Extracting More Intra-transaction Parallelism with Work Stealing for OLTP Workloads

Xiaozhou Zhou† Zhaoguo Wang‡ Rong Chen† Haibo Chen† Jinyang Li‡

†Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
‡Department of Computer Science, New York University
Contact: zhaoguo@nyu.edu

Abstract

Online transaction processing systems use two-phase locking (2PL) to guarantee serializability. However, traditional 2PL does not perform well under high contention, because a transaction will be blocked when it fails to acquire lock. This paper proposes a scalable work stealing algorithm for 2PL to leverage intra-transaction parallelism. The key idea is to parallelize the lock holder's work among lock waiters. Compared to traditional 2PL, our approach can achieve up to 2.8X throughput improvement for TPC-C new-order transactions under high contention.

1. Introduction

Existing online transaction processing (OLTP) systems use two-phase locking (2PL) [2, 14] or optimistic concurrency control (OCC) [2, 24] to ensure transaction serializability. However, these protocols do not scale on a multi-core platform, especially when the workload exhibits high contention. Under 2PL, a transaction must grab the lock of a record before accessing it. Thus, once concurrently transactions make conflicting access, their execution will be serialized. Under OCC, all but one conflicting transactions must abort and retry, resulting in worse performance than 2PL under contended workloads. There have been work parallelize highly contended workloads. Unfortunately, most of them target on specific workloads. For examples, some systems [35, 52, 56] require the workload to be static, while others [8, 12, 46–48] require the system to know the working set of each transaction before its execution. As a result, these work have limited usage scenarios in practice.

OCC and 2PL do not scale well because they enforce the execution order of conflicting transactions in coarse granularity (inter-transaction). Actually, there is unexploited parallelism in fine granularity (intra-transaction). For example, suppose two concurrent transactions, T_1 and T_2, attempting to update record a and then b. Under 2PL, if T_1 has locked a before T_2, T_2 cannot access a and thus idly waits until T_1 commits. However, instead of idly waiting, T_2 could help T_1 execute its second operation of updating b. In this way, a single transaction (e.g., T_1) can be parallelized. This paper presents StealDB, a new approach to improving multi-core database performance by unleashing the intra-transaction parallelism.

Existing approaches to exploit intra-transaction parallelism [22, 36, 41] dispatch the independent operations of a transaction to different worker threads. Unfortunately, this approach faces two technical challenges: 1) the dispatching component can become a performance bottleneck, and 2) it is hard to balance the load across worker threads. It is difficult to address these two challenges simultaneously: to balance the load, the system needs to check the current status of each worker before dispatching, thereby worsening the performance of the bottlenecked dispatcher.

StealDB's key ingredients in solving these challenges are: 1) *extract intra-transaction parallelism only for conflicting transactions* and, 2) *leverage work stealing to balance the load*. In the previous example, when T_2 is waiting to acquire the lock held by T_1, it will steal T_1's second operation to execute, resulting in intra-transaction parallelism. However, if T_2 accesses different records and does not conflict with T_1, then no work stealing happens and there is only inter-transaction parallelism.

Naively using existing work-stealing algorithms [5, 13] cannot scale up under highly contended workloads. The reason is that too many conflicting transactions end up attempting to steal operations from a single transaction. As a result, all of them contend on a single data structure to get the next runnable operation. Inspired by the scalable locking algorithms [9, 33], we propose a new scalable work-stealing algorithm to avoid contention. Specifically, StealDB organizes transactions that conflict on a record into a queue. A transaction steals work from the previous transaction and its work

APSys 2017, September 23, 2017, Mumbai, India.

DOI: http://dx.doi.org/10.1145/3124680.3124748

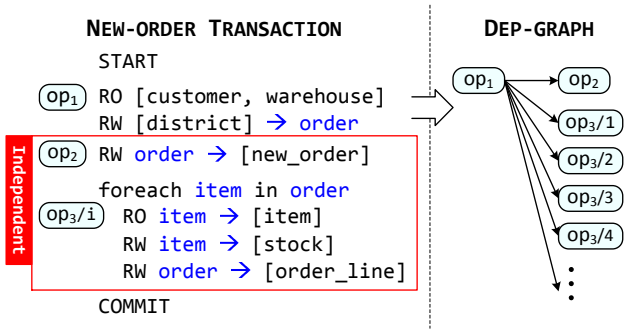

Figure 1. The pseudo-code and dependency graph (DEP-GRAPH) of the simplified TPC-C new-order transaction. In brackets are the database tables touched by each operation. RO and RW stand for read-only and read-write accesses to the tables respectively. op_3/N stands for the op_3 of the Nth iteration.

