Scaling Multicore Databases via Constrained Parallel Execution

Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi †, Haibo Chen†, Jinyang Li
New York University, † Shanghai Jiao Tong University

ABSTRACT

Multicore in-memory databases often rely on traditional concurrency control schemes such as two-phase-locking (2PL) or optimistic concurrency control (OCC). Unfortunately, when the workload exhibits a non-trivial amount of contention, both 2PL and OCC sacrifice much parallel execution opportunity. In this paper, we describe a new concurrency control scheme, interleaving constrained concurrency control (IC3), which provides serializability while allowing for parallel execution of certain conflicting transactions. IC3 combines the static analysis of the transaction workload with runtime techniques that track and enforce dependencies among concurrent transactions. The use of static analysis simplifies IC3's runtime design, allowing it to scale to many cores. Evaluations on a 64-core machine using the TPC-C benchmark show that IC3 outperforms traditional concurrency control schemes under contention. It achieves the throughput of 434K transactions/sec on the TPC-C benchmark configured with only one warehouse. It also scales better than several recent concurrent control schemes that also target contended workloads.

1. INTRODUCTION

Rapid increase of memory volume and CPU core counts have stimulated much recent development of multi-core in-memory databases [42, 15, 20, 27, 48, 51, 34, 54]. As the performance bottleneck has shifted from I/O to CPU, how to maximally take advantage of the CPU resources of multiple cores has become an important research problem.

A key challenge facing multi-core databases is how to support serializable transactions while maximizing parallelism so as to scale to many CPU cores. Popular concurrency control schemes are based on two-phase locking (2PL) [5, 19] or optimistic concurrency control (OCC) [26, 5]. Both achieve good parallel execution when concurrent transactions rarely conflict with each other. Unfortunately, when the workload exhibits contention, the performance of both schemes crumbles. 2PL makes transactions grab read/write locks, thus it serializes the execution of transactions as soon as they make a conflicting access to some data item. OCC performs worse under contention because it aborts and retries conflicted transactions.

There is much parallelism left unexploited by 2PL and OCC in contended workloads. As an example, suppose transactions 1 and 2 both modify data items A and B. One can safely interleave their execution as long as both transactions modify items A and B in the same order. Under 2PL or OCC, their execution is serialized. This paper presents IC3, a concurrency control scheme for multi-core in-memory databases, which unlocks such parallelism among conflicting transactions.

A basic strategy for safe interleaving is to track dependencies that arise as transactions make conflicting data access and to enforce tracked dependencies by constraining a transaction's subsequent data access. This basic approach faces several challenges in order to extract parallelism while guaranteeing serializability: How to know which data access should be constrained and which ones should not? How to ensure transitive dependencies are not violated without having to explicitly track them (which is expensive)? How to guarantee that tracked dependencies are always enforceable at runtime?

IC3's key to solving these challenges is to combine runtime techniques with a static analysis of the transaction workload to be executed. IC3's static analysis is based on the foundation laid out by prior work on transaction chopping [4, 41]. In particular, it constructs a static conflict graph (SC-graph) in which a transaction is represented as a series of atomic pieces each making one or more database access. Two pieces of different transactions are connected if both access the same table and one of the access is a write. A cycle involving multiple pieces of some transaction (i.e. an SC-cycle) indicates a potential violation of serializability when interleaving the execution of those pieces involved in the SC-cycle. Transaction chopping [41] precludes any SC-cycle in a workload by merging pieces into a larger atomic unit. By contrast, IC3 permits most types of SC-cycles and relies on its runtime to render them harmless by constraining the execution of the corresponding pieces.

The runtime of IC3 only tracks and enforces direct dependencies; this is more efficient and scalable than explicitly tracking and enforcing transitive dependencies, as is done in [33]. Without the aid of static analysis, this simplified runtime does not expose much parallelism: once a transaction T becomes dependent on another T', T has no choice but to wait for T' to finish in order to prevent potential

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2882934

violations of transitive dependencies. Static analysis helps unlock the parallelism inherent in a workload. If there exists an SC-cycle connecting the next piece p of T to some piece p' of the dependent transaction T', then IC3 can shortcut the wait and execute p as soon as p' has finished without waiting for the rest of T'. As most SC-cycles are among instances of the same type of transaction, this shortcut is very common and is crucial for IC3's performance. Static analysis also helps IC3 ensure all tracked dependencies are enforceable at runtime. In particular, we identify a class of SC-cycles as deadlock-prone in that they may cause deadlocks when IC3 tries to constrain piece execution. We remove the subset of SC-cycles that are deadlock-prone by combining pieces.

We contrast IC3's approach to existing work [36, 2, 47, 45, 16] that also expose parallelism among contended transactions (see § 7 for a more complete discussion). Dependency-aware transaction memory (DATM) [36] and Ordered shared lock [2] permit safe interleaving should it arise at runtime and otherwise abort. As such aborts may cascade, IC3's approach to pro-actively constraining execution for safe interleaving is more robust. Deterministic database [47, 45, 16] generates a dependency graph that deterministically orders transactions' conflicting record access in accordance with transactions' global arrival order. This fine-grained runtime technique avoids certain unnecessary constraining incurred by IC3 because static analysis cannot accurately predict actual runtime conflict. However, as the dependency graph is generated by one thread, deterministic database does not scale as well as IC3 when running on a large number of cores.

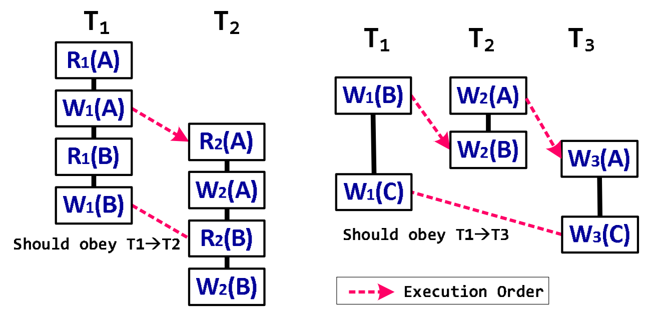
We have implemented IC3 in C++ using the codebase of Silo [48]. Evaluations on a 64-core machine using microbenchmarks and TPC-C [44] show that IC3 outperforms traditional 2PC and OCC under moderate to high amounts of contention. For the TPC-C benchmark configured with 1 warehouse, IC3 achieves 434K transactions/sec on 64 threads while the throughput of 2PL and OCC is under 50K transactions/sec. IC3 also scales better than deterministic lazy evaluation [16] and DATM [36] under high contention.

2. MOTIVATION AND APPROACH

In this section, we consider those parallel execution opportunities that are not exploited by 2PL and OCC. We discuss the challenges in enabling safe interleaving (§ 2.1) and explain IC3's key ideas at a high level(§ 2.2).

2.1 Parallelism Opportunity and Challenges

When concurrent transactions make conflicting database access, much opportunity for parallelism is lost when using traditional concurrency control protocols like 2PL and OCC. Consider two transactions, T_1 and T_2, each of which reads and modifies records A and B, i.e. T_1=$R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$ and T_2=$R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$. Both 2PL and OCC effectively serialize the execution of T_1 and T_2. However, safe parallel execution of T_1 and T_2 exists, as shown by the example in Figure 1a. In this example, once T_2 reads record A after T_1's write, T_2's subsequent read from record B will be constrained to occur after that of T_1's write to B, thereby ensuring serializability. Some existing protocols, such as Commit conflicting transactions [36] and Ordered sharing lock [2], augment OCC or 2PL to permit the safe interleaving in Figure 1a should it happen at runtime. However, if the actual interleaving happens to be unsafe (e.g. $W_1(A) \to R_2(A)$ but $R_2(B) \to W_1(B)$), these protocols [36, 2] abort offending transactions, resulting in a cascading effect because interleaved data access have read uncommitted writes. To avoid costly cascading aborts, it is better to actively constrain the interleaving to guarantee safety instead of passively permitting interleaving that happens to be safe.



(a) An example of safe parallel execution of two conflicting transactions

(b) Transitive dependency makes it difficult to track and enforce dependency.

Figure 1: Opportunity & Challenge.

The example of Figure 1a suggests an intuitive, albeit naive, solution to exploit parallelism. Specifically, the database could track the dependencies among transactions as they make conflicting data accesses (including read-write or write-write conflicts). It would then ensure that tracked dependencies are not violated later (i.e. no dependency cycles arise) by delaying a transaction's subsequent data access when necessary. For the example in Figure 1a, we would track $T_1 \to T_2$ when T_2 performs $R_2(A)$ after $W_1(A)$. Subsequently, the database would block $R_2(B)$ until $W_1(B)$ has finished in order to enforce the dependency $T_1 \to T_2$. While this naive approach works for the example in Figure 1a, it does not work for general transactions, due to the following challenges.

Knowing which data access to enforce dependency. In order to minimize unnecessary blocking, we need to know the series of records to be accessed by a transaction beforehand. In the example of Figure 1a, if T_2 knows that its dependent transaction T_1 will update record B (and T_1 makes no other access to B), T_2 only needs to wait for $W_1(B)$ to finish before performing its own access $R_2(B)$. If a transaction's record access information is not known, T_2 must wait for T_1 to finish its execution in entirety, leaving no opportunities for interleaving.

Handling transitive dependencies. The naive approach is not correct for general transactions, because it does not handle transitive dependencies. To see why, let us consider the example in Figure 1b, in which T_3 becomes dependent on T_2 (i.e. $T_2 \to T_3$) after T_3 has written to record A after T_2. During subsequent execution, T_2 becomes dependent on another transaction T_1, resulting in the transitive dependency $T_1(\to T_2) \to T_3$. This transitive dependency needs to be enforced by delaying T_3's write to record C after that of T_1. Techniques for tracking transitive dependencies at runtime have been proposed in the distributed setting [33], but the required computation would impose much overhead when used in the multi-core setting.

Tracked dependencies are not always enforceable. The naive approach assumes that it is always possible to enforce tracked dependencies. While this assumption holds for some workloads, it is not the case for arbitrary real-world transaction workloads. For example, if $T_1 = W_1(A), W_1(B)$ and $T_2 = W_2(B), W_2(A)$, no safe interleaving of T_1 and T_2 exists. If the database permits both to execute concurrently, it is impossible to enforce the tracked dependencies later.

