Deferred Runtime Pipelining for contentious multicore software transactions (extended version)

Shuai Mu
Stony Brook University

Sebastian Angel
University of Pennsylvania

Dennis Shasha
New York University

Abstract

DRP is a new concurrency control protocol for software transactional memory that achieves high throughput, even for skewed workloads that exhibit high contention. DRP builds on prior works that chop transactions into pieces to expose more concurrency opportunities, but unlike these works, DRP performs no static analyses and supports arbitrary workloads. DRP achieves a high degree of concurrency across most workloads and guarantees deadlock freedom, strict serializability, and opacity. We incorporate DRP into the software transactional objects library STO [18] and find that DRP improves STO's throughput on several STAMP benchmarks by up to 3.6×. Additionally, an in-memory multicore database implemented with our modified variant of STO outperforms databases that use OCC or transaction chopping for concurrency control. Specifically, DRP achieves 6.6× higher throughput than OCC when contention is high. Compared to transaction chopping, DRP achieves 3.3× higher throughput when contention is medium or low. Furthermore, our implementation achieves comparable performance to OCC and transaction chopping at other contention levels.

1 Introduction

The challenge of writing correct and efficient concurrent programs has inspired many new runtimes [4, 6, 15] and abstractions [16–18, 24, 29, 30]. For example, transactions, which can simplify concurrent application design, are commonly used in databases and distributed systems. However, transactions are rarely used in single-machine multi-threaded applications. A major reason for this is performance: using (software) transactional memory [29] introduces a lot of overhead. Instead, programmers turn to synchronization primitives such as locks and semaphores to build their concurrent applications.

Recent efforts on transactional data structures [18, 30, 41] allow programmers to write applications with transactions defined over particular objects (rather than arbitrary memory words) while guaranteeing atomicity and isolation. For example, STO [18] is a recent library that includes many transactional objects (e.g., arrays, maps, primitive types) that C++ developers can use in their transactions. STO relies on an optimized variant of optimistic concurrency control (OCC) [19] to process transactions. OCC works well in low contention environments (e.g., read-only workloads, applications with few threads) since it incurs low locking overhead and aborted transactions can be re-executed quickly due to memory's low latency. However, the number of aborts in OCC becomes excessive under high contention workloads (e.g., many threads competing to get the id of the next order), significantly degrading performance [5, 39].

The above is problematic since high-contention workloads are not uncommon, and include skewed workloads with hot items. To provide high throughput for such high-contention workloads, recent in-memory multicore databases [12, 37] propose the use of static analysis techniques that preprocess the workload to chop transactions into pieces [28]. The execution of these pieces can be pipelined at runtime [38], which exposes more concurrently since locks are held for shorter periods of time and are released early. The drawback of this approach is that relying on static analysis sacrifices generality by requiring all transactions to be known ahead of time.

Inspired by these works, this paper introduces a concurrency control protocol called *Deferred Runtime Pipelining* (DRP) and implements it in STO. DRP extends the *runtime pipelining* (RP) scheme of Xie et al. [38] to handle arbitrary transactions without static analysis, while still guaranteeing strict serializability (or opacity) and deadlock freedom. The key idea behind DRP is to defer the execution of transactions using *intentions* (similar to futures or promises in asynchronous programming), and pipelinine the lock acquisition and actual execution after a transaction's logic is known (or "commits"). This approach has several benefits (many of them previously documented in the context of lazy transaction processing [13]), but most importantly, it allows individual operations to be assigned arbitrary *ranks* that dictate the order within the pipeline, avoiding the use of static analysis.

A stumbling block in designing DRP is that some transactions cannot be fully deferred. As one example, consider a transaction that is not specified in full and that has logic that changes depending on the value of observed data. To account for this, we classify transactions into two types, *tame* and *wild*, and design compatible versions of DRP for each. Tame transactions have foreknowledge of the objects on which they will act before they execute. This is achieved by deferring the execution of their operations. Wild transactions, on the

EuroSys '19, March 25–28, 2019, Dresden, Germany.
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6281-8/19/03...$15.00
https://doi.org/10.1145/3302424.3303966

other hand, may alter their read/write sets based on the values they read. Programmers need not be aware of the tame/wild distinction, but the system acts slightly differently for the two, as we discuss in Section 6.2.

As a stepping stone to building DRP and proving its correctness, we first adapt Xie et al.'s RP scheme [38] to each transaction type: *Tame-RP* treats each object as a piece of unique rank (locks are acquired based on rank); *Wild-RP* executes transactions optimistically as in OCC, but crucially, acquires locks in rank order and then performs certification. The difference between these intermediate protocols and DRP is that they do not defer the execution of operations. Without deferring execution, Tame-RP must account for *rank mismatch*: the rank of an object might contradict a transaction's data dependencies or control flow. Consequently, Tame-RP must acquire all locks ahead of time (essentially devolving into predeclaration locking).

DRP's deferred execution mechanism is designed not only to determine a transaction's access set (and thereby make it tame), but also to avoid rank mismatch. In particular, DRP enqueues an intention (which is a piece of logic that encodes data-dependencies and is executed at commit time) into an object after acquiring a lock, and early-releases the lock even if data or control dependencies have not been met. DRP's use of intentions also allows wild transactions to pipeline their lock acquisition and early release locks while avoiding cascading aborts. Specifically, if a wild transaction needs to abort, it *nullifies* the intentions that it has enqueued without forcing other transactions to abort.

To better understand the benefits and limitations of DRP, we have built a modified version of STO [18]. Our experiments on a 64-core machine show that our modified STO achieves up to 3.6× higher throughput on the STAMP benchmark suite [8] than TL2 [11] and the existing STO [18]. Furthermore, our modifications improve the throughput of an implementation of the Silo multicore database [35] on STO by 6.6× at high contention; performance is comparable when contention is low. Compared to IC3 [37] (an in-memory multicore database that relies on static analysis to pipeline operations), our Silo implementation achieves 3.3× higher throughput at low contention, and comparable performance at high contention. Our results suggest that DRP is a good fit for STO, incurring modest operational overhead over OCC, while extracting concurrency from high contention workloads.

In summary, the contributions of this work are:

- An extension of the runtime pipelining protocol that works with tame and wild transactions without static analysis (§4). The resulting protocols guarantee strict serializability, opacity, and deadlock freedom (§4.3).
- The DRP concurrency control protocol, which uses deferred execution (via intentions) to allow tame transactions to execute while avoiding rank and dependency mismatch without having to statically parse their logic (§5). DRP also allows wild transactions to pipeline lock acquisition and avoid cascading aborts (§5.2).
- The implementation of DRP into STO (§6), and a thorough experimental evaluation of the resulting system (§7).

The rest of this paper is structured as follows. Section 2 discusses background on STO, opacity, and concurrency control protocols based on static analysis. Section 3 discusses the requirements of DRP, and Section 4 gives the design of Tame and Wild-RP. Section 5 discusses DRP and the use of intentions, and Section 6 describes the details of incorporating DRP into STO. We present our evaluation in Section 7, and survey other related work in Section 8. Finally, the Appendix presents the proofs of DRP's guarantees.

2 Background

In this work we focus on STO [18], a recent software transactional objects library that multi-threaded applications can use to simplify concurrent programming with the help of transactions. A "client" that issues transactions in the context of STO is one of the application's threads. The "server" that coordinates the processing of transactions is STO library's context (i.e., memory), since STO is just a set of functions that are invoked by threads.

The way that clients issue transactions in STO is different from a database. In a database, the server is typically a standalone service that receives requests from clients over the network in some standard format such as SQL. In STO, these transactions are specified directly in C++. Consider the following transaction in STO that transfers money from one user (`src`) to another user (`dst`), and increments a counter that tracks the number of transfers processed so far (`num`):

```
void transfer(TArray<int>& bal, TBox<int>& num,
              int src, int dst, int amt) {
  TRANSACTION {
    int bal_src = bal[src];
    int bal_dst = bal[dst];
    bal[src] = bal_src - amt;
    bal[dst] = bal_dst + amt;
    num = num + 1;
  } RETRY (true);
}
```

In this example, `TBox<int>` is a *transactional object* that acts like an `int` but keeps additional metadata (e.g., locks, version number) to ensure safe access by concurrent threads. Similarly, `TArray<int>` is a *transactional container*, which has a similar role to a table in a database. The `TRANSACTION...RETRY (true)` macro simply becomes a loop that automatically retries the transaction code until it successfully commits (in addition to generating begin and commit statements). When a thread issues operations on transactional objects or containers, each access is managed by STO's runtime, which implements OCC to provide serializability or TL2 [11] to guarantee *opacity* [14] (discussed below).

