Fast Abort-Freedom for Deterministic Transactions

Chen Chen*, Xingbo Wu†, Wenshao Zhong*, Jakob Eriksson*
*University of Illinois at Chicago
†Microsoft Research

2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-8711-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPS57955.2024.00067

***Abstract*—The efficiency of concurrency control protocols plays a crucial role in transaction processing systems. However, when it comes to deterministic transactions (i.e., transactions with known read/write key sets), existing concurrency control protocols are not optimized to make the most of the determinism. They either force transactions to be aborted and retried, which negatively affects system throughput, or use a centralized scheduler to organize transactions in a way that avoids aborts, but with limited system scalability.**

In this paper, we present DecentSched, a highly efficient decentralized concurrency control protocol for deterministic transactions. DecentSched employs fine-grained queuing and a decentralized scheduling algorithm to enable serializable concurrent transaction execution with a high degree of parallelism. Extensive evaluation results show that DecentSched can outperform state-of-the-art concurrency control protocols in representative benchmarks.

***Index Terms*—Transaction processing, Transaction scheduling, Concurrency control**

I. Introduction

Transaction processing systems employ advanced concurrency control protocols to exploit parallelism on modern hardware platforms. A major drawback of these protocols is that they inherently induce transaction aborts. For example, optimistic concurrency control (OCC) detects conflicts between concurrent transactions through commit-time validation [6, 20, 24, 26, 39, 43, 45]. It must abort transactions that violate the isolation property. A pessimistic concurrency control protocol like two-phase locking (2PL) identifies the dependencies between concurrent transactions via per-object locking [1, 3, 4, 15, 19]. When transactions form cycles in the dependency graph, some of them must abort to avoid deadlocks. A transaction aborted by the concurrency control protocol must keep retrying until committed successfully or aborted by program logic. Under a highly contentious workload, excessive aborts will result in a reduced level of parallelism, which overshadows the benefits of concurrent transaction execution.

Concurrency control protocol induced aborts are caused by the lack of *determinism* in transactions. For example, a transaction in OCC cannot determine whether it can successfully commit when the execution starts [6, 20, 24, 39, 43]. In Wound-Wait 2PL [3], a transaction that has successfully acquired locks can still be aborted by a transaction that started earlier. Concurrency control protocols are bound to abort and retry transactions due to the non-deterministic nature of general transaction workloads. This limits the cost-effectiveness of concurrent transaction processing in several aspects, including unpredictable end-to-end latency, overwhelming contention on shared data structures, and wasted computing resources.

In deterministic transaction processing systems, an opportunity exists to eliminate aborts and improve performance of OLTP workloads [33, 35, 36, 37]. A deterministic transaction has its read/write set known before it starts execution, which is practical for a subset of real-world applications, for example, financial transactions [37]. In such applications, concurrency control protocols can utilize this knowledge to execute transactions free of aborts. For instance, acquiring locks in a consistent order can avoid deadlocking. Therefore, 2PL protocol induced aborts can be completely eliminated. That said, a transaction may still wait on each lock before it can proceed to the next lock. Excessive waiting and lock contention can compromise inter-transaction parallelism. Previous works proposed scheduling transactions before execution by workload analysis in single-node [16, 17, 32] or distributed [33, 35, 37] systems. However, as the number of concurrent transactions grows, the scheduling cost ultimately saturates the resources of the monolithic scheduler. Therefore, a centralized scheduler often becomes a scalability bottleneck [16].

We present DecentSched—short for Decentralized Scheduling, a novel concurrency control protocol that gains high parallelism and low latency in deterministic transactions. In general, DecentSched exploits parallelism by properly scheduling transactions across worker threads. Unlike the existing approaches that use a centralized scheduler or contentious lock operations, scheduling in DecentSched is done by individual workers in a decentralized way. In DecentSched, a transaction can determine its execution order in a global schedule by itself, and the schedule is consistent across all concurrent transactions. Therefore, concurrent transactions can execute following their self-determined schedules without any aborts. DecentSched is built on two core ideas. First, it employs a queuing-based approach that allows transactions to exchange data access conflicts at a low cost. Second, it adopts a decentralized scheduling algorithm with which every transaction can obtain a globally consistent execution schedule without using a centralized scheduler.

DecentSched provides serializable isolation between concurrent transactions. In addition, it is adaptive to different workload patterns without optimistic or pessimistic assumptions. We implement DecentSched as a library that can be integrated into existing transaction processing systems. Evaluations of DecentSched on a single-node OLTP system show that it can achieve high throughput and retain low end-to-end latency in representative benchmarks.

This paper is organized as follows. Section II gives essential background information about concurrency control protocols.

1530-2075/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPS57955.2024.00067

Section III introduces the design of DecentSched. Section IV evaluates DecentSched through multiple benchmarks and compares it with the state-of-the-art. Section V discusses related research works, and Section VI concludes this paper.

II. Background and Motivation

A concurrency control protocol arranges the execution of simultaneous transactions to provide an illusion that a transaction has exclusive access to the data store (e.g., database), which satisfies serializability. To better exploit parallelism, modern transaction processing systems mainly employ two types of concurrency control protocols—2PL and OCC [3, 24].

2PL is a representative pessimistic concurrency control protocol that presumes frequent conflicts between concurrent transactions. In 2PL, a transaction needs to acquire locks for its accessed objects (e.g., rows in a database) before execution. When a lock is already acquired by another transaction, the transaction has to wait until the lock is released, which constructs a *wait-for* dependency between the two transactions. Since different transactions can access objects in different order, deadlocks can occur due to circular waiting. To solve this problem, 2PL must break cycles in the dependency graph by aborting specific transaction(s). For example, as shown in Figure 1, a transaction in Wait-Die 2PL [3] aborts itself if it finds itself younger than the current lock holder. Similarly, a transaction in Wound-Wait 2PL [3] kills (wounds) the lock holder if the holder is younger.

In contrast to 2PL, OCC presumes that conflicts are rare. A typical transaction in OCC first executes on a local copy of the accessed data without blocking other transactions. Then, locks are only briefly acquired for validation and committing. Commits on hot objects can invalidate local copies and force transactions that are running opportunistically to abort and retry. In the worst case, a transaction may never successfully commit due to conflicts on hot items [10, 19].

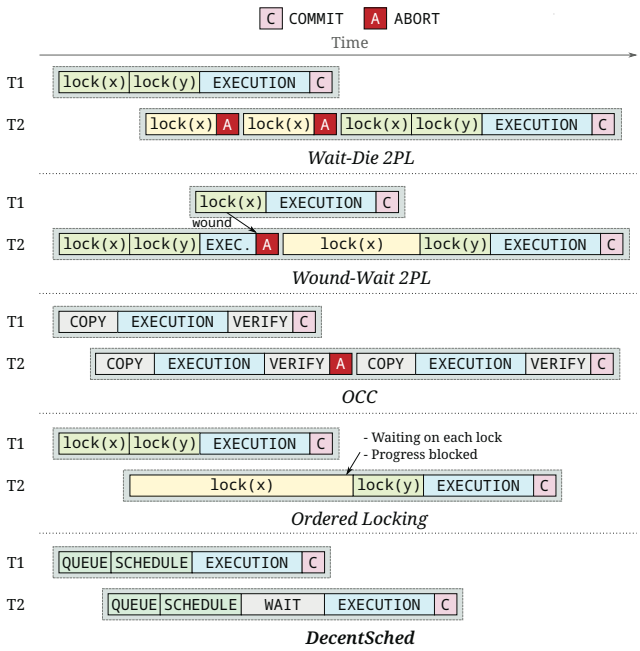


Fig. 1: Examples of concurrent transactions in different protocols. T1 has a lower transaction ID than T2 and they conflict on two shared variables x and y.

