# High Performance Transactions via Early Write Visibility

Jose M. Faleiro
Yale University
jose.faleiro@yale.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

## ABSTRACT

In order to guarantee recoverable transaction execution, database systems permit a transaction's writes to be observable only at the end of its execution. As a consequence, there is generally a delay between the time a transaction performs a write and the time later transactions are permitted to read it. This *delayed write visibility* can significantly impact the performance of serializable database systems by reducing concurrency among conflicting transactions.

This paper makes the observation that delayed write visibility stems from the fact that database systems can arbitrarily abort transactions at any point during their execution. Accordingly, we make the case for database systems which only abort transactions under a restricted set of conditions, thereby enabling a new recoverability mechanism, *early write visibility*, which safely makes transactions' writes visible prior to the end of their execution. We design a new serializable concurrency control protocol, piece-wise visibility (PWV), with the explicit goal of enabling early write visibility. We evaluate PWV against state-of-the-art serializable protocols and a highly optimized implementation of read committed, and find that PWV can outperform serializable protocols by an order of magnitude and read committed by 3X on high contention workloads.

## 1. INTRODUCTION

Over the past decade, concurrency control research has seen a renaissance due to the abundance of parallelism in multi-core servers and datacenters. Modern serializable protocols are explicitly designed to exploit this abundant parallelism [17,19,28–30,32,37,38, 44–46]. While these new protocols propose novel isolation mechanisms that address the incompatibility between conventional concurrency control protocols and massively parallel environments, they use ideas for *recoverability* [12] that are decades old. Indeed, the last widely-adopted research on recoverability, group commit [22], was proposed in the 1980s. These conventional recoverability mechanisms limit concurrency control protocols' ability to extract concurrency from a workload.

Recoverability is the property that all of a committed transaction's writes are made durable, and that none of an aborted transaction's writes are made durable or observed by committed transactions [12]. In order to guarantee recoverability, most concurrency control protocols only permit a transaction's writes to be read when it commits or at least finishes executing [16, 22]. These protocols effectively delay making a transaction's writes visible. This *write visibility delay* can adversely impact strong isolation levels such as serializability. This is because serializable isolation requires that transactions always read the latest value of any record; any delay in satisfying a read will delay the corresponding reading transaction.

Recoverability mechanisms employ delayed write visibility because database systems can arbitrarily abort a transaction prior to the point that its commit record is made durable; a database system may abort a transaction due to deadlock handling logic, failures, optimistic validation errors, or simply because the transaction consumes resources that are in short supply. Database systems' ability to arbitrarily abort transactions forces recoverability mechanisms to make extremely pessimistic assumptions about when a transaction's writes are safe from being rolled back.

This paper makes the case for curtailing database systems' ability to arbitrarily abort transactions. We show that if a database system only aborts transactions under a restricted set of conditions, then it can avoid pessimistic recoverability mechanisms based on delayed write visibility. In particular, if only a subset of a transaction's statements can cause it to abort, then the transaction is guaranteed to commit as soon as every such abortable statement has finished executing, even while one or more "non-abortable" statements remain to be executed. This enables a new write visibility discipline, *early write visibility*, which can safely make transactions' writes visible *prior* to the end of their execution, and, as a consequence, can reduce the duration for which concurrent transactions are disallowed from making progress due to conflicts.

This paper proposes a new deterministic concurrency control protocol, piece-wise visibility (PWV), explicitly designed to enable early write visibility. PWV employs deterministic execution to avoid arbitrarily aborting transactions. To enable early write visibility, PWV decomposes transactions into a set of sub-transactions or *pieces*, such that each piece consists of one or more transaction statements. PWV then schedules pieces such that their corresponding transactions execute in a serializable order. PWV makes a piece's writes visible as soon as its transaction commits, even if one or more pieces of the same transaction have not yet executed.

PWV decomposes transactions by performing a data-flow analysis on their control-flow graphs [8]. PWV's decomposition procedure has three important properties. First, it is modular; a transaction is decomposed based only on the data dependencies between its constituent statements. Second, it places no restrictions on the number of pieces that can potentially abort, while simultaneously preventing cascaded aborts. Third, it allows PWV to exploit intra-transaction parallelism by executing multiple pieces belonging to the same transaction in parallel. These three properties address lim-


*Proceedings of the VLDB Endowment,* Vol. 10, No. 5

itations that, to the best of our knowledge, are present in every prior transaction decomposition proposal [36, 43, 48, 52, 54, 55].

Our experimental evaluation shows that PWV's ability to produce aggressive serializable schedules results in significant performance gains. Under high contention workloads, PWV can outperform state-of-the-art serializable protocols, including transaction chopping, by over an order of magnitude. Furthermore, we show that PWV can even outperform a highly optimized implementation of read committed isolation by more than 3X, while still providing the stronger guarantee of serializable isolation.

In summary, this paper makes the following contributions:

- We identify write visibility delay as a significant impediment to the performance of strong isolation levels due to the *specifications* for strong isolation. This impediment is fundamental and cannot be avoided by designing better concurrency control protocols (Section 2).
- We propose a new write visibility discipline, early write visibility, that addresses the limitations of prior write visibility disciplines. We show that early write visibility can enable concurrency control protocols that allow a transaction's writes to be made visible prior to the end of its execution while still guaranteeing serializability and preventing cascaded aborts (Section 3).
- We design PWV, a concurrency control protocol that exploits early write visibility to obtain greater concurrency than conventional serializable protocols. We prove that if transactions' read- and write-sets are known a priori, it is impossible for any serializable concurrency control protocol which avoids cascaded aborts to extract more concurrency from a workload than an ideal implementation of PWV. We also discuss practical issues related to application corner cases (Section 4).
- We evaluate a multi-core optimized implementation of PWV against state-of-the-art pessimistic locking, optimistic concurrency control, transaction chopping, and a weak isolation read committed implementation. (Section 5).

# 2. BACKGROUND AND MOTIVATION

In order to guarantee serializable and recoverable execution of transactions, every widely deployed concurrency control protocol disallows a transaction's writes from being read until at least the end of the transaction's execution. This write visibility delay is intrinsic to concurrency control protocols such as two-phase locking and optimistic concurrency control due to their use of locks and private writes prior to validation, respectively. Furthermore, the requirement that database systems guarantee recoverability fundamentally limits them from making writes visible early, *regardless of concurrency control protocol*.

## 2.1 Isolation

Variants of two-phase locking [18] and optimistic concurrency control [31] are among the most widely deployed serializable isolation protocols in modern database systems. In order to correctly isolate conflicting transactions, both strict two-phase locking (2PL) and optimistic concurrency control (OCC) restrict interleavings among conflicting transactions. In particular, if a transaction $T_2$ reads $T_1$'s write to $x$, then practical implementations of both 2PL and OCC produce schedules in which $T_2$'s read *always* follows $T_1$'s completion. Under 2PL, transactions hold *long-duration* locks on records; any locks acquired by a transaction are only released at the end of its execution [11, 23]. This locking discipline constrains the execution of conflicting reads and writes; if transaction $T_2$ reads $T_1$'s write to record $x$, and $T_1$ holds a write lock on $x$ until the time it completes, $T_2$'s read can only be processed *after* $T_1$ completes.

OCC similarly constrains conflicting transactions. Transactions perform writes in a local buffer, and only copy these writes to the active database after validation [31]. Thus, a transaction's writes are only made visible at the very end of the transaction. Modern variants of OCC actually produce schedules of committed transactions that are provably equivalent to those produced by 2PL [46].

Both 2PL and OCC produce schedules in which there exists a delay between the time that a transaction writes a record, and the time that later transactions can read this write. This delay can significantly limit opportunities for concurrency under high contention.

## 2.2 Recoverability

Every transaction processed by a database system must either commit or abort. If a transaction commits, then all of its writes must be made persistent. In contrast, if a transaction aborts, its writes cannot be made persistent. Furthermore, most widely-used isolation levels – including Read Committed, Snapshot Isolation, and Serializability – require that an aborted transaction's writes must never be observed by a committed transaction [11]. If this is not the case, the committed transaction exhibits an aborted read anomaly [6].

In order to prevent aborted reads, concurrency control protocols must constrain the execution of transactions whose reads and writes conflict. Consider a transaction $T_1$ that writes record $x$. If another transaction, $T_2$, reads $T_1$'s write to $x$, then $T_2$ must not be allowed to commit before $T_1$ commits. This discipline prevents $T_1$ from aborting after $T_2$ (which read $T_1$'s data) has already committed. Schedules that satisfy this property are called recoverable [12]. Recoverable scheduling mechanisms must therefore control when a transaction's writes are made visible to other transactions. There exist two general write visibility disciplines:

- **Committed write visibility.** The database delays making a transaction's writes visible until the transaction is guaranteed to commit. Strict two-phase locking is one such strategy [12]. A transaction holds exclusive locks on a record it writes from the time it updates the record to the time the transaction commits. Holding exclusive locks until commit time prevents concurrent transactions from reading uncommitted writes.