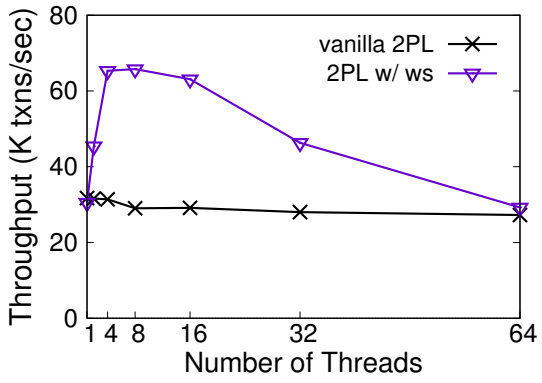

Figure 2. A throughput comparison of the TPC-C new-order transaction workload.

may get further stolen by the next transaction in the queue. This strategy avoids performance collapse when the number of worker threads increases.

We have implemented a preliminary version of StealDB in C++ based on Silo [49]. As an ongoing project, we have only evaluated StealDB with a micro-benchmark and TPC-C new-order transactions [45]. Compared to traditional 2PL, StealDB can improve the throughput by up to 2.8X for TPC-C new-order transactions under high contention and only incur negligible overhead under low contention.

2. Overview

2.1 Intra-transaction Parallelism in OLTP

Transaction processing systems support *inter-transaction* parallelism, i.e. the parallel execution of independent transactions, by assigning each transaction to a worker thread. Moreover, some operations within a single transaction can also be executed in parallel, which is called *intra-transaction* parallelism. For example, most of the operations in the TPC-C new-order transactions [45] can be executed concurrently. As shown in the left part of Figure 1, all operations except op_1 in the new-order transaction are independent of each other. Therefore, it is possible to execute these operations concurrently.

Although intra-transaction parallelism is widely used in OLAP workloads [1], it is difficult to exploit it for OLTP workloads. First, a typical OLTP transaction only accesses a moderate number of records, thereby limiting the amount of intra-transaction parallelism. For example, the TPC-C new-order transaction accesses 37 records on average. Second, parallelizing the execution of a single transaction incurs communication overhead among CPU cores, including dispatching the operations, passing execution results across operations, and coordinating the commit status of each operation. Therefore, to exploit intra-transaction parallelism efficiently, we need to fully utilize the limited potential parallelism and minimize communication overhead across CPU cores.

2.2 Basic Solution: Work Stealing

Our proposed approach *parallelizes a transaction only when there is contention*. For example, suppose there are two concurrent TPC-C new-order transactions, T_1 and T_2 (as described in Figure 1) and they conflict on op_1. Under 2PL, once T_1 has locked the $district$ table, T_2 cannot executes op_1 until T_1 commits. Existing system implementations either suspends T_2 and starts a new transaction, or keeps T_2 busy waiting until T_1 commits. StealDB worker also suspends the execution of T_2, but instead of executing a new transaction, StealDB will schedule T_2's worker to execute T_1's remaining operations. Consequently, T_1 is parallelized only when transactions conflict.

Using the basic work-stealing idea, T_1 will put all independent runnable operations (e.g. op_2 and op_3 of every item) to its runnable queue. When other transactions (T_2) are blocked at op_1 due to locks held by T_1, they will look up T_1's runnable queue and steal operations to execute. Figure 2 illustrates the performance of 2PL for TPC-C new-order transactions under high contention. Note that the workload only has one type of transaction accessing 37 records on average; all transactions conflict on the first operation $district$ record (op_1), and the other operations of the transaction are independent and access random records from different tables. Compared to vanilla 2PL, the throughput of 2PL with basic work stealing (WS) can scale to 8 cores and has 2.2X speedup (65.79K vs 29.02K TPS). However, the throughput degrades after 8 cores since too many transactions steal operations from a single transaction simultaneously, causing high contention in getting the available operations from a single runnable queue.

