No False Negatives: Accepting All Useful Schedules in a Fast Serializable Many-Core System

Dominik Durner
Technical University of Munich
dominik.durner@tum.de

Thomas Neumann
Technical University of Munich
thomas.neumann@tum.de

***Abstract*—Concurrency control is one of the most performance critical steps in modern many-core database systems. Achieving higher throughput on multi-socket servers is difficult and many concurrency control algorithms reduce the amount of accepted schedules in favor of transaction throughput or relax the isolation level which introduces unwanted anomalies. Both approaches lead to unexpected transaction behavior that is difficult to understand by the database users.**

We introduce a novel multi-version concurrency protocol that achieves high performance while reducing the number of aborted schedules to a minimum and providing the best isolation level. Our approach leverages the idea of a graph-based scheduler that uses the concept of conflict graphs. As conflict serializable histories can be represented by acyclic conflict graphs, our scheduler maintains the conflict graph and allows all transactions that keep the graph acyclic. All conflict serializable schedules can be accepted by such a graph-based algorithm due to the conflict graph theorem. Hence, only transaction schedules that truly violate the serializability constraints need to abort. Our developed approach is able to accept the useful intersection of commit order preserving conflict serializable ($COCSR$) and recoverable (RC) schedules which are the two most desirable classes in terms of correctness and user experience. We show experimentally that our graph-based scheduler has very competitive throughput in pure transactional workloads while providing fewer aborts and improved user experience. Our multi-version extension helps to efficiently perform long-running read transactions on the same up-to-date database. Moreover, our graph-based scheduler can outperform the competitors on mixed workloads.

I. Introduction

Concurrency control is one of the most important tasks of a database management system (DBMS). It is used to achieve the ACID (atomicity, consistency, isolation, and durability) properties. Although multiple concurrent tasks can run simultaneously, transaction isolation creates the illusion of being the only user of the database. This illusion is enforced with the help of the concept of serializability. Conflict serializability is the theoretical foundation that focuses on finding a serial schedule with equal read-write, write-read, and write-write tuple access pairs. The goal of concurrency control protocols is to allow as many theoretically possible conflict serializable schedules without reducing the query performance.

Although serializable transaction processing is a great concept, it seems hard to implement efficiently. Previous approaches need to trade maximum transaction throughput for allowed concurrency (reducing the number of allowed schedules). The classical way to ensure serializability is Two-Phase Locking (2PL) which is easy to implement [1]. Many traditional commercial systems, such as DB2, use two-phase schedulers [2]. Pessimistic protocols, such as 2PL, work well on highly contented workloads but incur severe overhead in read-mostly database systems. As a consequence the prominent class of concurrency control algorithms that are based on timestamps was developed. These schedulers work well in low contention workloads but introduce unwanted and unnecessary aborts if the workload is conflict heavy. Optimistic concurrency control protocols, which are often used in modern database systems, schedule many unnecessary aborts [3], [4], [5].

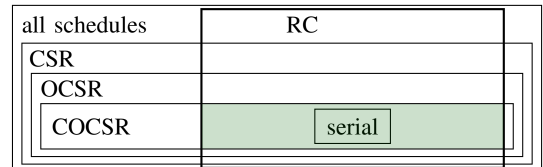


Fig. 1: Our approach accepts the useful intersection of $COCSR$ and RC schedules for correctness and superior user experience.

A particular challenge arises with long-running read transactions which are common in online analytical processing (OLAP) workloads. Many modern systems have either reduced online transaction processing (OLTP) throughput or use isolation levels that are more relaxed than serializable. These systems rely on multi-version concurrency control (MVCC) that retains the data in multiple versions which helps to process long-running readers [6], [7]. To achieve higher throughput most systems only guarantee the relaxed isolation level of Snapshot Isolation (SI). SI provides a high level of isolation; however, there are schedules that are not serializable but are allowed in SI and this results in unwanted and hazardous anomalies [8], [9].

Moreover, traditional concurrency control algorithms do not scale with new hardware trends. In particular, the development of many-core server architectures, which have more than hundred general-purpose cores, introduces many pitfalls. For example, optimistic algorithms often rely on an exclusively executed verification phase at commit time. Unfortunately, the high contention on the commit phase leads to very poor throughput performance. As a result, most DBMS perform poorly on OLTP workloads and new algorithms are required to help overcome such fundamental performance bottlenecks.

In this paper, we present a novel scheduler that achieves high throughput on many-core systems while reducing aborted schedules to a minimum. It is based on the concept of maintaining a conflict graph to detect serializability issues [10].

The graph-based scheduler was never really considered for practical applications as it seemed to be too expensive to continuously maintain an acyclic graph. However, we show that this assumption does not hold anymore because of modern multi-core processors and our innovative graph structure. Our key contributions are a novel graph structure that only needs transaction local locks for commit critical operations and our read-only multi-version concept for long-running read queries. With our new graph structure and modern hardware it is possible to scale the graph operations and achieve very high throughput performance for both OLTP and OLAP workloads. The conflict graph theorem asserts that the history is serializable if and only if the conflict graph is acyclic. Our approach is able to basically accept the most desirable intersection of commit order preserving conflict serializable ($COCSR$) and (log-) recoverable (RC) schedules which is shown in Figure 1. Other concurrency control algorithms can only accept a smaller class of these user wanted schedules. In practice, these schedulers use heuristics, such as 2PL or timestamp ordering, to approximate the class of serializable schedules. This leads to user visible but hard to explain behavior artifacts. We minimize aborted schedules, which require restarting, to avoid unnecessary system load. Aborted transactions are very unpleasant since user code or in the worst case the human user needs to handle these transaction failures. Our transaction system provides the highest isolation level of conflict serializability and does not compromise isolation for performance. Moreover, our findings can refute the assumption that the usage of graph-based schedulers as concurrency control algorithms is impractical [11].

The rest of this paper is structured as follows: Section II formalizes the properties of the single-version graph-based concurrency control algorithm and shows the steps necessary for achieving high performance on many-core systems. After discussing the theoretical properties of our approach, the long-running read optimized version is described in Section III. The algorithms are evaluated in Section IV according to multiple benchmarks and workloads. Section V discusses the related work before we draw conclusions in Section VI.

II. Graph-based Concurrency Control

The class of conflict serializable schedules (CSR) can be represented entirely with acyclic conflict graphs, as a schedule is conflict serializable if and only if the corresponding conflict graph is acyclic [6]. Therefore, we want to exploit this theoretical property by introducing our novel scheduler that maintains the conflict graph. Only transactions that break the conflict serializability properties need to abort. One challenge with the graph-based approach arises from the regular single-threaded execution of graph operations. The single-threaded conflict graph scheduler has the same unpleasant execution paths as other concurrency control algorithms such as the validation phase in optimistic protocols. In this work, we eliminate the performance problems of the graph-based scheduler by developing a many-core ready parallel graph structure without globally synchronized paths. We first give a short overview of the foundations of such a graph-based scheduler. Afterwards, we show the novel many-core optimizations that help scale the graph operations.

A. *Conflict Graph Properties*

Our scheduler is based on the idea of maintaining an acyclic conflict graph of the current schedule s which is denoted as $CG(s)$. Every transaction is represented by a node in the graph. Conflicting operations a_i of t_i and b_j of t_j of the form $a_i[x] <_s b_j[x], t_i \neq t_j$, with $(a, b) \in \{(r, w), (w, r), (w, w)\}$, are represented as an edge of the form (t_i, t_j).

Theorem 1 (Conflict Graph Theorem)**.**

$$s \in CSR \Leftrightarrow CG(s) \textit{ is acyclic}$$

The main proof idea of the conflict graph theorem is the existence of a serial ordering such that all conflict edges can be satisfied. A serial schedule with the same conflict pairs as the current executed schedule exists iff the graph is acyclic. The theorem can be proven by ordering the transactions according to their topological ordering in the graph.

