Transaction Scheduling: From Conflicts to Runtime Conflicts

Transaction Scheduling: From Conflicts to Runtime Conflicts

Yang Cao[1], Wenfei Fan[1,2,3], Weijie Ou[2], Rui Xie[2], Wenyue Zhao[1]
University of Edinburgh[1] Shenzhen Institute of Computing Sciences[2] BDBC, Beihang University[3]
{yang.cao@, wenfei@inf. wenyue.zhao}ed.ac.uk, {ouweijie, xierui}@sics.ac.cn

Abstract

This paper studies how to improve the performance of main memory multicore OLTP systems for executing transactions with conflicts. A promising approach is to partition transaction workloads into mutually conflict-free clusters, and distribute the clusters to different cores for concurrent execution. We show that if transactions in each cluster are properly scheduled, transactions that are traditionally considered conflicting can be executed without conflicts at runtime. In light of this, we propose to schedule transactions and reduce runtime conflicts, instead of partitioning based on the conventional notion of conflicts. We formulate the transaction scheduling problem to minimize runtime conflicts, and show that the problem is NP-complete. This said, we develop an efficient scheduling algorithm to improve parallelism. Moreover, for transactions that are not packed in batches, we show that runtime conflict analysis also helps reduce conflict penalties, by proposing a proactive deferring method. Using standard and enhanced benchmarks, we show that on average our scheduling and proactive deferring methods improve the throughput of existing partitioners and concurrency control protocols by 131% and 109%, respectively, up to 294% and 152%.

1 Introduction

There has been increasing demand on high volume of concurrent transactions from *e.g.,* e-commerce, FinTech and cloud applications [28]. This and the growing dominance of multicore machines highlight the need for pursuing higher throughput and parallelism of multi-thread transaction processing [4, 31, 38, 56]. Due to contended operations that read and write the same data items, transaction execution has to be guarded against concurrency anomalies to uphold the desired isolation levels. There are mainly two types of approaches to maximizing concurrency while providing isolation guarantees: (a) partition-based approaches [14, 21, 31, 34, 38, 45] that use transaction partitioners to decide a transaction-to-thread assignment for a batch of transactions before their execution; and (b) concurrency control (CC) based approaches [9, 24, 38, 49, 57] that, instead of searching for good transaction-to-thread assignments, focus on CC protocols to resolute contended transactions on-the-fly.

Transaction partitioning methods focus on a bundle of transactions that are *e.g.,* a set of transaction collected in a fixed duration [34] or workloads [21] for which the transaction logic is known in advance (*e.g.,* stored procedures and hard-coded templates), as targeted by deterministic databases [4, 36]. Such workloads allow the system to carry out analysis over the transactions, *i.e.,* transaction partitioning, to decide the best way of assigning transactions to threads. More specifically, given a workload $\mathcal{W}$ over k threads, a transaction partitioning method computes a partitioning $(P_1, P_2, \ldots, P_k)$ of $\mathcal{W}$ that assigns each P_i to a distinct thread i such that transactions in the same partition are executed serially, while concurrent execution is governed by CC for the desired isolation levels. Its objective is to minimize conflicts among transactions across different partitions to reduce CC cost, while maximizing load balance so as to minimize total parallel execution time.

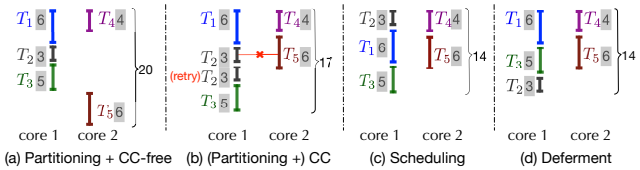


Figure 1: Different transaction executions in Examples 1-2

Instead of searching for the best thread assignment for the transactions, CC aims to resolute conflicts between transactions that are concurrently executed, and do not require the access sets of transactions before execution. There are two types of CC cost: (a) CC overhead incurred on every transaction and (b) conflict penalty when a conflict between transactions happens at runtime, *e.g.,* block (waiting) time with locking-based CC [10, 17] or abort/retry penalty with optimistic concurrency control protocols [5, 24]. In practice, CC protocols with higher overhead are more likely to reduce conflict penalties. Hence, the effectiveness of CC relies on delicate trade-offs between the overhead and the ability to reduce conflict penalties.

We refer to transactions targeted by the partitioning and CC approaches as *bundled transactions* and *unbundled ones*, respectively.

Runtime conflict. Conflicts are conventionally defined relative to the isolation level adopted by transaction systems. For serializability [20], transactions T and T' are in conflict if they access the same data item and at least one of them updates it. For snapshot isolation [7], T and T' are conflicting if they write to the same data item.

However, transactions that are conventionally considered in conflicts can be executed concurrently without conflicts at runtime as long as they are scheduled properly, as illustrated below.

Example 1: Consider a set $\mathcal{W}_0$ of transactions $\{T_1, T_2, T_3, T_4, T_5\}$:

- $T_1 = \mathtt{R}[x_2]\mathtt{W}[x_2]\mathtt{R}[x_3]\mathtt{W}[x_3]\mathtt{R}[x_4]\mathtt{W}[x_4]$,
- $T_2 = \mathtt{R}[x_1]\mathtt{W}[x_2]\mathtt{W}[x_1]$,
- $T_3 = \mathtt{R}[x_3]\mathtt{W}[x_3]\mathtt{R}[x_2]\mathtt{R}[x_3]\mathtt{W}[x_2]$,
- $T_4 = \mathtt{R}[x_5]\mathtt{W}[x_5]\mathtt{R}[x_6]\mathtt{W}[x_6]$, and
- $T_5 = \mathtt{R}[x_1]\mathtt{W}[x_1]\mathtt{R}[x_5]\mathtt{W}[x_5]\mathtt{R}[x_1]\mathtt{W}[x_1]$,

where $\mathtt{R}[x_1]$ denotes a read of data item x_1 and $\mathtt{W}[x_1]$ is a write to x_1. Assume that the targeted isolation level is serializability. Then T_1, T_2 and T_3 are in conflict; similarly for (T_2, T_5) and (T_4, T_5).

Assume that we have two cores. Following [34], we partition $\mathcal{W}_0$ into $P_1 = \{T_1, T_2, T_3\}$ and $P_2 = \{T_4\}$, along with T_5 as a cross-partition transaction, as shown in Fig. 1(a). One can execute P_1 and P_2 concurrently without CC, followed by T_5 after both P_1 and P_2 are completed. Assuming that each of read and write operation takes a unit time; then the makespan of such execution (the concurrent execution of the transactions) of $\mathcal{W}_0$ is 20 time units. Note that the workloads P_1 and P_2 are not balanced, and the second core may idle for long before the cross-partition transaction T_5 can start.

Alternatively, by following [14, 21], one can start T_5 as soon as T_4 completes, as shown in Fig. 1(b); however, this requires CC since T_5 conflicts with T_2 that is being concurrently executed at the

other core. This may cause a retry of T_2 (when using OCC), and a makespan of 17 for $\mathcal{W}_0$ (ignoring CC overhead for all transactions).

In contrast, if we schedule the transactions as two *queues* $Q_1 = \langle T_2, T_1, T_3 \rangle$ and $Q_2 = \langle T_4, T_5 \rangle$, *i.e.,* imposing an execution order of transactions in the partitions, we can process $\mathcal{W}_0$ at two cores concurrently *without* CC as shown in Fig. 1(c), by executing Q_1 and Q_2 in their orders at the two cores independently. Although T_2 of Q_1 is in conflict with T_5 of Q_2, their executions actually *do not overlap*. In fact, the executions are serializable even without CC. Note that the workloads are more balanced than the case with partitioning, and the makespan of such an execution of $\mathcal{W}$ is 14 instead of 20. □

This example suggests that we consider *runtime conflicts* instead for partition-based approaches, which provide a more accurate characterization of transaction executions and more opportunities to improve concurrency. Moreover, runtime conflicts could also help with CC-based approaches for unbundled transactions.

Example 2: Consider the execution of transactions in Example 1 following a CC-based approach: we execute T_1, T_2 and T_3 at core 1 and T_4, T_5 at core 2 in order, using OCC. Then T_2 retries due to conflict with T_5, causing a total execution time of 17 (as shown Fig. 1(b)).

If prior to the start of T_2, core 1 could find that T_5 is being executed at core 2, which will incur conflict with T_2, then a better option is to "defer" T_2 and execute T_3 first instead. In this way, T_3, T_5 and T_2 can all commit without retry with a total time of 14 instead of 17 as shown in Fig. 1(d). That is, we can reduce runtime conflicts during execution for CC-based approaches, by proactively deferring transactions and changing their execution order on-the-fly. □

Transaction scheduling. For bundled transactions that are handled by existing partitioners, we propose to schedule transactions so as to minimize runtime conflicts. Given a workload $\mathcal{W}$ that has been partitioned over k cores, we generate k queues $(Q_1, \ldots, Q_k)$ and a residual set $\mathcal{R}$, such that Q_i's consist of transactions in $\mathcal{W}$ without incurring runtime conflicts, and hence can even be executed without CC. As opposed to partitioning, transactions in Q_i are *scheduled*, by placing an ordering on the executions of their transactions. Transactions in $\mathcal{R}$ are executed over all cores using CC.

Intuitively, transactions in Q_i and those in Q_j $(i \neq j)$ have *no runtime conflict* if they are executed on schedule, although they may be conventionally considered in conflict; hence transactions in Q_i run *serially* at the i-th core, and *in parallel* with those in Q_j $(i \neq j)$ without conflicts. Transactions in $\mathcal{R}$ inflict runtime conflicts and hence are executed at all cores with CC. As shown in Example 1, scheduling allows more transactions to be put in $Q_1, \ldots, Q_k$ and executed concurrently without conflicts compared to partitioning.

We show that transaction scheduling is intractable. This is not surprising since it is more intriguing than the transaction partitioning problem, which is already NP-hard [34]. This said, we develop an efficient algorithm that is able to either refine a transaction partition into a schedule, *i.e.,* partitions with ordering, or compute a schedule starting from scratch when a partition is not available. Scheduling targets bundled transactions to which the partitioning methods are applicable. It not only reduces runtime conflicts but also balances workloads among the threads, improving the throughput. It is particularly effective for transactions with skewed costs or I/O latency, which existing partitioners do not handle very well.

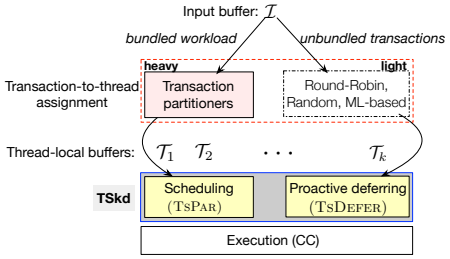


Figure 2: A typical transaction system with TSkd: TSkd *works as a plug-in for existing transaction systems that either use transaction partitioners to assign transactions to threads or directly rely on* CC *protocols. It sits between the transaction-to-thread assignment module and transaction execution engines to reduce runtime conflicts.*

Proactive transaction deferment. For unbundled transactions handled by CC-based approaches, we develop a lightweight method to reduce their CC cost by proactively deferring transactions on-the-fly that are on course to inflict runtime conflicts with other transactions being executed. Unlike partitioners that are often used for preprocessing batched transactions prior to their execution, CC is part of the execution phase and hence proactive deferment has to be at a low cost since otherwise the benefit of reduced runtime conflicts could be canceled out by the increased CC overhead.