2.3 Scalable Work Stealing Algorithm

To remedy the problem of basic work-stealing algorithm, we propose to pass the information of available operations among worker threads one-by-one, inspired by the scalable queue-based locking [9, 34]. The idea for queue-based lock-

ing is to maintain an explicit list of all waiting threads for each lock. Upon releasing the lock, the lock holder passes the lock to the next waiter in the list. This approach significantly reduces the cacheline invalidation events and avoids performance collapse as more CPU cores are involved. In our scalable queue-based work-stealing algorithm, all conflicting transactions are chained together as a linked list, and transaction T_i can only steal operations from its immediate previous transaction T_{i-1}.

Suppose there are three concurrent new-order instances (T_1, T_2 and T_3). They all conflict with each other on the first operation (op_1), as they access the same district table. If T_1 executes op_1 before T_2 and T_3, it will put all independent operations into its runnable queue and start to execute op_2. The successor (T_2) in the chain of blocked transactions will steal T_1's operations to execute. Similarly, T_3 may steal T_1's operations from T_2 to execute. Note that T_1 can further steal its operations from the tail of the chain (T_3). Consequently, the lock holder's operations can be efficiently parallelized across all lock waiters.

3. Design

In this section, we introduce how to generate dependency graphs for transactions (Section 3.1), how to exploit intra-transaction parallelism on contention (Section 3.2), and how to leverages queue-based mechanism to implement scalable work stealing (Section 3.3).

3.1 Offline Analysis

Before executing a new type of transaction, StealDB will first perform an offline analysis to construct a static dependency graph (DEP-GRAPH), which is used to parallelize the execution of the transaction. In *dep-graph*, the transaction is represented as a DAG of operations (*op*). Operation op_j is a child of op_i if op_j depends on op_i (e.g. op_j's execution needs op_i's result). We assume that a transaction does not have dynamic control flows; otherwise, it is not parallelized. Unlike prior work [35, 52, 56], the offline analysis in StealDB only needs to analyze an individual transaction.

3.2 Intra-transaction Parallelism

Upon receiving a request, StealDB creates a transaction instance with three pieces of data structures: 1) the DEP-GRAPH containing all its operations and their corresponding contexts. The context of an operation includes a pointer to the operation's function and the input/output of the operation (e.g., the input may contain the key of the record to be accessed). 2) the runnable queue containing all runnable operations that can be stolen by other transactions. 3) the steal context, which is a $\langle tid, index \rangle$ tuple. The tid is the transaction ID of the transaction instance T which others try to steal operations from. The $index$ indicates the last runnable operation in T's runnable queue. The steal context is a single 64-bit word to ensure atomic access.

Transaction execution and commit. Algorithm 1 shows how a worker thread in StealDB executes a transaction operation-by-operation (Line 1-2). StealDB executes each operation (*op*) in its runnable queue from the beginning. If all dependencies of some operation op_i are satisfied upon executing *op*, StealDB will insert op_i into the runnable queue after *op* finishes (Line 3-4). As StealDB uses 2PL to ensure serializability, each operation will hold the locks of records accessed until the end of execution. After all operations finish, StealDB checks the status of each finished operation (Line 6-7). If any operation is aborted, then StealDB will abort the entire transaction by discarding all buffered changes. Otherwise, it will commit the transaction. In either case, it releases all locks held by the operations before returning.

ALGORITHM 1: RUNTRANSACTION (T) :

Input: T: The current transaction

```
   // Execute transaction T
1: foreach op in T.runnable_queue do
2:     RunOp (op, T.tid)
3:     if ∃ op_i s.t. all op_i's dependencies are satisifed after
       executing op then
4:         insert op_i into T.runnable_queue
   // Commit transaction T
5: foreach op in T do
6:     wait for op to finish
7:     if op.status == aborted then
8:         abort all ops
9:         release locks held by all ops
10:        T.status = aborted
11:        return
12: commit all ops of T
13: release locks held by all ops
14: T.status = committed
```

Operation execution. The execution of each operation consists of three phases, as shown in Algorithm 2:

- **Check phase** (Line 1-2): in StealDB, a transaction T may try to execute an operation which has already been stolen by other transactions. To ensure exactly-once execution, StealDB performs a check before executing the operation. Specifically, each operation has a tid field, whose value is initialized to contain the tid of its corresponding transaction. Before executing an operation, StealDB sets the tid field to be $null$ ($\perp$) with the CAS atomic instruction. Therefore, StealDB can know if the operation is under execution or not by simply checking the tid field (Line 1).
- **Lock phase** (Line 3-9): StealDB locks each record before accessing, according to 2PL. StealDB provides a

new TRYLOCK interface: it returns 0 upon successfully acquiring the lock. Otherwise, it returns the latest transaction (tid_p) waiting on this lock. We will discuss the detail of the lock manager in Section 3.3. If StealDB fails to acquire a lock, it will get the transaction instance context (T_p) by using the tid_p returned by TRYLOCK. Afterwards, it steals operations to execute using the steal context kept by T_p (Algorithm 3), if T_p is still running. After finishing a stolen operation, it will check again whether the lock is acquired using CHECKACQUIRED.