- **Speculative write visibility.** Alternatively, the database system can allow transactions to read uncommitted writes (dirty reads), and enforce a commit discipline on transactions that perform dirty reads [7, 12, 25, 27, 41]. In practice, if transaction $T$ writes a record and later aborts, then any transaction that read $T$'s write also aborts.

Each of these write visibility disciplines has advantages over the other. Speculative write visibility is susceptible to cascaded aborts [12]. If transaction $T$ makes uncommitted writes visible to other transactions and later aborts, then any transaction $T'$ that read $T$'s uncommitted writes must abort. In turn, any transaction that read uncommitted writes by $T'$ must also abort, and so forth. In general, if transaction $T$ aborts, then the transitive closure of transactions linked via dirty reads dependencies to $T$ must also abort. Cascaded aborts can severely hurt performance because the database wastes cycles executing transactions that are later aborted.

Committed write visibility avoids cascaded aborts by disallowing dirty reads; transaction $T$ is never allowed to make uncommitted writes visible to other transactions. On the other hand, committed write visibility can inhibit performance by forcing transactions to wait for prior transactions' commit records to be made durable. This delay can lead to unacceptable performance when transactions' runtimes are much shorter than the time it takes to make

commit records durable, for instance, in main-memory database systems which maintain durable state on disk.

As a consequence of these tradeoffs, modern database systems employ *hybrid* disciplines that combine committed and speculative write visibility [26, 32, 38, 46]. The best known example of such a hybrid write visibility discipline is group commit [16, 22]. As originally proposed by Gawlick and Kinkade, transactions follow a locking protocol in which they hold locks for the duration of their execution, and release locks after their execution completes but before their commit records are made durable. Prior to releasing their locks, transactions write their commit records to an in-memory sequential log. The in-memory log is asynchronously flushed to disk in batches. Since a transaction $T$ holds its locks until its commit record is written to the in-memory log; if transaction $T'$ reads $T$'s writes, then its commit record will be logged after $T$'s commit record. This logging discipline guarantees recoverability; if transaction $T'$ commits, then all transactions whose commit records were logged prior to $T'$'s also commit, including those whose transactions whose writes were read by $T'$.

Most modern concurrency control protocols either use a form of committed write visibility or a variant of group commit based on delayed write visibility. In Section 2.3, we show that even the short delays in making writes visible in recoverability mechanisms such as group commit – compared to, for instance, delays with disk I/O on the critical path – can adversely impact serializable concurrency control protocols under contended workloads.

## 2.3 Interaction of write visibility and isolation

Database systems allow users to assign transactions an *isolation level*, which abstractly specifies the permissible set of interleavings among conflicting transactions [6]. Isolation levels allow each individual transaction to tradeoff consistency for performance. Strong isolation levels, such as serializability, permit fewer interleavings among conflicting transactions, which provides strong consistency at the expense of concurrency. In contrast, weak isolation levels, such as read committed, permit more interleavings among conflicting transactions, allowing transactions to observe inconsistent database states in order to improve performance.

One important restriction on interleavings is that serializability requires that transactions always observe the *latest* committed value of any record that they read. In contrast, read committed allows transactions to read *any* previously committed values of a record (reads can be arbitrarily stale). As a consequence, serializable concurrency control protocols must carefully constrain the execution of transactions whose reads and writes conflict, while read committed protocols can decouple conflicting reads and writes. Consider a scenario where record $x$ is first written by transaction $T_0$, and next written by transaction $T_1$ ($T_0$ precedes $T_1$). If a later transaction $T_2$ reads $x$, then under serializability, $T_2$'s read *must* return the value written by $T_1$. In contrast, read committed allows $T_2$ to read either of $T_0$ or $T_1$'s writes.

As Section 2.2 discussed, recoverability mechanisms based on group commit only permit transactions' writes to be observed at the end of their execution. Serializability's requirement that transactions observe the latest committed values of records interacts poorly with the delayed write visibility discipline employed by these recoverability mechanisms. In the above example, if $T_1$'s write to $x$ is followed by additional writes to records $y$ and $z$, then for recoverability purposes, $T_1$'s write to $x$ can only be read by $T_2$ after $T_1$'s additional writes to $y$ and $z$ complete. In contrast, under read committed, the database can allow $T_2$ to read $T_0$'s write to $x$ even as $T_1$'s writes are in progress. In order to guarantee recoverability, read committed must also delay making $T_1$'s write to $x$ visible until the end of its execution. However, this delay has no effect on $T_2$ because $T_2$ is permitted to read *earlier* transactions' writes to $x$.

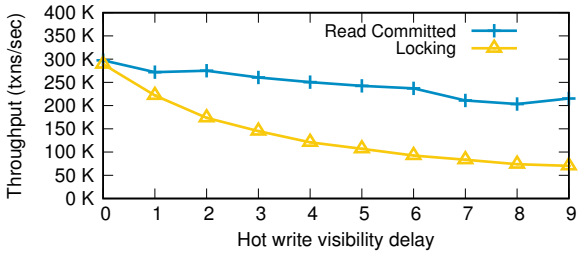


**Figure 1:** Effect of write visibility delay of hot record updates on transaction throughput using 40 threads.

Serializability's requirement that transactions observe the latest committed values of records is part of its *specification*. Therefore, every protocol that correctly implements this specification, that is, *every* serializable concurrency control protocol, is subject to the reduction in concurrency due to delayed write visibility. The fact that delayed write visibility limits concurrency cannot be circumvented by designing better protocols or more efficient implementations.

In order to substantiate this argument, we conducted an experiment to measure the interaction between write visibility delay and isolation levels. The experiment runs a workload consisting of transactions which perform 10 read-modify-write operations. The database consists of 1,000,000 records. We designate one record in the database as "hot", and force every transaction to update this hot record. As a consequence, every pair of transactions conflicts. The 9 remaining records updated by a transaction are chosen uniformly at random from the remaining 999,999 records. We compare the performance of a multi-core optimized implementation of serializable locking and read committed (Section 5 provides a detailed discussion of these algorithms and experimental setup).

In order to measure the impact of write visibility delay on each system, we vary the point at which each transaction updates the hot record. The earlier a transaction updates the hot record, the higher the write visibility delay. We measure write visibility delay as the number of updates that transactions must perform *after* updating the hot record. Figure 1 shows the results of the experiment. We plot the throughput of read committed and serializable locking as a function of increasing write visibility delay.

Figure 1 shows that locking's throughput decreases dramatically as visibility delay increases. The locking algorithm acquires an exclusive lock on a record prior to updating the record, and holds all acquired locks until the end of its execution. When the hot record update is performed at the end of each transaction (the left-most point on the x-axis), the lock on the hot record is only held while the transaction updates the hot record. In contrast, when the hot record update is performed at the beginning of each transaction (the right-most point on the x-axis), the lock on the hot record is acquired at the beginning of each transaction, and is held while the transaction updates every record in its write-set. Locking's throughput drops by nearly a factor 6 at maximum write visibility delay. In contrast, read committed's throughput drops by a more modest 30%. Increasing write visibility delay does not have as adverse an impact on read committed as serializability because read committed allows transactions to read stale values of records. The modest drop in throughput occurs because read committed acquires write locks on records at commit time in order to consistently order transactions' writes [6] (see Section 5). These commit time write locks are acquired in the same order as serializable locking, but held are held for a much shorter duration.

# 3. EARLY WRITE VISIBILITY

We now describe a new recoverability mechanism, early write visibility, that addresses the limitations of delayed write visibility. Early write visibility constrains a database system's ability to arbitrarily abort transactions. Early write visibility can be tailored to any database system which sufficiently constrains transaction aborts to a specific set of circumstances, such as explicit abort statements and constraint violations. Examples of such systems include deadlock-free locking systems [42] and deterministic database systems [19, 44, 45]. In this paper, we focus on deterministic database systems (although the ideas can be generalized to other systems).

## 3.1 Deterministic execution background

Transaction aborts can broadly be classified into logic- and system-induced aborts. Logic-induced aborts occur in order to prevent a transaction from writing state which violates application invariants. For example, a transaction may include an explicit abort statement which is conditionally triggered after reading a database record, or the transaction may be aborted if its updates cause a constraint violation. System-induced aborts are triggered by the database system, and occur independently of transactions' logic. Examples of system-induced aborts include aborts due to deadlock handling logic, failures, and validation errors in optimistic protocols.