To this end, we develop a lock-free structure that tracks the progresses of transaction execution of all the threads, which allows us to efficiently detect whether a runtime conflict would happen before executing a new transaction, during execution time. The lock-free design avoids data racing overhead when threads update and look up execution progresses concurrently. In addition, it has a parametric complexity to trade the detection overhead for larger reduced conflict penalties and hence higher throughput.

Prototype and evaluation. As a proof of concept, we develop TSkd, a lightweight tool as shown in Fig. 2 for improving existing partitioning-based or CC-based transaction systems via scheduling (module TsPar) and proactive deferment (module TsDefer).

Given a workload $\mathcal{W}$ for partitioners, TSkd (TsPar) first learns rough runtime estimates of the transactions in $\mathcal{W}$ via execution histories, partial dry-runs used by partitioners [4], or more sophisticated cost estimators [11, 42, 46, 51]. It then converts the partitioning of $\mathcal{W}$ into a schedule, *i.e., k* queues $(Q_1, \ldots, Q_k)$ and residual $\mathcal{R}$, to reduce runtime conflicts and improve load balance. The transactions in $Q_1, \ldots, Q_k$ can be concurrently executed as scheduled without CC if the estimates are accurate. A subtle issue arises when time estimates are not perfectly accurate; if so some transactions may end up inflicting runtime conflicts. To cope with this, TSkd uses both CC and proactive transaction deferment to execute the scheduled queues and $\mathcal{R}$. This guarantees that TSkd always executes transactions correctly, while still benefiting from reduced conflicts and balanced workload among the threads via scheduling.

For unbundled transactions that are executed directly using CC, TSkd employs proactive transaction deferment (TsDefer) to reduce conflict penalties. TsDefer detects potential conflicts between transactions concurrently executed at runtime, and it imposes no restrictions on the workload that CC schemes deal with.

To allow a fair and repeatable evaluation, we integrate TSkd into

DBx1000, a popular open-source testbed for in-memory transaction processing with built-in implementation of major CC schemes [2, 56]. Using TPC-C and YCSB, we verify that TSkd improves popular partitioners and CC protocols by 131% and 109%, respectively, and reduces their retries by 45.3% and 45.7%, up to 294% and 152% for throughput and 61.1% and 63.9% for retry reduction. Moreover, TSkd remains effective for transactions with varying degrees of long-tail I/O latency, making existing partitioners and CC schemes robust against *e.g.,* databases hosted in disks. We observe that TSkd is less effective for extremely short-run transactions as conflict penalties play a much smaller role in their performance.

Contributions & organization. To summarize, in this work we develop techniques that improve both transaction partitioners and CC schemes by reducing runtime conflicts. More specifically:

- We formalize a notion of runtime conflicts and propose transaction scheduling to improve transaction partitioners by making conflicting transactions conflict-free at runtime (Section 2.2).
- We propose proactive transaction deferment for CC protocols to reduce runtime conflicts during execution time (Section 2.3).
- We develop TSkd, a tool that implements both methods. It can serve as a plugin for existing transaction systems that use either transaction partitioners or CC protocols directly, providing delicate controls over and flexible trade-offs between transaction retry penalties and CC overheads (Section 3).
- We settle the complexity of scheduling and develop an efficient scheduling algorithm for transaction scheduling (Section 4).
- We develop a lightweight lock-free structure that enables proactive transaction deferment (Section 5).
- We empirically verify that TSkd improves the throughput of both partitioners and CC schemes, and makes them robust against large skewed I/O latency and runtime (Section 6).

We remark that TSkd is not to replace existing CC schemes or transaction partitioners. Instead, it is positioned as a tool to expose opportunities to improve their performance in a non-intrusive way.

We discuss related work in Section 7 and future work in Section 8.

2 Reducing Runtime Conflicts for Transactions

In this section, we first review the basics of transaction processing (Section 2.1). We then present the notion of runtime conflicts, based on which we propose transaction scheduling (Section 2.2) and proactive transaction deferring (Section 2.3).

2.1 Transaction Preliminaries

A transaction is a sequence of database actions that are to be executed as atomic work units, including reads from and writes to a database. The tuples in the database that are read (resp. written) by a transaction T are referred to as the *read* (resp. *write*) set of T.

Conflicts. Two transactions T and T' are *in conflict w.r.t.* a CC protocol ρ if they contain contended operations under ρ. Contention is defined relative to the particular isolation level [7] upheld by ρ. For instance, if ρ enforces serializability, then T and T' are in conflict if they both access (read or write) the same data item x and at least one of them writes x. If ρ enforces snapshot isolation, then T and T' are in conflict if they both write the same data item. As an example, under serializability, T_2 and T_5 of Example 1 are in conflict; however, they do not conflict under snapshot isolation.

Transaction T is *conflict-free* with T' if they are not in conflict. Conflicting transactions require CC to ensure the correctness at the particular isolation level upheld by the CC protocol ρ.

Workload model. As shown in Fig. 2, we consider multi-thread transaction systems that either use transaction partitioners for bundled transactions or directly use CC for unbundled ones. Indeed, the majority of transaction systems fall into these two categories. They consist of (a) input buffer $\mathcal{I}$ that receives transactions to be processed; and (b) thread-local buffers $\mathcal{T}_1, \ldots, \mathcal{T}_k$, where each $\mathcal{T}_i$ ($i \in [1, k]$) contains the transactions to be executed by thread i.

For *bundled* workloads $\mathcal{W}$, *i.e.,* a set of transactions revealed to $\mathcal{I}$ all at once prior to execution, the practice is to first decide an assignment for transactions in $\mathcal{W}$ to the k thread-local buffers, by partitioning them based on their read/write sets [4, 14, 21, 31, 34, 38, 45]. All threads then execute assigned transactions in their local buffers concurrently. By spending more preprocessing time, partitioners reduce transaction execution time with fewer aborts and less CC cost.

For transactions that are coming *unbundled* in $\mathcal{I}$, they are periodically flushed to the thread-local buffers via much lighter method than transaction partitioning, *e.g.,* round-robin, random or lightweight ML-based assignment methods [41]. All transactions are executed concurrently with CC. Hence, the impact of CC is greater on unbundled transactions when compared to bundled workloads.

Concurrency control (CC). When transactions in the local buffers are executed concurrently, conflicting transactions may cause anomalies that the desired isolation level prohibits, impairing the correctness of the execution results. To this end, transaction systems use CC protocols [9, 20, 24, 38, 49, 57] to prevent such anomalies. The cost of CC consists of (a) the overhead of the CC protocols, which is charged to every transaction even when it involves in no conflict, and (b) the cost of preventing anomalies when conflicts happen, *e.g.,* transaction blocking (waiting) time for locking-based CC and abort/retry costs for optimistic CC methods.

Transaction partitioning. The partitioning methods assign bundled transactions (workload) to threads for concurrent execution by analyzing the conflicts between transactions. It is typically used as a preprocessing method and incurs computation cost [4].

Conceptually, the process that transaction partitioners [4, 14, 33] partition a workload $\mathcal{W}$ can be modeled as a graph cut problem: by finding a minimum k-cut of the *conflict* graph G of $\mathcal{W}$, where vertices are transactions and an edge (T, T') indicates that T conflicts with T', one breaks $\mathcal{W}$ into k partitions, where the edge cut denotes conflicts between partitions. Each partition is executed serially by a dedicated thread and partitions are executed concurrently with CC.

One can also gathers all conflicting transactions into a single set $\mathcal{R}$ such that the k partitions become mutually conflict-free. We refer to $\mathcal{R}$ as a *residual* partition for $\mathcal{W}$ and the remaining k sets of transactions as *CC-free* partitions. It has been shown that by executing CC-free partitions without CC first, and then executing $\mathcal{R}$ with CC after all CC-free partitions complete, one can achieve higher throughput for high contention transactions [34].

In this setting, the objective of partitioning is to minimize the size of residual set (or k-cut) while balancing the partitions. For conve-

nience, we represent a partitioning of $\mathcal{W}$ simply as $(P_1, \ldots, P_k, \mathcal{R})$, where P_i's are CC-free partitions and $\mathcal{R}$ is the residual.

2.2 From Transaction Partitioning to Scheduling

We next propose transaction scheduling based on runtime conflicts.

Execution time. Transactions in practice often have varying execution time. The *execution time* of a transaction T, denoted by $\mathsf{time}(T)$, is the serial execution time of T by a single thread, from the start to the completion of T. It measures the duration (length) of T.

A variety of methods have been explored to estimate $\mathsf{time}(T)$, from a brute-force one that counts reads and writes in T (*e.g.,* Example 1), to advanced statistical and ML strategies [11, 42, 46, 51].

For a set P of transactions, $\mathsf{time}(P)$ denotes the total execution time of transactions in P, assuming that they are executed serially.

Transaction schedule. A *schedule* of a transaction workload $\mathcal{W}$ over k threads, denoted by $\mathcal{W}_f^{\prec}$, is a pair $(f, \prec)$, where

- f is a function that partitions $\mathcal{W}$ into $k+1$ disjoint sets $Q_1, \ldots, Q_k$ and $\mathcal{R}_s$ such that (a) $\bigcup_{i=1}^{k} Q_i \cup \mathcal{R}_s = \mathcal{W}$ and (b) T and T' are not in runtime-conflict for any $T \in Q_i$, $T' \in Q_j$ and $i \neq j$; and
- $\prec$ is an order relation on $\mathcal{W}$ such that for $i \in [1, k]$ and each Q_i, $\prec$ is a total order on Q_i.

We refer to Q_i as *a conflict-free queue.*

Similar to conventional transaction partitioners, function f clusters transactions in $\mathcal{W}$ by assigning them to threads. In contrast to partitioning, relation $\prec$ orders transactions assigned to each of the k queues. Moreover, the partition is guided by runtime conflicts (see below). As a consequence, the k conflict-free queues may include residual transactions for partitioning, and $\mathcal{R}_s$ only contains transactions that incur runtime conflicts with those in the queues.

Given a schedule $(f, \prec)$ over k threads, each core serially executes transactions assigned to it by f, in the order specified by $\prec$.

Runtime conflict. For a transaction $T \in \mathcal{W}$ that is assigned to queue Q_i, denote by $\mathsf{ts}(T)$ the *scheduled start time* of T by schedule $(f, \prec)$. It is defined as $\sum_{T' \in \mathsf{pred}(T)} \mathsf{time}(T')$, where $\mathsf{pred}(T)$ is the set of transactions in $\mathcal{W}$ that are assigned to the same Q_i as T by f and are ordered prior to T by $\prec$ (recall that $\mathsf{time}(T)$ is the serial execution time of T). Similarly, $\mathsf{tc}(T)$ denotes the *scheduled complete time* of T by $(f, \prec)$ and is defined as $\mathsf{ts}(T) + \mathsf{time}(T)$. The range $[\mathsf{ts}(T), \mathsf{tc}(T)]$ is called the *scheduled runtime* of of T.

Two transactions T and T' are *in conflict at runtime* by schedule $(f, \prec)$ if they are in conflict and their scheduled runtime overlap. If they are not in conflict at runtime, T and T' are called *RC-free.*

Consider a conventional transaction partitioner that splits workload $\mathcal{W}$ into a partition plan $f_c = (P_1, \ldots, P_k, \mathcal{R})$. Let $(f, \prec)$ be a schedule of $\mathcal{W}$ over k threads. Then we say $(f, \prec)$ *refines* partitioning f_c if P_i is a subset of Q_i partitioned by f for all $i \in [1, k]$.