The absence of determinism in general transactions often results in unavoidable aborts, constraining the efficiency of concurrency control protocols. By adopting a deterministic transaction model, there is a chance to achieve determinism in concurrent transaction processing while significantly reducing or even eliminating aborts [33, 35, 36, 37]. However, 2PL and OCC fail to utilize the determinism because they are incapable of capturing and resolving complex dependencies. A lock in 2PL only represents a unidirectional *wait-for* relationship. Therefore, abort-and-retry is the only available measure for avoiding deadlocks. A transaction in OCC is unaware of other conflicted ongoing transactions. Therefore, commits are purely opportunistic in OCC. On the other hand, existing solutions that utilize the determinism face other challenges, including contention and waiting with ordered locking and scalability bottleneck with centralized transaction scheduling, as discussed in Section I.

In summary, the aforementioned concurrency control protocols do not achieve maximal determinism while maintaining high efficiency. To get the best of both worlds, we present DecentSched, a concurrency control protocol that realizes *decentralized scheduling*, where individual transactions can obtain their execution order from a globally consistent schedule on their own. Its design combines and extends the core ideas of 2PL and centralized scheduling. First, it upgrades 2PL's per-object locks to per-object queues to capture essential ordering information of concurrent transactions on each object. Second, it adopts a decentralized scheduling algorithm with which a transaction can derive its execution order based on its queuing ranks in a decentralized way.

III. DecentSched

This section describes the design of DecentSched. We first introduce the system model of DecentSched (§III-A). Then, we present how DecentSched exploits determinism by introducing its per-object queuing and decentralized scheduling (§III-B). After that, we discuss the correctness of DecentSched (§III-C). Finally, we introduce DecentSched's implementation details, including queue implementation, memory management, transaction ID allocation, and optimizations (§III-D).

A. System Model

In DecentSched, the smallest data access unit is an *object* (often referred to as a row in a database table). An object has a globally unique ID (`oid`) and it can be reached by looking up an index using its `oid`. For simplicity, we use `objs[oid]` to represent an object access.

A transaction in DecentSched consists of a sequence of operations over a set of objects. Each transaction has an *access set* which records the `oid` of every object to be accessed,

Algorithm 1 Queuing

```
1: function ENQUEUE(T)
2:     T.ready ← false
3:     T.finished ← false
4:     for each access ∈ T.access_set do
5:         queue ← objs[access.oid].queue
6:         type ← access.type
7:         access.entry ← {T.tid, type}
8:         ATOMIC_ENQUEUE(queue, access.entry)
```

along with an access type (either read-only or read-write, denoted as `R` and `W` respectively). We assume the access set of a transaction is known before it starts execution, which is similar to previous works that target deterministic transactions [16, 17, 32, 33, 35, 36, 37]. Each transaction has a globally unique ID (`tid`), and is denoted as T_{tid}. The `tid`s are allocated in monotonically ascending order. DecentSched maintains an array of transaction metadata, indexed by `tid`s, that is accessible by all threads in the system. `txns[tid]` represents an access to a transaction's metadata.

DecentSched uses multiple *worker threads* for concurrent transaction processing. Each worker thread processes exactly one transaction at a time until the transaction successfully commits, or aborts due to program logic. Once a transaction commits, the updated objects (of read-write access type `W`) are made visible to all the worker threads in the system. If a transaction aborts, all pending updates are discarded.

B. Protocol Design

DecentSched processes a transaction in five phases — queuing, decentralized scheduling, waiting, executing, and committing. A transaction first uses queuing and decentralized scheduling in DecentSched to obtain its execution order in a globally consistent schedule. Then, it waits for its predecessors according to the schedule, executes the program logic, and finally commits the updates. The program logic can induce aborts in the executing phase. Compared to traditional transaction scheduling solutions, DecentSched differs primarily in the first two phases. Thus, we focus below on explaining how queuing and decentralized scheduling work.

1) Per-object Queuing: DecentSched uses a queuing-based approach to capture essential timing information of concurrent transactions, which determines their execution order. This approach is based on the intuition that multiple threads can access an object on a first-come, first-served (FCFS) basis.

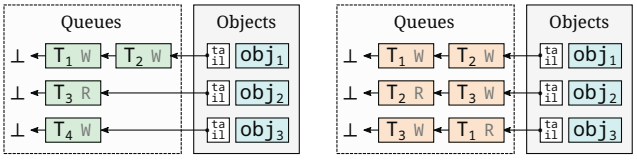
Using one queue to handle transaction scheduling is intuitive. However, a single queue, like a centralized scheduler, can easily cause scalability issues. Therefore, DecentSched employs per-object queuing by maintaining a queue for each object in the system. For each transaction, the corresponding worker thread enqueues on every object in the transaction's access set. A queue entry records the `tid` of the transaction as well as the access type on the object. The pseudocode for the queuing procedure is shown in Algorithm 1. Two atomic flags of the transaction, `ready` and `finished`, are initially unset. The usage of these flags will be introduced later.

If each transaction only accesses one object, the rank of a transaction in its corresponding queue can be used to determine the execution order. For example, in Figure 2a, each transaction only needs to wait for the transaction ahead of it to finish execution, so that all the transactions execute in a serializable schedule. In the example, T_1, T_3, and T_4 can execute simultaneously without any waiting, while T_2 must wait for T_1's completion.

However, a real-world transaction must be able to access multiple objects. Since multiple queue entries cannot be inserted atomically, the order of transactions might differ on different queues, which can cause *circular dependency*. For example, in Figure 2b, T_1, T_2, and T_3 form a cycle in the dependency graph. As a result, the naive solution that only identifies per-queue dependencies can cause deadlocks.

To make different worker threads derive a valid and consistent schedule when facing circular dependencies, all concurrent transactions in a cycle must be aware of the cycle in the first place. Therefore, transactions must obtain information beyond per-object dependencies. In addition, transactions in the same cycle must make mutually consensual scheduling decisions. In order to satisfy these requirements under a multi-queue setup, we develop a decentralized scheduling algorithm. In this algorithm, a transaction first collects sufficient dependency information using lightweight queue searching to reveal cycles, then determines its own execution order which is conflict-free and deadlock-free.

2) Dependency Tracking: The dependency set of a transaction in DecentSched consists of direct and indirect dependencies. For a transaction T_i, every T_j that is ahead of T_i in any object's queue is a direct dependency of T_i, unless T_i and T_j only perform read-only operations on every object accessed by both of them. All the transactions satisfying the above criterion form the *direct dependency* set of T_i. Then, the full dependency set of T_i can be derived by recursively including the dependencies of those in T_i's direct dependency set. In other words, for every T_j in T_i's dependency set, every transaction T_k in T_j's dependency set is also a dependency of T_i. Accordingly, the extra transactions to be included are included in T_i's *indirect dependency* set. The tracking of indirect dependencies allows transactions to observe cyclic dependencies. For an arbitrary pair of transactions T_i and T_j, if T_i and T_j are each other's dependency (either direct or indirect), they must appear in at least one same dependency cycle. Below, we use `T` to denote a transaction, or its assigned worker thread acting on its behalf.



(a) Single-object access (b) Multi-object access

Fig. 2: Examples of per-object queues.