Deterministic systems employ scheduling techniques that eliminate the vast majority of system-induced aborts in conventional systems. A deterministic system processes a transaction in the following three steps. First, any calls to non-deterministic functions, such as a random number generator or system clock, are evaluated in order to be used at execution time. Second, the transaction's logic, its input parameters, and all non-deterministic input are logged. Note that *all* transactions are logged, regardless of whether they eventually commit or abort. Third, the transaction is processed after its existence has been successfully (stably) logged. We next describe how deterministic systems execute transactions during normal case and recovery processing.

**Normal case processing.** Deterministic systems process transactions in an order that is equivalent to the order in which they are logged (as described above). Serializability is guaranteed by the fact that the log is totally ordered. A class of deterministic systems, exemplified by Calvin and Bohm [19, 45], use knowledge of transaction conflicts to relax the total order into an equivalent partial order. If transactions $T_1$ and $T_2$ conflict, such that $T_1$ precedes $T_2$ in the log, then $T_1$ will always be executed before $T_2$. The execution of non-conflicting transactions is not constrained.

These systems determine transactions' conflicts using a priori knowledge of transactions' read- and write-sets. The read- and write-sets are determined either via a static analysis of each transaction's logic, or via speculative execution of a subset of each transaction's logic (see below). These systems also use a priori knowledge of read- and write-sets to implement a deadlock avoidance strategy. For example, Calvin isolates transactions using a modified version of logical locking [45]. The scheduler acquires transactions' locks by sequentially scanning the input log. For every transaction in the log, the scheduler requests locks on *every* record in the transaction's read- and write-sets prior to its execution. A transaction is only permitted to execute when all of its locks have been acquired. This lock acquisition protocol avoids deadlocks because the set of locks required by transactions are known a priori and can be acquired in lexicographic order.

In certain applications, transactions' read- and write-sets are deducible from their input parameters, such as when all records involved in a transaction are accessed by their primary keys. In other applications, read- and write-sets depend on database state (such as secondary indexes). In the latter case, deterministic systems determine transactions' read- and write-sets using speculative execution. Speculative execution occurs as part of non-deterministic input processing prior to logging transactions (as described in the first step of transaction processing above). The obtained read- and write-sets are then logged along with transactions' other input parameters. At execution time, deterministic systems check that the speculatively obtained read- and write-sets are correct. This is done by adding a logical condition as early as possible in the transaction code to validate the speculatively generated read- and write-sets. If this condition fails, a deterministic logical abort results.

**Recovery processing.** Deterministic systems execute transactions only when they are guaranteed to be stable on the log [33]. Therefore, each in-progress transaction during a failure is guaranteed to be in the log. Furthermore, *all* transactions are logged, regardless of whether they eventually commit or abort. At recovery time, a deterministic system can play the log forward from the time of the last checkpoint. As mentioned previously, the only information contained in a log record is the transaction's logic, its input parameters, and any non-deterministic input. The log is played back in a serial-equivalent fashion using *the same* mechanism used during normal case processing; there is no difference between recovery and normal case processing. Since the non-deterministic inputs to a transaction are also logged, each transaction is guaranteed to deterministically write the same record values and arrive at the same commit decision during recovery.

## 3.2 A new write visibility discipline

We make the observation that the reduced scope of aborts in deterministic systems can be exploited to obtain a far more aggressive write visibility discipline than those used by conventional systems. Since deterministic systems only abort transactions due to logic- and speculation-induced aborts, a transaction is *guaranteed* to commit once all the operations that can cause logic- and speculation-induced aborts have finished executing. Importantly, this "commit point" can occur before the transaction has finished executing in its entirety. In other words, a transaction can have several operations pending after its commit point.

Early write visibility prescribes two write visibility rules; each applicable to writes that precede or follow a transaction's commit point:

- **Writes preceding the commit point.** Such writes can only be made visible once every other operation that precedes the transaction's commit point has finished executing. Delaying the visibility of these writes until a transaction's commit point ensures that they are never read by another transaction only to be later rolled back.
- **Writes following the commit point.** Such writes can be made visible immediately after their completion. A write that follows a transaction's commit point is guaranteed to never rollback because a transaction can never abort beyond its commit point.

The two rules above ensure that a transaction's writes are only made visible if it commits. Early write visibility therefore guarantees that a transaction never reads dirty data, which eliminates the possibility of a transaction reading a write that is later rolled back.

# 4. PIECE-WISE VISIBILITY

Although early write visibility makes a transaction's writes visible as soon as it is guaranteed to commit, conventional concurrency control protocols such as 2PL and OCC cannot simply re-

place their recoverability mechanisms with early write visibility. These protocols cannot use early write visibility as a "black box" for two reasons. First, delayed write visibility is intrinsic to both 2PL and OCC due to their respective use of locks and validation (Section 2.1). Second, existing concurrency control protocols use arbitrary transaction aborts pervasively; dynamic locking aborts transactions due to deadlocks, and OCC aborts transactions on validation failures. These arbitrary aborts preclude early write visibility, which requires that transactions are only aborted under a limited set of conditions (Section 3). While existing deterministic concurrency control protocols do not arbitrarily abort transactions (Section 3.1), they cannot exploit early write visibility because they schedule each transaction's logic as a single atomic unit [19,44,45].

This section presents piece-wise visibility, or PWV, a new deterministic serializable concurrency control protocol that schedules work at the granularity of subsets of transactions' individual reads and writes. This fine-grained scheduling allows PWV to fully exploit early write visibility. PWV *decomposes* the totally ordered set of statements that constitute a straight-line transaction into a partially ordered set of statements based on the transaction's data-flow and commit point. PWV then schedules each decomposed transaction's constituent statements using a deterministic scheduler. Intuitively, PWV can produce schedules that are similar to those produced by a locking-based concurrency control protocol that is *not two-phase*; that is, a protocol which releases locks on records, and then goes on to acquire more locks on different records later on, but nonetheless guarantees serializability.

## 4.1 Transaction decomposition

The input to PWV's scheduler is a totally ordered set of decomposed transactions. A decomposed transaction is a partially ordered set of the transaction's constituent statements. This partially ordered set can be represented by a directed acyclic graph (DAG) whose nodes we refer to as *pieces*. The edges of the DAG define the order in which its pieces can execute.

There exist two situations under which an edge is created from piece $p_1$ to $p_2$. First, the input of $p_2$ depends on the output of $p_1$ (data dependencies). Second, $p_2$ contains an update statement, and follows its transaction's commit point, while $p_1$ precedes the commit point (commit dependencies). Intuitively, commit dependencies prevent pieces that follow a transaction's commit point from performing updates until the transaction is guaranteed to commit.

Figure 2 shows an example of transaction decomposition. The transaction shows logic which is invoked when a customer attempts to purchase a set of items from hypothetical online shopping portal. The transaction first tries to decrement the count of each requested item (lines 1-8). The transaction aborts if any of the item's counts is zero (lines 4-5). The transaction then updates some application-specific statistics, in this case, the total number of items sold (lines 9-10). Finally, the transaction updates the outstanding amount due from the customer (lines 11-12).

The transaction's decomposition is shown below the straight-line code (in Figure 2). Edges corresponding to data dependencies are represented by solid arrows. Edges corresponding to commit dependencies are represented by dashed arrows. The transaction can safely commit after the count of every item requested by the customer is successfully decremented. Each item count decrement is represented by a piece $P_i$. Piece $S$ updates the total number of items sold, and only depends on the number of items the customer purchases. Importantly, the write in piece $S$ does not depend on the output of any other piece. However, because it follows the transaction's commit point, $S$ has a commit dependency on each $P_i$. In contrast, piece $C$, which updates the customer's outstanding bill,

```
1  price = 0
2  for p_id in p_id_list:
3      prod = DB.write_ref(p_id, "products")
4      if prod.count == 0:
5          ABORT()
6      else:
7          prod.count -= 1
8      price += prod.price

9  stats = DB.write_ref("statistics")
10 stats.num_purchases += p_id_list.size()

11 cust = DB.write_ref(c_id, "customer")
12 cust.bill += price
```

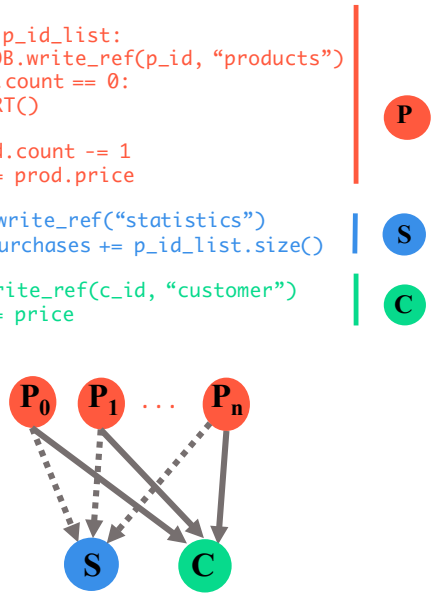


**Figure 2:** A straight-line transaction decomposed into a partially ordered set of pieces. Solid edges represent data dependencies. Dashed edges represent commit dependencies.

depends on the output of every piece $P_i$ (note that $C$ also has a commit dependency on each $P_i$). In particular, $C$'s write depends on the sum of the price of each item. An item's price is only obtained after the execution of the corresponding piece $P_i$. Pieces that are not ordered via an edge or path in the graph can be executed concurrently. In particular, each item update piece $P_i$ does not depend on any other item update piece.