2.1 Opacity

Guerraoui and Kapałka [14] introduce opacity as a correctness condition for transactional memory. At a high level, it captures strict serializability [7, 23] with the additional requirement that transactions (even those that will eventually abort) never observe inconsistent values. The rationale is that in transactional memory and transactional objects systems, observing inconsistent state can have dramatic consequences. For example, observing inconsistent state can lead to divide-by-zero exceptions or invalid pointer dereferences, which could crash not just the process executing the transaction but the entire transactional system. Another example includes loops: if values read act as loop bounds, it is possible for a thread to read inconsistent values that lead to an infinite loop.

2.2 Related concurrency control protocols

Concurrency control protocols used in recent in-memory multi-core databases [20, 35, 40] and software transactional objects libraries like STO [18] rely on optimized variants of OCC [19]. This choice results in great performance for low contention workloads, but can lead to many aborts when contention is high. To support environments with moderate contention, Cicada [20] proposes techniques to reduce aborts (e.g., keeping multiple versions of records), but these techniques do not guarantee opacity. In contrast, our proposed protocol, DRP, maintains good performance during high contention and guarantees opacity. DRP builds on the work of *transaction chopping* [28] and *runtime pipelining* [38].

Transaction chopping. Shasha et al. [28] describe a technique to increase concurrency by chopping transactions into smaller pieces; as long as the pieces of each transaction are executed serially (though pieces of different transactions may execute concurrently), the entire execution remains serializable. The benefit is that pieces are smaller than transactions and therefore locks are held for a shorter amount of time. In order to chop a transaction, a programmer performs static analysis of the workload to construct an *SC-graph* where vertices represent transaction pieces, "sibling" edges are added between pieces of the same transaction, and "conflict" edges are added between pieces of different transactions that have conflicting operations. A valid chopping requires that there be no cycles involving both kinds of edges in the SC-graph and that only the first piece has a rollback or abort statement.

Runtime Pipelining (RP). Xie et al. [38] extend transaction chopping with an enforcement mechanism that executes at runtime and allows transactions to be chopped into even finer pieces. This runtime mechanism allows even conflicting pieces to enjoy some degree of concurrency since their execution can be pipelined, as we discuss below.

RP's fine-grained chopping stems from its static analysis algorithm that inspects clients' transactions and derives a *rank* (not necessarily unique) for each table in the database. A piece corresponds to all operations of a transaction that access tables of the same rank (one can think of pieces as being assigned this rank). Within a transaction, pieces are processed sequentially according to their rank. Across transactions, pieces acquire and release locks in a pipeline: when transaction T_i acquires a lock on some rank r to process some piece, a conflicting transaction T_j cannot acquire locks on any rank $\geq r$. When T_i's piece completes, T_i releases the lock on r and acquires a lock on r' ($r < r'$) in order to process its next piece. This allows T_j to lock any table of rank $< r'$. This pipeline ensures serializability and avoids deadlock.

3 Problem statement

Recent multi-core databases [12, 37] implement similar mechanisms to RP, and demonstrate that this improves performance at high contention. Motivated by these results, we strive to extend STO with RP, which poses three main challenges.

- **Ad hoc transactions**. STO currently supports transactions that are defined at runtime and have arbitrary control flow and access patterns. This prevents the use of RP's static analysis to determine ranks and chop transactions.
- **Arbitrary data structures**. In a database, tables have a well-known structure (row-column, etc.). This known structure allows the runtime system to acquire locks on rows or tables, and maintain metadata on the side to track ranks and *predecessors* (prior pieces that have acquired a lock of a particular rank) in RP. In STO, programmers can add arbitrary data structures to the system. As a result, these data structures must themselves expose a locking abstraction and facilitate dependency tracking since the rest of the system is unaware of (and has no control over) the data structure layout and semantics.
- **Incremental deployment**. As mentioned above, transactional objects are arbitrary data structures that have been extended with a transactional interface. To support RP, the locks of all transactional objects need to record metadata that helps transactions track dependencies and ranks. Since this is an arduous process for existing transactional objects, we want the protocol to support transactions that operate over a mixture of *RP-enabled* (objects whose locks support this metadata) and legacy (unmodified) objects.

To address these challenges, we extend RP in several ways. Section 4.1 introduces a variant of RP, called Tame-RP, that works without static analysis and supports arbitrary data structures and legacy objects, but makes the assumption that a transaction declares which objects it will access before it starts executing. Section 4.2 introduces a second variant of RP which removes the requirement that transactions pre-declare which objects they will access. This variant, called Wild-RP, is inspired by TL2 [11], and combines ideas from pessimistic and optimistic approaches to concurrency control: it uses a local workspace like optimistic approaches, and Tame-RP (which is pessimistic) to pipeline the certification process. Section 5 then shows that Tame-RP is not easily implementable in STO (due to the need to predeclare objects), and introduces the *Deferred Runtime Pipelining* (DRP) protocol. Finally, Section 6 proposes an abstraction that encapsulates the logic of DRP in a simple lock primitive that STO developers can use to build new RP-enabled data structures.

4 Tame and Wild concurrency control

In this section we describe two protocols (Tame-RP and Wild-RP) that reap the pipelining benefits of RP [38] without requiring static analysis. These protocols push RP's rank assignment to the extreme: every object (e.g. tables, rows, arbitrary objects) is assigned a unique rank *r*. In contrast to RP and related systems [12, 36], Tame-RP and Wild-RP work with transactions that are defined at runtime. A second departure from RP is that transactions in Tame-RP can hold onto locks across multiple ranks, and can release locks before acquiring other locks. We start by defining the two transaction types.[1]

Definition 1 (Tame transaction)**.** A transaction is *tame* if the items it must lock are known before the transaction begins.

Definition 2 (Wild transaction)**.** A transaction is *wild* if the items it must lock are not known before the transaction begins.

A simple example of a wild transaction is one that credits the balance of a particular user:

```
void credit(TMap<String, int>& indices, TArray<int>& bal,
            String name, int amt) {
  TRANSACTION {
    int idx = indices[name];
    bal[idx] = bal[idx] + amt;
  }
}
```

Before the above transaction executes, the system does not know exactly which entry in `bal` it will access. But if the entry could be determined (perhaps by executing the transaction optimistically, as we explain later), then we could, in a second pass, treat such transactions like tame ones. In fact, our implementation hides the tame/wild distinction from the programmer entirely as we explain in Section 6.2. The runtime system treats all transactions as tame until they require data-dependent locks.

More complex transactions might perform operations based on non-key predicates, such as update the status of all employees between the ages of 25 and 35. These might seem to require special treatment since the set of rows that will be needed is not known in advance and can change over the course of the query. However, our approach handles these too (they execute as wild).

Note that it is sufficient to lock a superset of the items that the transaction accesses, so we could make any transaction tame by locking all objects. For practical reasons, we consider a transaction tame only if we know in advance which items the transaction will access either exactly or to such a close approximation that the extra locking overhead is not an issue.

[1]Tame and wild transactions correspond to "static" and "dynamic" transactions in TL2 [11]; we avoid these terms because they already have meanings in program analysis.

4.1 RP for tame transactions

Tame-RP ensures that tame transactions are serializable and deadlock-free (so they never abort for concurrency reasons). In Tame-RP, a transaction acquires locks in ascending order of rank, so a transaction may lock an item well before it is used. As in most locking schemes, the transaction must acquire a lock on an item *x* before accessing it, and hold onto it until after completing the access.

Tame-RP uses ranks to ensure that if transaction T_i conflicts with T_j, and T_i holds a lock on some object when T_j attempts to lock it, then T_j must never acquire a lock on an object of greater rank than the maximum rank of an object accessed by T_i for the rest of T_i's execution. Further, T_j must commit after T_i. In such a case we call T_i the *predecessor* of T_j. Tame-RP in fact ensures a stronger transitive condition: if T_k acquires a lock on some conflicting object after T_j, then T_k is required to acquire locks on any future conflicting objects after T_i (even though T_i and T_k may have had no prior conflicts). T_i is transitively a predecessor of T_k.

Assumptions. Tame-RP makes several assumptions. First, it assumes that the transaction's access set is known a priori. Second, transactions can be aborted one or more times (i.e., the effect of a transaction is only externalized after it commits). Last, the locks of RP-enabled objects have the ability to track the last *predecessor*, which is the id of the last transaction (or the last set of transactions in the case of shared locks) that held the locks. Note that our current implementation uses only mutexes so there is only one predecessor. We discuss the details of RP-enabled objects in Section 6.

We also introduce a new state for locks, called *relaxed*, that acts as a breadcrumb. A relaxed lock can be acquired by a tame transaction but not by a wild transaction. Its role in conflicts and in establishing precedence will be specified in the algorithms below.