Algorithm 2 Direct Dependency Discovery

```
1: function DISCOVER_DIRECT(T)
2:     T.direct ← ∅
3:     for each access ∈ T.access_set do
4:         entry ← access.entry
5:         while entry.next != nil do
6:             entry ← entry.next
7:             if entry.type ≠ R or access.type ≠ R then
8:                 SET_ADD(T.direct, entry.tid)
9:     T.ready ← true
```

Algorithm 3 Indirect Dependency Discovery

```
1: function DISCOVER_INDIRECT(T)
2:     T.indirect ← ∅
3:     for each tid ∈ T.direct do
4:         DISCOVER_INDIRECT_REC(T, tid)
5: function DISCOVER_INDIRECT_REC(T, tid)
6:     if MARK_VISITED(T, tid) = false then
7:         return
8:     SET_ADD(T.indirect, tid)
9:     Td ← txns[tid]
10:    wait until Td.ready = true
11:    for each tid′ ∈ Td.direct do
12:        DISCOVER_INDIRECT_REC(T, tid′)
```

A transaction T in DecentSched discovers its dependencies as follows. After the queuing phase of T, T scans all its accessed objects' queues to identify its direct dependencies. The pseudocode of this process is shown in Algorithm 2. After all the direct dependencies have been recorded in T's direct dependency set, T's ready flag is set. Accordingly, any transaction that queries T's direct dependency set must wait for T's ready flag to be set for completeness.

Then, T discovers its indirect dependencies based on the direct dependencies. The pseudocode is shown in Algorithm 3. In general, DecentSched adopts a depth-first search approach to recursively discover a transaction's indirect dependencies. On each transaction T_v that is being visited, T must wait until T_v's ready flag is set (i.e., T_v's direct dependencies have been fully discovered). Then, T adds T_v as its indirect dependency, and continues the recursive search over T_v's direct dependencies. Note that the recursive search continues regardless of whether T_v has finished or not because not all of its dependencies are guaranteed to have been finished according to the scheduling rules (introduced later in §III-B3). The search ignores visited transactions, including the searching transaction itself, to avoid redundant work. We further discuss the optimizations on searching in §III-B5.

We use T_1 in Figure 2b to show how a transaction discovers its dependency set and finds cycles. By scanning the queues of obj_1 and obj_3, T_1 sees T_3 as a direct dependency. Then, T_1 recursively searches on T_3's direct dependencies and sees T_2 as an indirect dependency. Finally, T_1 sees itself in T_2's dependency set and ends the search. In the meantime, T_2 and T_3 also perform searches in a similar way. As shown on the left in Figure 3, after dependency discovery, T_1–T_3 include each other in their dependency sets. T_1 observes that it must be in a cycle with T_2 and T_3 by finding itself in the dependency set of T_2 and T_3, respectively. Figure 3's right side shows a similar example where T_1 and T_2's positions are swapped on the first queue. In that case, T_1's dependency set contains T_2, but T_2 has no dependencies. Therefore, T_1 knows that it must not be in any cycle containing T_2.

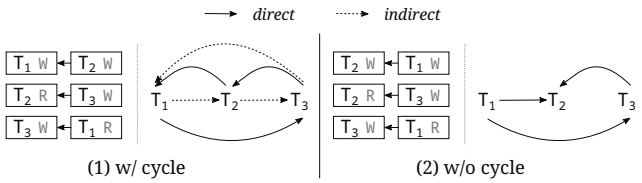


Fig. 3: Two examples of transaction dependency. An arrow from T_i to T_j means that T_j is T_i's dependency.

When a transaction exists in multiple cycles, the dependency tracking will add all the transactions in these cycles into the dependency set. As a result, the transaction cannot tell which dependencies belong to which cycles. In this condition, a transaction that appears in multiple cycles needs to derive its execution order without identifying different cycles. Meanwhile, all the other transactions must also obtain a globally consistent schedule. In the following, we explain how scheduling works without the knowledge of individual cycles.

3) Scheduling: A transaction T_i in DecentSched determines its execution order solely based on its dependency set. For each transaction T_j in T_i's dependency set, if T_i is not in T_j's dependency set, T_j must have been positioned ahead of T_i in all queues. Therefore, they are scheduled on a first-come, first-serve basis. Specifically, T_i needs to wait until T_j finishes execution.

Otherwise, when T_i is also in T_j's dependency set, they are likely in a cycle. In this scenario, their execution order is determined by their tids' ranks. Note that DecentSched's dependency discovery guarantees that running transactions connected by any cycles see each other in their dependency sets symmetrically, because an arbitrary pair of transactions in a dependency cycle will reach each other during their dependency discoveries. We use the following rule to order transactions in a consistent way. For T_i, if both T_i and T_j are in each other's dependency set and $i > j$, T_i needs to wait until T_j finishes execution.

The scheduling rules above can be implemented as follows. For a transaction T_i, it checks each of its dependencies (denoted as T_j). If $i > j$, T_i unconditionally waits for T_j. Otherwise, T_i waits for T_j until either T_j finishes execution, or T_i is added to T_j's dependency set by T_j's worker thread.

Under certain scenarios, the aforementioned scheduling methods can make a transaction perform unnecessary waits. Consider the example shown in Figure 4. After completing their dependency discoveries, T_1–T_3 have each other in their dependency sets. Following the scheduling rules introduced above, T_1 will execute first because it has the lowest tid in the cycles. In the meantime, T_2 needs to wait for T_1 as they're in each other's dependency set. However, T_1 and T_2 actually

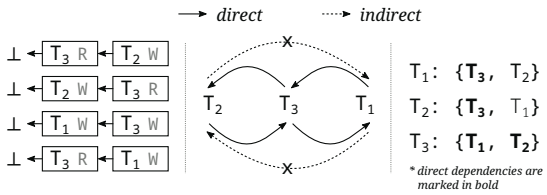


Fig. 4: Multi-cycle example.

reside in different cycles and they do not have real conflicts. Therefore, they should be able to execute in parallel.

DecentSched uses the following rule to avoid unnecessary waiting between non-conflict transactions. For two transactions T_i and T_j, if both of them are not each other's direct dependency, then we know that their access sets do not conflict. As a result, they do not need to wait for each other. In the example shown in Figure 4, T_2 does not need to wait for T_1 because they both only have T_3 as their direct dependency.

Algorithm 4 summarizes the scheduling rules DecentSched applies to every transaction to let them make mutually consensual scheduling decisions.

Algorithm 4 Scheduling

```
1:  function SCHEDULE(T)
2:      for each tid ∈ (T.direct ∪ T.indirect) do
3:          Td ← txns[tid]
4:          if tid ∉ T.direct and T.tid ∉ Td.direct then
5:              continue
6:          if T.tid > tid then
7:              wait until Td.finished = true
8:          else
9:              deps ← Td.direct ∪ Td.indirect
10:             wait until (T.tid ∈ deps or Td.finished = true)
```

4) Program Logic Execution and Commit: After the decentralized scheduling phase, a transaction can safely execute with exclusive access to its read-write objects, and no other transaction can modify its read-only objects during the execution. The transaction may abort due to program logic or successfully finish with its updates committed. Afterward, DecentSched sets the transaction's `finished` flag. Other transactions that are waiting for the transaction can proceed once they observe this flag change.

5) Search Pruning: Committed or aborted transactions can still have a negative impact on search efficiency. Every transaction has to scan queues of accessed objects to discover direct dependencies. When a transaction's `finished` flag has been set, other transactions may still recursively search on its dependency set because they may still find live transactions from there (as described in the indirect dependency discovery). The cost of scanning and search will keep increasing as the number of enqueued transactions on each queue grows.

DecentSched adds a flag on each transaction to enable the pruning of search paths. It marks a transaction `T` as *retired* when: (1) every transaction in `T`'s dependency set has finished, and (2) every transaction that is ahead of `T` in a queue has finished, regardless of whether the transaction is in conflict with `T` or not. The flagged transactions can help prune search paths. When discovering direct dependencies by scanning queues, `T` can stop traversing a queue when it reaches a retired transaction because all transactions ahead of the retired transaction have finished. Finished transactions can be safely ignored because they will not conflict with `T`. On the other hand, when discovering indirect dependencies by searching, `T` does not initiate a search starting from a retired transaction, which avoids unnecessary traversals in the search space.