Intuitively, the schedule is more tolerant than the partition by including conventionally conflicting transactions in the RC-free queues. As a consequence, $\mathcal{R}_s$ is a subset of residual $\mathcal{R}$, and hence more transactions can be executed concurrently.

Example 3: Continuing with Example 1, the queues Q_1 and Q_2 given there form a schedule for the workload $\mathcal{W}_0$. Moreover, the schedule refines the partitioning of $\mathcal{W}_0$ that consists of P_1 and P_2 and T_5. As shown there, the scheduling reduces the makespan of the execution of $\mathcal{W}_0$ from 20 to 14, and improves the throughput. □

2.3 Proactively Deferring Conflicting Transactions

For residual transactions or unbundled transactions that are directly executed using CC, we propose another method, called *proactive transaction deferment*, to further reduce conflict penalties for CC.

Recall that each thread i executes transactions in its local buffer $\mathcal{T}_i$ using CC so that anomalies can be prevented when conflicting transactions are being executed concurrently. Proactive deferment works by detecting whether runtime conflicts would happen when a thread i is about to execute the next transaction T in $\mathcal{T}_i$ after committing the current one; if so, it defers T and skips to the next transaction for execution. As will be seen in Section 5, the complexity of conflict detection is parameterized with knobs, to provide flexible trade-offs between the overhead of conflict detection and conflict penalties if runtime conflicts are not prevented in time.

For instance, as shown in Example 2, when core 1 detects that executing T_2 would inflict a conflict with T_5 being executed at core 2, it defers T_2 and skips to T_3, reducing retry cost of T_2.

We remark that proactive deferring is not a replacement for CC since it does not aim to detect and prevent *all* the runtime conflicts. Instead, it serves as a lightweight filter of transactions in the thread-local buffers before passing them to the transaction execution engine. It makes CC more robust against transactions with varying complexities by virtue of its parametric cost.

3 A Prototype Transaction Scheduler

In this section, we present an overview of TSkd, a lightweight tool for transactions to reduce runtime conflicts and CC penalties.

As shown in Fig. 2, TSkd works with existing transaction systems by serving as an intermediate layer between the transaction-to-thread assignment component and the execution engine. It consists of two components: (a) a *transaction scheduling* (TsPar) module that schedules and optimizes the partitioning generated by an existing transaction partitioner, and (b) a *proactive transaction deferment* (TsDefer) module that optimizes the thread-local buffers on-the-fly.

Scheduling (TsPar module). Given a transaction workload $\mathcal{W}$ and a partitioning of $\mathcal{W}$, TsPar computes a schedule $(f, \prec)$ that turns the partitioning into k RC-free queues $(Q_1, \ldots, Q_k)$ and a residual $\mathcal{R}_s$. TsPar then assigns transactions in Q_i to the local buffer $\mathcal{T}_i$ of thread i in the order; all k threads can execute the assigned transactions in parallel without CC if time estimates are accurate.

After all threads finish their local transactions, transactions in $\mathcal{R}_s$ are then executed with CC by all threads (with TsDefer enabled; more below). By default, TsPar assigns transactions to the thread by round-robin for its simplicity and load balance. One can also use other lightweight transaction-to-thread assignment methods supported by the underling systems, *e.g.,* random or ML-based [41].

To decide runtime conflicts for schedules, TSkd estimates the execution cost of the transactions in $\mathcal{W}$ by following *e.g.,* [11, 42, 46, 51]. TsPar *does not* rely on the actual transaction execution time; instead, it is only sensitive to the relative length of transactions. Hence, any estimates that roughly preserve the relative costs of transactions suffice. By default, TsPar uses execution histories to coarsely estimate costs; if T is instantiated with the same param-

eters as T' in the history, and if T and T' are based on the same template (*e.g.,* stored procedure), then TsPar uses the cost of T' as an estimate for T. When no T' has the same parameters as T, TsPar picks a T' with parameters close to that of T as a coarse estimate. For cases when execution histories are not available, TsPar adopts the partial dry-run approach used by *e.g.,* deterministic databases [4] to generate the estimates. The idea is to execute (samples) of transactions partially such that no writes are physically executed during the dry-run; this has also been shown effective to deduce access sets of transactions for partitioning [4]. In the extreme case, TSkd uses the sizes of the access sets of the transactions as a fallback.

To cope with inaccurate or even missing estimates, by default TSkd uses CC and TsDefer (more below) to guard the execution of RC-free transactions against potentially overlooked conflicts. This ensures that TSkd always upholds the desired isolation level no matter how bad the estimates can be. As will be shown in Section 6, TSkd still benefits from reduced conflicts and more balanced workload among the cores via scheduling, and yields higher throughput, especially for transactions with runtime skewness.

We will study the core problem underlying TsPar in Section 4.

Proactive deferment (TsDefer module). For unbundled transactions that are directly assigned to the thread-local buffers upon arrival or transactions of which runtime estimates are unavailable, TSkd uses TsDefer to reduce conflict penalties by further reducing runtime conflicts. TsDefer works by altering the ordering of transactions in the thread-local buffers during execution time, via carefully designed lightweight operations. It works as a dynamic filter of the transactions in the thread-local buffers and passes transactions that are not likely to cause runtime conflicts to the execution engine by deferring problematic transactions.

In a nutshell, TsDefer checks, prior to passing a transaction T for execution (with CC) at thread i, whether T is in conflict with transactions that are (very likely) being or will be immediately executed by some other thread. If so, it defers the execution of T by moving T to the end of the transaction queue in the local buffer for thread i; otherwise it passes T to the transaction execution engine.

We will discuss the implementation of TsDefer in Section 5.

Remarks. (1) TSkd aims to improve existing transaction systems, no matter whether partition-based [4, 14, 16, 31, 33, 34, 37, 38, 45, 54, 59], CC-based [2, 5, 8, 10, 15, 17, 18, 24, 26, 30, 40, 44, 52, 53, 57, 58] or hybrid [43, 45]. It neither imposes any new restrictions on nor makes assumptions about how these systems work. This enables existing systems to deploy TSkd without changes to benefit from the reduced runtime conflicts by transaction scheduling and deferment, via reduced CC overhead and conflict penalties.

(2) TSkd also uses TsDefer to reduce the execution cost of the unscheduled residual $\mathcal{R}_s$ returned by TsPar for bundled transactions.

(3) Neither TsPar nor TsDefer of TSkd is fixed to a specific isolation level such as serializability; instead, they work with arbitrary isolation levels that the underlying systems uphold, by observing conflicts according to the preferred isolation level.

Limitations. Because of its non-intrusive design, TSkd (TsPar and TsDefer) inherits some limitations of the systems it optimizes.

(1) Access sets. Transaction partitioners require that the access set of transactions is known upfront so that they could deduce conflicts between transactions. TsPar inherits this prerequisite to schedule the partitioning of bundled transactions. As a result, no random client-driven transactions are expected for both partitioners and TsPar; instead, they target transactions in the form of stored procedures or hard-coded templates, which are prevalent in *e.g.,* banking, e-commerce, business applications. Due to the importance of transaction partitioning, techniques have been developed to identify access sets in various scenarios [4, 36]. Moreover, for unbundled transactions where access sets are not known beforehand, we can fall back to CC-based approaches and use TsDefer instead.

Another limitation inherited is that TSkd executes range queries with CC, since partitioners do not optimize range queries for which read/write-sets are not available. On the positive side, TsPar inherits and reuses conflict graphs constructed by partitioners [14, 34] without reconstruction, as will be seen in Section 4.

(2) Application-specified dependencies. Similar to CC-based approaches for unbundled transactions, TsDefer operates on transactions visible during execution time only; hence, they do not have control on the global order of transaction execution to enforce transaction (*e.g.,* causal) dependencies implied by application logic. A possible way of mitigating this is to integrate consistency protocols with CC so that we have both isolation and causal consistency guarantees; TsDefer could be extended for such cases by only deferring a transaction when a limited number of transactions depend on it.

Different from CC and TsDefer, transaction partitioners and TsPar can readily incorporate transaction dependencies by enforcing dependencies in partitions and during scheduling.

(3) Generalization. In principal TsPar is not limited to the in-memory setting; it can be applied to shared-nothing distributed systems. In contrast, TsDefer cannot be trivially generalized as it relies on lightweight probing operations to detect runtime conflict at execution time; such operations will incur too much overhead in the shared-nothing architecture due to network latency involved.

4 Transaction Scheduling

In this section, we formulate the transaction scheduling problem, settle its complexity bound, and develop an efficient algorithm for it. The results serve as the foundation of the TsPar module.

Problem & complexity. The key to TsPar is to compute transaction schedules for transaction partitioners. Referred to as the *transaction scheduling problem*, this is abstracted as follows:

- Input: A transaction workload $\mathcal{W}$, number k of threads, and a partitioning of $\mathcal{W}$, *i.e.,* $(P_1, \ldots, P_k, \mathcal{R})$.
- Output: A transaction schedule $S = (f, \prec)$.
- Objective: For the k RC-free queues $(Q_1, \ldots, Q_k)$ and residual set $\mathcal{R}_s$ generated by S, to
  (a) minimize the *makespan* of the k RC-free queues; and
  (b) minimize the number of unscheduled transactions in $\mathcal{R}_s$.

Here (a) the makespan of the k RC-free queues is their concurrent execution time, calculated as the maximum of the serial execution time among all k queues. We (b) minimize the amount of work in the residual, and hence maximize work that is handled via RC-free queues. With both (a) and (b), we aim to find a schedule that mini-

mizes the total execution time of all the transactions in workload $\mathcal{W}$ and hence, improve the throughput for executing $\mathcal{W}$.

As opposed to transaction partitioning problems [14, 34, 38], this bi-criteria optimization problem is more challenging in that it considers runtime conflicts and execution time.

Theorem 1: *The transaction scheduling problem is* NP*-complete. It is already* NP*-hard to decide whether there exists a schedule such that*

(a) it has k non-empty RC-free queues; or

(b) the makespan of the schedule is no larger than a given number, even when $k = 3$.

The lower bounds hold even when all transactions take a unit time. □

Proof sketch: We give an NP algorithm that guesses a schedule for $\mathcal{W}$ over k cores, and checks in PTIME whether the schedule satisfies the conditions. We prove the NP-hardness of (a) by reduction from the maximum independent set problem [32], and (b) by reduction from the bounded independent sets problem [27]. □

Nonetheless, below we develop an efficient algorithm, denoted by TSgen, to compute transaction schedules for TsPar. Algorithm TSgen works in two settings. Given a workload $\mathcal{W}$ of transactions, (1) it can take as input a transaction partition plan and turns it into a schedule for $\mathcal{W}$, as shown in Algorithm 1. (2) It can also compute a schedule for $\mathcal{W}$ starting from scratch. Below we first present TSgen for case (1), and then show how it handles case (2).

From partitions to schedules. Given a workload $\mathcal{W}$ and a partition plan $f_c = (P_1, \ldots, P_k, \mathcal{R})$ of $\mathcal{W}$, TSgen refines f_c into a schedule $(f, \prec)$ for $\mathcal{W}$. It preserves transactions of cluster P_i in queue Q_i, makes as many residual transactions in $\mathcal{R}$ to be RC-free as possible, and balances the workloads of RC-free queues $(Q_1, \ldots, Q_k)$.