The concept of using the conflict graph as scheduler is known as Serialization Graph Testing (SGT) [10], [12]. Each operation first inserts all conflicting edges into the graph and only if the graph is still acyclic it is allowed to be executed. To ensure the acyclic property of the directed acyclic graph, cyclic dependencies result in aborts of transactions. If a cycle between two nodes exists, either transaction would need to be ordered before the other one in a conflict equivalent serial schedule. This impossible requirement shows that the schedule is not conflict serializable anymore. Due to the direct link to the conflict graph theorem, the SGT approach guarantees to accept all possible conflict serializable schedules. The main reason why SGT hitherto seemed to be impractical was the costs for cycle checking [11]. But modern multi-core processors allow us to efficiently schedule cycle checks nowadays. Especially, our novel many-core graph structure, that is able to perform simultaneous accesses in parallel, is a major benefit in comparison to the original SGT and many other modern concurrency control algorithms.

The SGT approach relies on conflict pairs to ensure a serializable execution. Therefore, the database system must guarantee that the execution of conflicting operations is in a fixed order. Each operation that affects a specific row is assigned an ordering local to a data element. Operations can only introduce conflicts on the same data element. Thus, the local ordering is sufficient to derive all conflict pairs for the scheduled operation. Further, all previous accesses to the tuple are stored within the tuple access history such that the different conflict pairs can be derived from the access history.

One central idea of SGT is that only transactions that are involved in a cycle actually break the conflict serializability constraints. The cyclic dependencies need to be resolved by aborting the node introducing the cycle. Transactions, denoted as t_j, that had a w-r / w-w edge from the aborted node t_a (edge of the form (t_a, t_j)) need to abort as well because t_j based a decision on a dirty value that should not have existed after all.

After these aborts the rest of the schedule is serializable. Livelocks can be detected and resolved as the aborting transaction knows the edges to its parents.

B. Contribution Overview

In the following we discuss our high-performance graph-based scheduler. First, the graph structure that allows concurrent accesses of different transactions is explained in detail. Optimizations such as the decoupling of the access history and deletions of transaction nodes are shown as well. After the algorithmic analysis, a short example highlights our approach. We argue about different approaches for the performance critical cycle checks to achieve maximum throughput. Before we conclude the section with the correctness analysis, all the important implementation details are explained.

C. Design Principles

In this section, we show the key changes in the algorithm that are necessary to fulfill the properties of an order preserving and recoverable concurrency control protocol. Moreover, the introduced changes help us to efficiently scale our graph-based scheduler.

Example 1 (Commit Requirements of SGT nodes)**.**

$$s_1 = r_0[x]\, w_0[x]\, r_2[x]\, w_2[x]\, r_1[x]\, r_1[y]\, w_1[y]\, c_1 \ldots c_0\, c_2$$

One pitfall of SGT is the timing of node deletions. The simple approach of deleting committed nodes can introduce serializability issues but a delayed deletion strategy can avoid these problems while reducing the memory consumption significantly. A committed node, denoted as node t_c, is not allowed to be removed from the graph before all edges of the form (t_i, t_c) have been removed. Without this requirement, a possible cycle after the commit remains undetected. In Example 1, transaction t_1 is not allowed to be removed from the graph after its commit c_1. If t_2 additionally requests $r_2[y]$ before the commit c_2 in s_1, the schedule s_1 would not be serializable anymore and a cycle between nodes $t_1 \leftrightarrow t_2$ would be inserted. Hence, both transactions schedule a write before the other one's read and no serial schedule can be found. If node t_1 is removed, the serialization graph does not encounter the cycle and the serialization issue would be undetected. Note that the commit of t_1 is allowed, just the deletion must be postponed in the original SGT definition.

Additionally, some schedules should not be allowed, in comparison to the regular SGT definition, to ensure the important recoverability property. The class of recoverable schedules is not directly related to conflict serializability. As a consequence, additional restrictions need to be enforced to ensure ACID properties. Every transaction needs to wait before committing until it does not depend on any other in-flight transaction. Thus, a transaction must not have uncommitted incoming write-read / write-write edges at commit time. Example 1 also shows a non log-recoverable schedule. t_1 is not allowed to commit before t_2. Otherwise, t_1 would base a decision on a read value that never existed if t_2 aborts. In the following, the even stronger property, that a transaction t_c cannot commit until all edges of the form (t_i, t_c) are removed, is ensured. Although read-write edges do not run into direct log-recoverability issues, these edges are treated the same as the other edges to generate only order preserving schedules. Assume $s_2 = r_1[x]\ w_2[x]\ c_2\ r_3[y]\ c_3\ w_1[y]\ c_1$. If the read-write edges would not lead to commit delays, the schedule s_2 would be accepted although it is not order preserving since $s_2' = t_3\ t_1\ t_2$. Our changes introduced to achieve recoverable and order preserving schedules also solve the deletion problem because every committed node can be deleted directly.

D. Many-Core Performance Tuning

A key challenge of using the serialization graph as concurrency control protocol is the scalability issues on multi-core servers that can have hundreds of general purpose cores. The simple approach of just locking the graph completely while a thread inserts edges or performs a cycle check has too many single-threaded synchronized paths. Therefore, this simple strategy does not perform well on multi-core servers although its theoretical properties are optimal.

We developed a local locking graph structure that allows for concurrent insertions and safe cycle checks. Hence, we can overcome the performance problems of a single-threaded graph-based scheduler while achieving the good theoretical properties of the graph-based scheduler. The proposed strategy uses locks on transaction granularity to ensure that all edges, possibly involving the transaction within a cycle, are inserted. This requirement must be met for safe commits and aborts, otherwise a concurrent edge insertion could introduce an undetected cycle or prohibit nodes from committing. For example, a transaction t_c is in the process of committing and has already deleted all outgoing edges. Another transaction t_k might read a value written by t_c and therefore introduces the edge (t_c, t_k) according to an inserted conflict. Thus, node t_k introduces an edge after the deletion routine is already finished and consequently the transaction t_k would always see the edge (t_c, t_k) although the node t_c does not exist anymore.

To prevent the issues arising from concurrent insertions and cycle checks, the transaction local locks have two modes; a shared insert / concurrent cycle check mode, and a final exclusive cycle check mode. During the shared lock, multiple threads can access the transaction's node for insertions and cycle checks. The final cycle check to determine whether a transaction is allowed to be committed is executed in exclusive mode. No parallel inserts can be performed introducing the above mentioned anomalies. So, either a conflict pair is inserted before the commit / abort or the conflict pair does not exist anymore since an inserting thread needs to wait until the commit is finished and the node has been deleted.

A deleted node t_d must still be accessible until no transaction t_i can possibly find a conflict and the corresponding edge (t_d, t_i) according to any tuple access history (reads and writes on a tuple). Each node needs to store the current state of the transaction (committed, aborted, running) for concurrently inserting transactions. If a node t_k is committed and another transaction t_i wants to access one of the data elements touched

Algorithm 1: Edge Insertion

```
input  : Node thisNode, Node fromNode, bool rwEdge
output : Boolean isStillSerializable
if {fromNode, rwEdge} ∉ thisNode.inSet then
    lockSharedGuard(fromNode)
    if fromNode.state == aborted && !rwEdge then
        return false
    else if fromNode.state == committed then
        return true
    thisNode.inSet.add({fromNode, rwEdge})
    fromNode.outSet.add({thisNode, rwEdge})
    cycle = cycleCheckFrom(thisNode)
    return !cycle
else
    return true //Edge already exists
end
```

Algorithm 2: Commit

```
input  : Node thisNode
output : Boolean successfullyCommitted
lockExclusiveGuard(thisNode)
if thisNode.inSet ≠ {} then
    return false
foreach outNode ∈ thisNode.outSet do
    lockSharedGuard(outNode)
    thisNode.outSet.remove({outNode})
    outNode.inSet.remove({thisNode})
end
thisNode.state = committed
epochGarbageCollector.scheduleDelete(thisNode)
return true
```

by t_k, the transaction t_i can perform its operation without actually inserting the edge (t_k, t_i). If the transaction t_k is aborted, the other transaction t_i needs to abort as well (w-r / w-w edge) since the undo of the abort might not be finished yet and the possible (t_k, t_i) edge indicates an access to a modified tuple. The existence of the removed node can be assured with an epoch based garbage collector that is also used for the concurrent data structures as well as for the multi-version graph-based scheduler, discussed later.