Details. For a given transaction T, let the initial value of $maxrank(T) = undefined$ and $predecessor(T) = \varnothing$. At any time *t* before transaction T commits, let $maxrank(T)$ be the highest ranked item that T has locked (with a read or write lock). $predecessor(T)$ is the set of transactions for which T had to wait directly. Tame-RP works as follows.

1. T acquires locks in ascending order of rank. A lock that is relaxed can be acquired by other tame transactions. Locks need not be acquired all at once; they can be acquired as the transaction executes (we discuss the mismatch between control flow and the ranks of objects in Section 5).
2. Before T acquires a lock on an object x, T waits so that for every $T' \in predecessor(T)$, $maxrank(T') \geq rank(x)$.
3. After T successfully acquires a lock on object x, T gets from x the identifier of the last transaction T' to hold a lock on x, and adds T' to $predecessor(T)$. If x is a

legacy object that does not support tracking predecessors, $predecessor(T)$ is unaffected by the acquisition.

4. After T acquires a lock on item x, T records the rank of that item in $maxrank(T)$ and performs its operation on x (read or write) when all data dependencies are satisfied.
5. T relaxes a lock on an item x after it no longer needs to access x. Relaxing a lock does not change $maxrank(T)$. Locks on legacy objects that do not track predecessors are never relaxed. T releases the locks on all items it holds when T commits or aborts.
6. T commits only after all transactions in $predecessor(T)$ have committed. If a transaction $T' \in predecessor(T)$ aborts and T' has written to an object that T has read, then T must abort too (otherwise T might have committed based on aborted data).
7. If transaction $T' \in predecessor(T)$ and T' commits or aborts, then remove T' from $predecessor(T)$.

Note that the above assumes the use of mutexes. If one wishes to also support shared locks, then $predecessor(T)$ must also include any transaction that currently holds a conflicting lock that is now relaxed on an object that T locks. Specifically, in Step 3, if the object x supports read locks, then T also gets from x the identifier of any transaction T' that currently holds a conflicting relaxed lock on x, and adds T' to $predecessor(T)$.

It might not be clear why T must wait for all transactions in $predecessors(T)$ to commit. We give an example to illustrate the need for this requirement. Suppose that $rank(x) < rank(y) < rank(z)$. Consider the partial interleaved execution:

$$W_3(y)\ relax_3(y)\ W_2(x)\ R_2(y)\ relax_2(x,y)\ R_1(x)$$

So $T_2 \in predecessor(T_1)$ and $T_3 \in predecessor(T_2)$. Suppose that we allow T_2 to commit before T_3 does. Doing so would allow the execution interleaving to continue as follows:

$$Commit_2()\ W_1(z)\ R_3(z)$$

But this would create the serialization graph cycle: $T_3 \rightarrow T_2 \rightarrow T_1 \rightarrow T_3$. Indeed, enforcing commit order is essential to guarantee serializability.

4.2 RP for wild transactions

One way to implement a variant of RP that supports wild transactions is to mimic Optimisitic Concurrency Control (OCC). In particular, transactions can do an optimistic pass where they access elements without locking. Then, transactions can validate their reads, but can do so by accessing elements in rank order (thereby being compatible with Tame-RP).

However, as we discuss in Section 2.1, opacity forbids any transaction T (even those that abort) from reading any transactionally inconsistent state. In the absence of user-defined aborts, tame transactions always commit, because Tame-RP is deadlock-free. Thanks to the absence of user-defined aborts and Tame-RP's pipelining, tame transactions always read from a transaction-consistent state, even if so far uncommitted. On the other hand, using OCC allows wild transactions to observe inconsistent states during the optimistic pass. For example, consider the partial history:

$$W_1(x)\ relax_1(x)\ R_3(x)\ W_2(y)\ relax_2(y)\ R_3(y) W_1(y)\ \ldots$$

Suppose that transactions T_1 and T_2 are tame, and transaction T_3 is wild and performs the reads during the first pass without acquiring any locks. This violates opacity, since T_3 observed an inconsistent state (the value of x after T_1 executes and the value of y before T_1 executes).

To avoid this, Wild-RP adapts STO's mechanism which is based on TL2 [11], and combines it with RP. In TL2, committed transactions must have monotonically increasing version numbers. This is achieved by maintaining a global version clock. When a wild transaction T_W begins, it reads the current global version, call it v, and before accessing any data item x, T_W will check certain conditions to ensure that x is transaction-consistent as of version v. If not T_W will abort.

In TL2, the version for the objects is incremented at commit time with atomic instructions after all write locks are held. In Wild-RP, a transaction T_W also increments and fetches the global version at commit time, and uses it to update the version of all objects that it modified. The incremented version is denoted $commitversion(T_W)$. Wild-RP works as follows.

1. Wild transaction T_W reads the global version number when it begins. Call that version number v.
2. T_W accesses all variables that it requires without obtaining any locks, but T_W aborts before accessing x if: (i) any object x that T_W reads has a version v' such that $v' > v$ or (ii) x's lock is a write lock that is acquired or relaxed; or (iii) there is a pending write on x.
3. T_W acquires locks in rank order on all the items in its write-set (but not its read-set), waiting until its predecessors have a higher rank. After T_W successfully acquires a lock on object x, T_W gets from x the identifier of the last transaction T' that held a lock on x. T_W adds T' to $predecessor(T_W)$. (As with Tame-RP, if the system supports read locks, then T_W gets from x the identifiers of any transaction that currently holds a conflicting lock on x, even if relaxed.)
4. T_W waits until all its predecessors have committed or aborted. T_W checks whether any object in its read-set has a version numbers greater than v or has a write lock. If an object x still has a version less than or equal to v, then x has not changed since Step 2. If x has a version number greater than v, then x may have changed, so T_W aborts.
5. T_W atomically increments and fetches the global version number, and the result is $commitversion(T_W)$.
6. T_W performs its writes and sets the version of all modified objects to $commitversion(T_W)$. As in Tame-RP, T_W can relax a lock on an item x after it no longer needs to access x. Locks on legacy objects that do not track predecessors are never relaxed. Relaxing a lock does not

change $maxrank(T_W)$.

7. T_W releases all of its locks when it commits or aborts.

Note that for wild transactions to coexist with tame transactions, Tame-RP must also atomically increment the global version number and update it at commit time (this is not required if there are only tame transactions).

4.3 Tame and Wild-RP's guarantees

We state our theorems here and highlight the key observation, but give the proofs in Appendix A.

Theorem 1. Tame-RP guarantees strict-serializability, even for transactions on legacy objects (i.e., objects without RP-enabled locks that do not track predecessors).

Our proof of Theorem 1 makes strong use of the enforcement of commit order based on predecessor relationships.

Theorem 2. Tame-RP is deadlock-free and guarantees opacity provided no tame transaction has user-defined aborts.

Our proof of Theorem 2 relies on the acyclicity in the waiting relationships that also extends to relaxed locks. Since deadlock-freedom and the no user-defined abort assumption guarantees no aborts at all, all reads will be of transaction-consistent states, thus guaranteeing opacity.

Theorem 3. A combination of Tame-RP and Wild-RP is strict serializable, deadlock-free, and guarantees opacity provided no tame or wild transaction has a user-defined abort statement.

To establish the good behavior (strict serializability and opacity) of the combination of Tame-RP and Wild-RP, we define the notion of the effective commit time of a transaction. For tame transactions, this is the commit time itself. For a wild transaction T_W that does not abort in Step 2 or Step 4, the $effectivecommit(T_W)$ is the moment after which T_W obtains all its write locks but before it checks that its read-set elements have version numbers less than $begin(T_W)$ (i.e., the moment before Step 4 begins).

Our proof of Theorem 3 shows that if T_W commits and all its operations happen exactly at $effectivecommit(T_W)$, then no reads-from or final write relationships [23] would be changed for any transaction (including T_W) from what they are in the actual execution. The reason is that for any committed wild transaction T_W: (i) no reader/writer of a data item x written by T_W will be able to access data item x in the write-set of T_W between $effectivecommit(T_W)$ and $commit(T_W)$ before T_W writes x, because T_W will hold write-locks on x as of $effectivecommit(T_W)$; (ii) no writer of data item x read by T_W can have changed x between the time T_W first read x and $effectivecommit(T_W)$, because of the version check of Wild-RP's Step 4 (otherwise the T_W would have aborted). Intuitively, the serialization order is based on *effectivecommit* time and that guarantees strict serializability. Opacity is guaranteed by the version check in Step 2.

Remark. While our algorithms guarantee opacity, it is possible to achieve better performance when opacity is not needed. Our algorithms can provide strict serializability and deadlock freedom for all transactions by changing Wild-RP to use per-object versions instead of a global version to check if objects have changed (STO also uses this approach).