TSgen (Algorithm 1) starts with empty RC-free queues Q_i and empty set $\mathcal{R}_s$. It examines residual transactions in $\mathcal{R}$ and decides whether to merge them into the RC-free queues of f and if so, how to do it. In the process, it schedules transactions in P_j's and preserves their assignment by f_c, *i.e.,* a transaction in P_j is added to queue Q_j at thread j. Along the way, TSgen generates RC-free queue P_j. The unscheduled residual transactions in $\mathcal{R}$ of f_c remain in $\mathcal{R}_s$.

More specifically, algorithm TSgen works as follows. Initially, the set $\mathcal{R}_s$ and RC-free queues Q_i are empty for all $i \in [1, k]$ (lines 1-2). TSgen iteratively expands Q_i by examining transactions in $\mathcal{R}$ one by one, following an ordering $\vec{R}$ of $\mathcal{R}$ (lines 4-14; by default, TSgen picks a random ordering of $\mathcal{R}$ as $\vec{R}$ for simplicity). For each transaction T_* in $\mathcal{R}$ (line 5), it checks whether it can be merged into the input CC-free partition P_l with the smallest total execution time (line 6), in order to balance the workload of RC-free queues.

To do this, it first finds all transactions in P_j ($j \in [1, k], j \neq l$) that are in conflict with T_*, appends them to the corresponding RC-free queue Q_j, and removes them from P_j (lines 7-9). It then checks whether appending T_* to RC-free queue Q_l would cause runtime conflict with transactions that are already in queue Q_j ($j \neq l$) via procedure ckRCF (omitted). If not, it appends T_* to Q_l, and adjusts the load (len_l) of thread l by including $\mathsf{time}(T_*)$ (lines 10-11). Otherwise, TSgen decides that T_* is a residual transaction that cannot be scheduled, and moves it into $\mathcal{R}_s$ (line 12).

After all transactions in $\mathcal{R}$ are examined and assigned to either one of the RC-free queues or $\mathcal{R}_s$, TSgen appends the remaining transactions in each partition P_i to the corresponding RC-free queue Q_i (lines 13-14). It returns the k RC-free queues $(Q_1, \ldots, Q_k)$, and the set $\mathcal{R}_s$ of unscheduled residual transactions (line 15)

ALGORITHM 1: Algorithm TSgen

Input: Transaction workload $\mathcal{W}$, a partition $f_c = (P_1, \ldots, P_k, \mathcal{R})$ of $\mathcal{W}$ over k threads.
Output: A transaction schedule $(f, \prec)$ that refines f_c for $\mathcal{W}$ into two k queues $(Q_1, \ldots, Q_k)$ and a set $\mathcal{R}_s$.

1 $\mathcal{R}_s \leftarrow \emptyset$;
2 **foreach** i in $[1, k]$ **do** $\mathsf{len}_i \leftarrow \sum_{T \in P_i} \mathsf{time}(T)$; $Q_i \leftarrow \emptyset$;
/* denote by G_c the conflict graph of transactions in $\mathcal{W}$ */
3 **while** $\vec{R} \neq \emptyset$ **do**
4 $T_* \leftarrow \vec{R}.\mathsf{pop}()$;
5 $l \leftarrow \arg\min_{i \in [1,k]} \mathsf{len}_i$; // pick the RC-free queue with the least load
6 **foreach** i in $[1, k]$ **do**
7 $S_i \leftarrow \{T \in P_i \mid (T, T_*)$ is an edge in $G_c\}$; // S_i: transactions in P_i that are conflict with T_* in P_i
8 append transactions in S_i to Q_i; $P_i \leftarrow P_i \setminus S_i$;
9 **if** ckRCF$(Q_1, \ldots, Q_{l-1}, Q_l \cup \{T_*\}, \ldots, Q_k) = \mathit{true}$ **then**
10 append T_* to Q_l; $\mathsf{len}_l \leftarrow \mathsf{len}_l + \mathsf{time}(T_*)$;
11 **else** $\mathcal{R}_s \leftarrow \mathcal{R}_s \cup \{T_*\}$;
12 **foreach** $i \in [1, k]$ **do**
13 **if** $P_i \neq \emptyset$ **then** append transactions in P_i to Q_i;
14 **return** $Q_1, \ldots, Q_k$ and $\mathcal{R}_s$;

Example 4: Given the partition of $\mathcal{W}_0$ in Example 1, *i.e.,* $P_1 = \{T_1, T_2, T_3\}$, $P_2 = \{T_4\}$ and residual $\mathcal{R} = \{T_5\}$, algorithm TSgen assigns T_5 of $\mathcal{R}$ to P_2, which turns to RC-free queue $Q_2 = \langle T_4, T_5 \rangle$, with $Q_1 = \langle T_2, T_1, T_3 \rangle$, exactly the same schedule as in Example 3. □

To complete TSgen, we next present how it identifies transactions that conflict with T_* from each input CC-free partition P_i (lines 7-9).

More specifically, to identify transactions in each input partition P_i that are in conflict with T_*, TSgen makes use of the conflict graph G_c of $\mathcal{W}$. Here G_c is an undirected graph in which (i) the nodes are the transactions of $\mathcal{W}$, and (ii) two transactions are connected by an edge if they are in conflict with each other. Note that variants of conflict graph have already been used in transaction partitioners (*e.g.,* [14, 59]), and are found efficient to construct and effective in practice; TSgen re-uses their conflict graphs when looking up conflicts. TSgen identifies transactions that are in conflict with T_* by searching the neighbor nodes of T_* in G_c, and checking whether they are in the input CC-free partitions P_i (line 8). It appends such conflicting transactions in P_i to RC-free queue Q_i (line 9), and checks whether T_* incurs runtime conflicts with the expanded Q_i (ckRCF; line 10). It appends T_* to Q_l if there is no runtime conflict.

As will be seen in Section 6, TSgen improves the throughput by 131% on average, up to 294%; the overhead it adds to transaction partitioners is less than 5% of that of the partitioners it optimizes.

Scheduling without input partition. Algorithm TSgen is also able to compute a schedule for workload $\mathcal{W}$ in the absence of a partition plan f_c. More specifically, given a transaction workload $\mathcal{W}$, we simply treat $\mathcal{W}$ as the residual R and runs TSgen with empty CC-free partitions, *i.e.,* $P_1 = \cdots = P_k = \emptyset$. TSgen works in the same way as how it refines a non-empty partitioning of $\mathcal{W}$.

Complexity. Algorithm TSgen can be implemented in $O(|\mathcal{W}| + (k-1)|\mathcal{R}|)$-time, where $|\mathcal{W}|$ (resp. $|\mathcal{R}|$) is the number of transactions in $\mathcal{W}$ (residual $\mathcal{R}$) and k is the number of threads, by reusing the conflict graph G_c from partitioners. That is, TSgen is linear in $|\mathcal{W}|$ when the number k of threads is a constant. Indeed, (a) each transaction in the CC-free partitions P_i is examined only once to decide its assignment. (b) Each transaction in $\mathcal{R}$ is examined by ckRCF in $O(k)$-time.

Remark. TSgen looks up conflicts between transactions from the conflict graph G_c of $\mathcal{W}$. However, it does not necessarily need to construct G_c during scheduling. Typically transaction partitioners already build G_c [14] or its variants [34] for partitioning in order to minimize cross-partition conflicts, and TSgen re-uses them from the partitioners. TSgen aims to strike a balance between the scheduling cost added to the partitioners and the quality of the schedules computed. It reduces the execution cost of residual queues by preserving conflict-free transactions of partitions P_i's in the RC-free queues and moving residual transactions of $\mathcal{R}$ to RC-free queues.

5 Proactive Transaction Deferment

In this section, we present how TsDefer reduces runtime conflicts during execution time for unscheduled residual $\mathcal{R}_s$ from TsPar and for unbundled transactions that are directly assigned to threads.

TsDefer acts directly on thread-local buffers and reduces runtime conflicts for CC by proactively deferring transactions that are likely to cause runtime conflicts with other transactions that are being executed. It is to reduce conflict panelties. Nonetheless, unlike transaction partitioners and TsPar, TsDefer inflicts extra overhead to the CC protocols for execution. Hence, it needs to be extremely lightweight while being effective since otherwise its overhead would cancel out any benefit from reduced conflicts.

This imposes challenges to the design of TsDefer. In particular, tracking and sharing execution progress among threads are essentially contended operations, while locking is not an option due to its large overhead on execution. Moreover, a thread cannot afford to simply look at all concurrent transactions to decide runtime conflicts due to the overhead of reading their read/write sets.

To tackle the challenges, TsDefer proposes two techniques.

(a) It uses a *lock-free* structure for all threads to keep track of their execution progress and look up the progress of other threads for conflict detection. This avoids the locking overhead and data races among threads on tracking and sharing progress.

(b) To suppress the overhead of checking conflicts among transactions on-the-fly, TsDefer randomly probes data items of the read/write sets of active transaction at other threads within a limited times, and defers the transaction with a certain probability if conflicting items are found. The crux is that each data item probe takes only a constant time in a lock-free manner, independent of both the size of transactions and the number of threads.

Runtime progress tracking. As shown in Fig. 3, TsDefer represents the transaction queue in each thread-local buffer as an array of transaction IDs; the buffer is shared globally across all threads. Each thread uses two pointers, `headp` and `tailp`, that point to the next transaction to be executed and the end of the transaction list, respectively. Both the array and the pointers are writable by its own thread and are read-only for other threads.

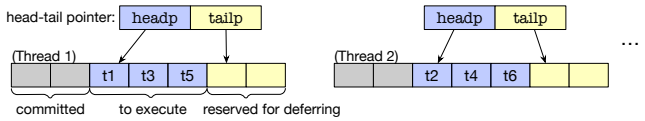


Figure 3: Structure for transaction progress tracking: *grey grids are committed transactions; blue grids represent transactions about to execute; yellow grids reserve slots for deferred transactions.*

TsDefer implements three operations for each thread i:

- regPos to update the progress of thread i by moving `headp` to the next transaction ID once the current one commits; and
- lookup that, upon each invocation, returns in a *constant* time a data item in the write set of some *active transaction* of another thread, where a transaction is active for thread i if it (a) is at thread j ($j \neq i$) and (b) is pointed by `headp` of thread j.
- defer that moves a transaction to the end of the queue that maintains the deferred transactions of the current thread.

Proactive deferment. With these, TsDefer works as follows.

(1) When thread i is about to execute a new transaction T in its local buffer $\mathcal{T}_i$, it checks whether T would cause a runtime conflict with active transactions at other threads by invoking a bounded `#lookups` number of lookup operations.

(2) Let d be the number of distinct data items retrieved from lookup. If `#lookups` $- d$ is above a threshold (typically 1), TsDefer decides that T is likely to cause a runtime conflict. In such cases, thread i defers T via the defer operation with a probability of `deferp%`.

(3) If T is deferred, thread i then moves to the next transaction and updates its progress via regPos. In addition, it records the deferred T at `tailp`, and then moves `tailp` to the next slot.

(4) If T is not deferred, thread i then goes on to execute T. If T subsequently commits successfully, then thread i updates its progress via regPos. If T aborts, thread i will retry T immediately until it commits. During the period, it does not update its progress.

