing is good enough to explore TransLen and ReadRatio for our benchmarks. For more complicated applications, it is possible to use three-way mix or greater.

5.3 Overhead and Optimizations

One major limitation of CormCC is that it requires each protocol maintain a separate metadata set for all records. However in our prototype, this is not a problem because PartCC has minimal metadata to maintain (i.e. partition locks) and prior work [24] shows that it is easy to merge the timestamp of OCC with the lock of 2PL into a single word. This optimization makes our protocol maintain the same size of metadata as systems with single OCC or 2PL. Another overhead of CormCC is that executing a transaction may involve logic of multiple protocols. We show in § 6.5 that this introduces minimal overhead compared to executing static single protocols.
CHAPTER 6
EXPERIMENTS

We now evaluate the effectiveness of mixed execution and online reconfiguration of CormCC. Our experiments answer the following questions:

- How does CormCC perform compared to state-of-the-art mixed concurrency control approaches (§ 6.3)?
- How does CormCC adaptively mix protocols under varied workloads over time (§ 6.4)?
- What is the performance benefit and overhead of mixed execution (§ 6.5)?
- What is the performance benefit and overhead of online reconfiguration (§ 6.6)?

All experiments are run on a single server with four NUMA nodes, each of which has an 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 [19], 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 maintains its own memory pool to avoid memory allocation contention across many
cores [36]. A coordinator thread is used for extracting features from statistics collected by workers and predicting the ideal protocol to be used. Our prototype supports automatically selecting PartCC from H-Store [10], OCC from Silo [28], or 2PL No-Wait from VLL [23] (which co-locates locks with records) for each partition using a predictive model based on Decision Tree [29] classifiers. Note that we have compared 2PL variants, and find that 2PL No-Wait performs best in most cases because of lower synchronization overhead of lock management [22]. Our comparison additionally includes a general mixed concurrency control framework based on Tebaldi [25] (denoted as Tebaldi) and a hybrid approach of OCC and 2PL [24, 30] (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 (i.e. 32000 records in total) and for our lowly conflicted workload no records will be locked and we use OCC for them. Note that we label conflicted records offline only for Hybrid, so there is no overhead for detecting conflicted records at runtime; for CormCC, we do online feature extraction to enable our adaptive protocol switching.

6.2 Benchmarks & Experiment Settings

We use two benchmarks in our experiment:

**YCSB** We generate one table for YCSB which 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** 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.

We configure the database to run with low sample rates to minimize overhead of feature
Table 6.1: Parameters for generating training cases for predictive model

<table>
<thead>
<tr>
<th></th>
<th>CrossRatio</th>
<th>RecDist</th>
<th>TransMix (YCSB)</th>
<th>TransMix (TPC-C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>B.S. [0, 100]</td>
<td>Zipf 0, 0.4, 0.8, 1.0, 1.2, 1.4, 1.5</td>
<td>RR 50, 70, 90, 95, 100</td>
<td>9 Pairwise Mix</td>
</tr>
<tr>
<td>C2</td>
<td>100</td>
<td>Zipf [0, 1.5] interval 0.1</td>
<td>TLen 10, 15, 20, 25</td>
<td></td>
</tr>
</tbody>
</table>

extraction. For record contention detection, we sample one from every 2000 record accesses. For the monitoring of PartConf, TransLen, and ReadRatio we sample one from every 500 transactions. We find such configuration has minimal overhead to our prototype and can guarantee accurate protocol prediction. We configure the database to always monitor and workers to report statistics or partial features to the coordinator every second. We find that the overhead of always running feature extraction is a 3–4% drop in throughput on average.

To generate varied workloads, we tune the following parameters:

- **CrossRatio**: CrossRatio represents the percentage of cross-partition transactions ranging from 0 to 100. We set the number of partitions a cross-partition transaction will access to 2 in our experiment.

- **TransMix**: TransMix represents the mix of stored procedures issued in the workload. Note that YCSB only has one stored procedure, and we tune the number of operations per transaction and the ratio of read operations.

- **RecDist**: 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.

Table 6.1 shows the parameters for training the classifier responsible for using PartCC or not (C1), and the classifier choosing OCC or 2PL (C2). For C1, we binary search (denoted as B.S.) CrossRatio and vary theta of Zipf for RecDist. For C2, we fix CrossRatio and vary theta from 0 to 1.5 with interval 0.1 for RecDist.