The edges in our many-core optimized graph are represented by sets of pointers to other transaction nodes. Each node consists of two sets of pointers that represent incoming and outgoing edges of this transaction. The edge sets need to guarantee thread-safe concurrent accesses (scans, insertions, and deletions) during the shared lock. If an edge is already present in the graph, the graph does not change and we do not need to perform any additional work. Otherwise, we acquire the shared lock such that the other transaction cannot be in its commit process. Due to the parallel execution, we check whether the node is still alive and not in the process of being collected from the garbage collector. If the other transaction is not finished yet, we establish the new edge by inserting it into the respective sets of incoming and outgoing node pointers. A final cycle check determines whether the schedule is still serializable. Algorithm 1 shows the edge insertion procedure.

For commit handling, we need to recall the requirements for recoverability. Following from the theoretical analysis, the node needs to be without incoming edges to restrict the set of allowed schedules to only recoverable ones. First, the transaction's own exclusive lock needs to be acquired to cope with concurrent transactions that want to access the node pointer sets. Because no incoming edges are allowed for recoverability, we can simplify the cycle check to a check for no incoming edges. If a transaction has no incoming edges, it obviously cannot lie on a cycle. All outgoing edges are deleted after acquiring the shared locks for the outgoing transactions. Since the shared locks are only acquired after the no incoming edges check, no deadlocks can be introduced by concurrent commit attempts. Finally, the node is added to the garbage collector that assures a deletion after no other thread can see an edge to this node anymore. The resulting commit process is shown in Algorithm 2.

Our graph structure is therefore suitable for many-core systems since final serializability checks do not need to traverse large parts of the graph and insertions into the graph only require transaction local locks to guard the critical commit procedure of the other transaction instead of a global mutex.

E. Optimized Graph Structure Example

In the following, we explain our proposed optimized graph model with a small example, whose schedule is shown as transaction timeline on the left side of Figure 2. The graph representing the state of the green highlighted area is illustrated on the right-hand side of the same figure. Transaction t_0 just acquired the exclusive lock to determine that its node has no incoming edges and all outgoing edges are safely inserted into the graph. Every other transaction cannot acquire a shared lock on t_0, thus no other transactions can currently perform edge insertions, edge deletions, or cycle checks involving t_0. Furthermore, transaction t_1 cannot positively validate the check for no incoming edges as long as transaction t_0 still points to it. Therefore, t_1 needs to wait until the commit of t_0 and the deletion of the edge (t_0, t_1). At this point in time, transaction t_2 is performing a read on value z. As a result, t_2 finds the conflict pair with t_1. The edge from t_1 is inserted by acquiring the shared lock of t_1 and adding the respective other node into the incoming and outgoing edge sets of the involved transactions. Because the insertion of the edge (t_1, t_2) is executed in shared lock, concurrent transactions accessing t_1 or t_2 might not be able to detect the edge at the moment. For instance, a concurrent cycle check could also acquire a shared lock on t_1 but the edge might not be inserted yet into the outgoing edge set. Hence, the final check for no incoming edges in exclusive mode is necessary to guarantee serializability.

F. Conflict Detection

The graph-based scheduler needs to derive conflict pairs to find serializability issues. Conflict detection requires an ordered access history for each data element. The ordering and storage of the accesses is guaranteed with two additional columns for each relation — a local sequence column and an ordered list of tuple access entries. An example of such a relation can be found in Figure 3. Algorithm 3 uses the tuple access history to represent the conflict pairs. The list orders the data accesses on a single data element and is therefore sufficient to detect

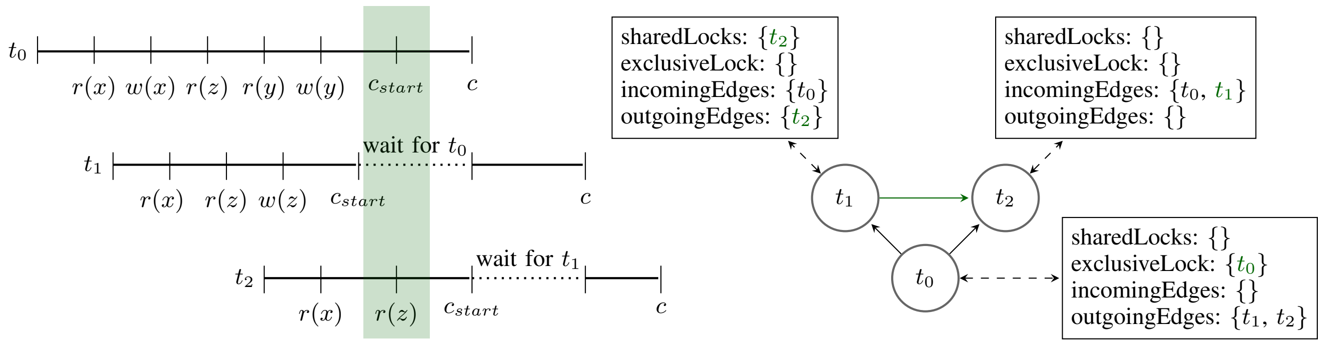


Fig. 2: Example of the transaction local locking graph

A	B	SeqNo	Op	{Transaction, Operation}
1	x	5	*	→ {0,w} → {1,r} → {1,w} → {2,r}
2	y	4	*	→ {0,w} → {2,r} → {2,w}
3	z	2	*	→ {0,w}

Fig. 3: Example relation and operations in the access list

Algorithm 3: Conflict Pair Detection

```
input  : Transaction tx, Operation op, List accessHistory, Cell
         seqNoEntry, SGT sgt
currentId = accessHistory.push(tx, op)
while seqNoEntry ≠ currentId do
  | wait()
end
foreach elem ∈ accessHistory do
  | if elem.id < currentId then
  |   | if isInConflict(elem.op, op) then
  |   |   | abort(!sgt.EdgeInsertion(elem.tx, tx)) (Alg 1)
  |   end
end
```

the conflict pairs. The local sequence numbering is used to determine the next operation that is allowed to be scheduled in the access list and then executed on the data element. The edges of the graph can simply be calculated by scanning over the access history list of all relevant rows. An edge is inserted into the graph if the other operation was scheduled beforehand and is in conflict with the current operation. The ordering is guaranteed by a spin-lock that waits until the sequence number matches the list position of the transaction's current operation. Note that these spin-locks just guard the read or write operation and are not hold until commit time of the transaction. Immediately after the tuple access, the sequence number is increased to allow the next transaction to operate on the tuple. Hence, multiple threads accessing the same elements do not need to wait for the other transaction to finish which is a major benefit in comparison to locking based schedulers.

G. Cycle Checking Strategies and Graph Scalability

The regular offline cycle checking algorithm for our SGT approach leverages a modified DFS search. For serialization graph testing it is actually sufficient to just traverse the part of the graph that needs to be validated. All other cycles within the graph do not lead to a direct abort of the transaction because no ordering constraints for this transaction are violated. In comparison to the final check for no incoming edges as shown in Algorithm 2, the cycle check only holds nodes in shared

Algorithm 4: Cycle Check with (reduced) DFS

```
input  : Node& currentNode, Set& visitedPath
visitedPath.insert(currentNode)
lockSharedGuard(currentNode)
if currentNode.isAlive() then
  | foreach inNode ∈ currentNode.inSet do
  |   | if visitedPath.contains(inNode) then
  |   |   | return true
  |   | else if cycleCheck(inNode, visitedPath) then
  |   |   | return true
  |   end
end
visitedPath.erase(currentNode)
return false
```

lock. As a consequence, concurrent edge insertions are possible such that a simultaneously inserted cycle might be undetected. Therefore, we require that the commit-critical final check for no incoming edges is guarded by an exclusive lock. On the other hand, cycles that are introduced concurrently will be detected with a following check or with an abort request from another transaction. Moreover, this helps to scale our approach to many cores because only the final check for no incoming edges shortly blocks other threads from accessing a specific node. Note that only nodes with incoming edges might be blocking shortly. However, the incoming edge check is the first operation and results in a direct release of the lock if it was negative. Algorithm 4 shows the reduced DFS cycle detection.