Similar reasoning also applies when `T` adds a transaction into its dependency set during dependency discovery. If the transaction is finished, `T` does not add it to `T`'s dependency set (either direct or indirect) because a finished transaction does not conflict with `T`. Note that `T` will still continue scanning the queue or initiate further dependency search if the transaction is only *finished* but not *retired* for completeness.

Whether a finished transaction is retired or not is actively evaluated by other transactions when they are searching in the transaction's dependency set. To facilitate retirement evaluation, a transaction's metadata also records all the unfinished transactions ahead of it in the queues. Since a worker thread in DecentSched runs one transaction at a time, the total number of transactions (including dependencies and transactions ahead) recorded in each transaction's metadata is bounded by the number of worker threads.

C. Correctness

We present the argument for the correctness of DecentSched from three aspects. First, the recursive exploration of indirect dependencies of a transaction is devoid of data races and terminates in a finite number of steps. Second, the decentralized scheduling is free of deadlocks. Third, DecentSched provides serializability, which renders concurrent execution of transactions equivalent to a sequential order of execution.

1) Search Termination and Race-freedom: A recursive search for indirect dependencies terminates in finite steps. This can be proved by contradiction. If a recursive search by a transaction does not stop, there is at least one search path that can be indefinitely extended by visiting an ever-increasing number of transactions. According to the search algorithm, all the transactions on this path will also encounter an indefinite search. Therefore, none of them can finish the search. Since every worker thread in DecentSched runs one transaction at a time, the number of running transactions is bounded. Thus, the ever-increasing number of transactions on the path contradicts the fact of having a bounded number of worker threads. Therefore, a recursive search process in DecentSched terminates. Since the dependency detection algorithm only reads other transactions' metadata, and it proceeds only when a transaction's `READY` flag is set, no data races would occur during this process. The reason is that dependency detection only requires direct dependencies of the transactions on the search path, and a transaction's direct dependencies are immutable after its `READY` flag is set.

2) Deadlock-free Scheduling: In DecentSched, a transaction recursively extends its dependency set by including its

indirect dependencies. Therefore, all the transactions in a dependency cycle are aware of the cycle and see each other in a symmetrical way. The scheduling rules guarantee that the transaction with the smallest tid in a cycle can start execution, followed by the other transactions ordered by their tids. When a transaction resides in multiple cycles, the transactions in all these cycles will be recursively added into each other's dependency set as if they're all in the same cycle. Similarly, these transactions can always make progress starting from the one of the smallest tid.

3) Serializability: DecentSched naturally provides serializability because transactions in conflict form a sequential schedule based on their tids. For an arbitrary pair of transactions $\mathtt{T_i}$ and $\mathtt{T_j}$, if they do not access the same objects or only read the same objects, their concurrent executions do not violate serializability. If $\mathtt{T_i}$ and $\mathtt{T_j}$ have read-write or write-write conflicts on some objects, one of the following stands: (1) $\mathtt{T_j}$ is in $\mathtt{T_i}$'s dependency set while $\mathtt{T_i}$ is not in $\mathtt{T_j}$'s dependency set. (2) $\mathtt{T_i}$ is in $\mathtt{T_j}$'s dependency set while $\mathtt{T_j}$ is not in $\mathtt{T_i}$'s dependency set. (3) $\mathtt{T_i}$ and $\mathtt{T_j}$ are in each other's dependency set. For all of the three cases, $\mathtt{T_i}$ and $\mathtt{T_j}$ will not execute in parallel according to the scheduling rules.

D. Implementation and Optimization

We implement DecentSched as a user-level library. In the following, we introduce the implementation details of the DecentSched library.

1) Queue: Each queue in DecentSched is a lock-free singly linked list. Before enqueuing, each transaction allocates memory for its queue entries and writes the corresponding tid and access types to the entries. Then, it enqueues on each queue with an atomic *compare-and-swap* (CAS) operation, plus occasional retries under contention.

The number of per-object queues in DecentSched grows as the number of objects increases. Maintaining a large number of queues can cause high memory usage and poor cache efficiency. To overcome this limitation, DecentSched employs *queue sharing* that bounds the number of queues to a configurable number m. Objects are mapped to the m queues by their object ID. Specifically, the queue index (qid) of an object is calculated as $\mathtt{qid} = hash(\mathtt{oid}) \bmod m$.

Before queuing, a transaction first calculates the corresponding qids based on its access set. Then, it follows regular queuing routines to enqueue. Note that if multiple objects in its access set map to the same queue, the transaction only allocates one entry for the queue and sets the access type accordingly. If there are mixed access types (both read-only and read-write), the access type of the entry will be set to read-write (W).

Queue sharing can incur false positives in conflict detection which causes unnecessary waiting. However, with a sufficient number of queues, the false positive rate can be controlled at a low level, and the performance impact is negligible in practice. More importantly, it does not affect DecentSched's correctness (i.e., deadlock-freedom and serializability). Queue sharing also helps retain the space efficiency of DecentSched and makes DecentSched tunable for different sizes of data. We further evaluate the effectiveness of queue sharing in Section IV.

2) Memory Management: DecentSched employs epoch-based memory management to avoid frequent memory allocations and expensive garbage collection operations. It splits time into *epochs* and an epoch ends when either of the following is satisfied: (1) a worker thread has processed at least k transactions; (2) the current epoch has lasted over t milliseconds. Both k and t are tunable parameters. At the end of each epoch, all worker threads synchronize on a barrier. After that, the memory used in the previous epoch is fully reclaimed at a low cost.

3) Transaction ID Allocation: In DecentSched, the transaction scheduling relies on the value of tids to decide the execution orders when dependency cycles appear. It is necessary to allocate tids in a fair way to avoid privileged worker threads. An intuitive tid allocation method is using a global integer and letting worker threads perform atomic *fetch-and-add* operations to obtain their tids. In this way, the transactions can be approximately ranked by start time. However, a globally shared variable can become a bottleneck on multi-core platforms [11, 39, 43].

Each worker thread in DecentSched allocates tids using a thread-local counter to avoid contention. With w worker threads in the system, a worker thread with worker ID i ($0 \leq i < w$) starts to assign tids from i, with an increment of w after each transaction. That said, the tids allocated by different worker threads may diverge between faster and slower workers over time. As a result, transactions running on worker threads with fewer executed transactions will always be prioritized when it's in a cycle. This can be unfair to faster workers. DecentSched mitigates the problem by reinitializing every allocation counter to the respective worker ID (i) at the beginning of each epoch. Therefore, the tid divergence in an epoch does not persist beyond the epoch. Additionally, the maximum number of transactions in an epoch is bounded so that tids never overflow.

4) Opportunistic Execution: While a transaction is waiting, its worker thread will try to execute the transaction logic locally. If the transaction begins its actual execution according to the global schedule and the local read sets have not been updated since the last attempted execution, it will be committed directly. To verify this, an atomic version number is assigned to each object in the system. When a transaction is committed, the version number of each object in its write set is incremented by 1.

5) Memory Footprint: DecentSched maintains a transaction metadata array and multiple queues. These structures are lightweight in-memory metadata. When there are 32 worker threads, 16384 queues, and 1024 transactions per worker thread per epoch, the memory allocated for all structures costs 62 MB of memory, which is not a point of concern in commodity servers.