We generate TransMix for YCSB by directly varying the percentage of read operations (denoted as RR in Table 6.1) and the number of operations per transaction has (denoted as...
TLen in Table 6.1). We then generate TransMix by mixing three major stored procedures NewOrder, Payment, and OrderStatus in pairs (introduced in § 5.2.3) for TPC-C (i.e., 9 transaction mixes in total), which is enough to explore the feature space of read rate and transaction length.

6.3 Comparison with Tebaldi

Tebaldi [25] 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 6.1 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 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 is reported to have the best performance for TPC-C [25].

We first compare CormCC with Tebaldi using TPC-C over mixing well-partitionable and non-partitionable workloads. We partition 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 result of 2PL.

Figure 6.2 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 become 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 [25], it can become a performance 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 6.3 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 Tebali 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 [25] 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 benefit 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. We have CormCC collect features every second and concurrently select the ideal protocol for each partition at runtime.

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 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 6.4 shows the test result 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 for partitions in the same way as YCSB test. We report the results of every 2 seconds in Figure 6.5. 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 6.4, 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 $\text{max}$ and $\text{min}$ respectively) for each varied workload (i.e. every 5s) of both benchmarks in Figure 6.6 and Figure 6.7. We additionally report the ratio of the mean throughput of CormCC to the average throughput of three single fixed protocols (labelled by $\text{avg}$) 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 Test Mixed Execution

In this section, 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 6.8 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 6.9 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 6.10 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 change protocols or add extra metadata operations to synchronize conflicts across protocols.

6.6 Test 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.

As shown in Figure 6.11, 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.
CHAPTER 7
CONCLUSION

By exploring the design space of mixed concurrency control, CormCC presents a new approach to generally mix 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


APPENDIX A
CORRECTNESS OF MEDIATED SWITCHING

We prove that CormCC maintains COCSR and recoverability during mediated switching. Note that by Theorem 2 mediated switching can avoid or detect deadlocks.

**Theorem 4.** Given that each protocol $p \in P$ manages a partition of database records, if i) a record set $RS$ of a partition managed by $p_1$ is additionally managed by a mediated protocol $p_{12}$, ii) any protocol $p \in P$ in CormCC is COCSR, then CormCC generates conflict serializable schedules.

**Proof.** To prove CormCC during mediated switching is COCSR, we need to prove for any two committed transactions $t(i)$ and $t(j)$ that have conflicts, their commit time follows $c(i) < c(j)$ if $t(j)$ depends $t(i)$ or vice versa (by the definition of COCSR).

Without loss of generality, suppose that $t(i)$ and $t(j)$ have conflicts on the record $r$ and $t(i)$ accesses $r$ before $t(j)$. Therefore, we have $t(j)$ depends on $t(i)$.

- if either $t(i)$ or $t(j)$ is not successfully committed, this violates our assumption that $t(i)$ and $t(j)$ are committed transactions.

- if both $t(i)$ and $t(j)$ are committed, and $r$ does not belong to $RS$, $c(i)$ must be less than $c(j)$ because $r$ is managed by a single protocol $p$, which is COCSR and ensures $c(i) < c(j)$.

- if both $t(i)$ and $t(j)$ are committed, and $r$ belongs to $RS$, $c(i)$ must be less than $c(j)$ because both $p_1$ and $p_{12}$ executes the concurrency control logic of $p_1$, which ensures $c(i) < c(j)$.

Therefore, we prove that CormCC during mediated switching is COCSR and thus conflict serializable. □
Theorem 5. Given that each protocol \( p \in P \) manages a partition of database records, if i) a record set \( RS \) of a partition managed by \( p_1 \) is additionally managed by a mediated protocol \( p_{12} \), ii) any protocol \( p \in P \) in CormCC is strict, then CormCC is strict.

Proof. We need to prove that for a record \( r \) that is written by a transaction \( t(i) \), \( r \) will not be read or overwritten by other transactions until \( t(i) \) commits or aborts. Consider two cases:

- if \( r \) does not belong to \( RS \), then \( r \) is managed by a single protocol \( p \) and \( p \) is strict. Therefore, \( r \) will not be read or overwritten by other transactions until \( t(i) \) commits or aborts.

- if \( r \) belongs to \( RS \), then \( r \) will not be read or overwritten by other transactions until \( t(i) \) commits or aborts because both \( p_1 \) and \( p_{12} \) executes the concurrency control logic of \( p_1 \) and \( p_1 \) is strict.

Therefore, CormCC is strict. \( \square \)