Lock-free Implementation. We next describe the implementation of the progress tracking structure. The key idea is to keep the overhead low and controllable; the rationale is that TsDefer does not replace CC, *i.e.,* it does not detect all runtime conflicts during execution time. In light of this, it implements both operations regPos and lookup with C++ atomic builtins [1] in a lock-free manner. Note that lookup may read slightly stale progress due to the lock-free design. However, we find that such staleness has negligible implication as TsDefer is supposed to be lightweight and does not aim to identify all potential runtime conflicts. For long-run transactions of which conflicts are costly, one can compensate this by instructing lookup to check transactions that are further in the future *w.r.t.* the one it sees from `headp`, within bounded steps.

To ensure each lookup has a constant complexity, each thread maintains the predicted access sets of all other transactions. This does not have to be exact since some of the read/write items are not possibly known without execution; for such cases, TsDefer only records the estimated access sets based on the constants and template of the transaction. For instance, the warehouse id (w_id), district id (d_id) and customer id (c_id) of an instantiated Payment transaction in TPC-C could largely determine its access sets; sim-

ilarly, the access sets of YCSB transactions can also be accurately inferred by the instantiated parameters (keys) in most cases.

Each invocation of lookup then randomly takes a thread ID j and index p, via reservoir sampling; it then returns the *p-th* item in the write set of the transaction that is currently pointed by `headp` of thread j. Note that each invocation requires only one read to the global structure, to retrieve the transaction ID determined by j, say T_*. Then lookup simply reads the data item of T_* indexed by p in its local copy of the read/write set of T_*.

Example 5: Recall thread-local buffers $\mathcal{T}_1 = \langle T_1, T_2, T_3 \rangle$ and $\mathcal{T}_2 = \langle T_4, T_5 \rangle$ from Example 2. Assume `deferp%` = 100%. With TsDefer, thread 1 can defer T_2 with a probability of 50% with 1 lookup (*i.e.,* `#lookups` = 1), and defer T_2 for certain with only 2 lookup (*i.e.,* `#lookups` = 2). Indeed, for thread 1 and T_2, the only active transaction is T_5, which has two data items, x_1 and x_5. Hence one lookup has 50% of chance of returning x_1, which witnesses a conflict with T_2, while two lookup calls return x_1 for certain, triggering TsDefer to defer T_2. Note that reading T_5 alone would already cost 6 reads of data items, already a higher overhead than the deferment of T_2. □

Parameters and trade-off. Observe that the accuracy and overhead of TsDefer is parameterized by the following two parameters:

- `#lookups`: the number of lookup operations invoked; and
- `deferp%`: the probability to deferring a candidate.

The overhead of TsDefer is determined by the number of lookup operations, *i.e.,* `#lookups`. With larger `#lookups`, however, more potentially contended operations in remote threads can be discovered; hence, the chance of *not* deferring a transaction that is in runtime conflict with other active transactions becomes lower. This helps us reduce the CC cost (abort/retry penalty) via reduced conflicts among transactions being executed. Probability `deferp%` allows TsDefer to adapt to varying contention levels: for extremely high contention workloads, TsDefer uses a relatively lower `deferp%` to avoid excessive number of transactions being deferred.

With these tunable parameters, TsDefer can adapt to transaction workloads with varying characteristics, by trading deferring overhead for the accuracy of conflict reduction for CC. When conflict penalties (abort/retry cost) make a larger factor, *e.g.,* for long-run transactions or transactions with heavy application-level abort penalties, one may prefer larger `#lookups` for a higher chance of identifying a runtime conflict, to avoid a costly abort/retry with a bit higher overhead. On the contrary, for light and simple transactions, *e.g.,* key-access over a key-value table as YCSB transactions do, one may prefer smaller `#lookups` since abort/retry is cheap and overhead is more sensitive. In the extreme case, one can disable TsDefer with `#lookups` = 0, avoiding any overhead to CC at all.

Such flexible trade-off between overhead and conflict penalty reduction is in particular viable for transactions that have complex application logic and hence have varying costs of aborts/retries.

6 Experimental Study

Using benchmarks, we evaluated the effectiveness of TSkd in improving both partitioning-based systems for bundled transaction workloads (Section 6.2) and CC-based systems for unbundled transactions (Section 6.3), in particular for transactions with varying runtime skewness. We specify the experimental settings in Section 6.1 and summarize the evaluation results in Section 6.4.

6.1 Implementation and Experimental Setup

Systems. We have implemented a prototype of TSkd as described in Sections 3-5, and integrated it with DBx1000 [2, 56] as the transaction execution engine, which implements multiple CC protocols. Since DBx1000 directly initializes thread-local buffers with instantiated unbundled transactions, to evaluate the effectiveness of TSkd for partitioning-based systems, we extend it by also porting external transaction partitioners into DBx1000, making it a full testbed for both partitioning-based transaction processing and non-partitioning CC-based transaction processing methods.

This is done by initializing the transaction buffer for each thread in DBx1000 with a partition generated by the partitioner, so that transactions are executed according to the partitioning. When TSkd is enabled for transaction partitioners, each thread local buffer receives the transaction queue generated by the partitioner and TSkd.

TSkd instances. We deployed five instances of TSkd, depending on whether it partitions transactions and what partitioner it employs:

- TSkd[`S`]: an instance of TSkd that targets systems with partitioning; it employs Strife [34], a recent partitioner that has been shown effective for highly contended transaction workloads [34]. We used its open source implementation from [3].
- TSkd[`C`]: an instance of TSkd that uses Schism [14] partitioner.
- TSkd[`H`]: TSkd with Horticulture [33] as the partitioner.
- TSkd[`0`]: TSkd without input partitions, *i.e.,* taking all transactions as the residual.
- TSkd[`CC`]: an instance of TSkd targeting unbundled transactions that are directly handled by DBx1000's default transaction-to-thread assignment with CC; it only uses TsDefer.

Among the three partitioners, Strife generates partitioning of transactions with an explicit residual set, while Schism and Horticulture do not. For the partitioning of Schism and Horticulture, TSkd first extracts a residual set that contains all those transactions that are in conflict with some other transactions from another partition, and then carries out the scheduling as it does with Strife.

Baselines. We compared the performance of the scheduling-based TSkd[`S`], TSkd[`C`] and TSkd[`H`] with their partitioning counterparts Strife [34], Schism [14] and Horticulture [33], respectively, for bundled transactions that are known to the partitioners prior to execution. Both Schism [14] and Strife [34] are general partitioners that work with any given transaction workloads with known read/write sets. Horticulture is hard-coded for TPC-C [48] and YCSB [12] workloads, and is not a full-fledged partitioner.

To study the effectiveness of TSkd (TsDefer) for unbundled transactions that are not known beforehand, we also compared TSkd[`CC`] with DBx1000's default configuration that executes transactions using CC without partitioning (denoted by DbCC).

Configuration. For all TSkd instances, all the transactions are executed with CC to guarantee correctness. This is a suboptimal implementation of TSkd and is in favor of the baselines since one can retain the lower cost of CC-free execution of the RC-free queues by enforcing the scheduled order via, *e.g.,* dependency tracking in

Parameter	Definition	Range	Default
c%	TPC-C cross-warehouse transaction(%)	[15%, 35%]	25%
#whn	# of warehouses in TPC-C	[20, 60]	40
θ	Zipfian skewness parameter in YCSB	[0.7, 0.9]	0.8
#core	# of threads for execution	[8, 32]	20
CC	CC protocols for execution	OCC, SILO, TICTOC	OCC
minT	minT · t_T = minimum transaction runtime lower bound, where t_T is the average transaction runtime	[1/8, 1]	1/2
p	p · minT = maximum runtime lower bound	[32, 64]	48
θ_T	the skewness parameter of the Zipfian distribution for runtime lower bounds	[0.7, 0.9]	0.8
l_{IO}	l_{IO} = max latency/min latency, where min latency is 5000 CPU cycles, about 1/6 (1/8) of the average TPC-C (YCSB) transaction runtime.	[0, 100]	50[1]
θ_{IO}	Zipfian parameter of I/O latency distribution; larger θ_{IO} means *longer tail* latency	[0.8, 1.6]	1.2
#lookups	# of lookup operations for TsPar	[1, 5]	2
deferp%	the probability to defer a candidate	[0.4, 0.8]	0.6

[1] by default we disabled I/O latency; we set $l_{\text{IO}} = 50$ for tests about I/O latency when varying θ_{IO}.

Table 1: Workload and system parameters: *blue/yellow for* TPC-C/YCSB *parameters; green for system parameters; red for runtime skeet's and I/O latency; and gray for* TsDefer *parameters. When varying a parameter, we use the default for all the other parameters.*

[35, 36]. TsPar uses the warm-up dry-run trails of DBx1000 as the source of histories to derive coarse cost estimates for transactions. The isolation level of all tests is set to serializability.

Benchmarks. We used two transaction benchmarks.

TPC-C [48]. We tested with full TPC-C transactions. Since the built-in TPC-C implementation in DBx1000 contains only the NewOrder and Payment transactions without insertion (update only), we extended it to cover full TPC-C. In particular, we enabled insertions in NewOrder and Payment and added OrderStatus, StockLevel and Delivery transactions to DBx1000 by following [6].

To evaluate the impact of contention levels, we also enabled TPC-C to vary the originally hard-coded percentage (c%) of transactions that cross multiple warehouses. We set c% to 25% and #whn/#core to 2 when varying the number of cores (#core), to simulate workloads with high contention, where #whn is the number of warehouses.

YCSB [12]. We also tested with the YCSB benchmark. We used the built-in YCSB driver in DBx1000, which implements the YCSB core A workload [55]. We used a single table of 20M records, where each record is 128 bytes in size and is accessed by a unique key; each transaction accesses 16 records. Contention in YCSB is configured by a Zipfian distribution that controls data skewness; we varied the θ parameter of Zipfian from 0.7 to 0.9 ($\theta = 0.8$ by default).

Extension with runtime skewness. Both TPC-C and YCSB transactions are short, *e.g.,* a YCSB transaction consists of key accesses in a key-value table. In practice transactions may have varying lengths (*i.e.,* runtime) and skewness in their distribution. To test more complicated transactions, we extend both TPC-C and YCSB by "lower bounding" the runtime of transactions: assuming that a transaction T is lower bounded by t_{min}, if the actual execution time of T exceeds t_{min}, then nothing changes and T commits as usual; otherwise, it delays its committing until the total runtime is t_{min}.

More specifically, we extend TPC-C and YCSB with three parameters minT, p and θ_T to emulate runtime skewness. Transactions are assigned with a minimum runtime *randomly* drawn from a range [minT·t_T, p · minT·t_T], following a Zipfian distribution with the skewness parameter θ_T. Here t_T is the average transaction runtime and minT (≤ 1) is a small coefficient such that minT · t_T serves as the "unit" time of transaction execution, and integer p constrains the maximum "lower bound" runtime. These lower bounds follow a Zipfian distribution with skewness degree varied by θ_T. By varying minT, p and θ_T, we emulate different runtime patterns of the transactions. We set minT small, *e.g.,* as low as 1/8 so that the original short-run TPC-C and YCSB transactions are included as a subclass of the workloads for all the tests. These parameters improve the expressiveness of TPC-C and YCSB, not to restrict the benchmarks.