2.2 IC3's Approach

To solve the challenges of safe parallel execution, one could rely solely on the runtime to analyze transactions' record access information and to enforce the dependency among them. Deterministic databases [47, 45, 16] take such an approach. They generate a dependency graph that deterministically order transactions' conflicting record access in accordance with transactions' global arrival order and enforce the dependency during execution, sometimes lazily [16]. While this approach can enforce dependency precisely, it does not scale well to a large number of cores, as the dependency analysis is performed by one thread and can become a performance bottleneck (§ 6). IC3 uses a different approach that augments runtime techniques with an offline static analysis of the workload. Below, we discuss the main ideas of IC3.

Static analysis. Most OLTP workloads consist of a mix of known transaction types. For each type of transaction, we typically know which table(s) the transaction reads or writes [4, 41, 33, 53]. Utilizing such information, static analysis chops each transaction into pieces, each of which makes atomic data access. IC3's runtime decides which pair of pieces needs to be constrained (e.g. if both make conflicting access to the same table) to enforce tracked dependencies. Moreover, static analysis can also determine when there may exist no safe interleaving at runtime and take preemptive steps to ensure all tracked dependencies are enforceable. The upside of static analysis is that it simplifies IC3's runtime dependency tracking and enforcement. The downside is that it can cause the runtime to unnecessarily constrain certain safe interleaving. We discuss techniques to mitigate the performance cost of over constraining (§ 4.1).

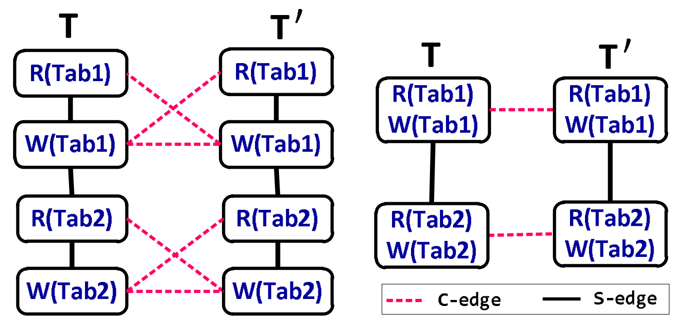
Efficient runtime dependency tracking and enforcement. IC3 tracks and enforces only direct dependencies which occur between two transactions when they make consecutive, conflicting access to the same record. By augmenting these direct dependencies with information from offline static analysis, IC3 can ensure all transitive dependencies are obeyed without explicitly tracking them.

To see how static analysis helps, let us consider a naive scheme that can obey all transitive dependencies while only tracking direct ones. In this scheme, every transaction T must wait for its directly dependent transactions to commit before T is allowed to commit or perform subsequent data access. Such conservative waiting ensures that all transitive dependencies are obeyed. For example, in Figure 1b, T_3 waits for its directly dependent transaction T_2 to commit before it proceeds to write to record C. Because T_2 also waits for its directly dependent transaction T_1 to commit before T_2 commits, we are able to ensure $T_1 \rightarrow T_3$ for their corresponding access to record C without ever explicitly tracking and discovering the transitive dependency $T_1 \rightarrow T_2 \rightarrow T_3$.

The naive scheme kills the parallel execution opportunity shown in Figure 1a. For example, it will prevent the interleaved execution in Figure 1a, as T_2 waits for T_1 to commit before it is allowed to perform $R_2(B)$. The key to make direct dependency enforcement work well is to leverage the static analysis of the workload. With the help of static analysis, we add two rules that bypass the default behavior of making T_2 wait for its direct dependency T_1 to commit. 1) if static analysis shows that no *potential* transitive dependencies may arise between T_1 and T_2 due to T_2's next data access, the runtime can safely let T_2 perform that access without blocking for T_1. 2) if static analysis shows that T_2's next data access might potentially conflict with some of T_1's later access, then T_2 only needs to wait for the corresponding access of T_1 to finish instead of waiting for the entirety of T_1 to finish and commit. The second rule applies to Figure 1a to allow its interleaving. Both rules are crucial for achieving good performance for practical workloads. We will elaborate the combined static analysis and runtime techniques in the next section (§ 3) and provide intuitions for their correctness.

3. BASIC DESIGN

This section describes the design of IC3. We discuss how IC3 analyzes static conflict information (§ 3.1), how its runtime utilizes this information to track and enforce dependency (§ 3.2), and lastly, how IC3 ensures that all tracked dependencies are enforceable (§ 3.3). We provide a proof sketch for correctness in § 3.5.



(a) SC-graph when considering each operation as a single piece

(b) SC-graph after merging

Figure 2: SC-graph [41] with two instances of the same transaction type.

3.1 Static Analysis

IC3 is an in-memory database with a one-shot transaction processing model, i.e., a transaction starts execution only when all its inputs are ready Many existing databases have the same transaction model (e.g., H-store [42], Calvin [45, 47], Silo [48]). IC3's current API provides transactions over sorted key/value tables whose primitives for data access are: *Get*("tableName", key), *Put*("tableName", key, value), *Scan*("tableName", keyRange)[1].

In this paper, we assume the transaction workload is static and known before the execution of any transaction. We relax this assumption and propose techniques to handle dynamic workloads and ad-hoc transactions in the technical report [50]. IC3 performs the static analysis by first constructing a static conflict graph (SC-graph), which is introduced by prior work on transaction chopping [4, 41]. In an SC-graph, each transaction is represented as a series of atomic *pieces* each making one or more data access. Pieces

[1] Get/Put/Scan also optionally take column names as parameters to restrict access the specified subset of columns.

within a transaction are connected by a S(ibling)-edge. To construct a SC-graph for a given workload, we include *two* instances of each transaction type and connect two pieces belonging to different transactions with a C(onflict)-edge if they *potentially* conflict, i.e., two pieces access the same column of the same database table and one of which is a write access. IC3 analyzes user transaction code to decompose each transaction into pieces and to construct the resulting SC-graph (see details in § 5). Suppose the example of Figure 1a corresponds to a workload consisting of the following one type of transaction:

MyTransaction(k_1,v_1,k_2,v_2):
  v_1 =**Get**("Tab1", k_1)
  Put("Tab1", k_1, v_1')
  v_2 =**Get**("Tab2", k_2)
  Put("Tab2", k_2, v_2')

The corresponding SC-graph for this workload is shown in Figure 2a. SC-graphs reveal potential violations of serializability at runtime. Specifically, if there exists an SC-cycle [41] which contains both S- and C-edges, then a corresponding cyclic dependency (i.e., a violation of serializability) may arise at runtime. Figure 2a has an SC-cycle containing two instances of the same transaction, therefore, there is a danger for T_1 and T_2 to enter a cyclic dependency at runtime, e.g., with execution order $R_1(A)$, $W_1(A)$, $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$, $R_1(B)$, $W_1(B)$. While transaction chopping [41] combines pieces to eliminate all SC-cycles, IC3 renders SC-cycles harmless at runtime by constraining piece execution to prevent the corresponding dependency cycles.

Algorithm 1: RunPiece(p, T):

```
   Input: p is a piece of transaction T
1
2  // Wait Phase:
3  if p does not have any C-edge                      // case-1
4     skip to execute phase
5  foreach T' in T.depqueue:
6     if ∃ p' of T' and p has a C-edge with p'        // case-2
7        wait for p' to commit
8     else                                            // case-3
9        wait for T' to commit
10
11 // Execute Phase
12 run user code to execute p
13
14 // Commit Phase
15 lock records in p's read+writeset (using a sorted order)
16 validate p's readset
17 foreach d in p.readset:
18    T_w ← last write tx in DB[d.key].acclist
19    T.depqueue += T_w
20    DB[d.key].acclist += (T,"reader")
21 foreach d in p.writeset:
22    T_rw ← last read/write tx in DB[d.key].acclist
23    T.depqueue += T_rw
24    DB[d.key].acclist += (T,"writer")
25    DB[d.key].stash = d.val
26 release grabbed locks
27
28 return status //whether p has committed or aborted
```

3.2 Dependency Tracking and Enforcement

IC3 executes a transaction piece-by-piece and commits the transaction after all its pieces are finished. The atomicity of each piece is guaranteed using traditional concurrency control (OCC). IC3 ensures the serializability of multi-piece transactions by tracking and enforcing dependencies among concurrent transactions. To control performance overhead and improve multi-core scaling, IC3 only tracks and enforces direct dependencies at runtime. Without static analysis, a transaction T must wait for *all* its dependent transaction to commit before T can perform *any* subsequent data access, which precludes most scenarios for safe parallel execution. Static analysis helps IC3 avoid or shorten such costly wait in many circumstances.

Algorithm 1 shows how IC3 executes each piece and Algorithm 2 shows how IC3 executes and commits a multi-piece transaction. Together, they form the pseudo-code of IC3. IC3 uses several data structures to perform direct dependency tracking. First, each transaction T maintains a dependency queue, which contains the list of transactions that T is directly dependent on. Second, the database maintains an accessor list [36], one per record, which contains the set of not-yet-committed transactions that have either read or written this record.

Piece execution and commit. The execution of each piece consists of three phases, as shown in Algorithm 1:

- *Wait Phase (lines 3-9).* Once a transaction T has become dependent on another transaction T', how long should T wait before executing its next piece p? There are three cases, as determined by information from the static analysis. In the first case (lines 3-4), piece p does not have any C-edges and thus is not part of any SC-cycle involving T and T'. This means one cannot violate $T' \to T$ by executing p immediately, without constraints. In the second case, piece p is part of a SC-cycle that involves only T and its dependent transaction T'. In other words, p has a C-edge connecting to some piece p' in T'. For case-2, p only needs to wait for p' to finish (line 6-7). The third case is when neither case-1 or case-2 applies (line 8-9), then T has to wait for T' to finish all pieces and commit. The intuition for why p only needs to wait for p' in the second case is subtle; basically, if IC3 ensures that no (direct) dependencies are violated due to the SC-cycle involving only T and T', then no transitive dependencies are violated due to SC-cycles involving T, T' and some other transaction(s). To see why, we refer readers to the proofs (§ 3.5 and Appendix A). Note that case 2 is actually quite common; SC-cycles are most likely to occur among instances of the same transaction type (Figure 2a shows such an example). These SC-cycles make case-2 applicable.