In recent years, new online algorithms were proposed that can be used to detect cycles in a directed graph. These online algorithms can reduce the worst-case runtime by maintaining a topological ordering of the graph [13], [14]. The algorithm developed by Pearce et. al. does not rely on custom and hard to parallelize data structures in contrast to the other proposed algorithms. First, it computes a forward oriented DFS and a backwards oriented DFS from the inserted edge. Afterwards, the new topological ordering is calculated and assigned atomically. Assuming an edge from $x \rightarrow y$ is inserted, the forward oriented DFS starts from y, whereas the backward oriented one starts from x. Hence, the algorithm only considers the affected region of nodes. The real advantage of this algorithm, compared with our optimized regular offline based cycle checker, is the number of DFS executed. The computations are only triggered if the ordering numbers between these two nodes are in the wrong order, so only if $ord(y) < ord(x)$. Unfortunately, multiple updates on the graph could break the topological ordering

requiring a single-threaded execution. However, maintaining and updating the topological order is expensive on many-core machines.

Today, database systems often span multiple nodes for increased availability and performance. Due to the scaling capabilities (Algorithm 4), our graph approach can be extended to leverage multiple nodes by introducing a special node type that indicates a remote server. Each server stores a local graph that represents conflicts between its local transactions but has incoming and outgoing edges from remote servers. The remote server node needs a mapping between the remote transactions and the host transactions such that all possible conflict edges can be determined due to multi-edges from different remote transactions. Since the final commit only considers the current node, the commit process remains unchanged. For edge insertion and cycle checks, remote accesses to the other server must be possible to acquire shared locks on remote transaction nodes and to traverse the fully connected graph (e.g. RDMA). With a partitioning scheme where conflicts mostly occur locally, only a few transactions need to traverse parts of the graph that are located on remote nodes such that the overall latency remains small. This is an interesting avenue for future work.

H. Implementation Details

Our many-core approach depends on massively parallel data structures. Our developed data structures use atomic operations to guarantee high operation throughput in a multi-access workload. Most concurrency control protocols require totally ordered transaction IDs to guarantee serializability. However, this leads to bottlenecks for myriads of small OLTP transactions due to many updates of global atomic counters. In our approach we can simply use the address of the transaction's node in the graph as the transaction ID to avoid the global counter problem. Furthermore, we can reuse these nodes, including their edge sets, after the commit and deletion of the transaction.

Commits are only possible if the transaction has no incoming edges and must be postponed otherwise. After every unsuccessful commit process a cycle check determines whether the transaction is now able to commit. In our implementation we repeatedly re-check the ability to commit because every transaction is pinned to a single hardware thread.

The graph-based scheduler allows to optimistically read dirty records which will introduce cascading aborts. Before removing the w-r / w-w edges from an aborted node, the child nodes are simply marked for abortion. At the time the children have no incoming edges, they are already marked and cannot commit anymore. If multiple uncommitted transactions could modify one data element, which is possible in the original SGT definition, the undo of transactions must be synchronized. In particular, the transactions need to undo their actions in the reverse order of the data element accesses. Accesses on different data elements might happen in an arbitrary ordering between multiple transactions. Hence, these cascading aborts lead to unnecessary system load and are not useful for both the user and the system. Therefore, only one uncommitted transaction is allowed to modify a data element.

I. Serializability Properties and Correctness

Because the theoretical aspects of SGT are well studied, we only need to show that every accepted (committed) transaction created by our SGT implementation would also be accepted by the theoretical concept of SGT. In the following $Gen(algo)$ represents the class of schedules an algorithm generates.

Theorem 2.

$$Gen(\textit{Our Many-Core Optimized SGT}) \subseteq CSR$$

Proof. Previous work shows that $Gen(SGT) = CSR$. Further, the nodes and edge insertions into the graph remain the same also for our approach. First, we show that our implementation cannot encounter a cycle while committing. The transaction that wants to commit must lock its own node exclusively. A successful lock guarantees that no other transaction currently holds any lock on this node nor will any lock request be accepted. Consequently, the node cannot be involved in a cycle check or edge insertion currently. If the node has no incoming edges, it is assured that the node is not on a cycle and no edges are in flight at the moment. Concurrent transactions that wait for the committing transaction could only introduce outgoing edges. Hence, it is allowed to commit according to the basic SGT idea and the waiting transactions can access the tuple logically after the commit.

Second, at the time the node has no incoming edges, it is also allowed to be removed from the graph according to the SGT definition without introducing cycles. After the commit no edges to the node can be introduced anymore and all outgoing edges of the committed transaction can also be deleted. □

Besides only accepting log-recoverable schedules our approach has the pleasant property of providing schedules that always have the same serialization order and commit order. A conflict equivalent schedule could just rearrange a later committed transaction before an already committed one without violating CSR constraints. On the other hand, users expect to schedule later transactions after the committed changes. User applications often rely on the order of executed statements (applications are usually sequential). Such schedules that restrict the reordering are also known as order preserving conflict serializable ($OCSR$). The commit order is often expected to be aligned to the serialization order which is known as commit order preserving ($COCSR$). We allow all possible interleavings of read and write operations that are CSR valid but might delay a transaction's commit to ensure the order preserving properties. Therefore, we also consider read-write edges before committing as explained in Section II-C.

Traditional schedulers such as strict 2PL also produce a subset of $COCSR$ but our approach is able to accept a larger fraction of order preserving schedules. For instance, $s_1 = r_1[x]\ w_1[x]\ r_2[x]\ r_2[z]\ w_2[z]\ r_3[y]\ w_3[y]\ c_3\ r_1[y]\ w_1[y]\ c_1\ c_2$ cannot be accepted by a two-phase scheduler because of the conflict between transaction t_1 and t_2, whereas our SGT implementation could accept this operation interleaving. If $s_2 = r_1[x]\ w_1[x]\ r_2[x]\ r_2[z]\ w_2[z]\ c_2\ r_3[y]\ w_3[y]\ c_3\ r_1[y]\ w_1[y]\ c_1$,

with $s_2 \in CSR$ and $s_2 \notin OCSR \Rightarrow s_2 \notin COCSR$ ($s'_2 = t_3\ t_1\ t_2$) is requested, our SGT implementation would just delay the commit of t_2. In particular, the schedule s_1 would be executed on our system which is also order preserving conflict serializable. S2PL (Strict 2PL) is log-recoverable but requires that all write locks are held until commit time. The two-phase property and the write lock rule accept fewer schedules than our approach, i.e., S2PL cannot generate a transaction interleaving that would be rejected by our graph-based scheduler. Our approach generates the useful schedules of $COCSR \cap RC$ because of the introduced commit delays. Schedules affected by heavy cascading aborts (multiple uncommitted writers on a tuple) are deliberately restricted to achieve low latency.

III. Multi-Version Extension

Having explained the traditional single-version graph-based scheduler, we briefly discuss optimizations that help to increase concurrency. Traditional concurrency control algorithms have difficulties in executing long-running readers simultaneously with update-heavy transactions. Previous work shows that multi-version concurrency control (MVCC) helps to address the problem of long-running read-mostly transactions. The difference to the traditional single-version approach is that every data element is now accessible in different versions and the scheduler has to decide which version to read. In our design we deliberately favor long-running read-only queries, which are the most prominent ones in OLAP workloads.

Writes store the updated data regulary in the base table but snapshot the row beforehand which creates a list of previous states of the row. Every snapshot is assigned with the current epoch state that is used within the epoch based garbage collector of the database system. The epoch state is used similarly to transaction ids in classical multi-version schedulers to determine the access of the correct version. In comparison to increasing transaction ids, the epoch state is more coarse-grained. Thus, it does not introduce a single update bottleneck because not every transaction requires an increment.