Extension with I/O latency. To evaluate the impact of I/O latency, we further extended DBx1000 with a new knob similar to [47], to add an artificial delay to simulate I/O latency at transaction commit time. The delays draw values from [0, l_{IO} · minIO] for a Zipfian distribution with skewness parameter θ_{IO}, where (a) minIO is set to 5000 CPU cycles, about 1/6 (resp. 1/8) of the average TPC-C (resp. YCSB) transaction runtime, and (b) l_{IO} varies from [0, 100]. By varying l_{IO} and θ_{IO}, we get various patterns of I/O latency to DBx1000. In particular, larger l_{IO} means longer worst-case I/O latency and higher θ_{IO} indicates a "longer-tail" latency distribution.

Metrics. We measure the performance of systems by the following:

- *throughput*: the number of transactions committed per second;
- #retry: the total number of retries per 100,000 transactions.

Experimental setup. The experiments were run on an AWS EC2 m5.8xlarge instance, with 32 vCPU and 128 GB of memory. Each experiment was run 3 times. The average is reported here.

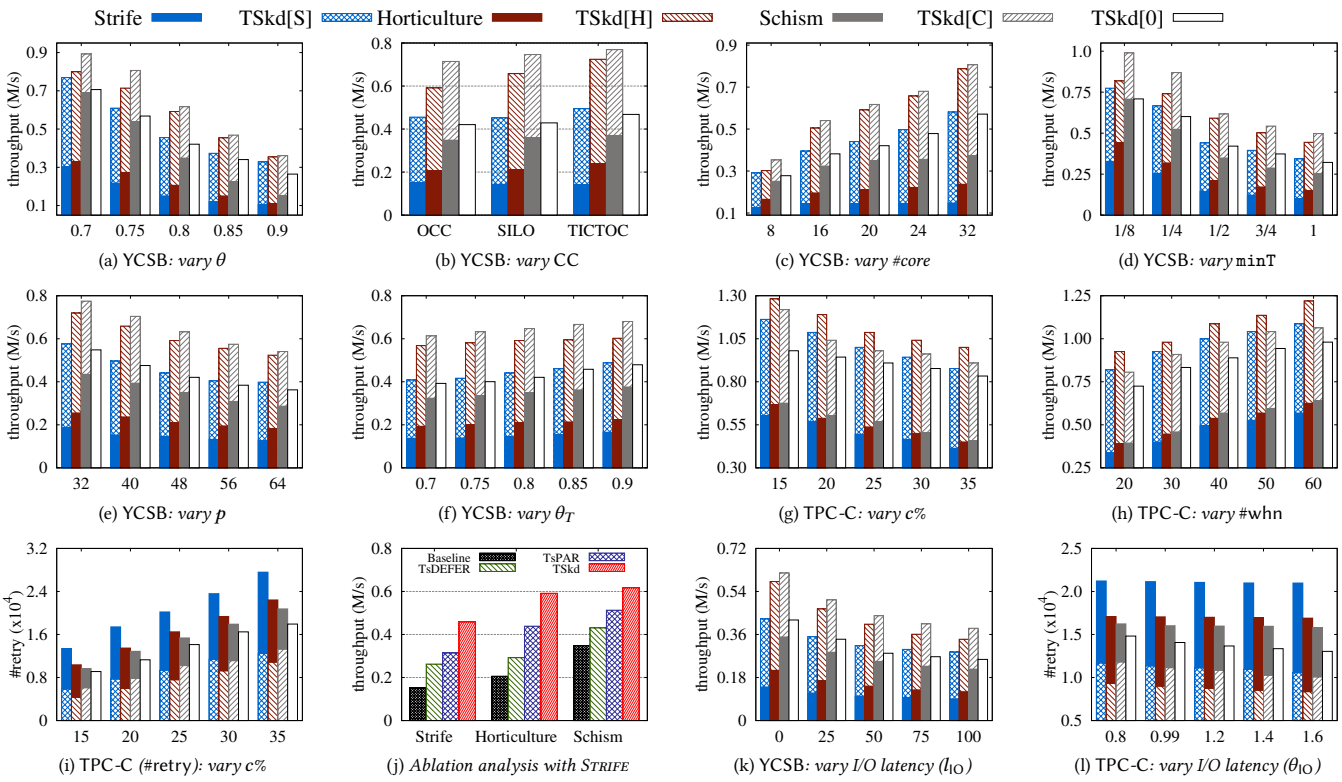
All the parameters, including their configuration ranges and default settings, are summarized in Table 1. When varying a parameter, we use the default configuration of all other parameters. By default, each bundle consists of 10,000 transactions.

6.2 TSkd on Partitioning-based Systems

We first evaluated the effectiveness of transaction scheduling in improving the throughput and #retry of partition-based systems. We compared the performance of TSkd[S], TSkd[C] and TSkd[H] with Strife, Schism and Horticulture, respectively, to find out improvement over transaction partitioners introduced by TSkd via scheduling. Varying each parameter, we tested the throughput and #retry of all methods on TPC-C, YCSB and their skewed extensions.

Throughput: scheduling vs. partitioning. We first compared the transaction throughput of schedules computed by TSkd[S], TSkd[C] and TSkd[H] with that of partitioning generated by Strife, Schism and Horticulture, respectively, with varying parameters. The results over YCSB and TPC-C are shown in Figures 4a-4h.

(1) Overall, TSkd consistently improves the throughput of Strife, Schism and Horticulture. For instance, on average TSkd[S], TSkd[C] and TSkd[H] improve the throughput of Strife, Schism and Horticulture by 211%, 78.9% and 184% over YCSB, respectively, up to 294%, 119% and 225%. Over TPC-C on average the throughput of TSkd[S], TSkd[C] and TSkd[H] is 133%, 75.2%



(a) YCSB: *vary* θ (b) YCSB: *vary* CC (c) YCSB: *vary #core* (d) YCSB: *vary* minT

(e) YCSB: *vary* p (f) YCSB: *vary* θ_T (g) TPC-C: *vary c%* (h) TPC-C: *vary* #whn

(i) TPC-C (#retry): *vary c%* (j) *Ablation analysis with* Strife (k) YCSB: *vary I/O latency* (l_{IO}) (l) TPC-C: *vary I/O latency* (θ_{IO})

Figure 4: Effectiveness of TSkd (TsPar) on Partitioning-based Systems (Section 6.2)

and 101% higher than that of Strife, Schism and Horticulture, respectively, up to 183%, 104% and 141%. The reason is two-fold.

(a) Higher concurrency. By scheduling (TsPar), TSkd achieves better balanced load among the cores, yielding a higher level of concurrency. In contrast, existing partitioners can be sub-optimal in balancing the load, *e.g.,* the average ratio of the largest partition over the smallest partition of Strife is 3.2 over YCSB, while this reduces to 1.2 after deploying TsPar atop Strife. Moreover, by executing $\mathcal{R}_s$ with CC and proactive transaction deferment (TsDefer), TSkd further balances the workloads across threads.

(b) Reduced #retry. Furthermore, by combining TsPar and TsDefer together, such higher level of concurrency does not incur larger CC costs due to the ability of TSkd to reduce runtime conflicts. Indeed, in contrast to the normal pattern that higher concurrency means larger #retry, the #retry of all systems even decreases with TSkd. For instance, the #retry of TSkd[S], TSkd[C] and TSkd[H] is consistently lower than that of Strife, Schism and Horticulture, in all cases (*e.g.,* Fig. 4i). On average, TSkd reduces 49.7%, 43.6% and 33.6% of the #retry of that of Strife, Schism and Horticulture on YCSB, respectively, and 54.4%, 36.7% and 53.9% over TPC-C.

(2) The effectiveness of TSkd is robust *w.r.t.* CC protocols; it is even more evident with added cores as shown in Figures 4b-4c. For instance, over YCSB, TSkd[S] consistently achieves over 203% higher throughput than Strife with either OCC, SILO or TICTOC. The gap even increases with lager #core, *e.g.,* TSkd[H] improves Strife by 133% with 8 cores, and 294% with 32 cores.

(3) Overall, TSkd is more effective for workloads with higher contention, *e.g.,* the throughput improvement of TSkd[H] over Horticulture increases from 143% to 225% when θ varies from 0.7 to 0.9 over YCSB, as shown in Fig. 4a; similarly, when c% increases from 15% to 35% for TPC-C, the throughput improvement of TSkd[C] over Schism increases from 80.5% to 98.2% as depicted in Fig. 4g.

(4) Without an input partitioning, TSkd[0] still achieves throughput on average 85.8%, 65.1% and 69.7% higher than Strife, Schism and Horticulture over TPC-C, respectively, and 184%, 29.1% and 101% higher over YCSB, as shown in Fig. 4. Compared to partitioners, TSkd[0] achieves better load balancing via scheduling, by treating all transactions as residual; in addition, it has low retries due to reduced conflicts. This suggests that TSkd[0] is an option for non-partitioned workloads or partitioners that do not produce a residual.

(5) To analyze the individual contribution of TsPar and TsDefer to the effectiveness of TSkd, we tested TSkd[S] against TsPar[S] and TsDefer[S] where TsPar[S] is TSkd on Strife with TsDefer disabled, and TsDefer[S] is TSkd with only TsDefer enabled to execute partitions of Strife; similarly for Schism and Horticulture. As shown in Fig. 4j, over YCSB, on average TSkd[S], TsPar[S] and TsDefer[S] improve the throughput of Strife by 203%, 108%, 73.2%, respectively; similarly for Schism and Horticulture and over YCSB. This shows that for bundled workloads, TsPar plays a bigger role in improving the performance. Moreover, TsPar and TsDefer perform the best when working together, *e.g.,* even better than the sum of the improvement by the two separately on Strife.

Runtime skewness. TSkd improves the robustness of all systems

	system	scheduled pct. s%	#retry/10k transactions	
			w/o TsDefer	w/ TsDefer
YCSB	TSkd[S]	64.5	1353	448
	TSkd[C]	52.7	2786	1029
	TSkd[H]	69.7	1614	611
TPC-C	TSkd[S]	34.6	542	255
	TSkd[C]	36.7	803	474
	TSkd[H]	20.8	416	253

Table 2: Accuracy of scheduling and effectiveness of TsDefer

for transactions with varying skewness, as shown in Figures 4d-4f over YCSB (results over TPC-C omitted due to space limit). In particular, the improvement in throughput by TSkd[S], TSkd[C] and TSkd[H] over Strife, Schism and Horticulture, respectively, even increases when transactions become longer (*i.e.,* larger `minT`) or more variable (*i.e.,* larger p and smaller θ_T). For example, on YCSB, TSkd[S] improves Strife in throughput by 136% with `minT` = 1/8 while this increases to 240% with `minT` = 1; similarly TSkd[H] improves Horticulture on YCSB by 171% with p set to 32, while this increases to 186% with $p = 64$. This is because the quality of partitions degrades with transactions of varying lengths, while with TSkd they can adapt to the variance and skewness in transaction runtime. Moreover, longer transactions inflict larger conflict penalties, and TsPar is more effective in reducing their runtime conflicts.

I/O latency. We next evaluated the impact of I/O latency. Varying l_{IO} and θ_{IO}, we tested the throughput of all methods; partial results over YCSB and TPC-C are shown in Figures 4k-4l. We find that TSkd still improves all three partitioners in all cases. Although the raw throughput degrades for all methods with larger l_{IO} or smaller θ_{IO} (*i.e.,* not so long-tail), the improvement brought by TSkd is relatively stable, *e.g.,* consistently around 205% for Strife over YCSB; similarly for other partitioners and over TPC-C. This is because while latency reduces the accuracy of scheduling by TsPar, it increases the cost of retries; and TSkd consistently reduces the retries of all partitioners (*e.g.,* Fig. 4l) and balances the partitions in the presence of long-tail latency. Hence, the effectiveness of TSkd remains robust against I/O latency. This suggests that TSkd can also work with transactions involving I/O or network latency.