- *Execution Phase (line 12).* In this phase, IC3 executes user code, and accesses database records. In the database, the value of a record reflects the write of the last committed transaction. In addition, each record also keeps a stashed value, which reflects the write of the last committed piece. A read of the record returns its stashed value, instead of the actual value. This ensures that a transaction T can read the writes of

Algorithm 2: RunTransaction(T):

```
1  // Execute Phase
2    foreach p in T:
3      RunPiece(p, T)
4      retry p if p is aborted (due to contention)
5
6  // Commit Phase
7    foreach T' in T.depqueue:
8      wait for T' to commit
9    foreach d in T.writeset:
10     DB[d.key].value = d.val
11     delete T from DB[d.key].acclist
12
13 return status //whether T has committed or aborted
```

completed pieces in another transaction T' before T' commits. This allows for more interleavings that are not available to 2PL or OCC. All the writes of a piece are simply buffered locally.

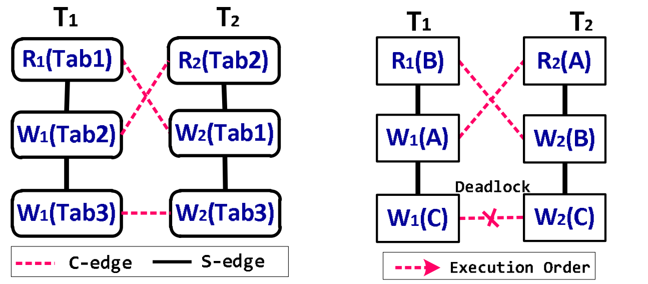
- *Commit Phase (lines 15-25).* First, IC3 must check that p has executed atomically w.r.t. other concurrent pieces. We perform the validation according to OCC: IC3 grabs all locks on the piece's readset and writeset (line 15) and checks if any of its reads has been changed by other pieces (line 16, with details ignored).

  Second, we augment T's dependency queue (T.*depqueue*) due to any write-read relations that arise from p's reads. Specifically, for each record in p's readset, we append to T.*depqueue* the last writer according to the record's accessor list (line 18-19). We then add T to the record's accessor list.

  Third, we update T.*depqueue* and the corresponding accessor lists based on p's writeset. This update is similar to that done for p's readset, except we must account for both read-write and write-write relations (lines 22-24). We then update the record's stashed value with the write value (line 25).

Transaction commit. After all its pieces have finished, transaction T proceeds to commit. First, T must wait for all its dependent transactions to finish their commits (lines 7-8). This is crucial because we must ensure that T's writes are not overwritten by transactions that are ordered before T in the serialization order. Then, it updates the value of the corresponding record in the database according to its writeset and removes itself from the record's accessor list (lines 10-11). We note that, unlike 2PL or OCC, the commit phase does not require holding locks across its writeset. This is safe because 1) T explicitly waits for all transactions serialized before itself to finish, and 2) conflicting transactions that are to be serialized after T wait for T's writes explicitly during piece execution (lines 5-9).

Examples. Now we use the examples in Figure 1a and Figure 1b again to see how IC3 works. After merging the deadlock prone SC-cycles shown in Figure 2a into Figure 2b (details in § 3.3), we can enforce the interleaving as follows: after T_1 has executed the piece $RW_1(A)$, IC3 appends T_1 to record A's accessor list. When T_2's first piece, $RW_2(A)$, finishes, IC3 checks A's accessor list and puts T_1 in T_2's dependency queue. When T_2 tries to execute its next piece, $RW_2(B)$, it will have to wait for $RW_1(B)$ to finish because the two pieces are connected by a C-edge and $RW_1(B)$'s transaction (T_1) appears in T_2's dependency queue. In Figure 1b, T_2 is in T_3's dependency queue when T_3 is about to execute $W_3(C)$. Since, $W_3(C)$ has a C-edge with some transaction but not with T_2, it has to wait for T_2 to finish all pieces and commit. As T_2 can only commit after its dependent transaction T_1 commits, $W_3(C)$ only executes after T_1 has finished, thereby ensuring $W_1(C) \rightarrow W_3(C)$ and guaranteeing serializability.



(a) $R1, W1, R2, W2$ is a deadlock-prone SC-cycle.

(b) An example runtime interleaving with deadlocks.

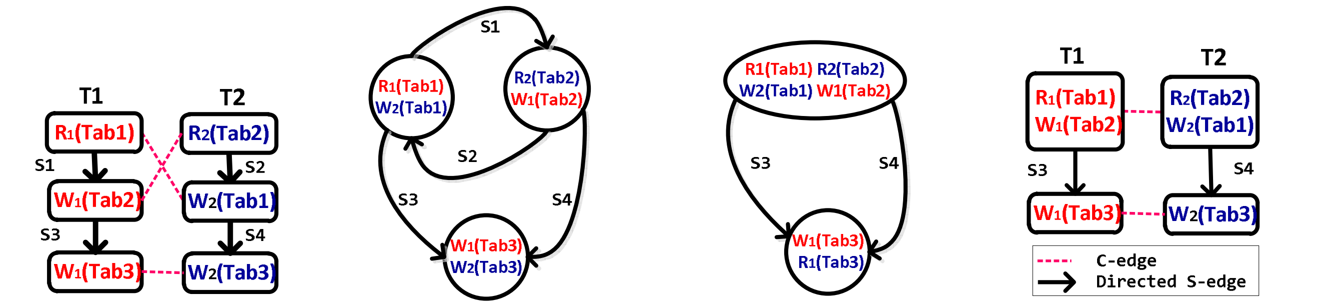
Figure 3: A deadlock-prone SC-cycle suggests potential runtime deadlock.

3.3 Ensuring Tracked Dependencies are Enforceable

As we discussed earlier, tracked dependencies are not always enforceable at runtime for arbitrary workloads. When this happens, the basic IC3 algorithm encounters deadlocks as pieces enter some cyclic wait pattern. The traditional approach is to run a deadlock detection algorithm and abort all deadlocked transactions. Unlike 2PL, however, in the context of IC3, aborts have a cascading effect: not only must we abort all deadlocked transactions, we must also abort all transactions that have seen the writes of aborted transactions and so forth. A number of concurrency control protocols adopt such an approach [2, 36]. However, as we see in Sec 6, cascading aborts can be devastating for performance when the workload has a moderate amount of contention.

IC3 adopts a different approach to prevent deadlocks. Again, we leverage the static analysis to identify those pieces that can cause potential deadlocks. We combine those pieces (belonging to the same transaction) together into one larger atomic piece. This ensures all tracked dependencies at enforceable at runtime with no risks of deadlocks.

Given a SC-graph, what patterns suggest potential deadlocks? Consider the example in Figure 3a in which T_1 accesses table Tab_1 before Tab_2 and T_2 does the opposite. In the underlying SC-graph, this corresponds to an SC-cycle whose conflicting pieces access different tables in an inconsistent order. We refer to this SC-cycle as a deadlock-prone SC-cycle. Deadlock-prone SC-cycles indicate potential runtime deadlocks. Figure 3b gives an example execution that is deadlocked: T_1 reads record B before T_2's write to B ($T_1 \rightarrow T_2$) while at the same time T_2 reads record A before T_2's write to A ($T_2 \rightarrow T_1$) This cyclic dependency will eventually manifest itself as a deadlock when T_1 and T_2 try to commit. In the example of Figure 3b, the deadlock appears earlier as T_1 and T_2 wait for each other before they attempt to modify record C. In addition to accessing different tables



(a) Make S-edge directed to reflect the execution order within a transaction.

(b) Put pieces connected with C-edges into the same vertex

(c) Make the graph acyclic by merging the vertexes involved in a cycle

(d) For each merged vertex, combine pieces of the same transaction into one piece.

Figure 4: How to combine pieces to remove deadlock-prone SC-cycles and avoid runtime deadlock.

in an inconsistent order, an SC-cycle is also deadlock-prone if multiple conflicting pieces of a transaction access the same table.

To eliminate deadlock-prone SC-cycles, IC3 combines pieces into a larger atomic piece using the following algorithm. First, we add direction to each S-edge to reflect the chronological execution order of a transaction(Figure 4a). Second, we put all pieces connected by C-edges into a single vertex. The first two steps result in a directed graph whose edges are the original S-edges (Figure 4b). Third, we iteratively merge all the vertexes involved in the same directed cycle into one vertex until the graph is acyclic (Figure 4c). Last, we combine those pieces of a merged vertex that belong to the same transaction into a single piece. Figure 4d shows the resulting SC-graph of Figure 3a after combining those pieces. Note that there are no longer deadlock-prone SC-cycles in Figure 4d, so IC3 is guaranteed to encounter no deadlocks when enforcing tracked dependencies.

3.4 Applicability of IC3.

IC3 is not beneficial for all application workloads; it is only effective for contentious workloads that access multiple tables within a single transaction. For these workloads, the static analysis is likely to produce transactions containing multiple pieces, thereby allowing IC3 to exploit the parallelism opportunity overlooked by 2PL/OCC by interleaving the execution of conflicting pieces. For workloads that only access a single table within a transaction, IC3 does not provide any performance gains over traditional concurrency control such as 2PL or OCC. For example, if all transactions only do read and then modify a single database record, then static analysis merges the two access so that all transactions consist of a single (merged) piece. As a result, IC3 behaves identically to the traditional concurrency control protocol which it uses to ensure the atomicity of a piece.

Workloads that access multiple tables within a transaction are quite common in practice. For example, all transactions in the popular TPC-C and TPC-E benchmarks access multiple tables and thus contain multiple pieces (Section 6 gives more details). We have also manually analyzed the 14 most popular Ruby-on-Rails applications on GitHub and found most of them contain transactions accessing multiple tables (Appendix C).

3.5 Proof Sketch

Due to space limitation, we only give a proof sketch here. A more rigorous proof is included in the appendix. We are going to prove that IC3 always ensure serializability by only generating acyclic serialization graph.