- **Execute phase** (Line 10-14): StealDB executes operation op and accesses the record. If op triggers a user-initiated abort, StealDB sets the status of op to be aborted. Otherwise, it considers op as committed.

ALGORITHM 2: RUNOP (op, tid) :

Input: op: The operation of transaction T
tid: Transaction ID of the current transaction

```
    // Check phase
1:  if op.tid != tid || CAS(&op.tid, tid, ⊥) != tid then
2:      return
    // Lock phase
3:  T = GETTXN(tid)
4:  op.lockCtx.tid = tid
5:  tid_p = TRYLOCK (op.record.lock, op.lockCtx)
6:  if tid_p != 0 then
7:      T_p = GETTXN(tid_p)
8:      while !CHECKACQUIRED(op.lockCtx) &&
        ISRUNNING(tid_p) do
9:          STEALOP (T, T_p)
    // Execute phase
10: success = op.execute()
11: if success then
12:     op.status = committed
13: else
14:     op.status = aborted
```

Work stealing. When a transaction fails to acquire a lock, it tries to steal operations from the previous transaction. Algorithm 3 shows how STEALOP works: Suppose T_p is T's previous transaction in the locking queue. Transaction T will steal operation to execute by using T_p's $stealCtx$ (Line 1). $stealCtx$ of T_p maintains the current stealing information: the transaction ID (tid_s) of the lock holder T_s and the index of the T_s's last runnable operation (i_{op}) (Line 3-4). If T_s has not finished, T will steal T_s's op_i from its runnable queue (Line 8 and 11). To enable T's successor transactions to steal T_s's operations, T also sets its $stealCtx$ with tid_s and $i_{op}-1$ if there are further operations to steal(Line 9-10).

ALGORITHM 3: STEALOP (T, T_p) :

Input: T: The current transaction
T_p: The previous transaction of T in the locking queue

```
1:  stealCtx = T_p.stealCtx
2:  if stealCtx != ⊥ then
3:      tid_s = stealCtx.tid
4:      i_op = stealCtx.op
5:      T_p.stealCtx = ⊥
6:      T_s = GETTXN (tid_s)
7:      if ISRUNNING (tid_s) then
8:          op = T_s.runnable_queue[i_op]
9:          if i_op > 0 then
10:             T.stealCtx = ⟨T_s, i_op − 1⟩
11:         RUNOP (op, tid_s)
```

3.3 Lock Manager

StealDB's lock manager provides three interfaces to the transaction runtime: TRYLOCK, CHECKACQUIRED and UNLOCK.

TRYLOCK will return 0 if the thread acquires a lock successfully. Otherwise, it will return the previous transaction's ID. For example, both T_1 and T_2 access record a, and T_1 acquires a's lock before T_2. When T_2 tries to lock a by calling TRYLOCK, it returns T_1's ID. CHECKACQUIRED will return `true` if the lock is granted to the current transaction. Otherwise, CHECKACQUIRED will return `false`. UNLOCK will release the lock of the record.

ALGORITHM 4: TRYLOCK ($lock$, $lockCtx$) :

Input: $lock$: The shared lock object
$lockCtx$: The lock context

```
1:  lockCtx.next = ⊥
2:  lockCtx.spin = true
3:  ctx_p = XCHG (lock.tail, lockCtx)
4:  if ctx_p == ⊥ then
5:      return 0
6:  tid_p = ctx_p.tid
7:  ctx_p.next = lockCtx
8:  return tid_p
```

To provide the semantic of TRYLOCK and CHECKACQUIRED, StealDB implements its lock mechanism based on queue-based lock algorithms (e.g. MCS lock [33] and CLH lock [9]). Algorithm 4 shows the basic logic of TRYLOCK which is extended from MCS lock. However, our algorithm is also suitable for other queue-based lock primitives (e.g., CLH queue lock). For each MCS lock, each transaction has a local lock context ($lockCtx$). It includes a pointer $next$, a boolean field $spin$ and the transaction's tid. Each lock