5 Deferred Runtime Pipelining (DRF)

Recall from Section 4 that, in principle, tame transactions can relax their locks whenever they no longer need to access an object, opening up concurrency opportunities. In practice, however, there are two main questions that we must answer before we can incorporate tame transactions into STO: (1) how does a transaction determine which items it will access? (2) How does a transaction know when it no longer needs to access an object so that it can relax the corresponding lock? In databases, this can sometimes be accomplished by parsing the query which is written in a domain-specific language (e.g., SQL) before execution. In STO, this is more difficult since transactions are arbitrary C++ code, and the objects are not known until runtime. One possibility is to predeclare the objects and the number of times that each object is accessed:

```
void transfer(TArray<int>& bal, TBox<int>& num,
              int src, int dst, int amt) {
  TRANSACTION {
    lock_objects([&bal[src], &bal[dst], num]);
    num_accesses([2, 2, 1]);

    int bal_src = bal[src];
    int bal_dst = bal[dst];
    bal[src] = bal_src - amt;
    bal[dst] = bal_dst + amt;
    num = num + 1;
  }
}
```

Beyond the error-prone nature of this approach (since a developer might change the transaction but forget to update the preamble), it is susceptible to *rank mismatch*.

Definition 3 (Rank mismatch)**.** If transaction T access objects x, y, and z (in that order), but $rank(z) < rank(y) < rank(x)$, then T must acquire all locks before accessing x.

Concretely, if the transaction acquires locks as it executes and the rank of `bal[src]` is higher than that of `bal[dst]`, the transaction must block or abort as soon as the execution reaches `bal[src]` (but Tame-RP cannot abort). This suggests that in the worst case all locks must be acquired at the beginning (due to rank mismatch). Furthermore, there is no easy way for STO to parse the transaction's logic to learn whether rank mismatch will occur or not since the transaction is not defined in "one shot": the calling thread issues one operation at a time (as it executes the instructions that make up the body of the `TRANSACTION` macro). Consequently, Tame-RP must assume the worst case (that ranks are the inverse of the transaction's data and control flow), and devolves into predeclaration locking (which supports less concurrency than two-phase locking and defeats the purpose of pipelining).

We observe that, fundamentally, rank mismatch stems from the fact that STO (and most transactional systems) execute transactions *eagerly*. That is, read operations must actually return the current value of an object. For example, after `int bal_src = bal[src];` executes, one expects `bal_src` to contain the actual value stored at `bal[src]`. Interestingly, Faleiro et al. [13] show that one can defer the execution of a transaction's operations until later in time (which results in better cache locality, load balancing, and avoids unnecessary work in some cases), while still guaranteeing ACID semantics. Our key insight is that *deferred execution can also be used to bypass rank mismatch in tame transactions, avoid cascading aborts in wild transactions, and blur the line between the two types of transactions*.

Below we introduce DRP, a variant of RP that defers the execution of operations and combines both Tame-RP and Wild-RP into a single protocol that can be implemented in STO. Under DRP, tame transactions are a special case of wild transactions in which the read-set is empty (since all reads have been deferred), and therefore Wild-RP's Steps 2 and 4 are skipped, and the lock acquisition in Step 3 is pipelined.

5.1 Deferring execution via intentions

DRP eliminates rank mismatch by deferring the execution of the transaction to after the commit point. We implement DRP by expressing operations (read, write, etc.) on transactional objects as *intentions*. An intention is a small piece of logic that will be executed on an object, and can (optionally) return a value. An intention takes as input concrete values (e.g., a new value to write to an object) or other intentions (which express data dependency).

For example, a transaction that increments a transactional integer and writes its value to another transactional integer can be represented as follows (this example is purposely verbose; we add syntactic sugar to make this easier on programmers):

```
TBox<int> x, y;
// val is old value; what's returned is new value
Intention<int>* i1 = new Intention<int>([](int& val){
  return ++val;
});
x.defer_write(i1);
// an intention can take other intentions as input,
// here it takes i1 as input and returns i1's result
y.defer_write(new Intention<int>([&](int& val){
  return i1->result;
}), {i1});
```

Unlike in eager evaluation, `defer_write` does not acquire any locks, nor does it execute any of the transaction's logic. Instead, it adds the intention to the object's thread-local buffer, and records which object is being accessed. This is the optimistic pass of Wild-RP (§4.2), except that deferred operations do not abort or observe state. This deferred pass does not block due to rank mismatch since no locks are acquired.

When the transaction calls commit, DRP runs the same algorithm as Tame-RP (§4.1) but instead of reading or writing values to objects, it appends intentions to the objects' *intention queue* (a queue that holds all the unprocessed intentions). Specifically, the transaction starts by acquiring the lock on the object with the lowest rank (call it *x*), and appends all the intentions associated with *x* in the thread-local buffer into *x*'s intention queue. Once the transaction appends its intention into *x*'s intention queue, it can relax *x*'s lock (if *x* supports relax) so that other transactions can write their intentions to *x*'s intention queue (since intentions are ordered and the queue is protected by a lock, intention execution is serialized). An intention in *x* is executed in the future when *x* is accessed again (e.g., by an eager read), or after all of its data dependencies are satisfied (i.e., when its input intentions are evaluated). We discuss the details in Section 6.2.

Since DRP requires extending objects to support an intention queue, tame transactions can be defined only over objects that support deferred execution (though these objects need not have RP-enabled locks). If an object does not support deferred execution (i.e., it does not implement the `defer_⋆` methods), the object is accessed eagerly and the transaction is treated as a wild transaction as we discuss in Section 6.2.

5.2 Benefits of deferred execution for wild transactions

In addition to avoiding rank mismatch in tame transactions, DRP also benefits wild transactions in three ways.

Pipelining without cascading aborts. In Wild-RP, the system needs to acquire the locks of all objects in the write-set before verifying the read (Section 4.2, Step 3). In principle, Wild-RP could install the values and relax the locks (but not release them), thereby pipelining lock acquisition. However, if Wild-RP's certification (Step 4) fails the transaction would abort, which violates opacity (since the predecessors of the aborting transaction would have observed state that will be aborted). Even if opacity were not needed, a failed certification would lead to cascading aborts since all predecessors would need to abort as well.

When DRP processes wild transactions it uses the same algorithm as Wild-RP except that transactions use intentions for write operations, which prevents cascading aborts. In particular, the transaction can pipeline the work of acquiring a lock, installing the intention, and relaxing the lock for each object in its write-set. The intentions, however, do not execute right away since they all take the certification result as input. If the certification fails, all these intentions will simply be skipped (which is equivalent to aborting), without affecting other transactions' pending intentions to the same objects.

Note that the only difference between the intentions of a tame transaction and those of a wild transactions is that the certification result is ignored for tame transactions.

Fewer aborts due to conflicts. DRP also avoids exposing inconsistent system state before a transaction commits because it exposes no state. This reduces the incidence of conflicts, and consequently of aborts. Of course, not all operations in a wild transaction can be converted into our style of deferred execution. For example, an object might not support deferred evaluation, or the transaction might need to read an object before deciding what to do next. However, DRP ensures that only those conflicting operations that execute eagerly can result in aborts.

Better temporal locality. With DRP, many intentions that have similar dependencies can be executed together, benefiting from temporal locality since the data is likely to be in the cache. In contrast, eager evaluation accesses data during the optimistic read and at certification time.

6 Implementing DRP in STO

As we discuss in Sections 4 and 5, DRP requires objects to be extended with additional functionality to track predecessors and queue intentions. To incorporate these extensions, we make two major changes to STO. First, we extend the interface of `TObject`, which is the abstract class inherited by all transactional objects, to support locks that can be relaxed and predecessor tracking. Second, we introduce new interfaces to support deferred execution; these interfaces vary depending on the transactional object (vector, list, map, etc.).

6.1 Changes to `TObject`

We require that all new objects implement the following interface (this is done by overriding the corresponding virtual functions of the `TObject` class). The first four operations are automatically supported by existing legacy objects in STO.

- **Lock**. Attempts to acquire the object's lock. This operation returns the id of the last transaction that acquired the lock. If the object supports read/write locks, this operation also returns the ids of threads that currently hold the lock in a relaxed state. A legacy lock returns only whether it was successfully acquired or not.
- **Unlock**. Releases the lock on the object.
- **Install**. Installs a value to the object.
- **Rank** (optional). Returns a non-zero rank for the object. By default, this is the object's memory address but can be tuned. Legacy objects return 0.
- **Relax** (optional). This operation is used in Step 5 of Tame-RP and Step 6 of Wild-RP. If the object does not support relax, this function simply returns `false`.
- **QueueIntention** (optional). Enqueues an intention into the object's execution queue. If the object does not implement this, the function returns `false`, which causes the engine to hold on to the lock until enough inputs are ready to eagerly execute the intention and install its value.