In a proof by contradiction we assume there is a cycle in the serialization graph, denoted by $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$. Treating each piece as a sub-transaction, the cycle can be expanded as $q_1..p_1 \rightarrow q_2..p_2 \rightarrow \ldots \rightarrow q_n..p_n \rightarrow q_1$. Because we use OCC to protect the atomicity of each piece, the directed edge in the cycle reflects the chronological commit order of pieces, therefore there must be at least a pair of q_i and p_i such that they are not the same piece. Then this cycle corresponds to an SC-cycle in the SC-graph, which means no piece in the cycle can skip checking direct dependent transactions in Algorithm 1. Moreover, the cycle necessarily implies a deadlock at transaction commit time, so none of the transaction in the cycle can commit.

Consider a fragment $p_i \rightarrow q_j \rightarrow p_j$ in that cycle. For p_j, because it can neither skip checking phase nor successfully wait for T_i to commit (T_i can never commit due to deadlock), p_j can only execute after it waits for the commit of a piece r_i in T_i that has a C-edge connection to itself. Our static analysis ensures that this r_i must be chronologically later than p_i, otherwise there will be a non-existent cycle $q_j \rightarrow p_j \rightarrow q_j$ in the SC-graph.

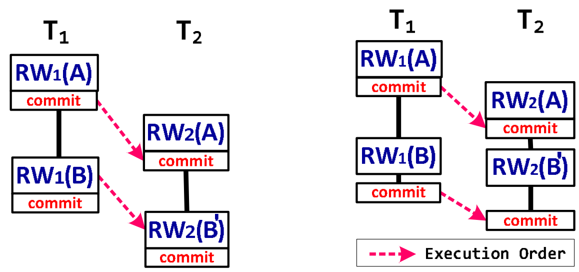
Using the above property, we can inductively "shrink" the cycle until there is no $q_i \rightarrow p_i$ in the cycle. Eventually after m ($m < n$) times iteration, we will get a new cycle, represented as $q_1^m \leftarrow p_1^m \rightarrow q_2^m \leftarrow p_2^m \rightarrow \ldots \rightarrow q_{n-m}^m \leftarrow p_{n-m}^m \rightarrow q_1^m$, which necessarily implies a directed cycle in the SC-graph we have already eliminated, which is a contradiction to the result of static analysis. Therefore, a cycle in the serialization graph cannot exist, hence IC3's scheduling is always serializable.

4. OPTIMIZATION AND EXTENSIONS

This section introduces several important performance optimizations and extends IC3 to support user-initiated aborts.

4.1 Constraining Pieces Optimistically

Static analysis can cause the runtime to unnecessarily constrain certain safe interleaving. Figure 5a shows an example runtime execution for the SC-graph in Figure 2. T_1 and T_2 write to the same record A in Tab1 but write to different records (B and B') in Tab2. However, under Algorithm 1, as T_2 is dependent on T_1 after both write to A, T_1 can not execute $W_2(B')$ until $W_1(B)$ is finished. Such unnecessary constraining sacrifices parallelism.



(a) Static analysis may constrain piece execution unnecessarily.

(b) Constrain piece optimistically by moving the waiting phase to be after piece execution and before its commit.

Figure 5: Constrain the interleaving optimistically to reduce false constraint

As each piece is protected by OCC, we can constrain its execution optimistically to unlock more parallelism. The basic idea to constrain each piece's commit phase instead of its execution phase. This can be done by reordering the wait phase and execution phase in Algorithm 1. For each piece, it first optimistically executes the piece without blocking (execute phase). Then, it then waits according to the rules of the wait phase. Last, it will validate and commit the piece.

Figure 5b shows the allowed interleaving of the previous example with optimistic constraining. Unlike before, $W_2(B')$ can execute concurrently with $W_1(B)$. However, before T_2 commits its second piece, it needs to wait for T_1's second piece to finish committing. As the commit time of each piece is typically shorter than its execution time, this optimization reduces the size of critical section in which execution needs to be serialized.

4.2 Rendezvous Piece

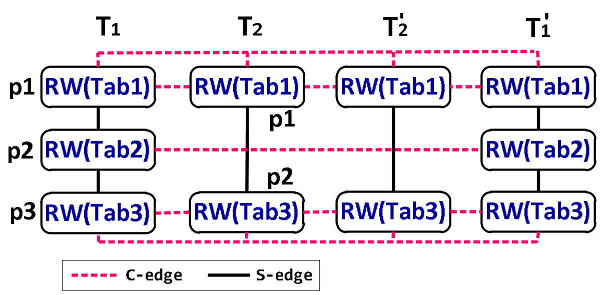


Figure 6: The SC-graph involving two types of transactions. T_2's P2 is the rendezvous piece of T_1's P2 and T_1''s P2.

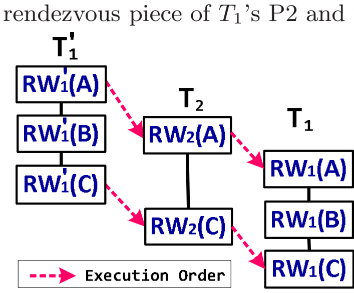


Figure 7: Constrain the interleaving with the rendezvous piece. Before T execute $W_1(B)$, it only needs wait T_2 finishes $W_2(C)$

When executing a piece p in T that is potentially involved in any conflicts (Algorithm 1, line 5-9), IC3 needs to find T's dependent transaction T', and then either 1) wait for a piece in T' that may potentially cause conflicts with p to finsih, or 2) wait for the entire T' to commit if no such pieces in T' exist. In the latter case, the wait could be expensive depending on how soon T' will commit. As an optimization to reduce this cost, we introduce a concept named *rendezvous piece*. For piece p in transaction T, its rendezvous piece in another transaction T' works as a promise that after this rendezvous piece finishes, T' can form no more conflicts with T involving p. More specifically, if we can find a later piece q in T (after p) and another piece r in T', such that q and r are connected by an C-edge in static analysis, r will be a rendezvous piece of p in T'. We then can use r to synchronize T with T'. In this case, IC3 can simply wait for r to finish before committing p, and doing so is sufficient to ensure no undesirable interference could ever happen.

During static analysis, after merging operations, IC3 looks for rendezvous pieces of every piece in other transaction types, and tags the information in the generated SC-graph. Afterwards, IC3 can leverage these information at runtime to cut down waiting time. As an example, Figure 6 shows an SC-graph with two types of transactions T_1 and T_2. The piece p_2 of T_1 has no C-edge directly connecting to T_2, but a later piece p_3 of T_1 directly connects the p_2 of T_2. During static analysis, p_2 is identified as the rendezvous piece of p_2 of T_1. Afterwards, during the execution, before T_1 executes p_2, it only needs to wait until T_2 finishes p_2, as shown in Figure 7.

4.3 Commutative Operation

Two write pieces commute if different execution orders produce the same database state. Therefore, it may seem unnecessary to constrain the interleaving of commutative pieces. However, this is not correct as write pieces do not commute with read operations that IC3 uses to generate a periodic consistent snapshots in the background [2]. IC3 solves this issue by deferring the execution of commutative operation to the transaction commit phase[3]. In our current implementation, users make commutativity explicitly by using a new data access interface, *Update*("tableName", key, value, *op*) where *op* is user-specified commutative accumulation function such as addition.

4.4 Handling User-initiated Aborts

So far, we have presented IC3 on the assumption that transactions are never aborted by users and thus can always be retried until they eventually succeed. We now explain how IC3 handles user-initiated aborts.

In IC3, when a transaction is aborted (by the user), all the transactions that depend on it must also be aborted. IC3 supports this cascading aborts by using an abort bit for each entry in the accessor list. When a transaction aborts, it unlinks its own entry from each accessor list that it has updated and sets the abort bit for all the entries appearing after itself. Before a transaction commits, it checks the abort bits of all its accessorlist entries. If any of its entry has the abort bit, then the transaction aborts itself.

[2] IC3 runs read-only transactions separately from read-write transactions by using thse snapshots, same as is done in Silo [48].

[3] Such deferring is always possible because commutative write pieces do not return any values used by later pieces. Otherwise they do not commute.

5. IMPLEMENTATION

We have built the IC3 runtime in C++ and implemented the static analyzer in Python. This section describes the basic implementation that performs static analysis in a separate pre-processing stage. We have also extended IC3 to perform online analysis in order to handle dynamic workloads, the details of which are given in [50].

5.1 Pre-processing and Static Analysis

The static analysis is done by a Python script in the pre-processing stage, before the system is to execute any transaction. Thus, the overhead of static analysis does not affect IC3's runtime performance. The analyzer parses the stored procedure code (in C++) to determine which table access belongs to which transaction piece and to construct the corresponding SC-graph among transactions. The granularity of the analysis is at the table and column level, i.e. two pieces are connected by a C-edge if both access the same column in the same table and one of the access is a write. To simplify code analysis, we restrict user code to specify the names of tables and columns accessed via the Get/Put/Scan API as constants. Thus, the analyzer can extract each table access simply, label each access as an individual piece, and determine which pieces are connected via C-edges. When processing an $if-else$ statement, the analyzer includes the table access of both branches in the corresponding transaction in the SC-graph. For table accesses within a for-loop, they are grouped into one piece (it is possible to split loops using known compiler techniques, but we did not implement it.)

After the analyzer obtains the raw SC-graph by parsing stored procedure code, it applies the algorithm in § 3.3 to merge some pieces if necessary. For each table access, the final output of the analyzer contains its corresponding piece identifier and its C-edge information (i.e. the identifiers for those pieces with a C-edge to it). The IC3 runtime loads the C-edge information based on the analyzer's output prior to execution. We currently rely on manual annotation to associate the piece identifier with each table access. If a piece contains more than one table access, users must enclose those access within a pair of begin_piece and end_piece statements. As a result of the pre-processing, the IC3 runtime has sufficient information to implement line 3 and 6 in Algorithm 1.

5.2 Runtime Implementation

IC3's runtime implementation is based on the codebase of Silo [48], a high-performance in-memory database. Silo implements OCC on top of Masstree [31], a concurrent B-tree like data structure. We change the concurrency control protocol to IC3 and reuse Silo's implementation to support read-only transactions using snapshots, durability, and garbage collection.