IV. Evaluation

We integrate DecentSched into the codebase of Bamboo [2, 19], which contains an up-to-date version of DBx1000, a multi-core in-memory transaction processing system often used as a benchmark suite for performance evaluation [12, 42]. DBx1000 implements various state-of-the-art concurrency control protocols that can be compared against DecentSched in the same system framework and benchmarks. To make sure the baseline protocols are correctly configured in our experiments, we tune the parameters of Bamboo's DBx1000 implementation and validate its performance against STOv2, another representative multi-core in-memory transaction processing system [21]. Specifically, we employ hash-based data indexing and NUMA-aware memory allocation based on `jemalloc` [22]. In terms of contention regulation, we use a randomized exponential backoff strategy and the backoff buffer stores up to three recently aborted transactions. These configurations achieve matched throughput results in the TPC-C-NP benchmark (see §IV-B) for the TicToc protocol implemented in both systems.

We run the experiments with DecentSched and four classes of protocols that provide serializable isolation: centralized scheduling, ordered locking, OCC, and 2PL. We evaluate more than one implementation for each of OCC and 2PL as listed below. To demonstrate the performance characteristics of different protocol classes, we consolidated the results of each class by picking the best result (the highest throughput) for each data point. For example, data points on a throughput curve of OCC can be a mix of Silo and TicToc's results, whichever has the highest throughput.

- **C-Sched**: A centralized scheduling approach using a dedicated thread to arrange all the transactions.
- **Ord-Lock**: A locking-based protocol where each transaction acquires locks in a consistent order.
- **Silo (OCC)** [39]: A variant of OCC with contention-free transaction ID allocation.
- **TicToc (OCC)** [43]: A variant of OCC that provides decentralized timestamp allocation for better scalability.
- **Bamboo (2PL)** [19]: An enhanced Wound-Wait 2PL that enables early visibility of uncommitted data.
- **Wait-Die (2PL)** [3]: The Wait-Die variant of 2PL which is mentioned in Section I.

We have put additional effort into finding available systems for evaluation. However, we are not aware of any open-sourced systems using transaction scheduling, including those applicable to a similar single-node setup as in the aforementioned solutions [16, 17, 32]. Therefore, we implement a centralized, abort-free scheduling mechanism (C-Sched) in the DBx1000 framework. C-Sched has a dedicated scheduler thread that uses a monolithic access map to arrange the execution of transactions in different worker threads. Each worker uses a private memory buffer to communicate with the scheduler in a lock-free fashion. Before executing a transaction, the worker copies its read/write set to the buffer to create a request. The scheduler keeps polling the requests from each worker's buffer. If a transaction has no conflict with other running transactions, the scheduler sets a flag in the buffer as a signal for allowing execution. Otherwise, the scheduler will skip the request and check it again in the next round of polling. The worker can execute the transaction once it sees the flag. The implementation is capable of processing 4.7 million requests per second with 32 worker threads when each transaction has one read variable and there is no contention between workers.

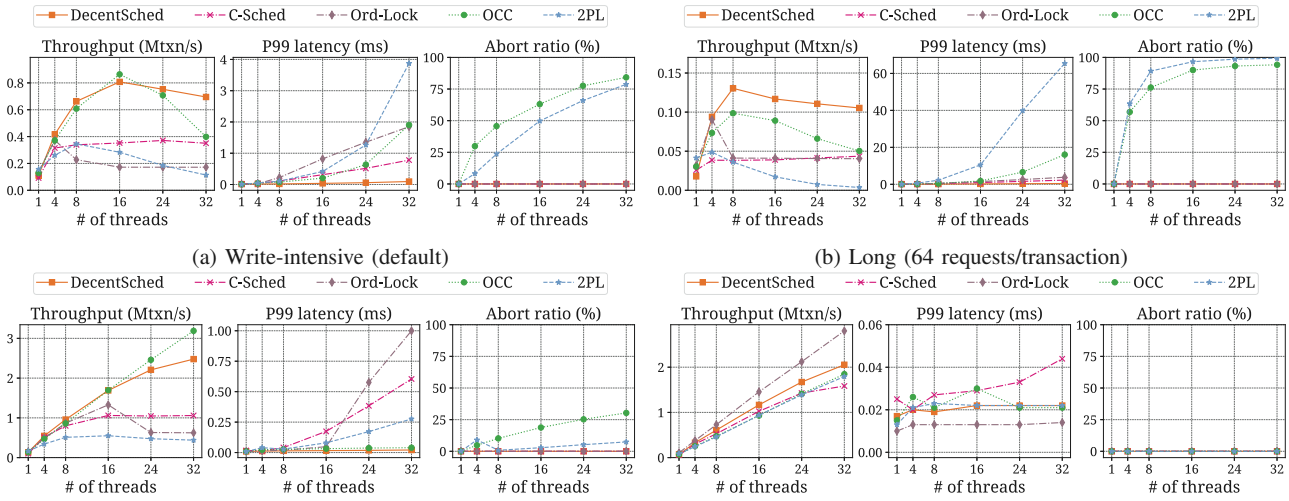
In order to provide sufficient insights about DecentSched's performance, we measure multiple performance metrics in the evaluation, including transaction processing throughput, end-to-end latency, and abort ratio. Specifically, for end-to-end latency, we focus on tail latency. Therefore, we mainly show the metrics of P99 latency. Through the experiments, we seek answers to the following questions: (1) How does DecentSched perform compared to other systems under different workload patterns? (2) What are the impact factors of the performance numbers? (3) How do the implementation and optimization techniques affect the performance of DecentSched?

All the experiments are performed on a server on CloudLab with dual 16-core Intel Xeon Gold 6142 CPUs and 384 GB of RAM. The server runs Ubuntu Linux 22.04 with kernel version 5.15 LTS. DecentSched is configured with 1 ms epoch timeout and 1024 per-worker transactions per epoch (whichever comes first), and 16384 shared queues. These numbers are chosen based on the observation that they can provide the most balanced performance on the server.

A. YCSB Experiments

We first use YCSB [9] to evaluate the performance of DecentSched. YCSB is a popular benchmark framework that adopts realistic workload patterns that are representative of large-scale Internet services. For all experiments in this section, we use a large database table that contains 100 million records. The default parameters are as follows unless otherwise specified. Keys are generated following the Zipfian distribution with $\theta = 0.99$. Each transaction accesses 16 records in the data store. Each access can be either a read or a write (update) and the read/write ratio is 1:1. Based on YCSB workload F that has transactional *read-modify-write* semantics, we adjust the read/write ratio, transaction size, and key distribution in the experiments. The results are summarized in Figure 5.

1) Write-intensive: The first YCSB experiment uses the default setup mentioned above, which demonstrates a scenario of skewed workloads with high contention and a high update ratio. The results are shown in Figure 5a. All protocols perform similarly with one thread. When the number of threads is greater than one, DecentSched outperforms other protocols, except for OCC when there are 16 threads. It achieves up to $1.7\times$ the throughput of the second-best-performing protocol (OCC, at 32 threads). This shows that DecentSched performs well under highly contentious workloads. OCC performs well with fewer than 16 threads. However, as the abort ratio increases, its throughput degrades when the number of threads goes beyond 16. Ord-Lock performs well with a low number of threads (≤ 8). However, with more than 8 threads, Ord-



(a) Write-intensive (default)

(b) Long (64 requests/transaction)

(c) Read-intensive (95% read, 5% write)

(d) Random (uniform key distribution)

Fig. 5: YCSB benchmark results. The default configuration is described in §IV-A.