Beyond per-object information, which can be obtained through the above interface, we introduce a data structure that is shared among all threads and stores the *maxrank* of each transaction (thread). Transactions update their *maxrank* in this data structure whenever they acquire a new lock. This allows transactions to check the rank of their predecessors to determine whether they can acquire the next lock or not.

The current implementation of STO does not support read and write locks; our current implementation of RP-enabled locks does not include this functionality either. In particular, locks in STO spinlock using atomic operations on a 64-bit integer (for ease of reference call it the "lock-integer"). STO uses one bit of the lock-integer to represent the state of the lock (acquired or free). STO uses another 5 bits to identify which thread is holding the lock (consequently the current implementation of STO supports up to 32 threads). The remaining bits of the lock-integer encode the version used by the OCC and TL2 algorithms (and DRP).

We repurpose one of the bits originally allocated to the version field in the lock-integer to represent the "relaxed" state. A thread relaxes the lock by setting this bit. If the relaxed bit is set, other tame transactions can acquire the lock by clearing this bit and resetting the last holder field with their id (wild transactions only look at the lock bit). At release time, the thread checks if it is the last holder of the lock (using the corresponding 5 bits of the lock-integer). By transitivity, if a tame transaction is the last holder and can commit (meaning all of its predecessors have already committed), then no other thread can have the lock in either the acquired or relaxed state (otherwise their id would be in the last holder field of the lock-integer). The thread then clears both the relaxed bit and the lock bit with a compare-and-swap operation.

Opacity. Opacity requires changes in each data structure to ensure that transactions observe the same global version. In STO, opacity can be turned on or off through compilation flags. This is achieved by an abstraction of `TVersion` and `TOpaqueVersion`. The latter obeys the TL2 protocol (and DRP). A macro controls which one is used by each data structure. If opacity is turned on, all transactions (both tame and wild) will increment the global version before they commit. The version will be treated as a special input to all intentions of the transaction. This does not block the pipelining, and ensures that no inconsistent states are exposed to ongoing wild transactions. If opacity is turned off, an executing intention will increment only the object-local version.

6.2 Support for deferred execution

In addition to the above interfaces inherited from `TObject` which are used by the transaction engine to commit the transactions, each data structure also needs to provide interfaces to buffer the read/write requests (intentions) at transaction parse time. In Section 5.1 we foreshadowed that a `TBox` has the interface `defer_write`. More complex data structures can provide other interfaces that lazily access the structure. For example, a transactional list should provide an interface to lazily index an element in the list. We have modified most of STO's transactional objects to have this deferred interface. To demonstrate their usage, below we give the deferred analog of the `transfer` example in Section 2.

```
void transfer(TArray<int>& bal, TBox<int>& num,
              int src, int dst, int amt) {
  TRANSACTION {
    // the next two lines explicitly use deferred
    // interfaces to buffer intentions locally
    Intention<int>* bal_src = bal.defer_at(src);
    bal.defer_update(src, new Intention<int>([&](int& val){
      return bal_src->result - amt;
    }, {bal_src}));
    // the next three lines use syntactic sugar based
    // on C++'s operator overloading and implicit
    // type conversion to achieve the same effects
    auto bal_dst = bal[dst];
    bal[dst] =  bal_dst + amt;
    num += 1;
  }
}
```

The example shows the `defer_at` and `defer_update` interfaces of a list that supports deferred execution. `defer_at` returns an intention that will return the actual element at the appropriate index when it is evaluated. `defer_update` updates the list with another intention. Both intentions will be buffered locally until after the transaction commits. The above example also has a few lines that look like eagerly evaluated code. In fact, these lines are also lazily evaluated but exploit C++'s operator overloading and support for implicit type conversion to make writing tame transactions easier.

An object can still provide interfaces for eager evaluation. A transactional array can have both `defer_at` and a normal `at` function. If an eager read interface is called, the data structure will add the read result to a thread-local read-set (supported by STO). At commit time, if the read-set is empty, then the system will treat the transaction as tame; otherwise the transaction is treated as wild. Consequently, distinguishing between tame and wild transactions is done automatically.

In fact, lazy and eager evaluation can share the same interface. An example of this is the `[]` operator in the above code snippet. It returns an `Intention<int>` object by default. But the returned object can be casted into an integer due to C++'s user-defined type conversion. In such a case, the conversion function evaluates the intention eagerly by reading the integer following Wild-RP's optimistic read. This hybrid style of lazy and eager evaluation simplifies writing transactions, and supports our goal of incremental deployability (§3). Furthermore, if a programmer accidentally triggers eager evaluation on an intention, the system can still function, though with lower performance due to the possibility of aborts.

Resolving intentions. An intention can be executed at any time when it is needed (e.g., in response to an eager read). We have chosen a simple strategy in our current implementation: a transaction waits until all its intentions finish executing before returning success. As we have shown in our examples, each intention maintains a set of its data dependencies. An intention will evaluate its dependencies (if they have not yet been executed), before it executes. An atomic flag within each intention is used to detect whether the intention has been evaluated. After an intention is evaluated, its results can be accessed via its `result` member variable.

More complex strategies could improve performance. For example, a thread could batch transactions and wait for their execution results. We consider this to be future work.

6.3 Intra-object concurrency

For a complex data structure (e.g., a tree), concurrency inside the data structure is important for overall system performance. The interfaces of `TObject` (Lock, Unlock, Install, etc.) may limit the concurrency that the data structure can achieve. STO solves this issue by having concurrent data structures define a finer grained unit of concurrency control. In particular, instead of locking the entire object (e.g., the entire tree), or serializing all intentions to the object, the data structure can identify a smaller unit on which the transaction engine can operate.

This finer-grained concurrency control is supported at the interface level by adding an extra argument `TransItem` to all of the interfaces we discussed in Section 6.1: from Lock to QueueIntention. A `TransItem` is created when the data structure is accessed (such as when `defer_at` is called). Each access creates a `TransItem` and appends it to a thread-local buffer. This item records the intention as well as the location that is being accessed in the data structure. At commit time, the transaction engine scans all the buffered `TransItem`s; for each item it follows DRP's protocol of locking, installing, and relaxing. `TransItem` is used in STO to record the read and write set, and we extend it to support deferred execution.

Rank tuning. In many cases a developer might wish to set the rank of an object for a given application in order to improve the pipelining (since intentions are resolved more effectively). We support this by treating the 64-bit integer representing the rank as a tuple of the form (`custom_rank, obj_rank`). The `custom_rank` is 16-bits and is set by the developer; it defaults to `0xFFFF` and can be changed with an object's `set_rank()` method. The `obj_rank` is 48-bits and corresponds to the 48 meaningful bits of the object's x86-64 virtual memory address. Comparing ranks is done using `custom_rank`, and ties are broken with `obj_rank`. `TransItem`s inherit the `custom_rank` of their container and have their own `obj_rank`.

7 Evaluation

This section studies the overhead introduced by DRP's mechanisms, and its performance on contended workloads. We use a combination of microbenchmarks and representative applications to answer five questions:

- What is the performance and memory overhead of managing transactions' local buffers, RP-enabled locks, and dependency tracking?
- How does DRP perform under low contention?
- How does DRP perform under high contention?
- How is the performance of DRP impacted by the fraction of tame transactions?
- What are the benefits of DRP over a simpler protocol sthat also defers the execution of operations?

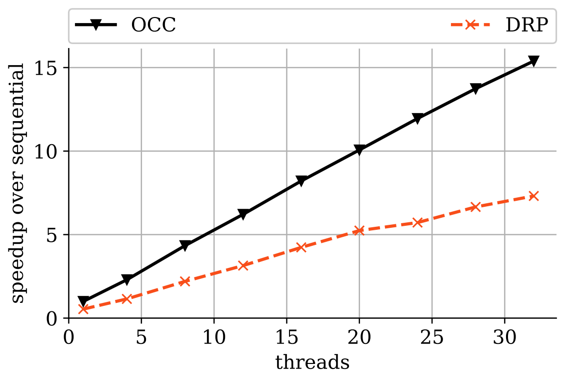


FIGURE 1—Simple microbenchmark with no contention to measure the cost of DRP's lock and metadata management (see text for details). The y-axis shows the speedup over running the operations one by one on a single thread (i.e., no transaction).

Experimental setup and baselines. We run our experiments on a 60-core Dell PowerEdge R920 with four Intel Xeon E7-4870 v2 processors (2.30 GHz, 30 MB L3 cache) and 256 GB of memory, running Ubuntu 16.04.1 (Linux kernel 4.15.0-39). Since STO supports up to 32 threads (§6.1), we use up to 32 cores in our experiments.