IC3 uses a *tuple* to represent each table record and its meta-data. Among others, the meta-data includes the record's accessor list (implemented as a linked list), the stashed value and a version number for ensuring piece atomicity using OCC.

IC3 uses n worker threads to execute transactions, where n is the number of cores in the machine. We implement MCS lock [32], a type of spinlock that scales well to many cores [9]. Each tuple is associated with a separate MCS lock which protects the atomicity of operations on the tuple.

Each running transaction is associated with a data structure, referred to as its context, that holds information such as the transaction ID, its dependency queue, the current piece being executed, and the status of the transaction. The context data structure has a fixed size and is reused for a new transaction.

Piece Execution and Commit (Lines 11-26, Algorithm 1). IC3 uses OCC to ensure the atomicity of each piece. During piece execution, IC3 buffers writes in the piece's write-set and copies the version number of each record being read in the piece's read-set. To commit piece p, IC3 first acquires the MCS locks for all tuples corresponding to p's read- and write-set. Then, IC3 checks whether the version numbers in the read-set differ from those currently stored in the tuples. Suppose validation succeeds for piece p of transaction T. Then, for each record accessed by p, IC3 adds the piece's transaction ID (T) to the tuple's accessor list, updates the tuple' stash value if the corresponding access is a write, and copies the transaction ID of the last conflicting entry in the tuple's accessor list to T's dependency queue (lines 17-25, Algorithm 1). Last, IC3 releases the MCS locks.

Piece wait (lines 2-9, Algorithm 1). There are three cases for how long a piece should wait before execution. The IC3 runtime can determine which case applies to piece p by examining p's C-edge information. This information is calculated by the static analyzer during pre-processing and loaded in the runtime's memory prior to execution.

By default, the worker thread waits by spinning. If piece p needs to wait for its dependent transaction T' to commit (line 8, Algorithm 1), the worker thread continuously reads several pieces of information in T''s context and checks 1) whether the status of T' becomes "committed". 2). whether the transaction ID stored in the context of T' has been changed; this occurs when T' finishes and its context data structure is re-used by other transactions. If piece p needs to wait for a specific piece p' in T' to commit (line 6-7, Algorithm 1), the worker thread also additionally spins to check whether the current piece being executed by T' is after p'. If so, piece p can proceeds to execution without waiting for T' to commit. Each read of the spin variables (i.e. transaction status, transaction ID, or current piece identifier) is done without locking because the underlying memory instructions involving no more than 8-bytes are atomic. IC3 adds a memory barrier whenever a spin variable is modified.

Apart from the default spin-wait strategy, we also explored a suspend-wait strategy in which a worker thread suspends a blocked transaction to execute others. The implementation of suspend-wait is done using C++ co-routines. If a piece needs to wait, the worker thread puts the current transaction in the thread's pending queue and switches to execute other runnable transactions (that the worker has previously suspended) or a new one. We implement the pending queue as a lock-free queue so that suspending a transaction will not block other worker threads.

We choose spin-wait as the default strategy because it leads to better performance in our experiments. For example, when running TPC-C new-order transactions with 64 worker threads accessing one warehouse, the spin-wait strategy has 2.14X throughput than the suspend-wait strategy (340K TPS vs. 158K TPS). This is because the overhead of the context switching and lock-free queue manipulation out-weights the overhead of spin-wait for short waits which are the case in TPC-C. Suspend-wait may be beneficial for workloads involving long waits.

5.3 Secondary Index Implementation

The current implementation of IC3 relies on users to manually update and lookup in secondary indexes, similar to what is done in Silo [48] and Calvin [16]. Specifically, users need to represent each secondary index as a separate table that maps secondary keys to the records' primary keys. To ensure consistency, one must access the secondary index table in the same user transaction that reads or writes the base table. For example, in our implementation of the TPC-C benchmark, the new-order transaction inserts into the base *order table* and then inserts to the *order table*'s secondary index, all in the same transaction. Because secondary indexes are read or updated as part of regular user transactions, we can use the same static analysis and runtime implementation.

6. EVALUATION

This section measures the performance and scalability of IC3. We first compare IC3 to OCC and 2PL using microbenchmarks and TPC-C. We use the OCC implementation of Silo [48]. We implement 2PL ourselves by associating each record with a scalable read-write lock [4]. Then, we present the comparison with other alternatives with contended workloads. Last, we analyze the effect of different optimization methods. Due to space limitation, we leave evaluations on the TPC-E benchmark, the effect of user-initiated aborts (which may cascade) and transitive dependencies in the technical report [50].

6.1 Experiment Setup

Hardware. Our experiments are conducted on a 64-core AMD machine with 8 NUMA nodes. Each node has 8 cores (Opteron-6474). Each core has a private 16KB L1 cache and every two cores share a 2MB L2 cache. Each NUMA node has a 8MB L3 cache and 16GB local memory (128GB in total). The machine runs a 64-bit 4.0.5 Linux kernel.

Workloads and metrics. This evaluation includes two benchmarks, a microbenchmark which does simple updates and the TPC-C [44] benchmark. We use TPC-C to compare the performance of IC3 with alternatives.

For each benchmark, we first evaluate the performance of IC3 with increasing contention rate. Then, we study system scalability under high contention.

Throughput is the primary metric in this evaluation. Every trial is run for a fixed duration (i.e. 30 seconds) to record a *sustainable throughput*. We manually pin each thread to a single core, and ensure that the number of worker threads equals to the number of cores. When disk logging is enabled, IC3 can only scale up to 16 cores due to the limitation of I/O bandwidth (two disks) in our test machine; thus, we disable logging by default.

6.2 Microbenchmark

We begin our evaluation with a simple microbenchmark involving one type of transaction. Each transaction executes a controllable number of pieces and each different piece accesses a different table. Each piece randomly updates 4 distinct records from the same table. Each piece is protected using 2PL. To avoid deadlock, the records are sorted when each piece starts. Each table has 1M records. Each record has a 100-byte value and a 64-bit primary key.

For each piece, the contention rate is controlled by varying the selection scope of its first operation between 10 and 1M. When a piece selects a record out of 10, it has 10% contention rate. Figure 8(a) shows the performance of workloads with each transaction with moderate size (10 pieces). When the selecting scope is 1,000,000 (contention rate is nearly 0), IC3's throughput is about 15% lower than that of 2PL and OCC (167K vs. 196K vs. 189K TPS). This extra cost is mainly from manipulating the linked list for dependency tracking. When the contention rate increases to 1% with 100 records to select from, the performance of IC3 remains unchanged (167K TPS) , while 2PL and OCC have 23% and 52% performance slowdown accordingly (149K vs. 196K and 93K vs.189K TPS). OCC's performance drops greater than 2PL because the 1% contention rate among pieces can cause 65% transaction aborts in OCC. When the contention rate is greater than 1%, IC3's performance degrades more gracefully. With 10% contention rate, IC3 has 45% performance loss, (90K vs. 167K TPS), while the throughput of 2PL and OCC drops 86% and 90% respectively (27K vs. 196K and 18K vs. 189K TPS).

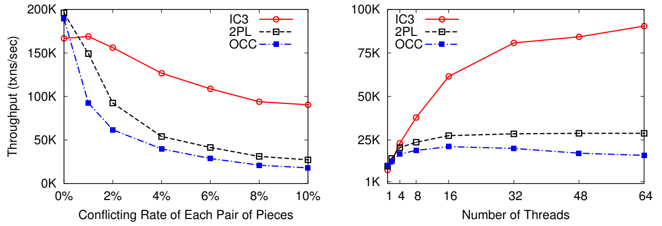
We also evaluate IC3's performance speedup with concurrent execution under highly contended workload. Figure 8(b) shows the throughput improvement under highly contented workload (10% abort rate) with an increasing number of worker threads. The throughput of IC3 keeps growing to 64 worker threads, while 2PL and OCC can only scale to 16 cores. With 64 worker threads, IC3 outperforms its single thread version by 9X (90K vs. 9.9K TPS), while 2PL gets 3X speedup (29K vs. 10K TPS) and OCC only gets 1.5X speedup (16K vs. 10K TPS).

We also analyze the effect of transaction length on performance. For the workload with short transactions having only 2 pieces, it has 70% performance degradation when the contention rate is increased from 0 to 10% (1350K vs. 410K TPS). Under high contention level with 10% abort rate, IC3 achieves 10X speedup with 62 worker threads compared with the performance with only 1 thread (43K TPS vs. 410K TPS). For the workload with longer transaction containing 64 pieces, it has 37% performance degradation when the contention rate is increased from 0 to 10% (23K vs. 15K TPS). For high contended workload with 10% abort rate, IC3 achieves 15X speedup (1K vs. 15K TPS) over a single thread. IC3 can improve the workloads' performance for workload under high contention regardless of the number of pieces in the transaction. However, IC3 achieves more speedup for longer transactions.

6.3 TPC-C

In TPC-C benchmark, three out of five transactions are read-write transactions that are included in the static analysis and processed by IC3's main protocol. The SC-graph is shown in Appendix B. The other two are read-only transactions and are supported by the same snapshot mechanism in Silo [48]. In TPC-C, we decrease the number of warehouses from 64 to 1 to increase the contention rate. When there are 64 warehouses, each worker thread is assigned with a local warehouse, so each thread will make most of the orders using its local warehouse. Only a small fraction of transac-

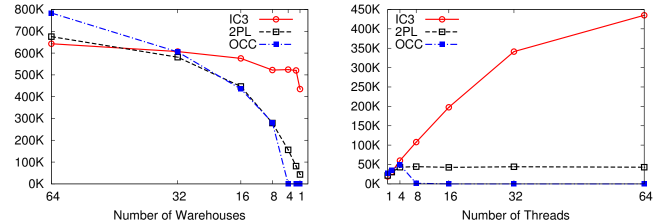
[4]Our implementation of 2PL eliminates the chance of deadlock by sorting the locks in a consistent order



(a) Throughput with increasing contention rate.

(b) Throughput with increasing number of worker threads.