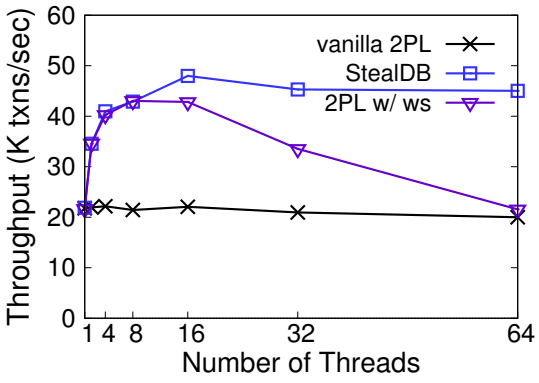

Figure 3. *The throughput comparison on the micro-benchmark with high contention.*

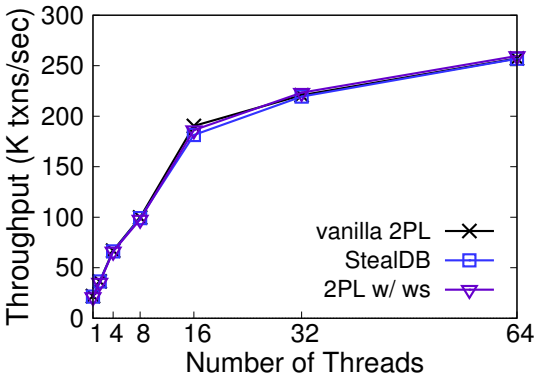

Figure 4. *The throughput comparison on the micro-benchmark with low contention.*

maintains an explicit list of $lockCtx$ structures. To acquire a lock, a transaction will add its $lockCtx$ in the lock's list (line 3). If there are already some other transactions waiting on this lock, it will check the $spin$ field by using CHECK-ACQUIRED. On releasing the lock, it just simply clears the $spin$ flag of next waiter on the waiting list.

4. Evaluation

This section evaluates the performance and scalability of StealDB and compare it to the traditional 2PL. Our evaluation seeks to answer the following three questions:

- What is the performance speedup of StealDB under high contention?
- Does StealDB introduce overhead under low contention compared to 2PL?
- What is the scalability of StealDB?

StealDB is implemented in C++ based on Silo [49], an open-source multicore in-memory database using Masstree [32] as a concurrent ordered index. We also implemented traditional 2PL with MCS lock [33] and the basic work stealing algorithm as baselines.

4.1 Experimental Setup

Hardware. All evaluations were conducted on a 32-core AMD machine (with hyper-threading), which consists of four 2.20GHz Opteron 6274 processors and 32GB DRAM. Each worker thread is bound to a CPU core.

Workload. There are two benchmarks in our evaluation: the micro-benchmark that performs random updates to each table, and the TPC-C new-order transactions. For each benchmark, we evaluate StealDB against the traditional 2PL (vanilla 2PL) under different contention levels and measure the throughput. In our experiments, we warm up all systems for 30s to get stable measurement results.

4.2 Micro-benchmarks

The micro-benchmark models a simple OLTP transaction. The database contains 32 tables in total, and each table contains 100K records with integer values. There is only one type of transaction running in the workload. During execution, each transaction chooses a random record from each table and increments the value of the record by one. We increase the contention level by restricting the number of records in the first table that the transactions can access. Under high contention, all transactions access the same first record.

Figure 3 shows the throughput with increasing number of threads under high contention. In this case, all waiting transactions are in the lock-waiting chain of the first record. 2PL has a flat throughput, because only one transaction can make progress and all other transactions are waiting for the lock of the first record. Basic work stealing scheme performs well with a small number of threads, but it cannot scale up because too many waiters will cause high contention. StealDB reaches 2.2X peak throughput (48.0K vs 22.1K TPS) at 16 threads and keeps the throughput nearly unchanged to 64 threads. At 16 threads, 25 out of 32 operations are executed by lock waiters, while the average execution time of operations increases by 3.3X (1.02us vs 3.36us) due to inter-core communication.

Figure 4 shows the throughput of various systems under low contention. Because transactions will hardly access the same record, all worker threads will not be blocked by locks, and almost no stealing happens. The throughput of StealDB is nearly identical to that of 2PL, since StealDB can also fully leverage the inter-transaction parallelism under low contention without additional overhead.