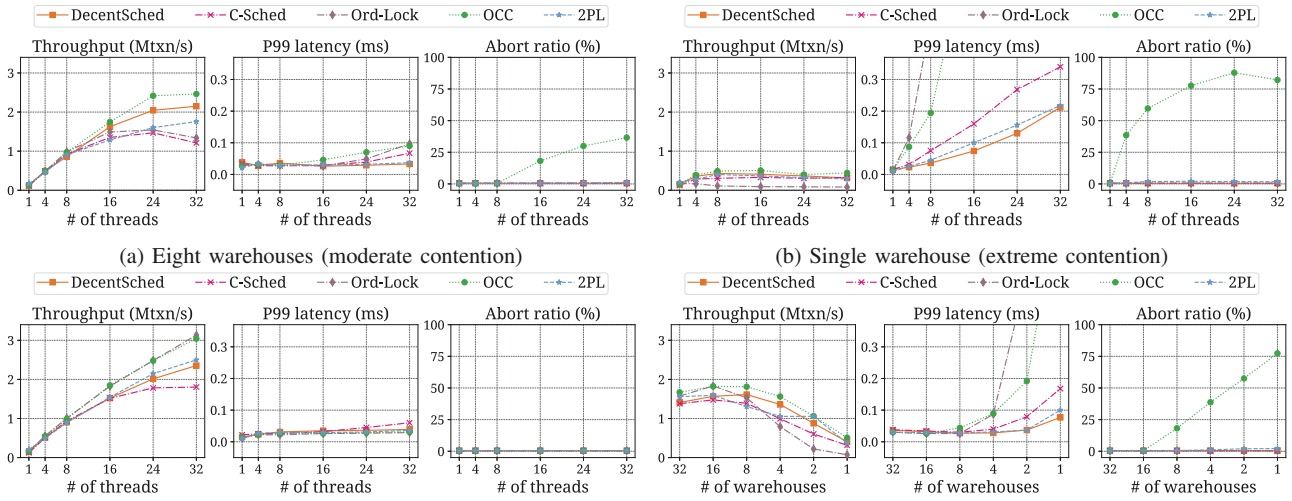
Lock suffers from excessive waiting on individual locks. The drastically increasing latency limits the scalability of Ord-Lock, which degrades the transaction processing throughput. The throughput of C-Sched does not scale with more than 8 threads because the scheduler becomes the scalability bottleneck. A transaction must wait until the scheduler updates the flag so that it can execute. With more concurrent transactions, the scheduler is saturated in detecting conflicts and scheduling transactions.

With more than 16 threads, DecentSched shows slightly degraded throughput. The reason is that with a large number of threads in contention, it costs a relatively longer time for a transaction in DecentSched to perform dependency discovery. In addition, the metadata of a transaction (including its dependency set) has a larger size, which reduces cache efficiency. Similar degradations are also shown in 2PL-based protocols. Throughout the experiments, DecentSched maintains a low end-to-end latency. Its latency is 5.7× lower than OCC, which shows higher throughput than DecentSched with 16 threads.

2) Long Transactions: We then run a YCSB experiment with the default parameters, except that the number of object accesses per transaction is increased to 64 to measure the performance of each protocol with long transactions that have higher abort penalties. The results are shown in Figure 5b. DecentSched retains its leading position in this experiment as it reaches up to 2.1× the throughput of the second-best-performing protocol (OCC, at 32 threads). However, since each transaction accesses 64 objects, the queuing cost of DecentSched is higher than that in the write-intensive experiment, and the dependency discovery takes a longer time to finish. As a result, the throughput of DecentSched decreases when the number of threads is higher than 8. The throughput of Ord-Lock is close to that of DecentSched with fewer than 4 threads. However, due to the increased waiting cost, it suffers performance penalties and is outperformed by OCC with more than 16 threads. Similar to that in the write-intensive experiment, the throughput of C-Sched is saturated with more than 4 threads. In addition to the increased scheduling cost, as the number of concurrent transactions grows, the larger read/write sets also incur higher memory access costs for analyzing conflicts. The P99 latency metrics of all the compared protocols are at the millisecond level with more than 16 threads, while the worst P99 latency in DecentSched is still under half a millisecond (with 32 threads).

3) Read-intensive: The third YCSB experiment is read-intensive with 95% reads and 5% writes. The results are shown in Figure 5c. Compared to the write-intensive experiment, this experiment exhibits lower contention on commonly accessed objects because most accesses to those objects are read-only. Therefore, all protocols show higher throughput than those in the write-intensive experiment. DecentSched shows good scalability because the number of dependencies of a transaction is small under low contention, so the cost of dependency discovery is low. OCC also scales well because of a low abort ratio.

4) Random Access: Finally, we run a YCSB experiment using a uniformly random key distribution. Each transaction randomly chooses 16 keys in the database (of 100 million keys) to access so that conflicts are rare. Note that other configurations are still default so this experiment is write-intensive with 50% reads and 50% writes. The results of this experiment are shown in Figure 5d. All protocols exhibit low abort ratios and scale well. Ord-Lock exhibits the highest throughput compared to other protocols. The reason is that it has the lowest software overheads as it uses a simple design that only maintains a reader-writer lock for each object. The throughput of C-Sched scales better than those in other YCSB experiments because most transactions do not



(a) Eight warehouses (moderate contention)

(b) Single warehouse (extreme contention)

(c) Variable warehouses (low contention)

(d) Fixed 16 threads (variable warehouses)

Fig. 6: TPC-C-NP benchmark results.

conflict so the scheduler is less pressured. DecentSched shows good performance numbers because, most of the time, the dependency set of a transaction is near-empty. OCC scales well but it exhibits lower throughput numbers. This is due to the nature that, unlike other protocols, OCC makes local copies for read and write requests, then copies local updates to the global data store when a transaction is committed. This incurs extra memory accesses to and from the global store, which limits its throughput.

B. TPC-C-NP Experiments

TPC-C [38] is an OLTP benchmark that has been the industry standard. It simulates a warehouse-centered order processing application and consists of nine database tables. We use TPC-C benchmark to evaluate the performance of DecentSched in more complex workload patterns.

In the experiments, we utilize the TPC-C benchmark in DBx1000 which models two transaction types in standard TPC-C benchmark—Payment and New Order. These two types account for 88% transactions in a TPC-C benchmark profile, which are representative. This modeling approach is also widely adopted in previous works [19, 26, 31, 34, 43]. Experiments in this section are named TPC-C-NP. We run TPC-C-NP with 50% Payment transactions and 50% New Order transactions. Since the number of warehouses is the main scaling factor, we start from experiments with eight warehouses, which represents a relatively large-scale order processing system with a moderate level of contention on our testing system. Note that we follow the TPC-C manual and randomly abort 1% of New Order transactions. These aborts are induced by program logic and cannot be avoided.

1) Eight Warehouses (Moderate Contention): We first run a TPC-C-NP experiment with eight warehouses. The results are shown in Figure 6a. With no more than 8 threads, all protocols perform similarly. They show very low abort ratios (around 0.5%) that are close to the overall proportion of random aborts in New Order transactions. The reason is that with a lower number of threads than the number of warehouses, most requests can be served by local warehouses so conflicts are rare. When the number of threads is higher than the number of warehouses, transactions on different threads become more contending. DecentSched outperforms other protocols by up to 1.8× except OCC. With 24 or more threads, OCC achieves up to 18% higher throughput than DecentSched, but it also has 2.7× higher P99 latency than DecentSched due to its frequent abort-retry and backoff activities.

Similar to the trend shown in the YCSB experiments, with more than 16 threads, Ord-Lock exhibits higher latency metrics and suffers from excessive waiting while the scheduler in C-Sched is saturated. This makes the throughput of Ord-Lock and C-Sched decrease significantly. For the case of 2PL (Bamboo), with 32 threads, its abort ratio is low, and its P99 tail latency is close to DecentSched's. Bamboo performs well because it allows transactions to read data from other uncommitted transactions to gain higher parallelism. However, it may lead to cascading aborts if an uncommitted transaction fails to commit. In TPC-C, updates to warehouses are committed by Payment transactions which only access three objects. Therefore, most of the transactions can finish execution quickly without causing aborts so that cascading aborts are rare. On the contrary, Bamboo suffers from frequent aborts in YCSB experiments because each transaction accesses at least 16 objects, and cascading aborts are much more frequent with a larger read/write set.