Write transactions and short readers (OLTP) always access the newest version (i.e. the base table version) and use the single-version scheduler mechanism as described in Section II. On the other hand, OLAP transactions access (i.e. read) a specific version calculated with the help of the current epoch state. The idea is that an OLAP transaction can only read a virtual state of the database in which no writers seem to be active anymore. Because OLAP transactions do not write any values, they do not need to insert edges into the conflict graph. Therefore, no conflicts are introduced because the read seems to be executed completely after all concurrent writers are finished. Moreover, all future writers are executed on a database state that is logically after the OLAP queries.

IV. Evaluation

In this section we evaluate our many-core optimized graph-based scheduler approaches. Our goal is to refute experimentally the assumption that an SGT scheduler is impractical. We show that our approach has very competitive OLTP throughput and that pessimistic schedulers such as our graph-based algorithm can outperform optimistic schedulers in high contention settings. Moreover, our approach has the lowest abort rate while providing expected user experience by retaining the order of already committed transactions. Further, we show that also in low contention settings the overhead of maintaining the graph is marginal. Our multi-version optimization helps us to efficiently process expensive and long-running read-only queries while executing transactional queries.

For the experiments we implemented a prototype database system that stores all its data in DRAM. Every transaction is scheduled on exactly one worker thread from the pool of available workers which are executed on a pinned core. Our system experiences truly concurrent transactions that are executed by the transaction's worker thread. Concurrent epoch-based garbage collection is used to minimize memory consumption. Aborts result in an undo and are rescheduled if needed with live-lock detection. We evaluate the following approaches in our prototype database (available at [15]).

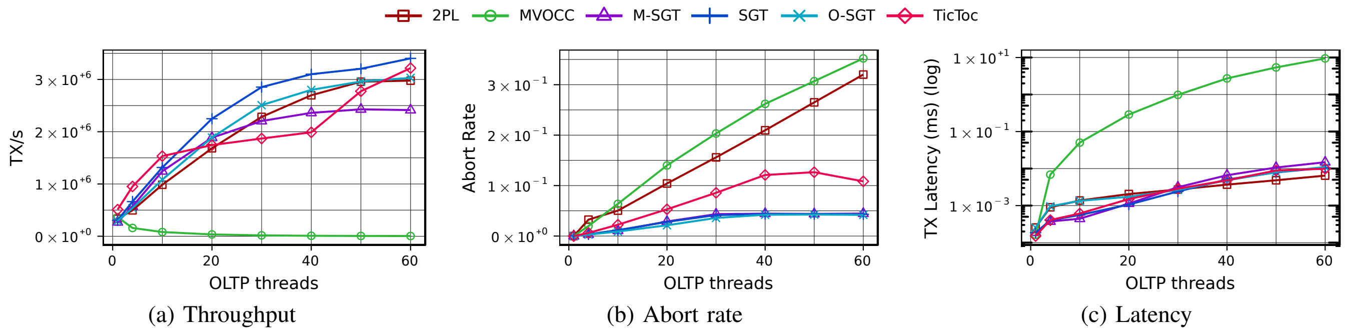
- *SGT:* Single-version graph-based scheduler with DFS
- *O-SGT:* Single-version graph-based scheduler with online topological ordering
- *M-SGT:* Multi-version graph-based scheduler with DFS
- *MVOCC:* Predicate multi-version OCC scheduler that uses a global lock for serializability validation [3]
- *TicToc:* TicToc with timestamp history derived from its original DBX1000 implementation [16]
- *2PL:* 2PL with row based read-write locks (using atomic CAS operations) and deadlock prevention (wait-die) [17]

All performance numbers were measured on a four-socket NUMA server (64-bit Ubuntu, gcc-7) with Intel(R) Xeon(R) CPU E7-4870 v2 processors. Each processor has 15 cores and twice as many hyper-threads. Every socket is connected to 256GB DRAM totaling 1024GB DRAM.

A. SmallBank

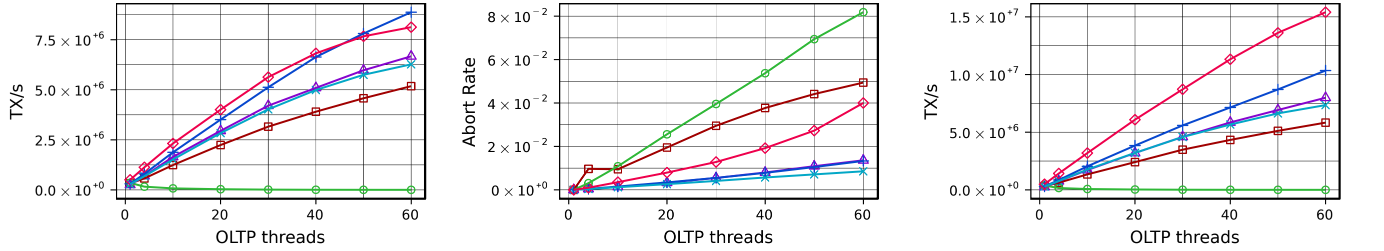
SmallBank mocks a simple banking database with financial typical transactions such as withdrawing money from checking accounts. It was designed to generate non-serializable schedules, in contrast to TPC-C, on isolation levels less restrictive than serializable [18]. OLTP-bench, which is a standardized benchmarking tool for databases, defines a configuration for SmallBank that we also use [19]. Thus, 25% of all operations on accounts are executed on a hotspot area of 100 tuples.

We start the experimental evaluation by studying unusual high contention. Figure 4 shows the throughput (commits per second), the abort rates for the different algorithms and the transaction latency in this high contention workload. All transactions request data elements of only 100 customers (uniformly distributed). Pessimistic approaches, such as locking-based schedulers, are known to perform well in high contention workloads. TicToc as the main OLTP competitor has a lower overall performance compared with our SGT implementation. Due to MVOCC's global locking approach to validate the read / write sets, the OLTP throughput is very limited on a NUMA scale server. The average transaction latency of all approaches,



(a) Throughput (b) Abort rate (c) Latency

Fig. 4: SmallBank workload with only 100 customers and high contention.



(a) Throughput medium contention for SmallBank with 1000 customers (b) Abort Rate medium contention (c) Throughput low contention for SmallBank with 10000 customers

Fig. 5: SmallBank workload with medium and low contention.

except for the global locking MVOCC, is similar. The abort rate ($\frac{|a|}{|a|+|c|}$) is higher for 2PL and MVOCC than for the other algorithms. 2PL and MVOCC abort directly at write conflicts even if the conflict is not involved in a non-serializable cycle. Reducing the contention to a more reasonable level results in performance improvements. The algorithms can achieve twice as much throughput if 1000 customers have accounts at the SmallBank as shown in Figure 5a. Our single-version SGT approach is still able to outperform TicToc in terms of minimizing aborts and many-core throughput (Figure 5b). In Figure 5c the number of customers is incremented resulting in low contention outside the hotspot area. The optimistic locking approach of TicToc performs best in very low contention settings as it does not encounter issues in the validation phase.

In all of the above settings, the online cycle checking SGT is not able to achieve better throughput than the regular DFS based one. The main reason is the maintenance of the topological order and the resulting contention on the order mapping. Since the multi-version SGT has a reduced performance compared to our single version SGT, introducing multiple data versions is not free after all. The deletion of versions and additional data local atomic compare and swaps at the version pointers slightly reduce the overall transaction throughput.