Overhead. We also tested the overhead that TsPar adds to the transaction partitioners, measured as the ratio of the runtime of TSgen to the partitioning time of the partitioners, denoted by overheadR. For workloads consisting of 100,000 transactions, we find that the average overheadR of TsPar over Strife and Schism over YCSB is 4.1% and 3.7%, respectively, and 4.6% and 4.4% over TPC-C. This verifies that TsPar has moderate overhead.

Cost estimation. By default, TsPar uses the warm-up process of DBx1000 to assess the cost of transactions when scheduling (recall Section 3). To assess the accuracy of scheduling with such coarse estimates, we measured (a) the average scheduled percentage (s%) of residual transactions that were merged to RC-free queues; and (b) the #retry when executing the RC-free queues, with and without employing TsDefer. The results are shown in Table 2.

We find that TSkd schedules a decent percentage of residual transactions, *e.g.,* 30.7% and 62.3% over TPC-C and YCSB, respectively. Due to the coarse estimates used by TSkd, it is understandable that RC-free queues incur conflicts and retries. Nonetheless, after employing TsDefer, TSkd significantly reduces #retry for the RC-free transactions. For instance, TSkd[S], TSkd[C] and TSkd[H] reduce 66.9%, 63.1% and 62.2% of #retry for the RC-free transactions over YCSB, and by 53.0%, 41.0% and 39.2% over TPC-C, respectively. As shown earlier, this even makes the #retry of all systems much lower than without scheduling, although they have much higher level of concurrency and better load balancing with TSkd deployed.

Although the percentage of the scheduled transactions over TPC-C is lower than YCSB, the throughput improvement of TSkd over TPC-C is comparable to that of YCSB. This is because with even a moderate number of scheduled residual transactions, TSkd is already able to make TPC-C partitions balanced.

6.3 TSkd on Non-partitioning CC-based Systems

We also evaluated the effectiveness of TSkd (TsDefer) for transaction systems that use CC without partitioning. To do this, we compared the performance of TSkd[CC] with that of DbCC, with varying parameters and CC protocols. The results over YCSB are shown in Figure 5 (TPC-C is similar and omitted due to space limit).

Contention. Varying θ from 0.7 to 0.9 for YCSB and c% from 15% to 35%, we tested the impact of transaction contention level on the effectiveness of TsDefer. We find that TsDefer consistently improves DbCC in both throughput and #retry. As shown in Fig. 5a, on YCSB, on average the throughput of TSkd[CC] is 111% higher than that of DbCC, while the #retry of TSkd[CC] is 49.8% lower. By digging into the profiling statistics, we find that the reduction of retries by TsDefer is largely consistent with that of mutex contention of the execution. In fact, TsDefer reduces #contended_mutex [29], the total number of times that a mutex was contended (already locked when a lock request was made) in DbCC by 53.8% on average.

Moreover, the improvement is even more significant with larger θ. For instance, TsDefer improves the throughput of DbCC by 44.8% and reduces the #retry by 43.9% when θ is 0.7, while these increase to 171% and 53.4%, respectively, with $\theta = 0.9$. The results over TPC-C are similar. This is because for workloads with higher contention, there exist more runtime conflicts, which could benefit from TsDefer with a higher chance. This is also confirmed by the runtime statistics, *e.g.,* TSkd[CC] reduces #contended_mutex of DbCC by 44.5% with $\theta = 0.7$ and this increases to 56.4% with $\theta = 0.9$.

Runtime skewness. Varying parameters `minT`, p and θ_T, we evaluated the effectiveness of TSkd in response to different runtime skewness. As shown in Figures 5d-5f, TsDefer is particularly effective with longer (larger `minT`) and more skewed and variable (larger p and lower θ_T) transactions. For instance, over YCSB when `minT` increases from 1/8 to 1, the throughput improvement of TSkd[CC] over DbCC increases from 34.5% to 119%. In contrast, when θ_T is 0.9, TSkd[CC] reduces #retry of DbCC by 49.5%, and this increases to 53.3% when $\theta_T = 0.7$; similarly over TPC-C.

I/O latency. Varying l_{IO} and θ_{IO}, we evaluated the impact of I/O latency on TsDefer for unbundled transactions. We find that with larger l_{IO} or smaller θ_{IO}, the throughput of both DbCC and TSkd[CC] decreases as transactions are prolonged by the I/O stalls; however, TSkd[CC] remains robust in improving DbCC in both throughput and retries, as shown in Fig. 6. This is because TsDefer is insensitive to transaction lengths via runtime progress tracking; hence its effectiveness is stable *w.r.t.* varying patterns of I/O latency.

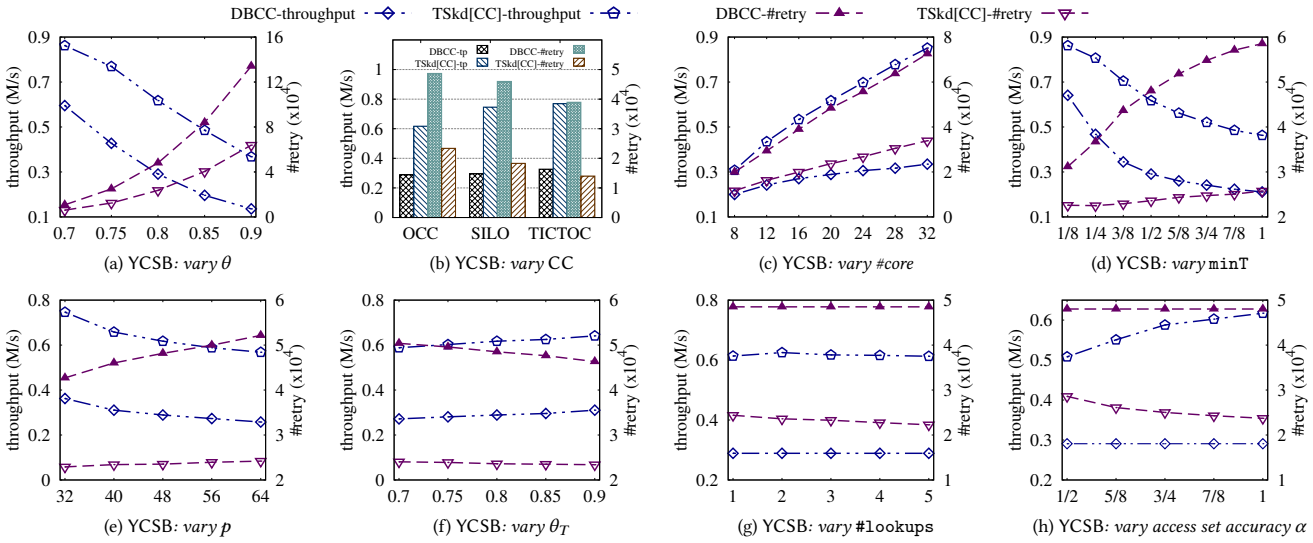


Figure 5: Effectiveness of TSkd (TsDefer) on CC-based Systems over YCSB (Section 6.3): *X-axis for throughput and Y-axis for* #retry.

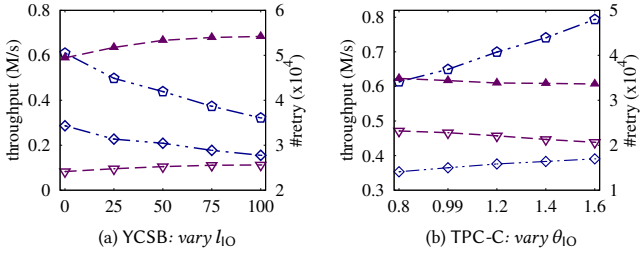


Figure 6: Impact of I/O latency on TsDefer (Section 6.3)

Trade-offs. We next examined the trade-offs between the effectiveness and overhead of TsDefer, by varying `#lookups` and `deferp%`. We find that larger `#lookups` naturally improves the effectiveness of TsDefer in reducing runtime conflicts and retries, *e.g.,* over YCSB the #retry of DbCC is reduced by 49.7% with TSkd enabled when `#lookups` = 1, and this increases to 54.1% with `#lookups` = 5 (as shown in Fig. 5g). However, larger `#lookups` comes with higher overhead, which would work with longer transactions whose runtime and retry cost could cover the overhead of TsDefer. For short-run TPC-C and YCSB transactions, we find that `#lookups` = 2 gives the best throughput, *e.g.,* TSkd[CC] improves TSkd by 116% over YCSB and 105% over TPC-C. Similarly, we find that higher `deferp%` gives better reduction on #retry for TSkd over DbCC.

CC and scalability. Varying CC protocols and the number of cores, we tested the impact of CC and the scalability of both TSkd[CC] and DbCC. The results over YCSB are depicted in Figures 5b-5c. We find that TSkd[CC] consistently improves the throughput and #retry of DbCC via TsDefer, by 117% and 51.8%, respectively, on average. Moreover, the gap even widens with larger #core. TSkd works the best with TICTOC, yielding a throughput improvement of 152% and a reduction of #retry by 63.9% over YCSB. It has the least advantage with OCC (the default CC used in all tests), but still improves DbCC by 116% for throughput and 52.0% for #retry.

Impact of inaccurate access sets. Finally, we evaluated the impact of inaccurately determined transaction read/write sets on the effectiveness of TSkd[CC]. To do this, we restricted TSkd[CC] such that it could only use an α-fraction of the actual access sets, *i.e.,* the determined transaction read/write sets have an accuracy of α. Varying α from [0.5, 1], we tested the performance of TSkd[CC] with DbCC. The results over YCSB are shown in Fig. 5h (the results over TPC-C are similar and thus omitted). We find that TSkd[CC] still improves the throughput of DbCC even when TSkd[CC] overlooked 50% of the actual access set, *i.e.,* when α is as low as 50% over YCSB. This is because TsDefer only randomly probes access sets of concurrent transactions to detect potential conflicts and it needs only part of the access sets; hence, it still observes conflicts and improves DbCC even when it overlooked reads/writes moderately. Naturally, the effectiveness of TSkd[CC] improves when the access sets are more accurately determined with higher α.

6.4 Summary

From the experiments we find the following.

(1) Transaction scheduling (TsPar) is effective in improving the performance of partitioning-based transaction systems, *e.g.,* it improves the throughput of state-of-the-art transaction partitioners by 131% on average, up to 294% over TPC-C and YCSB benchmarks.

(2) The benefit of TSkd (TsPar) is even more evident when a larger number of cores are available. For instance, over YCSB, on average TSkd improves the throughput of partitioners by 75.3% when 8 cores are used; the improvement increases to 214% with 32 cores.

(3) Proactive transaction deferment is effective for CC-based systems. On average it improves the throughput of DbCC by 109% and reduces its #retry by 45.7%, up to 152% and 63.9%, respectively.

(4) With TsDefer, TsPar can handle inaccurate cost estimates of transactions while still improving the throughput of partitioners.

(5) TSkd is particularly effective for transactions with skewed or long runtime or I/O latency. It makes both partition-based and CC-based systems robust against runtime and I/O skewness.

(6) TSkd works well with all CC protocols. Its improvement over partitioning and DbCC (CC-based) is evident over each CC tested.

7 Related Work

We categorize the related work as follows.