2) One Warehouse (Extreme Contention): We run the TPC-C-NP experiment with only one warehouse to demonstrate the performance of the protocols under extreme contention. In this experiment, all transactions access the same warehouse so that it represents the highest contention level of the TPC-C-NP

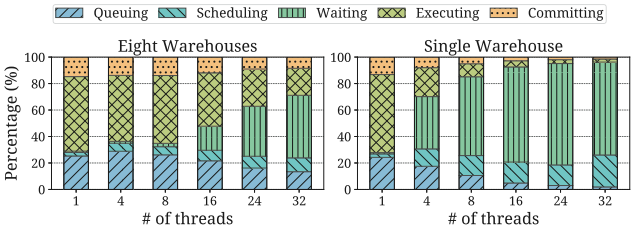


Fig. 7: The cost of each stage in DecentSched with varying number of worker threads in TPC-C-NP.

TABLE I: TPC-C-NP results of DecentSched with different numbers of queues, including the normalized throughput and latency values shown in parentheses.

# of queues	Throughput (Mtxn/s)	Abort (%)	P99 latency (μs)
256	0.68 (42%)	0.50	83 (319%)
1,024	1.14 (70%)		62 (238%)
4,096	1.49 (92%)		47 (181%)
16,384	**1.62 (100%)**		**26 (100%)**
65,536	1.60 (99%)		47 (181%)
262,144	1.45 (90%)		78 (300%)
1,048,576	0.71 (44%)		720 (2769%)

experiment. The results are shown in Figure 6b. DecentSched retains high throughput with more than one thread. However, all the protocols stop scaling with more than 8 threads because the parallelism they can exploit in New Order transactions is overshadowed by the increasing contention on updating a single warehouse in Payment transactions. Ord-Lock shows constantly decreasing throughput as the number of threads increases because it suffers from excessive waiting. The throughput of C-Sched stops scaling beyond four threads because the centralized scheduler once again becomes the system bottleneck. The OCC variants perform well and their opportunistic commit can help retain a high throughput. However, their P99 tail latency numbers are up to $4.3\times$ higher than that of DecentSched.

3) Variable Warehouses (Low Contention): In this experiment, the number of warehouses is equal to the number of threads, which represents a linearly scaling system. The results are shown in Figure 6c. Similar to the random YCSB experiment showcased in Figure 5d, all the protocols exhibit favorable scalability as the number of threads increases. Notably, Ord-Lock stands out among the protocols, achieving the highest throughput due to its minimal software overhead. Despite OCC involving additional local copies, the TPC-C-NP benchmark's smaller data store size and improved access locality (non-random) enable efficient data copying. As a result, the throughput of OCC is less constrained compared to that in the random YCSB experiment.

4) Fixed Threads: This TPC-C-NP experiment is with a fixed number of 16 threads and different numbers of warehouses, where the workload is more contentious with fewer warehouses. It shows the performance of each protocol under different contention levels. The results are shown in Figure 6d. Although 16 warehouses help to reduce contention in a workload, DecentSched reaches its peak throughput with 8 warehouses, and its throughput slightly degrades with 16 and 32 warehouses. The reason is that the database tables scale up with more warehouses. Therefore, with more warehouses, the transactions access more objects in the databases. DecentSched successfully exploits parallelism under highly contentious workloads. However, more queues are accessed with more warehouses. As a result, the throughput of DecentSched with 16 warehouses is 13% lower than that with 8 warehouses.

5) Performance Breakdown: Figure 7 reports the time spent in each DecentSched stage (as described in Section III). when running the TPC-C-NP experiments with eight warehouses and a single warehouse. As the number of worker threads grows, the increasing contention between concurrent transactions makes worker threads spend more time waiting. With the same number of threads, the scheduling cost of DecentSched in the experiment with one warehouse is relatively higher than that in the experiment with eight warehouses. The reason is that having only one warehouse also leads to increased contention, which makes the dependency detection more expensive as a transaction needs to scan most of the running transactions' dependency set. That said, the combined queuing and scheduling cost of DecentSched is between 19% to 35% in all the experiments, and it remains low as the number of threads increases.

6) Impact of Queue Sharing: DecentSched adopts queue sharing that allocates a fixed number of queues for serving an arbitrary number of objects. The number of queues in DecentSched is performance-sensitive because having fewer queues can increase the false positive ratio. To measure the performance impact of queue sharing, we rerun the 16-thread TPC-C-NP experiment with eight warehouses with different numbers of queues. The results are shown in Table I. The normalized throughput and latency results are also shown in parentheses. As we change the number of queues, we observe degraded throughput and increased P99 tail latency. We chose to use 16384 queues in our evaluation, which provides a good trade-off between a low false positive ratio and a low memory access cost. When DecentSched is deployed on different systems, it supports an automatic tuning phase which can determine the best number of queues for the target system specification and workload pattern.

C. Discussion

DecentSched achieves high throughput and low tail latency with abort freedom, but it does not lead in all the experiments. In this section, we summarize the key observations from the experimental results and discuss the best-case workload patterns for the compared protocols.

In workloads with very low contention (Figures 5d and 6c), i.e., there are barely any write conflicts that cause waiting, Ord-Lock achieves the best performance because of its low constant cost. However, when the update ratio on contending objects increases (Figure 5c), Ord-Lock starts suffering from performance degradation. The problem is that Ord-Lock can

only make progress on one lock at a time during the locking phase. Therefore, parallelism cannot be exploited before all locks have been acquired. OCC-based protocols also achieve high throughput because aborts are rare. However, the extra data copying of OCC limits the maximum throughput of the system when the data store is large or the access pattern exhibits weak locality. DecentSched is able to retain high throughput and low tail latency in this scenario. This is achieved by its decentralized scheduling design. On the one hand, the queuing phase in DecentSched has a low constant cost. On the other hand, the scheduling in DecentSched incurs minimal interaction between concurrent transactions. Once the order is secured, the scheduling rules and opportunistic execution can exploit parallelism immediately.

In more contentious workloads (Figures 5a, 5b, 6a, and 6b), DecentSched shows good performance compared to the baseline protocols in most experiments. Its advantage is especially prominent when transactions are long, which causes a higher abort penalty. Transactions in DecentSched have deterministic global orders so that concurrency control induced aborts are eliminated. Therefore, the end-to-end latency of DecentSched is well controlled and the system throughput is maintained at a high level. In some of these experiments, OCC-based protocols can achieve higher throughput than DecentSched. However, this comes with the cost of drastically higher end-to-end latency, which is a key factor that contributes to the system responsiveness.