B. OLAP Extended SmallBank

Many workloads in practice are a mixture of OLTP and OLAP transactions. Therefore, we introduce a transaction that tells the bank employees how much money is currently within all bank accounts. As a result, a complete table scan transaction, which is often scheduled in real world applications, is added. We assume that a query optimizer can decide if a transaction is executed as OLTP or OLAP transaction due to the existence of writes and the amount of read operations. The experiment was designed such that the system uses a fixed number of worker threads (all available cores) and splits the workload between OLTP and OLAP threads. In Figure 6 the extended benchmark is shown in low contention setting. Due to the updates during the long-running reader, TicToc is able to produce the best OLTP throughput but is not able to commit OLAP transactions. Even with transaction history only a small amount of updates can be handled before the read versions are not matching anymore and a transaction needs to abort. The validation of the read set at the commit time is very expensive for TicToc. On the other hand, MVOCC has good OLAP throughput but the low performance on concurrent OLTP transactions prohibits its usage for mixed workload. Our multi-version SGT approach can handle both scenarios well and has the lowest overall transaction latency. The OLTP performance is not as good as the one of TicToc due to the overhead for multi-versioning but still very high. Because our epoch-based multi-version approach uses batch synchronization many small OLAP readers (10,000 tuples) perform better on MVOCC. Nevertheless, the overall performance is nicely balanced such that our multi-version approach is the best choice for mixed workloads.

C. YCSB

The YCSB benchmark, originally developed as a key-value store benchmark, is often adopted for benchmarking classical concurrency control protocols by using multiple queries per transaction [20]. Because of the zipfian distribution (θ value) of the tuple accesses, varying contention levels can be simulated. We implemented the standard workload A and used 16 queries per transaction of which 50% were writes. In Figure 7a the throughput is shown according to different contention levels

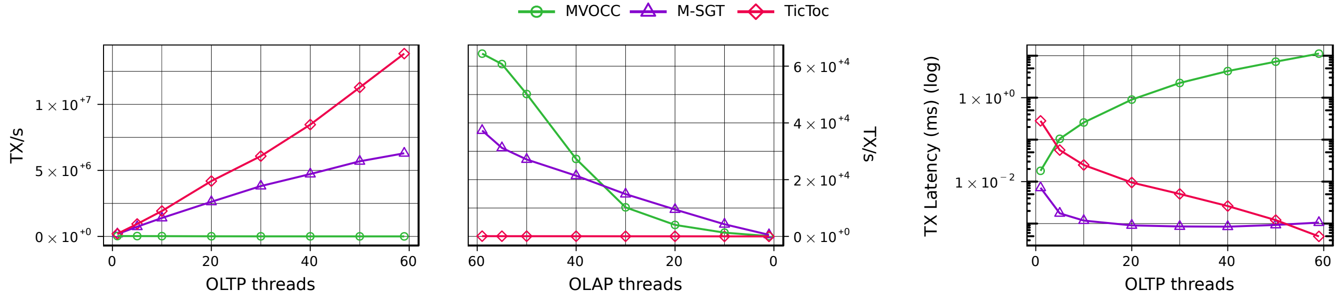


Fig. 6: Mixed workload for SmallBank with 10000 customers. The experiment was designed such that #OLTP threads + #OLAP threads = 60. The combined average transaction latency is shown on the right graph.

using 10 million records and 60 cores. Our approaches have very competitive throughput to the best performing algorithm but reduce the number of aborts significantly as shown in Figure 7b. In the very high contention setting of $\theta = 0.9$ (0.1% of the data is accessed by 35% of the queries), TicToc aborts twice as many transactions than our SGT variants. Since ten string columns (10 * 100B / tuple) are stored, multi-versioning is expensive in an update-heavy workload.

We also implemented the scan operation defined in YCSB. Each scan accesses 1% of the data. The benchmark, shown in Figure 7c, executes workload A with additional scan operations in a varying percentage ($\theta = 0.7$). Our multi-version approach can outperform the other algorithms as soon as scans are necessary. In comparison to TicToc, MSGT has a throughput boost of 1.75x. With an increasing number of expensive scans, the relative performance of global locking MVOCC improves.

The standard workload B describes a typical read-mostly transaction system. For testing concurrency protocols we again use 16 queries of which 95% are reads. The performance was evaluated with a varying contention factor, shown in Figure 8. In low and medium contention settings our approach has competitive throughput compared with TicToc and outperforms 2PL. Already at medium contention 2PL's approach of holding the locks until the end of the transaction decreases the throughput significantly. In very high contention scenarios, the short-term access locks of our approaches reduce the performance. On the contrary, optimistic protocols, such as TicToc, benefit from very few updates, almost no aborts, and an increasing cache-locality. Hence, the optimistic reads succeed and other threads invalidate cache lines only infrequently.

D. TATP

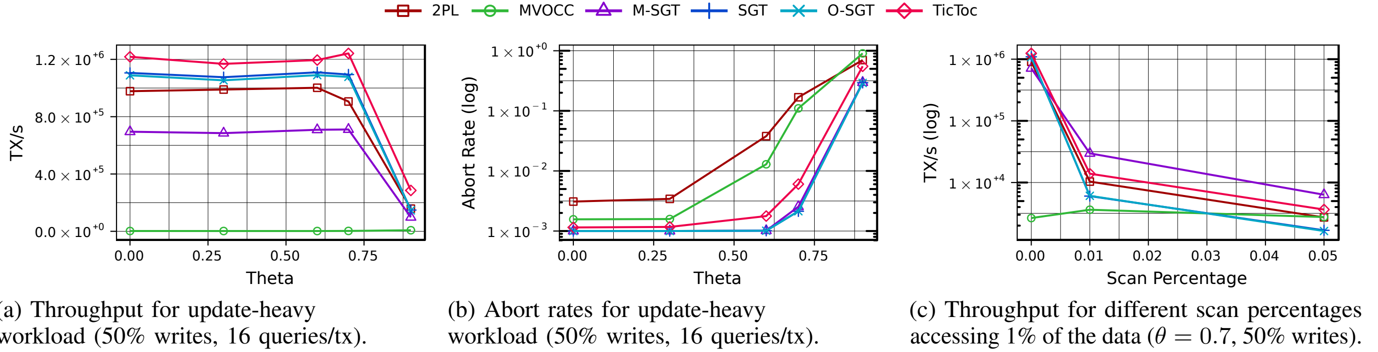
Further interesting properties of the algorithms can be shown with the TATP benchmark [21]. The primary key distribution is skewed to get a non-uniform row access pattern which increases tuple contention within the workload. TATP transactions can hardly generate conflict cycles. Therefore, a scheduler accepting the complete schedule class CSR should almost never abort. Our SGT approach achieves a performance of more than 10 million transactions per second at the highest contended version as shown in Figure 9. The throughput of our SGT implementation is higher than that of all the competitors. All the SGT versions as well as TicToc have very low abort ratios - as expected. Both schedulers are able to exploit a large fraction of the CSR class and do only need to abort if the UpdateSubscriber transaction is interleaved with the same keys. In contrast, MVOCC aborts if two transactions want to write on the same data element. 2PL with deadlock prevention (wait-die) aborts the newer transactions if two writes are requested to avoid deadlocks. Due to the simplicity of TATP, the online cycle checking of our SGT approach suffers from inserting and updating the ordering map and is therefore only as fast as 2PL.

E. Space and Time Requirements

Another interesting property is the CPU overhead of maintaining an ACID-consistent database state with conflict graphs. For better insights we tested how many CPU cycles are needed for pure cycle checking. Table I shows the results for the previously tested workloads in terms of average cycles needed per cycle check as well as per transaction. With increasing write contention the cycle checks get more expensive as the graphs tend to grow larger. For example, the TATP (100) workload does not require aborts and has short graph paths. On the other hand, YCSB A with θ = 0.9 has long paths that are frequently cyclic. Usually, O-SGT requires a few instructions less but has the additional overhead of inserting and deleting transactions from the ordering map which can be a bottleneck for myriads of very small transactions. Altogether, the amount of cycles needed is only a small fraction of the total transaction costs.

We designed an experiment that can separate the concurrency control overhead from the actual transaction workload. In this experiment, we executed the previous benchmarks but turned off the tuple access history, conflict detection, cycle checks, and restart handling. The only concurrency requirement for "No SGT" is that tuple accesses are serialized. The results are shown in Table II. Our approach needs to spend roughly one third of the time for concurrency control in the high contention setting of TATP and SmallBank (100). On the other hand, low contention benchmarks such as YCSB B (θ = 0.6) do not show any significant performance reductions. For YCSB A (θ = 0.9), a large fraction of the time is spent on concurrency control due to additional restarts. Turning off concurrency control applied to other protocols showed very similar results. For instance, TicToc also spends 85.7% of its time for retaining ACID.