We run the following baselines to compare with DRP:

1. STO with OCC and TL2. These two protocols are part of STO's codebase [3]. STO extends standard OCC with an optimization that improves performance under contended workload. In particular, STO's OCC aborts transactions early if it encounters conflicts (as opposed to during a certification phase). An active transaction leaves marks on data items, and later transactions accessing the same data items will see the marks and terminate early due to conflicts. In our tests, we find that this optimized OCC achieves performance similar to 2PL at high contention.
2. STM-TL2 [11] (only for the STAMP benchmarks). We use STAMP's implementation of TL2; the code we use is available in STO's repository [1].
3. IC3 [37] (only for the TPC-C benchmark). IC3 extends Silo [35] with static analysis to chop transactions, and leverages runtime pipelining to achieve good performance at high contention. Note that this is not a fair comparison because IC3 assumes full knowledge of the workload whereas DRP does not. Nevertheless, we compare to IC3 to demonstrate that DRP is competitive with systems that make use of static analysis.
4. STO with Deferred 2PL (only for factor analysis). This is a version of STO where we implement DRP's deferred execution, but not the pipelined lock acquisition; instead, it uses two-phase locking (with a deterministic order) to acquire locks.

7.1 Lock management overhead

This section discusses the overhead introduced by DRP's lock management, dependency tracking, and traversal of objects' intention queues over the existing STO. We write a simple microbenchmark where transactions perform little computation (index an array of 1 million integers and increment 50 entries) and there is no contention (each thread accesses a shard of the array). This scenario is ideal for optimistic protocols since transactions never abort, and highlights the performance penalty that DRP's bookkeeping introduces in cases where tracking and locking is unnecessary.

Figure 1 depicts the results. DRP consistently achieves 47–53% of OCC's throughout. The majority of the overhead comes from maintaining dependencies and ranks; rank management is done through a data structure shared across threads whose accesses lead to cache line bounces. In terms of memory consumption, DRP uses 2.1× more memory than OCC (50 MB vs 24 MB). This is consistent with DRP's new data structures and the added fields for the 1 million locks. For comparison, the sequential implementation uses 10 MB.

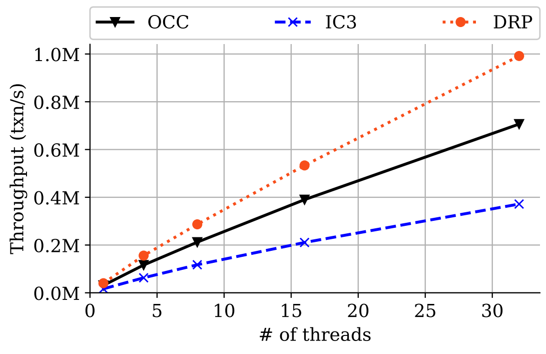
As we show next, the benefits of DRP's lack of aborts for tame transactions and additional concurrency outweigh its bookkeeping overheads, even under light contention.

7.2 TPC-C benchmark

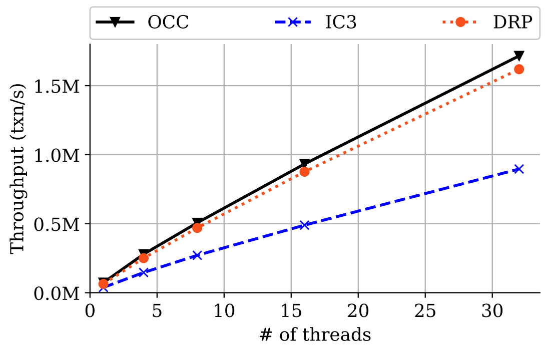
Workload and setup. We use the same TPC-C workload used in STO's experiments. TPC-C simulates an e-commerce service where products are stored in *warehouses*. Each warehouse supplies 10 *districts*, which sell products to customers and request products from warehouses. There are 5 possible types of transactions: new order, payment, delivery, stock level, and order status. Each thread picks a random warehouse and district (served by the warehouse) and performs a transaction (e.g., new order, which simulates a new purchase).

We make 3 out of the 5 transaction types tame (new order, payment, and delivery) by modifying STO's TPC-C workload code to use deferred operators. This required fewer than 500 lines of code. Making the remaining two transactions (stock level and order status) tame would require either locking the entire database, or tuning ranks in such a way that they are compatible with transactions' data and control flow. Indeed, IC3 performs a somewhat similar kind of tuning during its static analysis pass (it creates pieces and assigns ranks to be compatible with transactions data dependencies and control flow). Since we believe that this type of intrusive workload-dependent tuning would weaken our argument that static analysis is not needed for good performance, we run these two types of transactions as eagerly evaluated wild transactions without any modifications.

There are two ways in which we can control the level of contention in TPC-C: (1) modify the workload itself by biasing the distribution with which warehouse and districts are chosen, or by increasing the number of operations per transaction; (2) modify the number of warehouses and threads. We take the second approach to retain the standard TPC-C benchmark. Fewer warehouses increases the probability that two threads will pick the same warehouse and issue

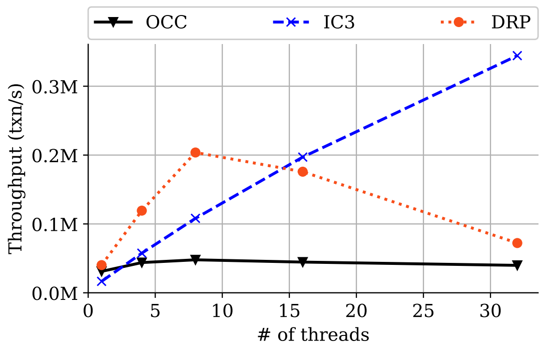


(a) Mixed workload

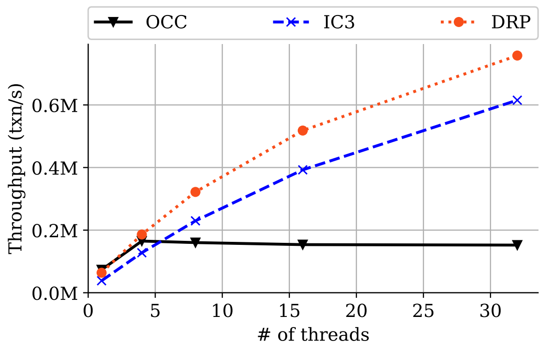


(b) 100% new-order transactions

FIGURE 2—TPC-C with a constant warehouse-to-thread ratio of 1. The level of contention is relatively low. As required by TPC-C, throughput measures only completed new-order transactions.



(a) Mixed workload



(b) 100% new-order transactions

FIGURE 3—TPC-C with a single warehouse and varying threads. Contention increases with the number of threads (to the right). Anything beyond 16 threads is very high contention.

conflicting transactions; more threads increases the load and the probability of conflicts.

We evaluate DRP's performance under three scenarios:

- *constant ratio*: warehouse-to-thread ratio is 1.
- *constant warehouses*: 1 warehouse, varying threads.
- *constant threads*: 32 threads, varying warehouses.

For each scenario we run a *mixed workload* (all five TPC-C transactions in their specified ratios [32]) and a workload of only new-order transactions. As stated in the TPC-C specification, we measure throughput as the number of completed new-order transactions per second. We run scenarios five times and report the mean (standard deviations are $< 5\%$ of the means). In addition to the above scenarios, we also evaluate the impact of tame transactions on overall performance by varying the fraction of transactions that are tame.

Constant ratio. Figure 2a depicts the throughput of Silo built on the existing STO [2] (labeled as "OCC") and our modified version (labeled as "DRP"), as well as IC3 under a constant ratio of warehouses to threads. This is the "standard" TPC-C benchmark, and has low contention. DRP's throughput on the mixed workload is similar to OCC's, and 3.3× higher than IC3. This is due to much fewer aborts in DRP when compared to OCC, and the finer grained pipelining when compared to IC3. In particular, IC3 splits transactions into relatively large pieces; pieces hold onto the locks of the tables they access until they complete. DRP, by contrast, acquires locks on individual rows and relaxes locks as early as possible.

We also show the case for the 100% new-order workload (Figure 2b), where the level of contention is even lower. Two transactions rarely access the same warehouse; even when they do, the probability that they access the same item is very low (transactions access on average 10 out of 100K items).

As we discuss in Section 7.1, DRP has relatively high bookkeeping costs; at low contention the benefit of DRP is only enough to cover its own overhead. DRP's performance under this regime is comparable to that of OCC.