In our current implementation, transactions are decomposed by hand. However, PWV's decomposition procedure can be made fully automatic, and thus would not require any developer effort. In its full generality, the algorithm for automatically decomposing transactions is beyond the scope of this paper, but we describe a simplified algorithm that creates a piece for each transaction statement: First create one piece per unique record that is read or written by the transaction. Data dependencies between pieces can be created using the following three steps. First, construct a transaction's statement-level control flow graph (where each statement corresponds to a single piece). Second, perform a reaching-definitions analysis on the control flow graph [8]. Third, for every definition that reaches a particular statement, construct an edge to the piece from the piece that creates the definition. For commit dependencies, create an edge to a writing piece that follows a transaction's commit point from every abortable piece.

PWV's decomposition algorithm is *modular*; a particular transaction's decomposition does not depend on any other transaction. As a consequence, transactions can be decomposed on clients or by admission control prior to being submitted to PWV. For the remainder of this section, we assume that PWV takes transactions' decomposed pieces as input.

## 4.2 Rendezvous points

PWV must execute the pieces of a decomposed transaction in an order that is consistent with the DAG constructed via the analysis of a transaction's data and commit dependencies. PWV coordinates the execution of pieces whose execution must be ordered using *rendezvous points*, a mechanism for synchronizing a partially ordered set of transaction statements originally proposed by Pandis

et al. in their work on data oriented transactions [40]. In addition to using rendezvous points to coordinate the execution of dependent pieces, PWV re-purposes rendezvous points (RVPs) to implement a lightweight transaction commit protocol [12].

**Coordinating dependent pieces.** PWV associates a single rendezvous point (RVP) with every piece that has at least one dependency. For instance, in the decomposed transaction described in Figure 2, PWV associates a RVP with the customer update piece $C$, since it depends on each product update piece $P_i$. Furthermore, two or more pieces can share a single RVP if they share the same set of parent pieces. In Figure 2, both $C$ and $S$ have the same set of parent pieces; $P_0, ..., P_n$. $C$ and $S$ can therefore share a RVP.

A RVP is used to determine when *all* of a dependent piece's parents have finished executing. For this purpose, a RVP uses a counter whose value is initialized to the number of parent pieces of a particular dependent piece. Each parent piece contains a reference to this RVP, and decrements the RVP's counter when it completes executing. When the value of the counter reaches zero, the downstream pieces associated with the RVP are ready to execute. In Figure 2's decomposed transaction example, the RVP counter associated with $C$ and $S$ is initialized to $n + 1$ (corresponding to its parent pieces pieces $P_0, ..., P_n$).

**Committing transactions.** PWV associates a single RVP with every abortable piece in a transaction. We refer to this RVP as the transaction's *commit RVP*. The commit RVP's counter is initialized to the number of abortable pieces. When an abortable piece finishes its execution and determines that it can commit, it decrements the value of the counter. However, if a piece must abort, it atomically sets the value of the counter to -1. The final value of the counter is either 0 or negative. If the value is 0, the transaction can commit. If the counter's value is negative, the transaction must abort.

## 4.3 Piece ordering constraints

In order to guarantee serializable and recoverable execution, PWV must appropriately order pieces corresponding to *different* transactions. PWV must deal with write-read, write-write, and read-write conflicts between pieces. Among these classes of conflicts, only write-read and write-write conflicts can impact recoverability. Read-write conflicts cannot impact recoverability because the abort of a reader has no impact on a later writer.

If transaction $T_1$ is serialized before $T_2$, and one or more pairs of their constituent pieces conflict, then PWV imposes constraints on the order in which $T_1$ and $T_2$'s pieces can execute. Assume that pieces $P_1$ and $P_2$ conflict (where $P_1$ and $P_2$ correspond to $T_1$ and $T_2$, respectively). PWV orders the execution of $P_1$ and $P_2$ based on **Constraints 1 and 2** below. **Constraint 1** captures ordering due to write-read and write-write conflicts, and is divided into two cases depending on whether the preceding writer can abort. **Constraint 2** captures ordering due to read-write conflicts.

- **Constraint 1.** There exists a write-read or write-write conflict between $P_1$ and $P_2$.

  **a) $\mathbf{P_1}$ can abort** because it precedes $T_1$'s commit point. In this case, *all* of $T_1$'s abortable pieces (including $P_1$) must execute before $P_2$.

  **b) $\mathbf{P_1}$ cannot abort** because it follows $T_1$'s commit point. In this case, $P_1$ must execute before $P_2$.

- **Constraint 2.** There exists a read-write conflict between $P_1$ and $P_2$. In this case, $P_1$ must execute before $P_2$.

The constraints above ensure that PWV produces only serializable and recoverable schedules. For serializability, PWV ensures that if $T_1$ is serialized before $T_2$, then $P_1$ is always executed before $P_2$. For recoverability, PWV ensures that transactions never observe dirty reads using **Constraint 1a**.

## 4.4 Executing pieces

This section describes a multi-core optimized implementation of PWV which respects the ordering constraints from Section 4.3.

### *4.4.1 System model and assumptions*

PWV divides the records in the database across a set of mutually exclusive partitions. Each partition processes pieces that read and write records in its partition. PWV can guarantee that a piece always writes records in a single partition by assigning each read or update statement its own piece. The techniques described in this section assume that PWV is deployed on a single multi-core server, such that a single CPU core is assigned a partition of the database.

Intuitively, PWV imposes two total orders; first, a total order on each transaction's pieces, second, a total order on transactions themselves. PWV processes transactions in batches, ordered as follows: Each transaction's pieces are ordered according to a topological sort of the decomposed transaction's DAG (Section 4.1). Pieces from different transactions are ordered according to transaction order in the input log; if transaction $T_1$ precedes transaction $T_2$ in the log, then *all* pieces of $T_1$ precede *all* pieces of $T_2$.

Given a batch of totally ordered pieces (as described above), each core only processes pieces that read or write records on its partition. In certain cases, it may be impossible to deduce in advance which partition must execute a piece. Such an ambiguous piece is replicated and processed by every partition. Upon ascertaining the correct partition at runtime, irrelevant replicas immediately commit without executing any logic, while the relevant piece is executed as usual (as Section 4.4.2 will describe).

### *4.4.2 Partition local concurrency control*

PWV must ensure that it executes pieces such that the constraints in Section 4.3 are satisfied. PWV ensures that the serialization order of transactions in a particular batch is *exactly the same* as the total order in which the transactions are received as input.

When a partition receives a batch of pieces, it first constructs a dependency graph whose edges represent conflicts among pieces within the partition. PWV constructs this dependency graph in its entirety (for a particular batch), before it executes the first piece in the batch. This partition-local dependency graph captures read-write, write-read, and write-write conflicts among pieces. PWV's dependency graphs are similar to those used in prior deterministic concurrency control algorithms [19, 20, 49]. These prior algorithms use dependency graphs in a *shared-everything* context, while PWV's dependency graphs are partition-local.

In order to construct a batch's dependency graph, PWV needs to determine piece-wise conflicts. This either requires determination of read- and write-sets as is done in other deterministic systems [19, 20, 45], or alternatively, a piece can conservatively request to access to arbitrary ranges of records, such as a partition or an entire table (Section 4.5.2 discusses this issue in detail).

A piece can be in one of three states, **Unexecuted**, **Executed**, and **Completed**. Non-abortable pieces can either be in state **Unexecuted** or **Completed**. After executing, abortable pieces first transition to **Executed**, and eventually transition to **Completed** after their corresponding transactions' commit decisions are determined. All pieces are initially **Unexecuted**.

Once a partition core constructs a batch's dependency graph, it progresses through the total order of pieces generated in Section 4.4.1 and performs three checks to see if it can immediately

execute that piece. The first check ensures that a piece $P$ is correctly ordered with respect to pieces from *other* transactions. The second and third checks ensure that $P$ is correctly ordered with respect to pieces in its own transaction.

- First, for every piece $P'$ in the partition-local dependency graph which $P$ depends on, the partition core checks whether $P'$ is **Completed**. This step ensures that conflicting pieces execute according to the pre-determined total order of transactions.
- Second, the partition core checks that $P$'s RVP counter is zero. This step ensures that $P$'s data dependencies have been satisfied.
- Third, if $P$ is not abortable then the partition core checks whether $P$'s corresponding transaction has obtained a commit decision (by checking the corresponding commit RVP). This step ensures that non-abortable pieces only execute if their corresponding transactions commit. This check always holds if $P$ is abortable.