4.3 TPC-C New-order Transactions

We plan to run the whole TPC-C benchmark to show the real-world performance of StealDB. Currently, however, StealDB can only run the new-order transaction since we

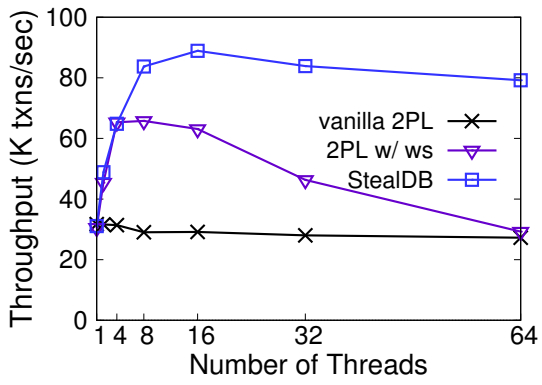

Figure 5. *The throughput comparison on TPC-C new-order transactions with high contention.*

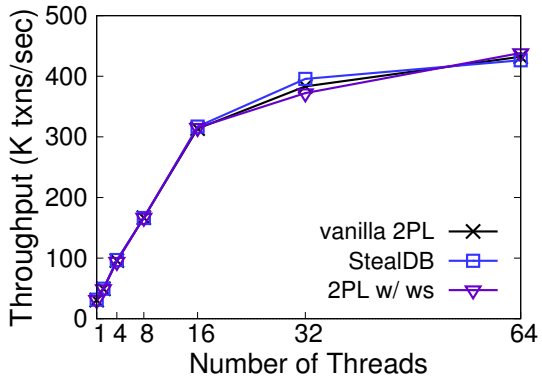

Figure 6. *The throughput comparison on TPC-C new-order transactions with low contention.*

have not implemented the read-write lock and deadlock detection.

The new-order transaction first reads a warehouse, then locks a district under that warehouse. The rest of the transaction does not cause much contention. We evaluated two extreme cases of contention level. Under high contention, there is only one warehouse with one district table. Under low contention, each worker thread accesses its private warehouse. The new-order transaction involves 26 operations on average.

Figure 5 shows StealDB outperforms 2PL by up to 2.8X under high contention. At 16 threads, 80% operations of a transaction are executed by other worker threads, and the average execution time of operations increases by 65% (0.96us to 1.58us) due to inter-core communication. StealDB cannot scale further with increasing threads because the first two operations and the lock releasing part cannot be executed in parallel.

Further, as shown in Figure 6, StealDB can provide a similar performance with 2PL under low contention.

5. Future work

StealDB is still an ongoing project, we are going to solve following issues in future:

Extend to read-write and range lock. The work-stealing algorithm can be extended to handle reader-writer lock. The queue-based reader-writer lock maintains lock waiters in a linked list like MCS lock [34]. Unlike the exclusive lock, multiple readers can be the lock holders of a reader-writer lock. Therefore, the lock waiters need to steal operations from multiple readers efficiently. We also plan to support range queries by supporting range locks to StealDB so that transactions blocked by a range lock can also steal operations from the lock holder.

Add deadlock detection. StealDB sorts all locks and acquires locks in the same order to avoid deadlock, but this lock ordering can be violoated by work stealing. In our current implementation, if lock acquiring time exceeds a threshold, StealDB will abort the transaction and roll back all changes. We planned to add a wait-for graph deadlock detector which will select a victim transaction to abort when there is a cycle in the wait-for graph.

Analyze transactions automatically. Currently, all transactions are implemented manually. The users of StealDB need to manually decompose a transaction into operations to enable work stealing. Similar to DORA [37], we will make a tool to analyze the dependencies inside transactions and generate code automatically.

6. Related Work

There are existing work [7, 22, 41] to improve database performance by leveraging intra-transaction parallelism. These systems [22, 41] dispatch independent sub-transactions to different CPU cores at the beginning of transaction. Colohan et al. [7] also propose to speculatively execute independent sub-transactions concurrently. By contrast, StealDB only exploits the intra-transaction parallelism when transactions conflict.

Database partitioning [15, 20, 44, 50] is another approach to take advantage of intra-transaction parallelism. Database records are divided into multiple disjoint partitions, and all transactions are also divided into a couple of sub-transactions, which will be dispatched to different partitions in advance. However, since each partition is protected with a global lock, it is hard to scale when there are lots of conflicting cross-partition transactions. DORA [36, 37] provides a fine-grained approach to associating each record with a private lock. However, it would increase the complexity of deadlock detection.