Transaction partitioning. There has been a host of work on transaction partitioning for parallel OLTP systems [14, 33, 37, 54, 59]. The methods are designed for distributed systems under a shared-nothing architecture, where the database is partitioned across multiple computing nodes to minimize transactions that access data from multiple partitions since it requires costly distributed CC.

Closer to this work is the study of multi-core transaction processing [16, 31, 34, 38, 45]. Instead of assigning transactions to cores, [31] decomposes transactions into smaller read and write actions, and assigns such actions to threads to minimize contention. Adaptive concurrency control is proposed by [45] for changing workloads, by dynamically clustering data and adopting the optimistic CC (OCC) protocols for each cluster. [16] uses transaction batching and reordering during the validation phase to reduce the retries of OCC-based protocols. [38] develops Orthrus, a multicore transaction system that separates transaction logic and concurrency control, and employs a set of dedicated cores for the latter; the deterministic approach of [39] pre-processes transaction workloads via partitioning. In particular, [34] proposes Strife, a transaction partitioner to dynamically cluster finer-grained batches of contended workloads that do not have a good static partition.

Different from the partitioning strategies, we propose to schedule transactions of possibly varying execution times; transaction scheduling not only partitions transactions but also sorts them in each partition. This is based on runtime conflicts, which characterize contention among transactions at a finer-grained granularity than traditional transaction conflicts, and allow conventionally conflicting transactions to be executed without conflicts. We also develop an algorithm that can refine existing transaction partitioning and make it a transaction schedule while minimizing runtime conflicts. In addition, we develop a strategy that proactively detects and defers transactions that are highly likely to cause runtime conflicts on-the-fly, to reduce aborts and retries. Our method is not restricted to OCC.

Transaction assignment for OLTP. Most OLTP systems use random or round-robin like strategies to assign unbundled transactions that are coming and processed on-the-fly [60]. Unlike partitioners, they do not have prior knowledge of the entire workload, and hence assign transactions one by one upon arrival. [41] proposes to use a lightweight ML model that predicts, for an incoming transaction, which thread it should be assigned to in order to have lower abort chance. Our work complements these methods by altering the assignment during execution, by tracking thread progress and proactively deferring transactions that are likely to cause conflicts.

Concurrency control. Concurrency control (CC) provides isolation guarantees for concurrent transaction processing. A variety of CC protocols have been proposed, which mostly fall in two classes: locking-based and timestamp based. Locking-based ones, such as two-phase locking [10, 17] and its variants, are pessimistic in that a transaction accesses a tuple only after it acquires a lock with the required permission. For high-contention transactions, timestamp-based protocols such as OCC [5, 24], multi-version concurrency control (MVCC) [9] and their variants [8, 15, 18, 26, 30, 40, 44, 52, 53, 57, 58] have been verified effective in reducing conflicts and blocking time. Hybrid approaches that combine the two strategies have also been explored [43, 45]. There has also been work on learned CC to further mix CC parameters and specialize to a given workload [50].

While these CC protocols provide controlled concurrent execution of conflicting transactions and ensure isolation guarantees, they do not specify how the transactions should be allocated to the threads and in what order they should be executed; both of these have significant impact on the performance of concurrent transaction execution. Our work aims to bridge the gap by minimizing runtime conflicts via transaction scheduling and proactive transaction deferment, as an alternative to conventional approaches that only cluster transactions or directly invoke CC protocols.

Multi-core in-memory OLTP systems. A number of multi-core in-memory OLTP systems have been developed recently, *e.g.,* TicTok [57], Cicada [26], Foedus [23], ERMIA [22], Silo [49] and Orthrus [38]. These systems focus on the design of new CC protocols and architectural optimizations for OLTP workloads.

Different from these systems, we do not aim to build yet another full-fledged OLTP database system. Instead, we present TSkd as a lightweight tool that can be incorporated into these OLTP database systems to improve their throughput by reducing runtime conflict.

Deterministic approaches. Related is also the work on deterministic databases (see [4] for a survey), which execute bundled transactions via a predetermined order that is typically decided via transaction dependency analysis [13, 19, 25, 35, 36, 38, 47]. They scale well in distributed systems, where the determined ordering ensures transactions to always read consistent data across multiple cache copies.

Different from these, our work studies how to reduce runtime conflicts by proposing transaction scheduling and proactive deferring. It neither imposes restrictions on how CC should work, *e.g.,* deterministically or nondeterministically, nor breaks down transactions. This said, this work and the previous work on deterministic databases complement each other. For instance, their techniques for determining read/write sets and analyzing workloads can be used by transaction partitioners for clustering and assigning transactions. Their dependency-based execution method can be adopted by transaction scheduling when runtime estimates are not reliable.

8 Conclusion

We have shown that by considering runtime conflicts, more transactions can be executed concurrently with low conflict penalties than adopting the conventional notion of conflicts. We have proposed transaction scheduling by placing an ordering on the execution of transactions, instead of transaction partitioning. We have shown that the scheduling problem is NP-complete. This said, we have developed an efficient algorithm to refine a partitioning into a schedule for bundled transaction. Moreover, for unbundled transactions targeted by CC protocols, we have proposed a proactive deferring strategy to reduce conflict penalties. Our experimental study has verified that transaction scheduling and deferment are promising in improving throughput and reducing conflict penalties.

One topic for future work is to develop ML models that decide TSkd parameters specialized for given workloads. Another topic is to combine transaction scheduling with other strategies, *e.g.,* transaction decomposition [31] and dynamic adjustment [45].

References

[1] [n.d.]. Atomic Builtins - Using the GNU Compiler Collection (GCC). https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html.

[2] [n.d.]. DBx1000. `https://github.com/yxymit/DBx1000`.

[3] [n.d.]. Strife. `https://github.com/zhr1201/STRIFE-public`.

[4] Daniel J. Abadi and Jose M. Faleiro. 2018. An overview of deterministic database systems. *Commun. ACM* 61, 9 (2018), 78–88.

[5] Rakesh Agrawal, Michael J. Carey, and Miron Livny. 1987. Concurrency Control Performance Modeling: Alternatives and Implications. *TODS* 12, 4 (1987), 609–654.

[6] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. 2022. The Full Story of 1000 Cores: An Examination of Concurrency Control on Real (ly) Large Multi-Socket Hardware—Measurements, Logs, Plots. *The VLDB Journal* (2022).

[7] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*. 1–10.

[8] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221.

[9] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *TODS* 8, 4 (1983), 465–483.

[10] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. 1979. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Trans. Software Eng.* 5, 3 (1979), 203–216.

[11] Aleksey V. Burdakov, Victoria Proletarskaya, Andrey D. Ploutenko, Oleg Ermakov, and Uriy A. Grigorev. 2020. Predicting SQL Query Execution Time with a Cost Model for Spark Platform. In *IoTBDS*. 279–287.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. 143–154.

[13] James Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *USENIX ATC*. 223–235.

[14] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *PVLDB* 3, 1–2 (2010), 48–57.

[15] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. 2016. Repairing Conflicts among MVCC Transactions. *CoRR* abs/1603.00542 (2016).

[16] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving Optimistic Concurrency Control Through Transaction Batching and Operation Reordering. *PVLDB* 12, 2 (2018), 169–182.

[17] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976), 624–633.

[18] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *PVLDB* 10, 5 (2017), 613–624.

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

[20] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2002. *Database systems - the complete book (international edition)*. Pearson Education.

[21] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*. 603–614.

[22] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD*. 1675–1687.

[23] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *SIGMOD*. 691–706.

[24] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *TODS* 6, 2 (1981), 213–226.

[25] Jialin Li, Ellis Michael, and Dan RK Ports. 2017. Eris: Coordination-free consistent transactions using in-network concurrency control. In *SOSP*. 104–120.

[26] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD*. 21–35.

[27] Zbigniew Lonc. 1991. On complexity of some chain and antichain partition problems. In *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 97–104.

[28] Microsoft. [n.d.]. In-Memory OLTP overview and usage scenarios. https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/overview-and-usage-scenarios?view=sql-server-ver16#usage-scenarios-for-.

[29] Mutrace. [n.d.]. Measuring Lock Contention. http://0pointer.de/blog/projects/mutrace.html.

[30] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*. 677–689.

[31] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-Oriented Transaction Execution. *PVLDB* 3, 1 (2010), 928–939.

[32] Christos H Papadimitriou. 1994. *Computational Complexity*. Addison-Wesley.

[33] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*. 61–72.

[34] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *SIGMOD*. 527–542.

[35] Thamir M. Qadah and Mohammad Sadoghi. 2018. QueCC: A Queue-oriented, Control-free Concurrency Architecture. In *Middleware*. ACM, 13–25.

[36] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *SIGOPS*. 180–194.

[37] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. 2013. SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads. In *EDBT*. 430–441.

[38] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *SIGMOD*. 1583–1598.

[39] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2014. An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. *PVLDB* 7, 10 (2014), 821–832.

[40] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. 2014. Reducing Database Locking Contention Through Multi-version Concurrency. *PVLDB* 7, 13 (2014), 1331–1342.

[41] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. 2019. Scheduling OLTP transactions via learned abort prediction. In *aiDM*. ACM, 1:1–1:8.

[42] Rekha Singhal and Manoj K. Nambiar. 2016. Predicting SQL Query Execution Time for Large Data Volume. In *IDEAS*. 378–385.

[43] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. 2017. Bringing Modular Concurrency Control to the Next Level. In *SIGMOD*. 283–297.

[44] Dixin Tang and Aaron J. Elmore. 2018. Toward Coordination-free and Reconfigurable Mixed Concurrency Control. In *USENIX ATC 18*. USENIX Association, 809–822.

[45] Dixin Tang, Hao Jiang, and Aaron J. Elmore. 2017. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All. In *CIDR*.

[46] Enrico Tedeschi, Tor-Arne S. Nordmo, Dag Johansen, and H*rule*(a)vard D. Johansen. 2019. Predicting Transaction Latency with Deep Learning in Proof-of-Work Blockchains. In *BigData*. IEEE, 4223–4231.

[47] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*. 1–12.

[48] Transaction Processing Performance Council. [n.d.]. TPC-C Benchmark. *http://www.tpc.org/tpcc/*.

[49] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SIGOPS*. 18–32.

[50] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control. In *OSDI*. 198–216.

[51] Gerhard Weikum, Arnd Christian Koenig, Achim Kraiss, and Markus Sinnwell. 1999. Towards self-tuning memory management for data servers. *IEEE Data Eng. Bull.* 22, 2 (1999), 3–11.

[52] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB* 10, 7 (2017), 781–792.

[53] Yingjun Wu, Chee Yong Chan, and Kian-Lee Tan. 2016. Transaction Healing: Scaling Optimistic Concurrency Control on Multicores. In *SIGMOD*. ACM, 1689–1704.

[54] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-performance ACID via modular concurrency control. In *SOSP*. 279–294.

[55] Yahoo! [n.d.]. YCSB Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[56] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *PVLDB* 8, 3 (2014), 209–220.

[57] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD*. 1629–1642.

[58] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-Memory Databases. *PVLDB* 9, 6 (2016), 504–515.

[59] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2020. Chiller: Contention-Centric Transaction Execution and Data Partitioning for Modern Networks. In *SIGMOD*. 511–526.

[60] Tieying Zhang, Anthony Tomasic, Yangjun Sheng, and Andrew Pavlo. 2018. Performance of OLTP via Intelligent Scheduling. In *ICDE*. 1288–1291.