If all three of the above checks hold, there exist two cases depending on whether $P$ is abortable. If $P$ is abortable, then it is executed. If $P$ is non-abortable and the third check determines that $P$'s transaction committed, then $P$ is executed. However, if the third check determines that $P$'s transaction aborted, then $P$ can be ignored and its state directly transitioned to **Completed**. If any of the above three checks does not hold, $P$ is added to a list of pending pieces, and the core moves on to the next piece in the batch.[1]

On executing piece $P$, its core decrements the count on each of $P$'s children's RVPs (Section 4.2). If $P$ is non-abortable, its state transitions to **Completed**. If $P$ is abortable and can commit, the partition core decrements the corresponding commit RVP's counter and $P$'s state transitions to **Executed**. If $P$'s commit decrements the RVP's counter to zero, then it means $P$'s transaction has committed, and the partition core proceeds to transition every abortable piece's state to **Completed** (including $P$'s). If $P$ is the first piece to abort, the partition core undoes the writes of the pieces that committed before $P$ (even if the pieces reside on remote partitions), and mark their state as well as $P$'s as **Completed**. Note that remote undo is safe because the abortable pieces are still in state **Executed**, and as a consequence, later conflicting pieces from different transactions are blocked because of the first check above.

The above partition-local mechanisms guarantee that the ordering constraints of Section 4.3 hold. The first step, which checks whether conflicting pieces have finished executing, ensures **Constraint 1b** and **Constraint 2** hold, while the first step and the commit protocol above together ensure that **Constraint 1a** holds.

PWV's constraints enable implementations that exploit both intra-transaction parallelism and early write visibility. Our implementation uses the a priori total ordering of transactions and their pieces to correctly order pieces across different transactions. Prior decomposition algorithms cannot exploit intra-transaction parallelism. This limitation is fundamental because these algorithms cannot make any a priori ordering guarantees across multiple pieces corresponding to a pair of conflicting transactions [36, 43, 48, 52, 55].

## 4.5 Discussion

### 4.5.1 Deferred constraint checks

Database constraints on records are usually checked as soon as update statements that could invalidate the constraints are evaluated. However, certain types of constraints (such as those involving the values of two or more records) are rendered temporarily inconsistent if evaluated after a single update statement. Transactions typically fix these constraints with later updates. Database systems therefore allow certain constraint checks to be deferred to the end of transactions' execution [3].

If applications use deferred integrity constraints, then PWV must place the commit point of any transaction that triggers the deferred constraint check at the end of its execution. Importantly, this *does not* mean that PWV cannot be used by such applications, only that certain transactions' writes cannot be made visible early. Furthermore, because PWV uses a modular decomposition procedure, deferred constraint checks in one transaction do not affect the commit point of other transactions, even if they conflict. It should be noted that practitioners have proposed that applications should avoid deferred constraint checks when possible; for example, by grouping updates together using multiple assignment operators [1, 2].

### 4.5.2 Comparison to prior deterministic systems

As explained in Section 3.1, certain deterministic systems use speculative execution to determine a transaction's read- and write-sets prior to its execution. These systems use read- and write-set knowledge to relax the pre-determined total order on transactions into a partial order [19, 45].

PWV similarly needs to determine conflict information prior to transaction execution. In particular, PWV leverages piece-wise conflict information in order to execute pieces out of order on each partition (Section 4.4.2). However, PWV's early write visibility discipline enables an alternative mechanism to the speculative execution required by prior deterministic systems.

If their read- and write-sets are unknown, pieces can conservatively specify ranges of records that they may need to access. These ranges can be arbitrarily imprecise; for instance, a piece may request exclusive access to an entire table even though it only updates a handful of records in the table. While conservatively requesting coarse-grained access to a range of records can limit concurrency in conventional serializable protocols [42, 46], PWV's piece-wise execution reduces the duration for which conflicting transactions are blocked, and thus substantially ameliorates any performance penalty associated with coarse-grained requests for record access.

Even when read- and write-sets are known in advance, we have found that coarse-grained conflict specification can improve the performance of PWV — especially under lower contention workloads. Coarse-grained conflict information allows PWV to tradeoff logical concurrency between conflicting pieces for reduced concurrency control overhead, in the spirit of hierarchical locking [23]. We explore this tradeoff in our experimental evaluation (Section 5.3).

## 4.6 Proof of optimality

This section proves that if transactions' read- and write-sets can be deduced a priori, then PWV's piece ordering constraints are necessary and sufficient to guarantee serializability (SR) and avoid cascaded aborts (ACA) in the absence of failures. We term these two properties together as SR ACA. Our proof pertains to transaction histories permitted by PWV's piece ordering constraints (Section 4.3), not our specific implementation of these constraints (Section 4.4). The implication of this proof is that, when transactions' read- and write-sets are known a priori, it is impossible for an SR ACA protocol to extract more concurrency from a workload than an ideal implementation of PWV.

Our proof asserts that, under the assumptions above, PWV permits *all* valid serializable transaction histories (unlike, for instance, two-phase locking, which cannot permit certain valid serializable histories [12, 18]). Section 4.3 showed that PWV's constraints on pieces are sufficient to guarantee serializability. We prove that these

[1] As an optimization, our implementation performs the third check even if $P$ is abortable. If its transaction aborted, $P$ does not need to execute, regardless of whether or not it is abortable, and can directly transition to **Completed**, even if the first two checks do not hold.

constraints are also *necessary* if transactions' read- and write-sets are known a priori.

We define a transaction's constituent read, write, and commit/abort statements as its *operations*. We denote read, write, and commit operations of transaction $T_i$ as $r_i[x]$, $w_i[x]$, and $c_i$, respectively (where reads and writes occur on record $x$). We denote operation $o_1$ preceding operation $o_2$ as $o_1 < o_2$.

If a transactions' read- and write-sets are known a priori, PWV can assign each read and write operation to a single piece. The proof sketch below therefore describes constraints on individual read and write operations, not pieces. Given two conflicting transactions, $T_i$ and $T_j$ such that $T_i$ is serialized before $T_j$, PWV imposes the following constraints on the order in which their operations can execute:

- **Case 1a.** If $w_i[x]$ conflicts with $o_j[x]$, and $w_i[x] < c_i$ (where $o_j$ is either $r_j$ or $w_j$), then PWV ensures that $c_i < o_j[x]$. No SR ACA concurrency control protocol can produce the order $w_i[x] < o_j[x] < c_i$ because $T_i$ may abort, and $o_j[x]$ would have observed an uncommitted write ($w_i[x]$) which could have rolled back prior to $c_i$. Furthermore, no serializable concurrency control protocol can produce $o_j[x] < w_i[x]$ because doing so would violate the assumption that $T_i$ is serialized before $T_j$.

- **Case 1b.** If $w_i[x]$ conflicts with $o_j[x]$, and $c_i < w_i[x]$, then PWV ensures that $w_i[x] < o_j[x]$. No SR ACA concurrency control protocol can produce $o_j[x] < w_i[x]$ because doing so would violate the assumption that $T_i$ is serialized before $T_j$.

- **Case 2.** If $r_i[x]$ conflicts with $w_j[x]$, then PWV ensures that $r_i[x] < w_j[x]$. No SR ACA concurrency control protocol can produce $w_j[x] < r_i[x]$ because it violates the assumption that $T_i$ is serialized before $T_j$.

## 4.7 Corner cases

PWV classifies each of a transaction's pieces as abortable based on the transaction's logic (specifically, the location of explicit abort statements) and constraints on database values, such as integrity constraints. Above, we classified all other types of aborts as system-induced, which are eliminated by deterministic database systems. However, there exist corner cases that fall in between these two categories, in which even deterministic systems would abort the transaction, but nonetheless are not caused by abort statements in transaction logic or integrity constraints. Two examples of such corner cases are integer overflows and infinite loops. One naïve corner case handling mechanism is to consider all pieces that modify integers or involve loops (and so on for all corner cases) as abortable.

Obviously, this naïve solution would lead to a large number of pieces being marked as abortable, precluding PWV's ability to make many writes visible early. A better approach is to engineer a solution to deal with each corner case individually. For loops, it is possible to use static analysis to detect loops that will definitely terminate. Only for the case where the static analysis fails to guarantee that a loop will terminate do the pieces corresponding to the loop's logic need to be marked as abortable. For integer overflows, the system could simply allow integers to overflow (as is the default setting in many modern database systems). Alternatively, the size of the integer (or entire column) can be dynamically increased in order to accommodate the overflow.