V. Related Work

a) Concurrency Control Protocols: There are abundant studies on extending the original 2PL and OCC protocol. For 2PL, several approaches propose violating the original 2PL protocol to extract more parallelism from the workloads [1, 18, 19, 23]. Bamboo [19] is a state-of-the-art solution which allows transactions to read data from uncommitted transactions and it uses extra metadata on locks to capture the new read-uncommitted dependencies to ensure serializability. However, it suffers from overwhelming cascading aborts with long transactions. In the line of work on OCC, Silo [39, 45] presents an efficient transaction ID allocation mechanism to avoid contentions so that the validation phase of transactions can be done efficiently. TicToc [43] targets the centralized timestamp allocation bottleneck and provides a solution based on decentralized and postponed timestamp allocation for better system scalability. Several previous works exploit the isolation provided by MVCC [25, 26, 30, 41]. They require maintaining historical versions of data using extra space and actively reclaiming memory [5], which is different from 2PL and OCC. There are also a lot of works focusing on concurrency control under a distributed setup [7, 14, 28, 40, 44]. They inevitably induce aborts and retries of conflict transactions. DecentSched lets individual transactions derive their serializable execution order in a deadlock-free way. Concurrency control protocol induced aborts are not possible with DecentSched when processing deterministic transactions.

b) Transaction Scheduling: Instead of applying concurrency control to individual transactions, prior studies also proposed processing transactions in batches and executing them in generated schedules. This approach necessitates foreknowledge of a transaction's read and write key sets to generate an optimal schedule. Consequently, this technique is primarily suitable for deterministic transactions, and the likelihood of concurrency control protocol-induced aborts is decreased. To support general transactions without known read/write key sets, reconnaissance queries may be utilized [33, 36, 37] at the cost of abort-freedom. QueCC [32] uses separate scheduling threads and executing threads to process transactions. It adopts a priority queue oriented approach to dispatch transactions to different partitions to avoid aborts. PWV [17] breaks transactions into multiple atomic pieces and builds a dependency graph to schedule and commit transactions in finer granularity. Similar ideas are also applied in various other works [13, 16, 29, 31] where transactions are executed in finer-grained pieces and/or separate phases. This type of approach is also actively used in distributed systems that require high determinism to reduce the cost of inter-node synchronization and crash recovery [8, 27, 37], which is out of the scope of this paper. DecentSched adopts the idea of transaction scheduling but the scheduling is done by individual transactions instead of large batches.

VI. Conclusion

In this paper, we present DecentSched, a new concurrency control protocol that provides high determinism for deterministic transaction processing systems. It eliminates aborts induced by concurrency control protocols with low cost using its queuing-based approach and decentralized scheduling algorithm. Extensive evaluations show that it can outperform state-of-the-art concurrency control protocols with higher throughput and lower tail latency in both YCSB and TPC-C benchmarks.

References

[1] Divyakant Agrawal, Amr El Abbadi, Richard Jeffers, and Lijing Lin. "Ordered shared locks for real-time databases". In: *The VLDB Journal* 4.1 (1995), pp. 87–126.

[2] *Bamboo Source Code*. https://github.com/ScarletGuo/Bamboo-Public.

[3] Philip A. Bernstein and Nathan Goodman. "Concurrency Control in Distributed Database Systems". In: *ACM Comput. Surv.* 13.2 (1981), pp. 185–221.

[4] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[5] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Scalable Garbage Collection for In-Memory MVCC Systems". In: *Proc. VLDB Endow.* 13.2 (2019), pp. 128–141.

[6] M.J. Carey. “Improving the Performance of an Optimistic Concurrency Control Algorithm Through Timestamps and Versions”. In: *IEEE Transactions on Software Engineering* SE-13.6 (1987), pp. 746–751.

[7] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. “Fast and General Distributed Transactions Using RDMA and HTM”. In: *Proceedings of the Eleventh European Conference on Computer Systems*. 2016.

[8] Pierpaolo Cincilla, Sébastien Monnet, and Marc Shapiro. “Gargamel: boosting DBMS performance by parallelising write transactions”. In: *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. IEEE. 2012, pp. 572–579.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking Cloud Serving Systems with YCSB”. In: *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC’10)*. 2010, pp. 143–154.

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. “Spanner: Google’s Globally Distributed Database”. In: *ACM Trans. Comput. Syst.* 31.3 (2013).

[11] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. “Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 33–48.

[12] *DBx1000 Source Code*. https://github.com/yxymit/DBx1000.

[13] Bailu Ding, Lucja Kot, and Johannes Gehrke. “Improving Optimistic Concurrency Control through Transaction Batching and Operation Reordering”. In: *Proc. VLDB Endow.* 12.2 (2018), pp. 169–182.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. “No Compromises: Distributed Transactions with Consistency, Availability, and Performance”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015, pp. 54–70.

[15] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. “The Notions of Consistency and Predicate Locks in a Database System”. In: *Commun. ACM* 19.11 (1976), pp. 624–633.

[16] Jose M. Faleiro and Daniel J. Abadi. “Rethinking Serializable Multiversion Concurrency Control”. In: *Proc. VLDB Endow.* 8.11 (2015), pp. 1190–1201.

[17] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. “High Performance Transactions via Early Write Visibility”. In: *Proc. VLDB Endow.* 10.5 (2017), pp. 613–624.

[18] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. “Controlled Lock Violation”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 2013, pp. 85–96.

[19] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. “Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking”. In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 658–670.

[20] Theo Härder. “Observations on Optimistic Concurrency Control Schemes”. In: *Inf. Syst.* 9.2 (1984), pp. 111–120.

[21] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. “Opportunities for Optimism in Contended Main-Memory Multicore Transactions”. In: *Proc. VLDB Endow.* 13.5 (2020), pp. 629–642.

[22] *jemalloc*. https://github.com/jemalloc/jemalloc.

[23] Hideaki Kimura, Goetz Graefe, and Harumi A Kuno. “Efficient locking techniques for databases on modern hardware.” In: *ADMS@ VLDB*. Citeseer. 2012, pp. 1–12.

[24] H. T. Kung and John T. Robinson. “On Optimistic Methods for Concurrency Control”. In: *ACM Trans. Database Syst.* 6.2 (1981), pp. 213–226.

[25] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. “Multi-Version Range Concurrency Control in Deuteronomy”. In: *Proc. VLDB Endow.* 8.13 (2015), pp. 2146–2157.

[26] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. “Cicada: Dependably Fast Multi-Core In-Memory Transactions”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 21–35.

[27] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. “Aria: A Fast and Practical Deterministic OLTP Database”. In: *Proc. VLDB Endow.* 13.12 (2020), pp. 2047–2060.

[28] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. “Extracting More Concurrency from Distributed Transactions”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. 2014, pp. 479–494.

[29] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. “Phase Reconciliation for Contended In-Memory Transactions”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. 2014, pp. 511–524.

[30] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. “Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 677–689.

[31] Guna Prasaad, Alvin Cheung, and Dan Suciu. "Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 527–542.

[32] Thamir M. Qadah and Mohammad Sadoghi. "QueCC: A Queue-Oriented, Control-Free Concurrency Architecture". In: *Proceedings of the 19th International Middleware Conference*. 2018, pp. 13–25.

[33] Dai Qin, Angela Demke Brown, and Ashvin Goel. "Caracal: Contention Management with Deterministic Concurrency Control". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 180–194.

[34] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. "Design Principles for Scaling Multi-Core OLTP Under High Contention". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 1583–1598.

[35] Kun Ren, Dennis Li, and Daniel J. Abadi. "SLOG: Serializable, Low-Latency, Geo-Replicated Transactions". In: *Proc. VLDB Endow.* 12.11 (2019), pp. 1747–1761.

[36] Kun Ren, Alexander Thomson, and Daniel J. Abadi. "An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems". In: *Proc. VLDB Endow.* 7.10 (2014), pp. 821–832.

[37] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. "Calvin: Fast Distributed Transactions for Partitioned Database Systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 1–12.

[38] *TPC-C Benchmark*. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[39] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. "Speedy Transactions in Multicore In-Memory Databases". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 18–32.

[40] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. "Fast In-Memory Transaction Processing Using RDMA and HTM". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015, pp. 87–104.

[41] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. "An Empirical Evaluation of In-Memory Multi-Version Concurrency Control". In: *Proc. VLDB Endow.* 10.7 (2017), pp. 781–792.

[42] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. "Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores". In: *Proc. VLDB Endow.* 8.3 (2014), pp. 209–220.

[43] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. "TicToc: Time Traveling Optimistic Concurrency Control". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 1629–1642.

[44] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. "Building Consistent Transactions with Inconsistent Replication". In: *ACM Trans. Comput. Syst.* 35.4 (2018).

[45] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. "Fast Databases with Fast Durability and Recovery through Multicore Parallelism". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. 2014, pp. 465–477.