There also have been several efforts [35, 52, 56, 59] to improve database performance under high contention via static analysis. They usually assume a static workload and build a conflict graph among all transaction types in advance [3, 4, 42]. Compared to StealDB, these approaches are not suitable for those workloads in which new types of

transactions are encountered during the runtime. Some other approaches [8, 12, 46–48] enforce the execution order of all transactions to ensure serializability. However, they require knowing each transaction's access pattern before execution [40]. In contrast, StealDB has fewer constraint on the workload.

Multicore in-memory databases provide low latency and high throughput for transaction processing comparing to disk-based databases, and several commercial databases [11, 26, 29] have already taken this advantage. StealDB continues this line of research by optimizing transaction processing in multicore and in-memory databases [23, 25, 26, 28, 36, 49, 54, 58, 60]. Yu et al. [58] studied the scalability of seven concurrent control mechanisms by simulating up to 1024 cores, and identified the scalability bottlenecks of these concurrency control mechanisms. Silo [49] uses a epoch based transaction ID generation mechanism and a decentralized validation protocol to avoid contention points in OCC; it achieves near-linearly scalability under low contention. Cicada [28] executes transactions in a optimistic multi-version fashion, and uses multiple loosely synchronized clocks to do the scalable timestamp generation. Cicada also includes several optimizations to handle varied workloads, such as choosing a maximum backoff time dynamically to adapt to different contention level. Recently, some databases start to improve multicore scalability by eliminating centralized locks and latches [16, 17, 19, 39] for databases implemented using 2PL. StealDB is orthogonal with these approaches and can be applied to these systems.

Hardware transactional memory (HTM) has recently appeared in the latest commercial processors (e.g., Intel's RTM). Since HTM features like atomicity, consistency and isolation (ACI) make it very promising for database transactions [6, 27, 53–55], it is interesting to explore how to utilize HTM for even better intra-transaction parallelism. Although, we focus on a single multi-core machine, it is possible to extend our scalable work-stealing scheme to distributed in-memory systems with highly concurrent transaction and query processing [43, 55].

The idea of StealDB is also influenced by multicore research in identifying and avoiding unwanted interleavings [10, 18, 30, 31, 51, 57]. Examples include identifying harmful interleavings to detect concurrency bugs [30, 31], and constraining interleavings to avoid bugs [18, 38], making multithreading stable [10, 57], and reducing nondeterminism for state machine replication [21].

7. Conclusion

This paper proposes a scalable work-stealing algorithm to efficiently exploit intra-transaction parallelism among conflicting transactions. It uses run-time information of running transactions to dynamically parallelize OLTP workload among waiting threads, so that it can adapt to different contention levels. Our resulting system can significantly improve the performance of OLTP workloads under high contention without sacrificing the performance under low contention. Our research opens a new way to extract intra-transaction parallelism for OLTP workloads.

Acknowledgments

We thank the anonymous reviewers for their insightful suggestions. This work is supported in part by the National Key Research & Development Program (No. 2016YFB1000500), the National Natural Science Foundation of China (No. 61402284, 61572314, 61525204), the National Youth Top-notch Talent Support Program of China, Singapore NRF (CREATE E2S2), and National Science Foundation under CNS 1218117.

References

[1] F. Akal, K. Böhm, and H.-J. Schek. Olap query evaluation in a database cluster: A performance study on intra-query parallelism. In *Advances in Databases and Information Systems*, pages 181–184. Springer, 2002.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987.

[3] P. A. Bernstein and D. W. Shipman. The correctness of concurrency control mechanisms in a system for distributed databases (sdd-1). *ACM Transactions on Database Systems (TODS)*, 5(1):52–68, 1980.

[4] P. A. Bernstein, D. W. Shipman, and J. B. Rothnie Jr. Concurrency control in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 5(1):18–51, 1980.

[5] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28. ACM, 2005.

[6] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using rdma and htm. In *Proc. EuroSys*, pages 26:1–26:17, 2016.

[7] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. Optimistic intra-transaction parallelism on chip multiprocessors. In *Proceedings of the 31st international conference on Very large data bases*, pages 73–84. VLDB Endowment, 2005.

[8] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, pages 21–21. USENIX Association, 2012.

[9] T. Craig. Building fifo and priority queuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, University of Washington, 02 1993.(ftp tr/1993/02/UW-CSE-93-02-02. PS. Z from cs. washington. edu), 1993.