Although dealing with corner cases on a case-by-case basis using software engineering techniques, such as static analysis and exception handling, is likely the optimal solution, our current implementation uses a more general approach. Our implementation optimizes for the common case where corner cases do not occur, and suffers from reduced performance when they do. In particular, upon encountering a corner case, such as an integer overflow or infinite loop, our implementation treats this as a full system failure, trashes the current database state, reloads state from the most recent checkpoint, expunges the problematic transaction from the log, and replays the log forward from the checkpoint without the problematic transaction. Clearly, optimizations of this algorithm are possible. For example, instead of trashing the entire database state and replaying the entire log from a checkpoint, one could use piece-wise conflict information to selectively re-execute only those pieces which may have read the aborted transaction's writes.

By selectively aborting problematic transactions, the above dynamic error handling mechanism prevents these expunged transactions from affecting stable database state. However, it does not prevent the writes performed by such aborted transactions from being visible to the application running over the database (for instance, via simple read queries). Our current implementation delays returning results of any data to the application until any transactions that contributed to these results have finished execution. We implemented this via an epoch-based external visibility mechanism, similar to the mechanism used in Silo [46], where read results are returned to the user at the end of each batch of transactions.

# 5. EVALUATION

This section evaluates PWV against three serializable protocols – locking, transaction chopping, and optimistic concurrency control (OCC) – and a read committed protocol.

**Locking.** This implementation is based on two-phase locking. The implementation acquires locks in lexicographic to eliminate deadlocks [42]. To avoid the overhead of maintaining a separate lock table, logical locks are implemented as MCS reader-writer latches co-located with records [35, 47].

**Transaction chopping.** This implementation is based on Wang et al.'s IC3 protocol [48]. IC3 uses a serializable protocol to schedule transactions' constituent pieces, and dynamically enforces causal dependencies across conflicting pieces. Our chopping implementation uses locking to guarantee serializable execution of pieces.

**OCC.** This implementation is based on Silo [46]. OCC validates transactions using decentralized timestamps, and avoids writing shared-memory for records that are only read.

**Read committed.** We implemented read committed isolation (RC) by modifying the OCC algorithm above. Our RC implementation provides PL-2 isolation [5], which imposes two constraints on transactions' reads and writes. First, transactions can only read committed values of records. Second, if two transactions perform conflicting writes, then their writes must be consistently ordered [5, 6]. Our RC implementation buffers transactions' writes until commit time. A writer will therefore only interact with a reader at commit time. RC uses a record latch – Silo's per-record TID word [46] – to ensure that reads observe only committed values of records. A writer acquires this latch while copying a record's updated value from its local buffer. Readers spin on the latch until it is free. RC deals with write-write conflicts using MCS latches, which, as in our locking implementation, are co-located with records [34, 47]. At commit time, a transaction acquires its write latches in lexicographic order, and then copies updated records' values from its local write buffers. Our RC implementation provides PL-2 isolation, which provides more concurrency than the PL-2L isolation provided by most real-world implementations of RC [5, 11, 23, 32].

We conduct our experimental evaluation on a single 40-core machine, consisting of four 10-core Intel E7-8850 processors and 128-GB of memory. Our operating system is Linux 3.19.0. All experiments are performed in main-memory, so secondary storage is not utilized for our experiments. Every implementation explicitly pins long running threads to CPU cores.

## 5.1 Effect of contention

In this experiment, we use the Yahoo! Cloud Serving Benchmark to understand PWV's basic performance characteristics [14]. The database consists of a single table of 1,000,000 records. Each record is 1,000 bytes in size. The workload in this section consists of a single type of transaction; an update transaction that performs 20 read-modify-write (RMW) operations. The records updated by each transaction are chosen from a zipfian distribution. We vary contention by varying the zipfian parameter, *theta* [24]. PWV's batch size is set to 10,000 transactions. We partition data using a random hash function, as a consequence most PWV transactions span more than 10 partitions.

The experiments in this section assume that transactions do not contain any abort statements, that is, they are guaranteed to commit before they begin executing. As a consequence, in PWV, there is no delay from the time that an individual update is performed, and the time it is made visible to other transactions. We perform three sets of experiments, one under low contention, one under high contention, and one under varying contention (Figure 3). Transaction chopping does not provide any benefit to our locking implementation in this experiment because it decomposes transactions based on table-level accesses. We thus omit transaction chopping from this set of experiments.

Figure 3a shows the results of the low contention experiment. We measure the throughput of each implementation while varying the number of available CPU cores. The zipfian parameter, theta[2], is set to 0. Figure 3a indicates that each system scales similarly under low contention because conflicts among transactions are rare.

Figure 3b shows the results of the same experiment under high contention. In this case, the records updated by each transaction are chosen from a zipfian distribution with theta set to 0.9. Locking and OCC's throughput drops significantly as compared to the low contention experiment. This is because in the high contention experiment, the likelihood that a pair of transactions conflicts is much higher. Figure 3b also shows that RC's throughput significantly decreases under contention. Although RC does not impose any order among conflicting reads and writes, writes across transactions must still be consistently ordered. Accordingly, our RC implementation acquires write locks on records at commit time, before transactions copy updated values from their buffers into the database (Section 5). The decrease in RC's throughput under high contention occurs due to transactions acquiring the same write locks at commit time. Importantly, these locks are acquired at commit time, and held for much shorter duration than locks acquired by serializable locking. This explains why RC can attain a much higher peak throughput than locking under high contention. It should be noted that since transactions perform only updates, locking-based PL-2L implementations which hold long-duration write locks on records would perform the same as serializable locking.

Finally, we find that PWV's throughput trend is completely different from the other concurrency control algorithms. The locking and OCC lines remain nearly flat, while RC peaks at 12 cores and plateaus thereafter. In contrast, PWV's throughput increases with core count without plateauing. Since transactions contain no abort statements, PWV can make an update visible as soon as it completes, without waiting for the corresponding transaction to finish executing in its entirety. This decoupling of a single transaction's constituent writes is the reason that contention does not adversely affect PWV's throughput. At 40 cores, PWV outperforms OCC, Locking, and RC by 15x, 7x, and 3x, respectively.

[2]Theta can take values between 0 and 1. Larger values of theta correspond to higher contention.

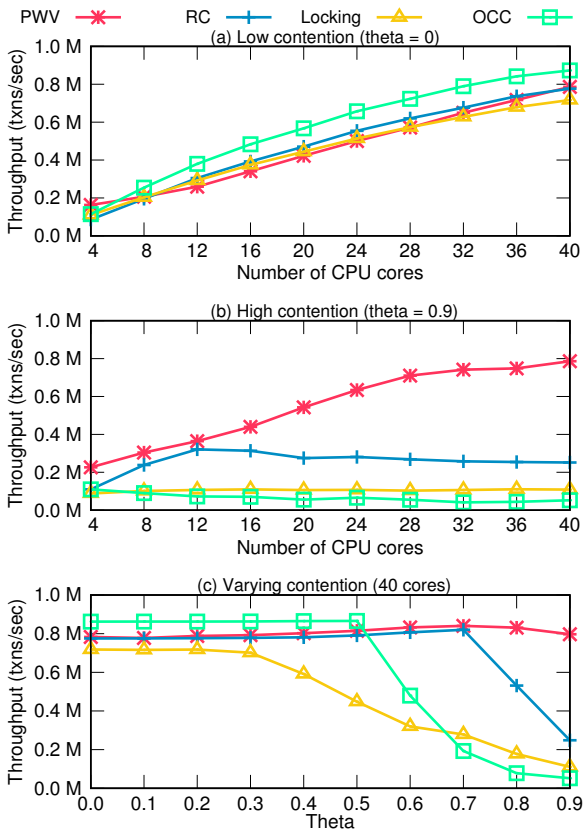


**Figure 3:** Effect of contention on throughput.

To better understand the behavior of each algorithm, we measured each algorithm's throughput while varying contention. Figure 3c shows the results of the experiment (contention increases with increasing theta). Locking and OCC's throughputs decrease at medium-low levels of contention (theta range of 0.3 to 0.6). RC and PWV's throughput remain very similar from low to medium contention. RC's throughput decreases when theta increases from 0.7 to 0.8 (medium to medium-high contention). PWV's throughput remains nearly constant despite variations in contention.

These experiments show that PWV is highly robust to contention in the ideal scenario that transactions never experience logic-induced aborts. Under high contention, PWV can outperform conventional serializable protocols by more than order-of-magnitude. PWV even outperforms our highly-optimized *non-serializable* read committed implementation, indicating that PWV can provide applications that fit this ideal with fast serializable isolation.

## 5.2 Effect of commit point

In this experiment, we limit PWV's ability to make individual writes visible as soon as they complete. We augment transactions' logic with explicit abort statements. By varying the position of transactions' abort statements with respect to its update statements, we can control which writes can be made visible immediately.