Figure 8: *Microbenchmark*



(a) Throughput with increasing contention rate.

(b) Throughput with increasing number of threads.

Figure 9: *TPC-C Benchmark*

tions will access remote warehouses. When there is only one warehouse, all worker threads make orders from this global shared warehouse, suggesting a high contention rate.

Figure 9(a) shows throughput of 64 worker threads with the increasing contention rate. Under low contention, IC3's throughput is 6% lower than 2PL due to the overhead of dependency tracking. However, both IC3 and 2PL have around 15% performance degradation than OCC due to mandating locks on read-only accesses. For IC3, except manipulating the accessor list, it also needs to acquire the lock on read-only operations if the access is included in the conflict piece. For 2PL, it needs to acquire the read lock before each read-only access.

Under high contention rate, IC3 performs much better than 2PL and OCC. As the number of warehouses decreases to 1, the throughput of IC3 only drops by 32% (435K vs. 643K TPS). In contrast, the performance of 2PL and OCC degrades dramatically when the total number of warehouses is less than 16. OCC suffers 64% throughput degradation at 8 warehouses with 63% abort rate. When there are only 4 or less warehouses, OCC has only a few hundreds of TPS (e.g. 341 TPS for 4 warehouses) as the abort rate is more than 99%. For 2PL, its performance degrades 59% with 8 warehouses and 93% with 1 warehouse.

Figure 9(b) shows the scalability of IC3 when sharing a global warehouse. IC3 can scale to 64 threads with 22X speedup over one thread (435K vs. 19K TPS)). However, the performance is only improved by 27% from 32 cores to 64 cores. The major reason is that all payment transactions update the same warehouse record and contend on the same cache line. This can be further optimized by distributing the record, such as phase reconciliation [34]. One may find that IC3's scalability with TPC-C is better than the one with microbenchmark. This is because not all pieces' execution in TPC-C will be constraint (i.e., commutative pieces, read-only pieces and read-only transactions). Such pieces can be concurrently executed with the conflicting pieces, which increases the concurrency. Both OCC and 2PL can only scale to 4 cores, with 49K TPS and 43K TPS accordingly. After 4 threads, the performance of 2PL does not change notably, but the performance of OCC degrades drastically because conflicting transactions are frequently aborted.

6.4 Comparison with Alternative Approaches

This subsection continues to use TPC-C to compare IC3 with four important alternative approaches under highly contended workloads.

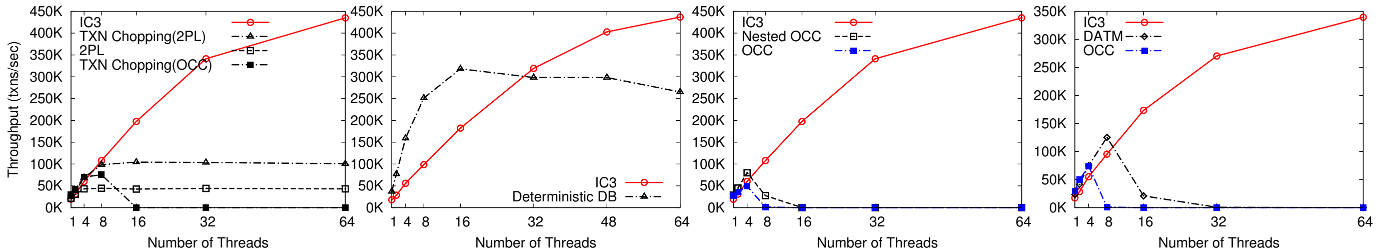
Transaction chopping: We first apply transaction chopping [41] to TPC-C and merge pieces to eliminate SC-cycles. Each merged piece is protected using 2PL or OCC. We also exclude C-edges between any commutative operations. Figure 10a shows the performance when all worker threads share a global warehouse. Transaction chopping with 2PL only gets marginal benefit compared to 2PL: it cannot scale beyond 8 threads, as most operations of new-order and delivery transaction need to be merged. After 8 threads, IC3 scales better than transaction chopping and its performance is 4X under 64 threads(435K vs. 100K TPS). When using OCC to protect the merged piece (Chopping-OCC in Figure 10a), it has similar performance as chopping-2PL when the number of threads is less than 8. With 8 worker threads, its performance is worse than chopping-2PL (75K vs. 99K TPS) due to piece aborts and retries. As contention further increases with more than 8 worker threads, the throughput of chopping-OCC degrades significantly.

Deterministic database with lazy evaluation: We also compare against deterministic database with lazy evaluation [16]. Deterministic database divides each transaction into two phases: now phase and later phase. The system needs to execute all transactions' now phases sequentially to generate a global serialized order and analyze the dependency of their later phases. Then the later phases can be executed deterministically according to the dependency graph generated in the now phase. Execute high conflicting operations in the now phase can improve the temporal locality and increase the concurrency of later phases.

We use the implementation released with the paper. Figure 10b shows the scaling performance of deterministic database under one global TPC-C warehouse. We use one extra worker thread to run all the transaction's now phases and use the same number of worker threads with IC3 to run the later phase concurrently. We tune the setting such that the number of transaction can be buffered in the now phase and choose the threshold (i.e., 100) with the highest throughput. We issue 1M transactions to the system for processing to get the throughput result.

With a single thread, lazy evaluation is better than IC3 (38K vs. 19K TPS). This is because: 1) One extra thread is used to execute all highly contended operations which parallelise the execution and achieve good cache locality; 2) Some operations are deferred and even never executed; 3) Different code base for both benchmark and system implementation also contribute the performance difference. When the size of transaction batch is 100, their system achieves highest throughput under 16 worker threads (318K) [5]. However, its performance degrades after 16 threads and IC3 can even

[5] This number is lower than what is reported in [16] because their evaluation used a more powerful CPU (Intel Xeon E7-8850)



(a) Transaction chopping (b) *Lazy evaluation* (c) OCC across pieces (d) Dependency-aware OCC

Figure 10: *Comparison with prior approaches under high contention. New-order only for dependency-aware OCC*

achieve better performance after 32 threads.

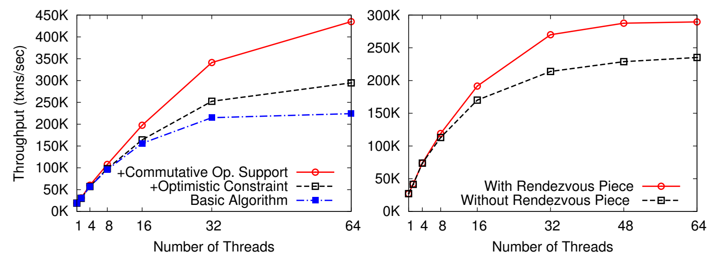
Using OCC to ensure serializability across pieces. We evaluate a another strategy, called Nested OCC, that chops each transaction into pieces and uses OCC to ensure atomicity within and across pieces. At the end of each piece, it validates the records read by the piece and retries the piece if validation fails. At the end of the transaction, it validates the records read in all pieces. Similar to IC3, commutative operations are not included in the conflicting pieces.

Figure 10c shows performance of Nested OCC for TPC-C under high contention. Since the contention within a piece only causes the re-execution of the piece instead of the whole transaction, Nested OCC has slower abort rate than OCC with 4 worker threads (15% vs. 42%). As a result, its performance is 62% better than OCC and 30% better than IC3 with 4 worker threads. With an increasing number of threads, the throughput of Nested OCC drops significantly like OCC due to dramatically increased abort rate. With 64 worker threads, its abort rate is 98% which is the same as OCC. This is because, with more worker threads, pieces are more likely to interleave in a non-serializable manner, thereby causing aborts. By contrast, as IC3 enforces the execution order among the concurrent conflicting pieces, thereby avoiding aborts and achieving good scalability with many threads.

Dependency-aware transactional memory (DATM) [36] is the last compared algorithm. Under DATM, a transaction can observe the update of conflicting uncommitted transactions. All conflicting transactions will commit successfully, if the interleaving of the conflicting operations are safe. However, this method also allows unsafe interleaving which will incur cascading abort.

We port DATM by modifying OCC to be dependency aware. However, since original algorithm targets software transaction memory, we make slightly optimized choices for our implementation of DATM. Each record keeps an accessor list to track the accessed transactions. However, our implementation only keeps the memory pointers of the received or written value. We can check if a read is stale by checking the memory pointer, which saves the cost from memory compare. Like [36], we use timeout for deadlock prevention.

As DATM is designed for software transactional memory, it doesn't support range query and deletion. Thus we only evaluate it with new-order transactions in TPC-C under high contention: all workers share the same warehouse. After profiling with different timeout thresholds, we found the best performance can be achieved when the timeout threshold is set to 0.45 ms on our 2.2GHz CPU. Figure 10d shows the performance. We also include OCC and 2PL as reference. Since DATM can manage dependency among uncommitted conflicting transactions, DATM scales to 8 threads with 125K TPS, which is better than all others. With 8 threads, IC3 gets 95K TPS while 2PL and OCC achieve 105K TPS and 1K TPS accordingly. However, the performance of DATM degrades with the increasing number of cores, mainly due to more cascading aborts and an increased cost per abort. With 16 threads, the abort rate is 81% and the abort rate increases to 99% (with 92% for observing an aborted transaction's update).



(a) Optimistic interleaving constraint & Defer the commutative operation. (b) Rendezvous piece

Figure 11: *Analysis the effect of each optimization method. Optimization is applied from bottom to top for each group.*

6.5 Factor Analysis

To understand the overhead and the benefit of each optimization, we show an analysis with TPC-C (Figure 11) when all threads share one warehouse (high contention). Figure 11a shows the analysis result. "Basic" is the performance of the basic algorithm(§ 3.2). Enforcing the interleaving optimistically ("+Optimistic Constraint") improves the performance by 32% with 64 threads (224K vs. 296K TPS). The performance is improved by 40% (296K vs. 435K) if we run the commutative operations without constraints.