Our implementation as well as most other concurrency control protocols rely on additional column(s) to determine serializability. In particular, one column for the tuple access entry lists and one column for the next sequence number are



(a) Throughput for update-heavy workload (50% writes, 16 queries/tx).

(b) Abort rates for update-heavy workload (50% writes, 16 queries/tx).

(c) Throughput for different scan percentages accessing 1% of the data ($\theta = 0.7$, 50% writes).

Fig. 7: Different YCSB A workloads with a table size of 10 million entries using 60 cores.

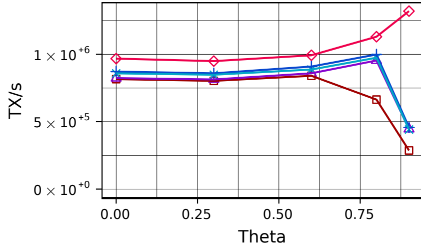
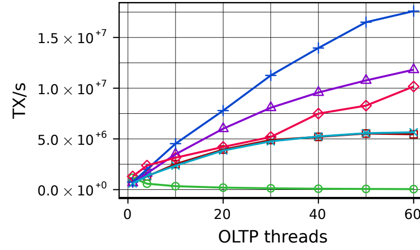
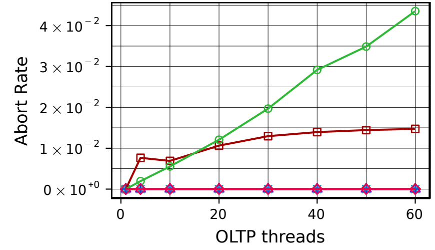


Fig. 8: Read-Mostly YCSB B workload with 16 queries/tx: 95% reads, 5% writes.



(a) Throughput (100 entries)



(b) Abort rate (100 entries)

Fig. 9: TATP workload with high contention.

Benchmark	SmallBank	TATP	YCSB A (0.9)	YCSB B (0.6)
SGT	901	681	1997	783
O-SGT	899	654	1962	681
SGT (tx)	1033	687	4516	788
O-SGT (tx)	1026	656	4428	682

TABLE I: Cycles needed for cycle testing with 60 cores.

Benchmark	SmallBank	TATP	YCSB A (0.9)	YCSB B (0.6)
No SGT	5.09M	24.1M	1.55M	0.94M [TX/s]
SGT	3.25M	17.4M	0.15M	0.91M [TX/s]
Overhead	36.1%	27.8%	90.6%	2.8%

TABLE II: Overhead of maintaining accesses, conflict detection, cycle tests, aborts, and live-lock handling [TX/s with 60 cores].

required. An implementation needs additional 16 bytes for the list header (atomic pointer and order number) and 8 bytes per tuple for the sequence number. For example the YCSB benchmarks store 1008 bytes per tuple, hence the overhead equals to 2.4%. Moreover, an optimization is possible such that only a pointer is stored. The list header and the sequence number are thereafter created lazily. Thus, only currently active tuples need the list and sequence number overhead which reduces the static memory requirements to the same as needed for TicToc or 2PL. Besides the static overhead our approach needs to store the transaction's access history and its graph node. Due to concurrent garbage reclamation the memory consumption is bounded according to the number of simultaneously active transactions. In Table III we evaluate the average amount of memory needed to store a transaction's access history and graph requirements. The space overhead depends on the density of the graph edges as YCSB workloads have higher requirements. However, the overhead is bounded.

Benchmark	SmallBank	TATP	YCSB A (0.9)	YCSB B (0.6)
Memory	499 B	456 B	876 B	544 B

TABLE III: Average transaction memory consumption

F. *Evaluation Summary*

Our novel graph-based scheduler is able to outperform the competitors on high-contention transactional workloads while aborting the fewest amount of transactions. With less contention our approach is able to achieve very competitive throughput in comparison to highly optimized optimistic schedulers — without introducing unexpected aborts. Thus, database users only encounter issues if the schedule violates the conditions for conflict serializability. We provide the best level of transaction isolation and are able to scale perfectly with an increasing number of execution cores. Moreover, our read-only addition helps to outperform the competitors on mixed workloads since other protocols are optimized for either OLAP or OLTP. We also show that in every contention setting the overhead of maintaining the graph is marginal. Thus, the assumption that graph-based schedulers are impractical can be refuted.

V. Related Work

The concept of ACID transactions is one of the oldest and most prominent features of databases. We give an overview of different approaches in particular those that are closely related to our concurrency control algorithm.

A. *Concurrency Control Strategies*

Concurrency control algorithms are used to isolate transactions such that every transaction seems to run exclusively on the data. Two-phase locking [1] protocols are serialization strategies that produce a real subset of $OCSR$ schedules [6].

In two-phase locking, all locks must be acquired before any lock can be released. By design, no two conflicting locks can be interleaved between transactions (deadlock and abort) in 2PL. Strict and strong 2PL grant their locks the same way as normal 2PL but release their write locks / read & write locks at once, reducing the accepted schedules to a true subset of $COCSR$ [11]. Deadlock detection is usually handled with a wait-for graph (WFG) [17] but also deadlock prevention is possible by reducing the number of accepted schedules [22]. Many traditional database systems use 2PL as their concurrency control strategy such as IBM DB2 and MySQL [2].

Because the locking introduced by 2PL degrades performance in today's multi-core architectures, most of the new database systems use an optimistic version of timestamp allocation. The basic idea for both optimistic and classic timestamp protocols is the allocation of a timestamp for each transaction (e.g. from a global natural number counter). Afterwards each operation of the transaction is assigned to this timestamp. Conflicting operations must be ordered according to the timestamps which leads to the following basic timestamp ordering rule if $p_i(x)$ and $q_j(x)$ are in conflict. $p_i(x)$ is executed before $q_j(x)$ iff the timestamp of t_i is smaller than the timestamp of t_j [23], [24]. All timestamp algorithms can only be a subset of CSR [11]. In modern many-core architectures global counters can already be performance bottlenecks although only a single atomic add instruction is needed. As a result, most modern timestamp ordering concurrency control schedulers cannot scale good enough with a large number of cores [5], [25]. Optimistic ordering can be classified into three phases — read, validation, and write. In the read phase all reads and transaction local writes are performed on the data. During validation the database scans for conflicts with concurrent transactions. If the transaction is positively validated, the local changes are written to the database. The backward oriented optimistic concurrency control (OCC) protocol validates the read set of the transaction in validation (t_v) with the write set of all transactions that were not committed before the start of t_v. For the forward oriented validation the write set of t_v needs to be disjoint with the read set of all concurrent read-phase transactions [4], [26].

In recent years, the focus of optimizing these protocols shifted from reducing memory consumption to better performance on multi-core machines. The problems that arise from global timestamp counters limit the scale up especially for OLTP workloads. The concurrency control scheduler used in SILO centers around a new calculation of transaction ids [5]. Transaction ids contain the system-wide global epoch counter of the time the transaction commits in the higher bits. Further bits are used to distinguish transactions within the same epoch. The local epoch bits do not represent the relative order among transactions. Only the read-after-write dependencies can be captured. As a result, the amount of concurrency is restricted. A more promising OCC protocol is TicToc which is a time traveling optimistic concurrency control scheduler [16]. TicToc avoids a global timestamp by calculating timestamps with the help of several parameters such as the read and write sets. These timestamps are computed lazily and thereafter checked whether they are valid. TicToc uses the concept of dynamic timestamp allocation, but instead of assigning them to transactions, they are assigned to tuples [27].