We term the point at which a transaction contains an abort statement its *commit position*. The value of a transaction's commit posi-

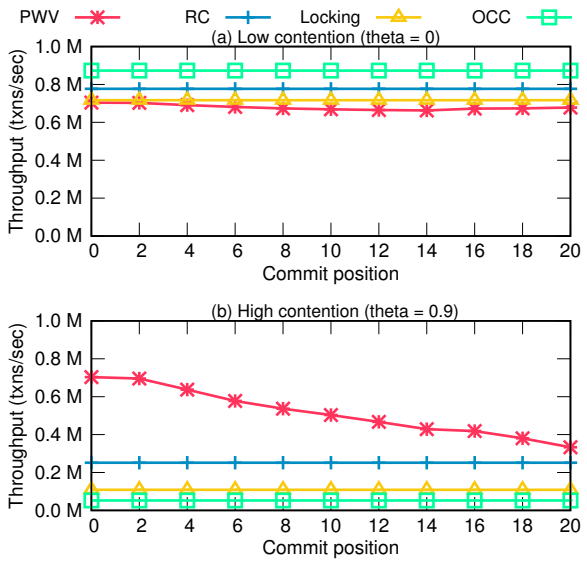


**Figure 4:** Effect of commit point on throughput.

tion corresponds to the number of write operations that precede it. We measure the effect of changing a transaction's commit position under low and high contention. As in Section 5.1, we use transactions that perform 20 read-modify-writes. The parameters of the low and high contention experiments are the same as Section 5.1.

Figure 4a shows the result of the low contention experiment. Locking, OCC, and RC's throughputs do not change while varying a transaction's commit position. The fact that a transaction contains an explicit abort statement has no effect on these protocols. PWV's throughput also remains nearly constant while varying transactions' commit position. However, small variations occur because each core needs to perform slightly more work to execute transactions; PWV must execute the RVP-based commit protocol (Section 4.2), and writing pieces that follow a transaction's commit point must wait for the commit protocol to complete.

Figure 4b shows the result of the high contention experiment. As before, locking, OCC, and RC's throughputs do not change with commit position. In contrast, PWV's throughput decreases significantly with increasing commit position. This is because PWV cannot make a transaction's writes visible until its commit position. Under high contention, this delayed visibility hurts throughput. Unlike the other algorithms, PWV employs intra-transaction parallelism by executing a single transaction's updates in parallel on multiple cores. Intra-transaction parallelism minimizes the execution time of an individual transaction, which reduces write visibility delay. Thus, PWV's throughput remains significantly higher than locking and OCC's, even when every algorithm makes writes visible at the end of each transaction (the right-most point of Figure 4b).

## 5.3 Reducing concurrency control overhead

We now examine the performance implications of PWV's coarse-grained conflict isolation mechanisms, whereby transactions can request access to a set of keys that is a guaranteed to be a superset of the keys they actually access. We show that coarse-grained isolation can significantly improve PWV's performance under low contention, while preserving its advantages under high contention.

We evaluate the throughput of each system under a workload consisting of an equal mix of TPC-C NewOrder and Payment transactions [4]. We use two versions of PWV for this benchmark: standard PWV, and PWV-coarse. In order to access the District, Customer, NewOrder, OrderLine, Order, and History tables, both PWV and PWV-coarse specify read and write requests at the granularity of (warehouse-id, district-id) foreign-key pairs. We partition these tables by (warehouse-id, district-id) pairs. Pieces which request access to a table via the same (warehouse-id, district-id) foreign-key pair are always processed by the same partition, but records with the same (warehouse-id, district-id) foreign-key from different tables, say Customer and NewOrder, can reside on different partitions. PWV and PWV-coarse differ in their conflict specification mechanisms for Stock records. PWV isolates pieces at the granularity of Stock record primary keys, while PWV-coarse isolates conflicting accesses to the Stock table at the granularity of warehouses. In PWV-coarse, each piece effectively requests exclusive access to the entire set of stock records within a single warehouse. Thus, if two pieces update stock records within the same warehouse, then PWV-coarse determines that they conflict, even if their read- and write-sets do not overlap.

We also measure the impact of coarse-grained isolation on conventional recoverability mechanisms by implementing a version of coarse-grained locking. Like PWV-coarse, coarse-grained locking protects all the stock records within a warehouse with a single lock. Furthermore, we include two versions of our implementation of IC3's transaction chopping protocol [48]. A standard version of IC3, which isolates pieces at the granularity of reads and writes, and a version which exploits commutativity. We refer to these algorithms as chopping and chopping-comm, respectively. Note that PWV does not make any commutativity assumptions.

We first show the results of a low contention experiment in which we simultaneously vary the number of database cores and warehouses (the number of warehouses is equal to number of cores). The non-PWV algorithms affinitize a core to a particular warehouse; requests which originate at a particular warehouse are always processed by the same core. This experiment therefore represents the *best case* scenario for these systems; locality is maximized and conflicts are minimized because transactions on a particular origin warehouse are always processed by the same core [42, 46].

Figure 5a shows the results of the experiment. Every algorithm's throughput scales with increasing core count. However, there exist significant differences in absolute throughput achieved by each algorithm. First, both chopping and chopping-comm are outperformed by locking because of the extra overhead they impose on tracking dependencies between pieces at runtime. Chopping-comm outperforms chopping despite the lack of conflicts because it maintains less chopping-related meta-data corresponding to updates by commuting pieces. Locking outperforms locking-coarse because locking-coarse induces unnecessary conflicts. Since approximately 10% of NewOrder transactions update stock records from remote warehouses, these transactions will block due to spurious stock update conflicts. This reduction in concurrency outweighs any potential benefit in reduced concurrency control overhead.

In contrast, the opposite effect is observed for PWV-coarse, where the reduced overhead greatly outweighs the reduction in concurrency. PWV-coarse *pipelines* the execution of transactions to minimize the impact of coarse-grained isolation on blocking. If two transactions conflict, PWV-coarse only imposes an order between the transactions' conflicting pieces, not entire transactions.

Figure 5b shows the result of a high contention experiment in which we fix the number of warehouses to 1, and vary the number of database cores. In non-PWV algorithms, the entire database is shared across all cores of the system. Due to increasing contention, OCC, locking, and locking-coarse's throughput remain mostly stagnant with increasing core count. Surprisingly, chopping

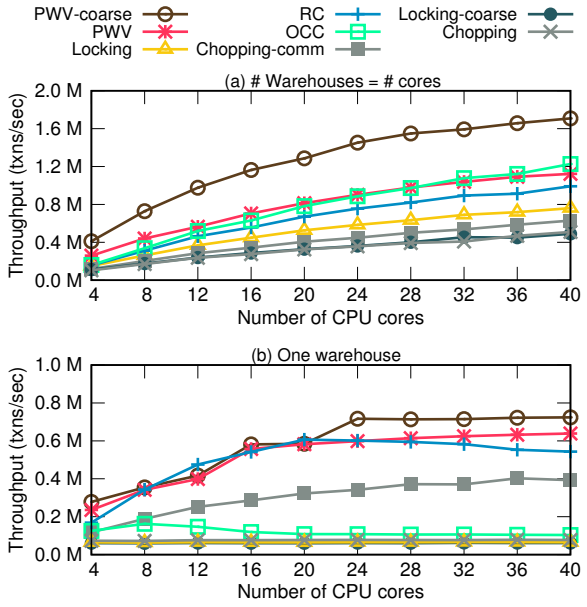


**Figure 5:** TPC-C NewOrder and Payment throughput.

is unable to outperform any of these systems. There are two reasons for this. First, if two pieces corresponding to a pair of transactions conflict, then later non-conflicting pieces corresponding to the same pair of transactions are constrained [48]. Second, dynamically tracking dependencies across pieces imposes overhead at runtime, which worsens under increased contention.

In contrast, PWV and PWV-coarse are able to outperform both RC and chopping-comm, despite the fact that they provide serializable isolation and make no assumptions about commutativity of pieces. PWV and PWV-coarse outperform chopping-comm because they impose no constraints on the execution of non-conflicting pieces, while chopping-comm must constrain non-conflicting pieces (although these constraints are minimized due to commutativity). The difference between RC and PWV is smaller than prior experiments because RC must acquire fewer commit time write locks and consequently holds write locks on the most contended records (Warehouse and District records) for short durations.

| | PWV-coarse | Locking | OCC | RC | Chopping-comm |
|---|---|---|---|---|---|
| 40 warehouses | 2ms | 8.2ms | 3.4ms | 4.5ms | 42ms |
| 1 warehouse | 7.3ms | 73ms | 534ms | 10ms | 136ms |

**Figure 6:** $95^{th}$%ile latency for batches of 5000 NewOrder and Payment transactions under maximum throughput (40 cores).

Finally, Figure 6 shows the $95^{th}$ percentile latency of processing batches of 5,000 TPC-C transactions. Figure 6 shows that PWV-coarse's multi-core scalability does not come at the expense of latency; PWV-coarse's $95^{th}$ percentile latency latency is lower than that of every other protocol under both high and low contention.