Optimization with rendezvous piece. Because the TPC-C benchmark contains no rendezvous pieces, we modified the benchmark to include a new transaction called *last-order-status*. *Last-order-status* checks the status of the latest order made by some new-order transaction. This transaction contains three pieces: it first reads the *next_o_id* from a random district, then uses this order id to read records from the *ORDER* table, and lastly, it reads from the *ORDER-LINE* table. The corresponding SC-graph involving new-order and last-order-status is shown in Figure 12 (Appendix B). The second and third pieces of last-order-status are rendezvous pieces for the new-order transaction. We configure a benchmark with 50% new-order and 50% last-order-status. Figure 11b shows the evaluation result. Compared with the basic protocol, the performance is improved by 23% with rendezvous piece with 64 cores (289K vs. 235K TPS).

7. RELATED WORK

One purpose of concurrency control is constraining interleavings among transactions to preserve some serial order, using various approaches like two-phase locking [1, 26, 8, 14], timestamp ordering [10, 5, 29] and commit ordering [37, 38]. One main difference with prior approaches is that IC3 constrains interleavings at a much finer granularity, i.e., a transaction piece instead of a whole transaction. This exposes more concurrency for database transactions, while still preserving serializability.

IC3 is built upon prior work on statically analyzing transactions to assist runtime concurrency control. Bernstein et al. [7, 6] use a conflict graph to analyze conflicting relationships among transaction operations and preserve serializability by predefining orders of transactions. However, they handle cyclic dependency by quiescing all other transactions other than a transaction in the cycle, while IC3 dynamically constrains interleaving of transaction pieces to preserve serializability, which may lead to more concurrency. Others have proposed decomposing transactions into pieces [18, 4, 13, 41]. To preserve serializability after chopping, Garcia-Molina [17] shows that if all pieces of a decomposed transaction commute, then a safe interleaving exists. This, however, is a strong condition and many OLTP workloads do not suffice. Shasha et al. [41] show that serializability can be preserved if there is no SC-cycle in an SC-graph. Zhang et al. [53] further extends this theory to lower latency in distributed systems. In contrast, IC3 tracks dependency at runtime and constrain the interleaving with SC-graph. Cheung et al. [11] also use static analysis to optimize database applications. However they target the code quality of the database applications which are running on application servers.

There exist some approaches [36, 2] that try to improve performance by being aware of dependencies. Ordered sharing lock [2] allows transactions to hold locks concurrently. It ensures serializability by enforcing order protected operation and lock releasing must obey the lock acquired order . Ramadan et al. [36] developed a dependency aware software transactional memory. They avoid false aborts by tracking dependency. However, both permit unsafe interleaving which cause cascading aborts. IC3 only permits safe interleaving during the execution. Callas [52] is a piece of work done concurrently with ours that also constrains the interleaving of transaction pieces to ensure serializability. Although the basic algorithms of Callas and IC3 are similar, they are designed for different settings (distributed vs. multicore). Furthermore, the details of the algorithms are different, e.g. Callas does not support dynamic workloads and does not include the technique of § 4.1.

Deterministic database [47, 46, 45] leverages a sequencing layer to pre-assign a deterministic lock ordering to eliminate a commit protocol for distributed transactions. Faleiro et al. [16] proposes a deterministic database with lazy evaluation on multicore settings. However, they only allow concurrency after the system knows exactly the data accessed information [40]. Thus, the sequencing layer can become a performance bottleneck with large number of cores.

Recently, there has been work to re-schedule or reconcile conflicting transactions to preserve serializability. For example, Mu et al. [33] reduces conflicts by reordering conflicting pieces of contended transactions to achieve serializability. However, they require the entire dependency graph is generated to before reordering. This approach is not practicable on multicore settings. Narula et al. [34] utilize commutativity of special transaction pieces to mitigate centralized contention during data updating. Compared to this work, IC3 is more general and can be applied to general transactions.

Researchers have also applied other techniques to reduce or even eliminate concurrency control [24, 12, 3]. H-store and its relatives [42, 24, 20, 49] treat each partition as a standalone database and thus local transactions can run to completion without any concurrency control. Cross-partition transactions can then be executed using a global lock. Hence, performance highly depends on whether the partition of database fit the workload and the performance would degrade noticeably when cross-partition transactions increase [48]. Granola [12] requires no locking overhead for a special type of distributed transactions called independent transactions.

IC3 continues this line of research by optimizing transaction processing in modern multicore and in-memory databases [35, 25, 27, 28, 15, 48, 30, 54]. Recently, some databases have started to improve multicore scalability by eliminating centralized locks and latches [22, 23, 39, 21] for databases implemented using 2PL. However, the inherent limitation of 2PL such as read locking constrains its performance under in-memory settings. Several recent systems instead use OCC [1, 26] to provide speedy OLTP transactions, using fine-grained locking [48] or hardware transaction memory to protect the commit phase [29, 51]. As shown in this paper, IC3 notably outperforms both OCC and OLTP under contention.

Subasu et al. [43] describe a hybrid design by using several replicated database engines, each running on a subset of cores, where a primary handles normal requests and other synchronized replicas handle read-only requests. In this case, IC3 could be used to accelerate the primary copy. Dora [35] uses a thread-to-data assignment policy to run each piece accessing a partition to reduce contention on the centralized lock manager. Though it also decomposes the execution of a transaction into pieces, each piece still uses locks to ensure an execution as a whole, while each piece in IC3 executes concurrently with others rather than respecting the synchronization constraints.

8. CONCLUSION

Multi-core in-memory databases demand its concurrency control mechanism to extract maximum parallelism to utilize abundant CPU cores. This paper described IC3, a new concurrency control scheme that constrains interleavings of transaction pieces to preserve serializability while allowing parallel execution under contention. The key idea of IC3 is to combine static analysis with runtime techniques to track and enforce dependencies among concurrent transactions. To demonstrate its effectiveness, we have implemented IC3 and evaluations on a 64-core machine using TPC-C showed that IC3 has better and more scalable performance than OCC, 2PL, and other recently proposed concurrent control mechanisms under moderate or high levels of contention.

Acknowledgements

The authors would like to thank Dennis Shasha, Lidong Zhou and all the reviewers for their insightful comments and feedback on this work. This work is supported in part by National Science Foundation under award CNS-1218117.

References

[1] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1):45–59, 1987.

[2] D. Agrawal, A. El Abbadi, R. Jeffers, and L. Lin. Ordered shared locks for real-time databases. *The VLDB Journal*, 4(1):87–126, 1995.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3), 2014.

[4] A. J. Bernstein, D. S. Gerstl, and P. M. Lewis. Concurrency control for step-decomposed transactions. *Information Systems*, 24(8):673–698, 1999.

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

[6] 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.

[7] 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.

[8] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):4–24, 1990.

[9] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Linux Symposium*, 2012.

[10] M. J. Carey. Modeling and evaluation of database concurrency control algorithms. 1983.

[11] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers. Using program analysis to improve database applications. *IEEE Data Eng. Bull.*, 37(1):48–59, 2014.

[12] 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.

[13] C. T. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.

[14] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):44–62, 1990.

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

[16] 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.

[17] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems (TODS)*, 8(2):186–213, 1983.

[18] H. Garcia-Molina and K. Salem. *Sagas*, volume 16. ACM, 1987.

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

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

[32] 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.

[33] 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.

[34] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 511–524. USENIX Association, 2014.

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

[36] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an stm. In *ACM Sigplan Notices*, volume 44, pages 163–172. ACM, 2009.

[37] Y. Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *VLDB*, volume 92, pages 292–312, 1992.

[38] Y. Raz. Serializability by commitment ordering. *Information processing letters*, 51(5):257–264, 1994.

[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] 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.
[42] 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.
[43] T. Subasu and J. Alonso. Database engines on multicores, why parallelize when you can distribute. In *Proc. Eurosys*, 2011.
[44] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, 2007.
[45] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.
[46] A. Thomson and D. J. Abadi. Modularity and Scalability in Calvin. *IEEE Data Engineering Bulletin*, page 48, 2013.
[47] 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.
[48] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proc. SOSP*, 2013.
[49] L. VoltDB. Voltdb technical overview, 2010.
[50] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. Technical Report TR2016-981, New York University, 2016. http://ic3.news.cs.nyu.edu/techreport16.pdf.
[51] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proc. EuroSys*, 2014.
[52] 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.
[53] 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.
[54] 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.

Appendix A: Proof

Definition: Let each S-edge in an SC-graph become directed following the chronological order of pieces. Also, repeatedly merge vertexes (pieces) connected by C-edges to a single vertex. Afterwards, a cycle in the graph is defined as an *x-cycle*. Similarly, a path is defined as an *x-path*.

FACT 1. *The offline static analysis ensures there is no x-cycle.*

We use the concept of serialization graph as a tool to prove serializability. A schedule is serializable iff the serialization graph is acyclic. Because we use OCC to protect the execution of each piece, if we consider each piece as a (sub-)transaction, we have the following property.

FACT 2. *The serialization graph of pieces is acyclic.*

If two pieces p_i and p_j from two different transactions are connected as $p_i \rightarrow p_j$ in the serialization graph, they have a chronological commit order as $t_c(p_i) < t_c(p_j)$, and p_i, p_j should be also connected by a C-edge in the SC-graph. We can denote the chronological commit order as $p_i \xrightarrow{c} p_j$.

If two pieces q_i and p_j are connected by an S-edge in the SC-graph, and q_i is ahead of p_i in chronological order, we denote this as $q_i \xrightarrow{s} p_j$.

FACT 3. *IC3 tracks per-record longest path in the serialization graph.*

If $T_i \rightarrow T_j$ (expanded as $p_i \rightarrow p_j$) is a longest path in the serialization graph, after p_j is committed, T_i will appear in T_j's depqueue. This also suggests that T_j will only enter its commit phase after T_i commits.

THEOREM 1. *The schedule of IC3 is serializable as it always generates acyclic serialization graph.*

Now we are going to prove IC3 is serializable by proving the serialization graph is acyclic. Assume there is a cycle in the serialization graph, let it be $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$; We are going to prove that the cycle necessarily implies there is an x-cycle in the SC-graph, which leads to a contradiction to Fact 1.