[10] H. Cui, J. Wu, C.-c. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proc. OSDI*, 2010.

[11] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Servers memory-optimized OLTP engine. In *Proc. SIGMOD*, 2013.

[12] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 15–26. ACM, 2014.

[13] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.

[14] J. Gray and A. Reuter. Transaction processing: concepts and techniques, 1993.

[15] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *Proc. SIGMOD*, pages 981–992. ACM, 2008.

[16] T. Horikawa. Latch-free data structures for DBMS: design, implementation, and evaluation. In *Proc. SIGMOD*, pages 409–420. ACM, 2013.

[17] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. EDBT*, pages 24–35. ACM, 2009.

[18] H. Jula, D. M. Tralamazza, C. Zamfir, G. Candea, et al. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proc. OSDI*, volume 8, pages 295–308, 2008.

[19] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *Proc. SIGMOD*, pages 73–84. ACM, 2013.

[20] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *VLDB Endowment*, 1(2):1496–1499, 2008.

[21] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about eve: execute-verify replication for multi-core servers. In *Proc. OSDI*, 2012.

[22] H. Kaufmann and H.-J. Schek. Extending tp-monitors for intra-transaction parallelism. In *Parallel and Distributed Information Systems, 1996., Fourth International Conference on*, pages 250–261. IEEE, 1996.

[23] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE, 2011.

[24] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[25] 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 *Proc. VLDB*, 2011.

[26] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krueger, and M. Grund. High-performance transaction processing in sap hana. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.

[27] V. Leis, A. Kemper, and T. Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. In *Proc. ICDE*, 2014.

[28] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35. ACM, 2017.

[29] J. Lindström, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila. Ibm soliddb: In-memory database optimized for extreme speed and availability. *IEEE Data Eng. Bull.*, 36(2):14–20, 2013.

[30] S. Lu, S. Park, and Y. Zhou. Finding atomicity-violation bugs through unserializable interleaving testing. *IEEE Transactions on Software Engineering*, 38(4):844–860, 2012.

[31] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *Proc. ASPLOS*, pages 37–48. ACM, 2006.

[32] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. EuroSys*, pages 183–196, 2012.

[33] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[34] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *ACM SIGPLAN Notices*, volume 26, pages 106–113. ACM, 1991.

[35] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distribted transactions. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 479–494. USENIX Association, 2014.

[36] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *VLDB Endowment*, 3(1-2):928–939, 2010.

[37] I. Pandis, P. Tözün, M. Branco, D. Karampinas, D. Porobic, R. Johnson, and A. Ailamaki. A data-oriented transaction execution engine and supporting tools. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1237–1240. ACM, 2011.

[38] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proc. SOSP*, pages 235–248. ACM, 2005.

[39] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *Proceedings of the VLDB Endowment*, 6(2):145–156, 2012.

[40] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proceedings of the VLDB Endowment*, 7(10):821–832, 2014.

[41] M. Rys, M. C. Norrie, and H.-J. Schek. Intra-transaction parallelism in the mapping of an object model to a relational multi-processor system. In *VLDB*, pages 460–471, 1996.

[42] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)*, 20(3):325–363, 1995.

[43] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proc. OSDI*, pages 317–332, 2016.

[44] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proc. VLDB*, pages 1150–1160, 2007.

[45] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, 2007.

[46] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.

[47] A. Thomson and D. J. Abadi. Modularity and Scalability in Calvin. *IEEE Data Engineering Bulletin*, page 48, 2013.

[48] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[49] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proc. SOSP*, 2013.

[50] L. VoltDB. Voltdb technical overview, 2010.

[51] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proc. OSDI*, pages 281–294, 2008.

[52] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1643–1658. ACM, 2016.

[53] Z. Wang, H. Qian, H. Chen, and J. Li. Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In *Proc. Apsys*. ACM, 2013.

[54] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proc. EuroSys*, 2014.

[55] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proc. SOSP*, pages 87–104, 2015.

[56] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 279–294. ACM, 2015.

[57] J. Yang, H. Cui, J. Wu, Y. Tang, and G. Hu. Determinism is not enough: Making parallel programs reliable with stable multithreading. *Communications of ACM*, page 75, 2014.

[58] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment*, 8(3):209–220, 2014.

[59] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proc. SOSP*, pages 276–291. ACM, 2013.

[60] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 465–477. USENIX Association, 2014.