Constant warehouses. Figure 3 shows a scenario where contention increases with the number of threads (since the warehouse-to-thread ratio decreases). In this scenario, OCC's performance plateaus after 4 cores for both the mixed workload and the 100% new-order. Additional threads lead to higher contention which leads to more aborts. Actually, if we disable the optimization of the early aborts—which makes it a more common version of OCC—the throughput will drop to almost zero after 16 threads (not shown in the figures). DRP's pipelining performs well at this higher level of contention, achieving 6.6× higher throughput than OCC.

IC3 performs better than DRP in the mixed workload af-

	1 thread			4 threads			8 threads			16 threads			32 threads		
	50%	90%	99%	50%	90%	99%	50%	90%	99%	50%	90%	99%	50%	90%	99%
OCC	14	22	26	22	68	109	26	104	193	28	182	378	26	472	845
IC3	28	42	52	33	49	65	35	53	68	39	60	79	44	72	94
DRP	16	25	29	21	34	51	23	39	54	25	43	61	26	42	52

FIGURE 4—Commit latency (in microseconds) of TPC-C new-order transactions corresponding to the experiment in Figure 3a (mixed workload with varying threads and 1 warehouse). We report the 50/90/99-th percentile latencies (lower numbers are better).

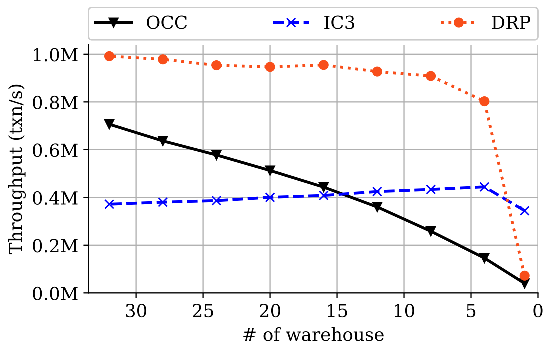
ter 16 threads for several reasons. First, recall that two of DRP's transaction types are wild. Transactions of these types will likely abort (due to the high contention) and retry over and over—preventing the corresponding threads from issuing other transactions in the meantime (OCC has the same issue but with all transaction types). Second, unlike DRP, IC3 statically analyses the workload and optimizes transactions' data flow. The benefit of this optimization grows as contention increases, and more than makes up for the limitations of coarse-grained chopping when contention is very high. In all other cases, DRP is better (up to 2×) than IC3.

We also evaluate the commit latency of (new-order) transactions for all three schemes in this setup. Unlike throughput, which is impacted by wild transactions that abort because threads issue one transaction at a time in a closed loop, the latency of tame transactions is unaffected. We present the 50/90/99-th percentile lantecy results in Figure 4.

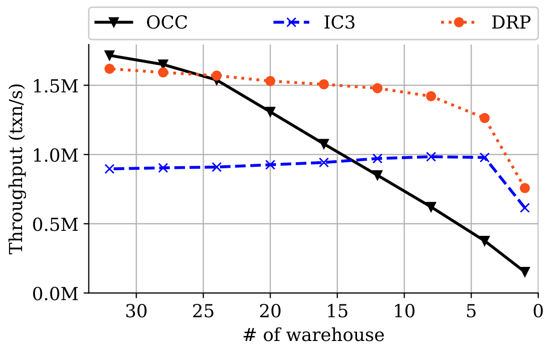
With 1 thread, OCC has the lowest latency because it has the lowest overhead and there is never any contention. The commit latency with 1 thread under DRP is lower than IC3 because transactions in IC3 are split into dozens of smaller transactions, each of which contains `TX_BEGIN` and `TX_END` statements that introduce overhead. As contention increases (more threads), the latency of OCC rises to account for aborts and repeated retries; at 32 threads, the 99-percentile latency increases to 845μs. In contrast, since new-order transactions in DRP are tame and never abort, any additional latency stems from waiting in the pipeline. At 32 threads, the 99-percentile latency of DRP is an order of magnitude lower than OCC.

Constant threads. Finally, we show the effect of contention on each system by holding the number of resources (threads) constant and varying the amount of choice (number of warehouses). Figure 5 depicts the result. As the amount of contention increases (from left to right in the figure), all systems (aside from IC3) experience performance degradation. The primary difference is the extent of this degradation. IC3's performance is stable due to its initially low starting point and the fact that its pipelining exposes roughly the same amount of concurrency across different contention levels.

Factor analysis. In order to understand the performance gains that DRP gets from tame transactions, we run an experiment where the workload consists of new-order transactions on a setup with 1 warehouse and 32 worker threads. The experiment includes a mix of unchanged new-order transactions



(a) Mixed workload



(b) 100% new-order transactions

FIGURE 5—TPC-C with 32 threads and varying warehouses. Contention increases as the amount of choice decreases (to the right).

previously implemented in STO (running as wild transactions), and our modified version (running as tame transactions). We vary the fraction of new order transactions that are tame and we report the results in Figure 6.

As we increase the ratio of wild transactions in the workload mix, the performance drop is evident. Having a mere 10% of transactions be wild is sufficient to cut the system's performance in half (258K vs 534K transactions per second). The reason is that wild transactions often abort and retry, wasting work. Furthermore, since tame transactions never abort, there could be situations in high contention regimes where wild transactions can starve.

To understand the gains that DRP gets from pipelining—as opposed to the benefits of deferred execution which have already been identified in prior works [9, 13]—we run an experiment where the workload consists of new-order trans-

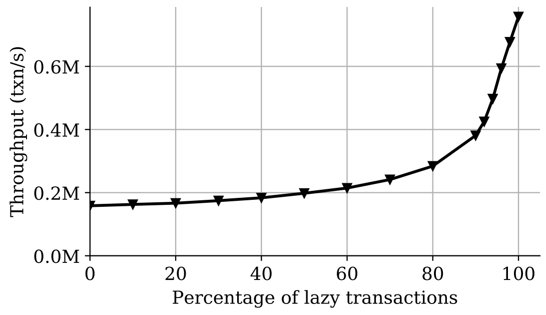


FIGURE 6—TPC-C (new-order only) with 32 threads and 1 warehouse (high contention), varying the percentage of tame transactions.

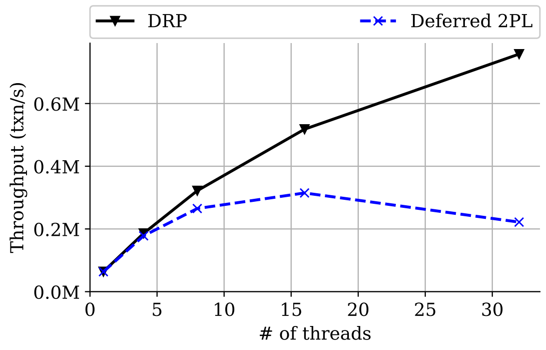


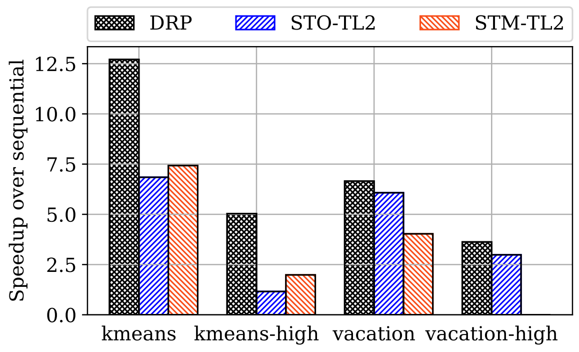
FIGURE 7—TPC-C new-order transactions with 1 warehouse and varying threads. Contention increases with more threads.

actions on a setup with 1 warehouse and varying threads (contention increases as the number of threads increases). As a baseline we use a concurrency control protocol that defers the execution of operations as in DRP, but acquires locks using two-phase locking (2PL). The locks are acquired following a pre-defined order to avoid the overhead of deadlock detection mechanisms.

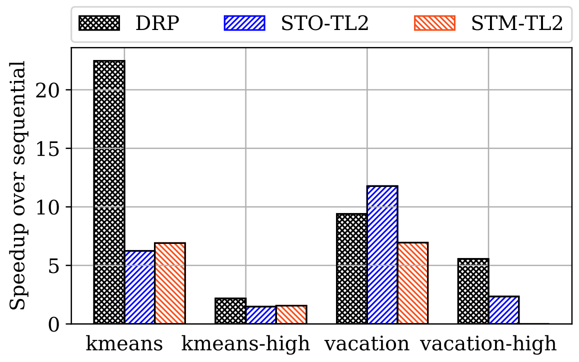
As shown in Figure 7, when the contention level is low (fewer than 4 threads) the deferred 2PL protocol has similar performance to DRP; at medium and high contention, DRP performs significantly better (over 3× higher throughput at very high contention). The main advantage of DRP is that it can pipeline the accesses to “hot” items, whereas 2PL’s transactions sit busy waiting until other transactions complete to acquire the locks for these items.