Expand each transaction in the cycle $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$ to pieces, as

$$q_1 \overset{s}{+} p_1 \xrightarrow{c} q_2 \overset{s}{+} p_2 \xrightarrow{c} ... \xrightarrow{c} q_n \overset{s}{+} p_n \xrightarrow{c} q_1$$

The symbol "$\overset{s}{+}$" above (between a pair of $q_i \overset{s}{+} p_i$) represents following three possible cases:

1. q_i and p_i are the same piece, i.e. $q_i = p_i$.
2. q_i and p_i are different pieces, they are connected by an S-edge, and q_i is chronologically ahead of p_i, denoted by $q_i \xrightarrow{s} p_i$.
3. q_i and p_i are different pieces, they are connected by an S-edge, and q_i is chronologically behind p_i, denoted by $q_i \xleftarrow{s} p_i$.

To simplify without loss of accuracy, we use the symbol $\overset{s}{\nrightarrow}$ to represent the combination of the first and second cases; $\overset{s}{\nleftarrow}$ to represent the first and third cases.

LEMMA 1. *No transaction in cycle can commit.*

According to Fact 3, the cycle $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$ necessarily implies an chronological cycle in the commit order, i.e. $t_c(T_1) < t_c(T_2) < ...t_c(T_n) < t_c(T_1)$, which is not possible. According to IC3's protocol, a cycle in the serialization graph will necessarily cause a deadlock in transaction commit phase, which means no transaction is able to commit. Actually, such deadlock can never form in the first place. The following part explains that.

LEMMA 2. $\exists\ i : q_i \neq p_i$

Assume for every pair of q_i and p_i, they are the same piece. Then the cycle leads a contradiction: the cycle should not exist according to Fact 2.

LEMMA 3. $\exists\ i, j : q_i \xrightarrow{s} p_i$ *and* $p_j \xrightarrow{s} q_j$

Proof by contradiction. Without loss of generality, assume $\forall i : q_i \xleftarrow{s} p_i$. Then the cycle will necessary imply an x-cycle in static analysis, which contradicts with Fact 1.

LEMMA 4. *For a fragment* $p_i \xrightarrow{c} q_j \xrightarrow{s} p_j \xrightarrow{c} q_k$, *there will be a piece* r_i *in* T_i, *such that* $p_i \xrightarrow{c} r_i \xrightarrow{c} q_k$.

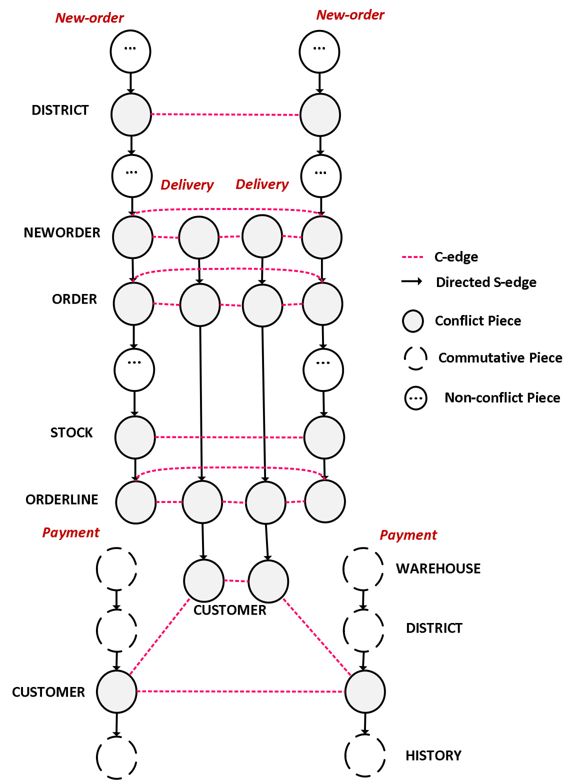


Figure 13: *The SC-graph of 3 Read Write Transactions in TPC-C*

$q_j \overset{s}{\rightarrow} p_j$ refers to that before p_j executes, q_j already commits, leaving T_i in T_j's depqueue. Therefore, when p_j commits, it shall either wait T_i commits, or it waits until all pieces in T_j that has C-connection to itself commits. According to Lemma 1 it has to be the latter case. Assume the last piece in T_i that has C-connection to p_j is r_i. r_i cannot be neither be p_i or any piece before p_i. Otherwise $r_i \overset{s}{\nrightarrow} p_i \overset{c}{\rightarrow} q_j \overset{s}{\rightarrow} p_j \overset{c}{\leftarrow} r_i$ will form a x-cycle, which contradicts with Fact 1.

With Lemma 4, we can inductively prove the cycle in serialization graph will necessarily implies an x-cycle in static analysis.

In the cycle:

$$q_1 \overset{s}{\nrightarrow} p_1 \overset{c}{\rightarrow} q_2 \overset{s}{\nrightarrow} p_2 \overset{c}{\rightarrow} \ldots \overset{c}{\rightarrow} q_n \overset{s}{\nrightarrow} p_n \overset{c}{\rightarrow} q_1$$

If there exists j that $q_j \overset{s}{\rightarrow} p_j$, without loss of generosity, we can safely assume $j = n$. We can shrink the cycle following Lemma 4, then the cycle becomes as follows.

$$q_1 \overset{s}{\nrightarrow} p_1 \overset{c}{\rightarrow} q_2 \overset{s}{\nrightarrow} p_2 \overset{c}{\rightarrow} \ldots \overset{c}{\rightarrow} q_{n-1} \overset{s}{\nrightarrow} p_{n-1} \overset{s}{\rightarrow} r_{n-1} \overset{c}{\rightarrow} q_1$$

Transform the above cycle into:

$$q_1^1 \overset{s}{\nrightarrow} p_1^1 \overset{c}{\rightarrow} q_2^1 \overset{s}{\nrightarrow} p_2^1 \overset{c}{\rightarrow} \ldots \overset{c}{\rightarrow} q_{n-1}^1 \overset{s}{\nrightarrow} p_{n-1}^1 \overset{c}{\rightarrow} q_1^1$$

Repeatedly reduce the cycle following Lemma 4, for m times ($m < n$), until we have:

$$q_1^m \overset{s}{\nleftarrow} p_1^m \overset{c}{\rightarrow} q_2^m \overset{s}{\nleftarrow} p_2^m \overset{c}{\rightarrow} \ldots \overset{c}{\rightarrow} q_{n-m}^m \overset{s}{\nleftarrow} p_{n-1}^m \overset{c}{\rightarrow} q_1^m$$

Consider each pair of $q_i^m \overset{s}{\nleftarrow} p_i^m$ in the above cycle, there exists at least one pair of q_i^m and p_i^m such that $q_i^m \neq p_i^m$. Otherwise we will have $q_1^m \overset{c}{\rightarrow} q_2^m \overset{c}{\rightarrow} \ldots \overset{c}{\rightarrow} q_{n-m}^m \overset{c}{\rightarrow} q_1^m$, which is not possible. Then means the above cycle necessarily implies an x-cycle in the SC-graph, which is a contradiction to the result of static analysis.

Q.E.D.

Appendix B: SC-graph of TPC-C benchmark

Figure 13 shows the SC-graph for the 3 read-write transactions in the TPC-C benchmark. Each piece includes at least one operation. For the delivery transaction, the for loop is split into four small loops and each one only accesses one table.

Figure 12 shows the SC-graph of a modified TPC-C benchmark including the new-order transaction with a new transaction called last-order-status.

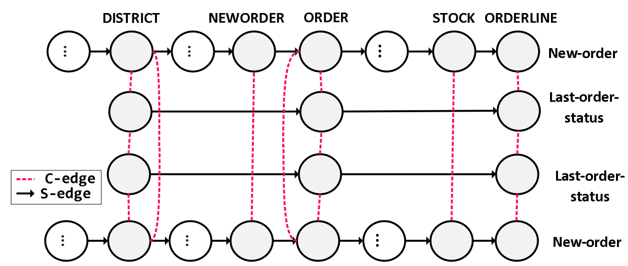


Figure 12: SC-graph with new-order and last-order-status

Appendix C: Web Applications Analysis

To determine if real-world applications can potentially benefit from IC3, we analyze the user-defined transactions in 14 most popular ruby-on-rails web applications on GitHub according to their GitHub stars. As most applications use the active record interface instead of SQL, we analyze the user defined transactions manually. Table 1 shows the results. 11 out of 14 applications have transactions that access more than one table. We construct the SC-graphs of these 11 applications and classify the SC-cycles found. As shown in Table 1, the SC-cycles found in most of these applications are *not* deadlock-prone. Thus, these workloads are likely to benefit from IC3. Even for workloads that contain deadlock-prone SC-cycles, there is much opportunity for parallel execution. For example, the most complex workload Canvas LMS has 46 user defined transactions, 30 of which access more than one table. Among these 46 transactions, 12 transactions are involved in a deadlock-prone SC-cycle. After merging the deadlock prone SC-cycles into atomic pieces, there are still 20 transactions which have more than one conflicting piece. All of them can benefit from IC3.

Name	Description	Total	Mul-Tables	D-SC	IC3-SC	GitHub Stars
Canvas LMS	Education	46	30	12	20	1,542
Discourse	Community discussion	36	28	3	26	16,019
Spree	eCommerce	7	3	-	3	6,802
Diaspora	Social network	2	1	-	1	10,331
Redmine	Pro ject management	17	8	-	8	1,560
OneBody	Church portal	2	1	-	1	1,229
Community Engine	Social networking	2	-	-	-	1,117
Publify	Blogging	5	1	-	1	1,497
Browser CMS	Content management	4	3	-	3	1,200
GitLab	Code management	8	5	-	5	16,523
Insoshi	Social network	3	-	-	-	1,585
Teambox	Project management	4	1	-	1	1,873
Radiant	Content management	2	-	-	-	1,609
GitLab CI	Continuous integration	2	-	-	-	1,420

Table 1: Ruby-on-rails applications used in analysis (**Total**: the total number of user defined transactions in the workload. **Mul-Tables**: the number of transactions which access more than one table. **D-SC**: The number of transactions which are involved in a deadlock prone SC-Cycle. **IC3-SC**: The number of transactions which have more than one conflicting piece and are involved in a COMPATIBLE SC-CYCLE.). GitHub starts record the number of stars on GitHub as of October 2015.