## 6. RELATED WORK

**Transaction decomposition.** Transaction chopping is the most well-known mechanism for serializable transaction decomposition [43], but has two important limitations relative to PWV; it permits only a single sub-transaction to abort, and only permits a pair of decomposed transactions to conflict on a single sub-transaction. As a consequence of these two limitations, chopping produces coarse-grained decomposed transaction. Indeed, recent systems based on transaction chopping, such as transaction chains [55] and Salt [51], resort to the use of application semantics to reduce conflicts among sub-transactions. In contrast, PWV places no restrictions on which pieces of a decomposed transaction can abort and the number of piece-wise conflicts between a pair of decomposed transactions.

Rococo dynamically fixes serializability violations among pieces of a decomposed transaction with no outgoing data dependencies [36]. However, pieces with outgoing data dependencies must be scheduled using conventional concurrency control. Recent work in optimistic systems that employ delayed write visibility proposes addressing serializability violations by selectively re-executing a subset of transactions' logic [15, 50]. In contrast, PWV never violates serializability at runtime; transactions' pieces are executed in an order that is consistent with a pre-determined total order.

IC3 [48] and Runtime Pipelining [52] address limitations in conventional transaction chopping by dynamically enforcing causal dependencies across pieces. Furthermore, both IC3 and Runtime Pipelining deal with aborts via cascaded rollbacks or defaulting to a rollback-safe decomposition based on conventional transaction chopping. Unlike IC3 and Runtime Pipelining, PWV uses a pre-determined total order to enforce causal dependencies across pieces, and avoids cascaded rollbacks via early commit points.

Faleiro et al. propose a decomposition mechanism that breaks a transaction's logic is into two pieces, an eager and a lazy piece [20]; a transaction's eager piece is executed as soon as the transaction enters the database, while the lazy piece is deferred. Lazy transactions are coarsely decomposed based on transaction commit points. In contrast, PWV decomposes a transaction using both, a transaction's commit point *and* its data-flow. PWV can therefore decompose transactions at a much finer granularity than lazy transactions.

**Write visibility.** Jones et al. propose a speculative write visibility discipline to avoid making new transactions wait on distributed coordination required to commit earlier transactions [27]. OPT is a distributed commit protocol where transactions are permitted to read the uncommitted writes of transactions in two-phase commit's prepare phase [25]. Reddy et al. propose a speculative write visibility discipline in which a transaction executes against *both* the pre-image and after-image of preceding uncommitted transactions [41]. This forked execution prevents cascading aborts at the cost of extra CPU and memory resources. Agrawal and El Abbadi propose ordered shared locks [7], which permit transactions to read uncommitted writes of preceding transactions. Each of these write visibility mechanisms must permit dirty reads to avoid the overhead associated with delayed write visibility. In contrast, PWV prevents dirty reads altogether, and instead relies on deterministic execution to arrive at early transaction commit decisions.

**Exploiting application semantics.** A Saga is decomposed transaction that exploits application-specific semantics to avoid serializable isolation and tolerate aborted reads [21]. Alonso et al. propose a logical inverse undo operation to avoid the overhead of delayed write visibility while simultaneously avoiding cascaded aborts [9]. Gawlick and Kinkade, and O'Neil proposed variants of the escrow method to exploit commutative operations on contended records [22, 39]. Doppel exploits commutative operations on hot records to replicate such records, and allow concurrent commutative updates to each replica [37]. Bailis et al. propose an application dependent correctness criterion *I-confluence*, that determines whether a coordination free execution of transactions will preserve application invariants [10]. Conway et al. propose using monotonicity analysis to eliminate coordination in distributed applications [13]. Each of these prior techniques exploits application semantics to enable

either a new recoverability or isolation mechanism, or both. In contrast, PWV makes no assumptions about application semantics, while still guaranteeing recoverability and serializability. PWV's novelty lies in its use of early write visibility and piece-wise scheduling of transactions to significantly reduce the duration of conflict-induced blocking.

**Transaction scheduling mechanisms.** QURO reorders transaction statements to reduce contended lock hold times [53]. DORA is a partitioned system that exploits intra-transaction parallelism on multi-core servers [40]. Both DORA and QURO use two-phase locking to guarantee serializability, and hence inherit the limitations of delayed write visibility. Whitney et al. [49], and Faleiro and Abadi [19] propose using dependency graphs to schedule transactions in deterministic systems. PWV uses a similar scheduling mechanism within a partition, but at the granularity of transaction pieces. Faleiro and Abadi's work on deterministic multi-version concurrency control is complimentary to PWV; PWV can use multi-versioning to ensure that reads never block writes.

# 7. CONCLUSIONS

This paper identifies write visibility delay as an important inhibitor of database concurrency and introduces early write visibility, a recoverability mechanism that enables writes to become visible as soon as a transaction executes any statements that could cause it to abort. To enable early write visibility, we designed PWV, a new concurrency control protocol that leverages database determinism to prevent arbitrary transaction aborts, and found that PWV can significantly outperform modern serializable and non-serializable concurrency control protocols.

**Acknowledgments.** This work was supported by the NSF under grant IIS-1527118.

# 8. REFERENCES

[1] How to write correct sql and know it: A relational approach to sql. https://goo.gl/WDpTDU.
[2] The multiple assignment operator and deferred constraints. https://goo.gl/84aGMR.
[3] Oracle database online documentation 10g release 2 (10.2). https://goo.gl/g9oFBs.
[4] TPC Council. TPC Benchmark C revision 5.11. 2010.
[5] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, 1999.
[6] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *ICDE*, 2000.
[7] D. Agrawal and A. El Abbadi. Locks with constrained sharing. In *PODS*, 1990.
[8] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques*. Addison Wesley, 1986.
[9] G. Alonso, D. Agrawal, and A. El Abbadi. Reducing recovery constraints on locking based protocols. In *PODS*, 1994.
[10] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *PVLDB*, 8(3), 2014.
[11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD*, 1995.
[12] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
[13] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SoCC*, 2012.
[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.
[15] M. Dashti, S. B. John, A. Shaikhha, and C. Koch. Repairing conflicts among MVCC transactions. *CoRR*, abs/1603.00542, 2016.
[16] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, 1984.
[17] B. Ding, L. Kot, A. Demers, and J. Gehrke. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *SoCC*, 2015.
[18] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *CACM*, 19(11), 1976.
[19] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11), 2015.
[20] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD*, 2014.
[21] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, 1987.
[22] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS fast path. *DE Bull*, 8(2), 1985.
[23] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared database. In *IFIP*, 1976.
[24] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, 1994.
[25] R. Gupta, J. Haritsa, and K. Ramamritham. Revisiting commit processing in distributed database systems. In *SIGMOD*, 1997.
[26] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *PVLDB*, 3(1-2), 2010.
[27] E. P. C. Jones, D. J. Abadi, and S. R. Madden. Concurrency control for partitioned databases. In *SIGMOD*, 2010.
[28] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *SIGMOD*, 2013.
[29] A. Kemper and T. Neumann. Hyper: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.
[30] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *SIGMOD*, 2016.
[31] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2), 1981.
[32] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4), 2011.
[33] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *ICDE*, 2014.
[34] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS*, 9(1), 1991.
[35] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPoPP*, 1991.
[36] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, 2014.
[37] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *OSDI*, 2014.
[38] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD*, 2015.
[39] P. E. O'Neil. The escrow transactional method. *TODS*, 11(4), 1986.
[40] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1-2), 2010.
[41] P. K. Reddy and M. Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *TKDE*, 16(2), 2004.
[42] K. Ren, J. M. Faleiro, and D. J. Abadi. Design principles for scaling multi-core oltp under high contention. In *SIGMOD*, 2016.
[43] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *TODS*, 20(3), 1995.
[44] M. Stonebraker, S. R. 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 *VLDB*, 2007.
[45] A. Thomson, T. Diamond, S. chun Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.
[46] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *SOSP*, 2013.
[47] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2), 2016.
[48] Z. Wang, S. Mu, H. Y. Yang Cui, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. 2016.
[49] A. Whitney, D. Shasha, and S. Apter. High volume transaction processing without concurrency control, two phase commit, SQL or C++. In *HPTS*, 1997.
[50] Y. Wu, C.-Y. Chan, and K.-L. Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *SIGMOD*, 2016.
[51] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining acid and base in a distributed database. In *OSDI*, 2014.
[52] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance acid via modular concurrency control. In *SOSP*, 2015.
[53] C. Yan and A. Cheung. Leveraging lock contention to improve oltp application performance. *PVLDB*, 9(5), 2016.
[54] C. Yao, D. Agrawal, G. Chen, Q. Lin, B. C. Ooi, W.-F. Wong, and M. Zhang. Exploiting single-threaded model in multi-core in-memory systems. *TKDE*, 28(10), 2016.
[55] 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 *SOSP*, 2013.