Optimistic protocols perform well if the amount of conflicts is small but introduce many aborts with a rising number of conflicts. The mostly-optimistic concurrency control (MOCC) has been developed to address the problems of highly contended workloads [28]. For highly contended accesses, MOCC maintains a modified version of 2PL to lock on a tuple granularity, on less contended tuples it relies on OCC's high-performance without taking read locks. One challenge of MOCC is the detection of such "hot" tuples. We evaluated both optimistic protocols and 2PL in our experiments section which are the components of MOCC. Optimistic protocols can benefit from better abort strategies. In particular, BCC proposes to abort according to commit-critical patterns in comparison to a changed read-set [29]. Intelligent graph-based batching and transaction reordering during the validation phase can increase throughput for optimistic protocols. However, batching based systems include trade offs between transaction latency, reordering freedom, and amount of aborts [30].

B. Multi-Versioning

Storing data in multiple versions helps to gain performance boosts for long-running readers [31]. Multi-version concurrency leads to the development of many schedulers [31], [6] which are used in state-of-the-art databases such as SAP HANA [32], [33], Microsoft Hekaton [34], [35], HyPer [36], and PostgreSQL [37]. Empirical results show that the concurrency control protocol choice is a crucial step in gaining good performance [38]. Many multi-version based systems only provided Snapshot Isolation (SI) instead of full serializability. However, Berenson et. al. shows that there exist schedules that are valid in SI but invalid in the context of conflict serializability [9]. To guarantee conflict serializability the concept of serialization certifiers was developed. It is based on anti-dependencies between two transactions. Anti-dependencies occur if a transaction creates a new version of a tuple and its previous version was already read by another transaction. When two consecutive anti-dependencies are detected, one of the transactions involved is aborted [39], [40]. PostgreSQL uses this Serializable Snapshot Isolation (SSI) strategy to enforce ACID [37]. PSSI is a graph-based scheduler on top of anti-dependencies to minimize the unnecessary aborts [41]. However, multi-threaded performance is limited due to a global lock for the certifier graph.

Hekaton and HyPer use a multi-version optimistic concurrency control algorithm to achieve full serializability. Hekaton stores both read and write sets to validate transactions. HyPer implements precision locking to reduce storage and validation efforts for readers. Our multi-version implementation is a heterogeneous system since read-only transactions are treated differently and are executed on a snapshot of the database. HyPer initially used a fork-based heterogeneous database but moved to a homogeneous one later [42], [3]. Kernel optimizations can help to speed up heterogeneous processing [43].

VI. Conclusion

In this paper, we presented a graph-based scheduler that scales perfectly with a rising number of cores which is one of the most important properties in the many-core age. Our concurrency control algorithm is able to handle both OLTP and OLAP workloads efficiently with the help of our heterogeneous multi-versioning extension. For read-heavy applications further optimization, such as precision locking to track the read accesses or unchanged data intervals, can be integrated as well [3]. Due to the nature of detecting conflict cycles, the presented scheduler can accept basically all useful schedules that are recoverable and commit order preserving conflict serializable. Because of the high throughput of our approach, the assumption that SGT is impractical can be refuted. The heterogeneous multi-versioning approach separates the transactional workload from the analytical one. This helps to gain high analytical performance while providing the best isolation level for the transactional workload as well as for the analytical transactions. In particular, the anomalies arising from the widely used snapshot isolation level require additional application code to handle write-skew issues which results in reduced performance. Further, our approach minimizes the number of aborted schedules due to the usage of the conflict graph which greatly improves user experience. All in all, we show that the almost forgotten concept of graph-based concurrency control can be used as high-performance and theoretical superior scheduler on many-core servers.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).

References

[1] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, no. 11, 1976.
[2] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter, "Priority mechanisms for oltp and transactional web application," in *ICDE*, 2004.
[3] T. Neumann, T. Mühlbauer, and A. Kemper, "Fast serializable multi-version concurrency control for main-memory database systems," in *SIGMOD*, 2015.
[4] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *TODS*, vol. 6, no. 2, 1981.
[5] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *SOSP*, 2013.
[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
[7] C. Mohan, H. Pirahesh, and R. Lorie, "Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions," in *SIGMOD*, 1992.
[8] A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions," in *ICDE*, 2000.
[9] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," in *SIGMOD*, 1995.
[10] M. A. Casanova, *The Concurrency Control Problem for Database Systems*, ser. Lecture Notes in Computer Science. Springer, 1981, vol. 116.
[11] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Elsevier, 2001.
[12] T. Hadzilacos and N. Yannakakis, "Deleting completed transactions," *JCSS*, vol. 38, no. 2, 1989.
[13] D. J. Pearce and P. H. Kelly, "A dynamic topological sort algorithm for directed acyclic graphs," *JEA*, vol. 11, 2007.
[14] D. Ajwani, T. Friedrich, and U. Meyer, "An $\mathcal{O}(n^{2.75})$ algorithm for online topological ordering," *Electronic Notes in Discrete Mathematics*, vol. 25, 2006.
[15] D. Durner, https://github.com/durner/No-False-Negatives, 2019.
[16] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "Tictoc: Time traveling optimistic concurrency control," in *SIGMOD*, 2016.
[17] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency control performance modeling: Alternatives and implications," *TODS*, vol. 12, no. 4, 1987.
[18] M. Alomari, M. Cahill, A. Fekete, and U. Rohm, "The cost of serializability on platforms that use snapshot isolation," in *ICDE*, 2008.
[19] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," in *PVLDB*, vol. 7, no. 4, 2013.
[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010.
[21] S. I. Technology, "Telecommunication Application Transaction Processing (TATP) Benchmark Description," 2009.
[22] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II, "System level concurrency control for distributed database systems," *TODS*, vol. 3, no. 2, 1978.
[23] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *TODS*, vol. 4, no. 2, 1979.
[24] P. A. Bernstein and N. Goodman, "Timestamp-based algorithms for concurrency control in distributed database systems," in *VLDB*, 1980.
[25] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," in *PVLDB*, vol. 8, no. 3, 2014.
[26] T. Härder, "Observations on optimistic concurrency control schemes," *Information Systems*, vol. 9, no. 2, 1984.
[27] R. Bayer, K. Elhardt, J. Heigert, and A. Reiser, "Dynamic timestamp allocation for transactions in database systems," in *DDB*, 1982.
[28] T. Wang and H. Kimura, "Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores," in *PVLDB*, vol. 10, no. 2, 2016.
[29] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, and X. Zhang, "Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases," in *PVLDB*, vol. 9, no. 6, 2016.
[30] B. Ding, L. Kot, and J. Gehrke, "Improving optimistic concurrency control through transaction batching and operation reordering," in *PVLDB*, vol. 12, no. 2, 2018.
[31] D. P. Reed, "Naming and synchronization in a decentralized computer system," Ph.D. dissertation, Massachusetts Institute of Technology, 1978.
[32] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: Data management for modern business applications," *SIGMOD Record*, vol. 40, no. 4, 2012.
[33] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient transaction processing in sap hana database: The end of a column store myth," in *SIGMOD*, 2012.
[34] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine," in *SIGMOD*, 2013.
[35] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, "High-performance concurrency control mechanisms for main-memory databases," in *PVLDB*, vol. 5, no. 4, 2011.
[36] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *ICDE*, 2011.
[37] D. R. Ports and K. Grittner, "Serializable snapshot isolation in postgresql," in *PVLDB*, vol. 5, no. 12, 2012.
[38] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, "An empirical evaluation of in-memory multi-version concurrency control," in *PVLDB*, vol. 10, no. 7, 2017.
[39] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," *TODS*, vol. 34, no. 4, 2009.
[40] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making snapshot isolation serializable," *TODS*, vol. 30, no. 2, 2005.
[41] S. Revilak, P. O'Neil, and E. O'Neil, "Precisely serializable snapshot isolation (pssi)," in *ICDE*, 2011.
[42] H. Mühe, A. Kemper, and T. Neumann, "Executing long-running transactions in synchronization-free main memory database systems," in *CIDR*, 2013.
[43] A. Sharma, F. M. Schuhknecht, and J. Dittrich, "Accelerating analytical processing in mvcc using fine-granular high-frequency virtual snapshotting," in *SIGMOD*, 2018.