7.3 STAMP benchmark

STO, as an STM-like system, supports general applications beyond databases. STAMP proposes 8 representative applications. We port 2 of them (*kmeans* and *vacation*) to use transactional objects that support deferred execution so they can be tame. We chose these two applications because they were the easiest to port and they did not require rank tuning. For completeness, we also experiment with the unmodified version of the other four STAMP applications implemented by STO (*labyrinth*, *genome*, *bayes* and *intruder*) and observe



(a) STAMP applications on 16 threads



(b) STAMP applications on 32 threads

kmeans	-m160 -n160 -t0.001 -i inputs/random-n262144-d32-c16.txt
kmeans-hi	-m40 -n40 -t0.00001 -i inputs/random-n262144-d32-c16.txt
vacation	-n2 -q90 -u98 -r1048576 -t4194304
vacation-hi	-n4 -q1 -u90 -r184857 -t12194304

(c) Parameters used for STAMP applications

FIGURE 8—Speedup over a sequential implementation of STAMP applications. Bars depict the mean speedup across 5 trials. The postfix “hi” signifies a higher contention setting of the application. For STO-TL2, we use predicates on vacation and vacation-hi to improve its performance [18]. STM-TL2 did not finish within 20 minutes on vacation-hi.

that these achieve the same performance as the existing STO. In other words, since these unmodified applications run using DRP’s eagerly evaluated wild transactions, we are able to confirm that DRP does not add overhead over TL2 (which is used in STO). We do not discuss these further.

Figure 8 depicts the results of the two ported applications at 16 and 32 cores. We report the speedup over running these applications on a sequential implementation that does not use transactions. DRP’s memory consumption on all STAMP applications is less than 5% higher than STO’s.

STO’s performance advantage over STM-TL2 comes from its awareness of data structure semantics. As a result, it can avoid locking at word granularity. Since DRP runs on top of STO, it inherits the same benefits. DRP is comparable or outperforms STO-TL2 on every application, sometimes by up to 3.6×. As in the TPC-C benchmark, the extent of the benefit depends on the level of contention. At low contention (e.g., kmeans at 16 cores or vacation), DRP is comparable to STO-TL2; at medium or high contention (e.g., kmeans-hi, vacation-hi), DRP's gains are significant ($> 2\times$) due to the absence of aborts.

8 Related work

We discuss closely related work to DRP in Section 2. We now discuss work that relates more generally.

Taming transactions. Many DBMSes [10, 22, 31, 35, 40, 43] restrict their focus to "one-shot" or "static" transactions, where the transaction's logic or its read/write set are specified a priori. Silo [35] observes that a benefit of one-shot transactions is that since they are specified in one shot, they avoid potential stalls by slow clients (e.g., when a client gets the result of a read and thinks for a while before issuing the next operation). DRP also leverages one-shot transactions. A key difference is our motivation: these works aim to cut transactions' latency to reduce the probability that new operations will arrive and cause conflicts and therefore aborts, since their algorithms are based on optimistic concurrency control schemes. In contrast, DRP acquires transactions' read/write sets through the use of intentions, and uses them to pipeline lock acquisition.

DRP's mechanisms are similar to those of deterministic database systems [12, 13, 27, 33, 34] in two ways. First, DRP avoids deadlock by acquiring locks in a prescribed order. Second, DRP leverages knowledge of transactions' logic (e.g., their read/write set) to induce a serial execution. Deterministic database protocols also ensure that any interleaving results in a precise serial execution. One distinction is that while DRP leverages transactions' logic, it does not constrain their interleaving (which exposes more concurrency) because it does not need to guarantee a particular serial outcome.

DRP is also similar to Diamond's DOCC protocol [42], DASTM [26]. Like those schemes, DRP collects and manages runtime information to improve performance: Diamond and DASTM track dependencies to prevent superfluous aborts in optimistic concurrency control, while DRP leverages transaction logic to early release ("relax") locks as soon as possible.

Deferred execution. DRP's use of deferred execution is inspired by prior works [9, 13, 25, 34]. A key difference is that we use lazy evaluation to avoid the rank mismatch problem inherent with tame transactions (§5), and to allow wild transactions to pipeline lock acquisition during certification without cascading aborts (§5.2). To our knowledge, both of these applications are novel. In contrast, these systems use lazy evaluation for a variety of other purposes. For example, lazy transactions [13] and Sloth [9] use lazy evaluation to batch queries together to reduce the number of round trips between clients and database servers, exploit temporal locality, and achieve better load balancing. Calvin [34] and QueCC [25] capture the transactions' control and data flow, create an execution plan, and defer their execution via pushing the operations into per-partion/object queues.

9 Discussion and summary

We now revisit our problem statement in Section 3. Our goal was to incorporate the runtime pipelining protocol into in-memory transactional systems like STO in hopes of reaping the benefits of pipelining for medium to high contention workloads, but we faced several challenges (and opportunities) along the way. For instance, wanting to support arbitrary transactions defined at runtime (i.e., wild transactions) led us to develop two new protocols that are heavily influenced by runtime pipelining's design. Furthermore, being unable to preprocess or parse transactions (since they are not expressed in one-shot) led us to a new application of deferred execution: allowing transactions to prove that they are tame at commit time (by having an empty read set) or be treated as wild transactions instead (§5.1). In particular, deferred execution combats the rank mismatch problem whereby the rank associated with objects contradicts control flow or data dependencies. In turn, this allowed us to assign arbitrary ranks to objects avoiding the need for static analysis.

Another one of our goals was incremental deployment. We stay true to this goal in several ways. First, DRP supports existing objects that do not have RP-enabled locks that can set the relax bit. Second, DRP supports both wild and tame transactions simultaneously, and programmers need not specify the type of a transaction; this is inferred automatically (§6.2). Third, our implementation supports existing unmodified transactions as wild transactions that execute operations eagerly. Last, our implementation allows wild transactions to have some of the operations be lazily evaluated, which has several benefits (§5.2).

We believe that DRP is a good addition to STO and other in-memory multi-core transactional systems that currently rely on OCC, as it improves performance for medium to high contention workloads, and remains comparable at low contention. While DRP is not as efficient as IC3 at very high contention (see for example Figure 3a) since DRP does not optimize control flow for the particular workload using static analysis, a similar effect can be achieved with rank tuning (§6.3). In particular, if one can ensure that the ranks of objects that are read are lower than those of corresponding write operations, rank mismatch can be reduced or in some cases eliminated. An interesting avenue for future work is to tune ranks dynamically as transactions execute in the background. This could be done either with a greedy search or with reinforcement learning.

A Proofs of Tame-RP and Wild-RP

This is an extended version of [21]. This section contains the proofs of the Tame-RP and Wild-RP algorithms.

We assume that algorithms proceed as before except that legacy data items do not maintain a precedence graph. Instead, a transaction T that accesses one or more legacy data items hold locks on those data items until T commits. We call this Tame with Legacy. If T_2 waits-for T_1 on a legacy item, we say $T_1 \in \mathit{waitlegacy}(T_2)$. Call the union of the waitlegacy and precedence graphs the waitpredecessor graph.

Theorem 1. Tame-RP guarantees strict-serializability, even for transactions on legacy objects (i.e., ones without RP-enabled locks that do not track predecessors).

Proof. Observe that if T_1 precedes and conflicts with T_2 on some data item, then there is an edge in the serializability graph $T_1 \rightarrow T_2$ and, unless T_1 has already committed, $T_1 \in \mathit{predecessor}(T_2)$ (because of acquired or relaxed locks) or $T_1 \in \mathit{waitlegacy}(T_2)$ (i.e., $T_1 \in \mathit{waitpredecessor}(T_2)$ in the case of legacy objects). Each edge in the waitpredecessor graph implies a commit ordering: $T_1 \in \mathit{predecessor}(T_2)$ implies that T_1 must commit before T_2 by the Tame-RP algorithm; $T_1 \in \mathit{waitlegacy}(T_2)$ implies that T_1 must commit before T_2 because (for legacy objects) T_1 will not release its lock before it commits.[2]

Suppose that there is a cycle in the serialization graph of committed transactions (aborted transactions whose writes no transaction reads do not matter): $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ then by construction T_1 must commit before T_2 which must commit before $\ldots T_1$. This yields a contradiction.

Our proof of strict serializability is also by contradiction. Suppose T_1 commits before T_2 begins, but every equivalent serial execution places T_2 before T_1. That implies that there is a path in the serialization graph from T_2 to T_1: $T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_m \rightarrow T_1$. By construction that would imply that T_2 must commit before T_3 which must commit before $\ldots T_1$, which contradicts that T_1 commits before T_2 begins. □

Lemma 1. The waitpredecessor graph engendered by the Tame-RP algorithm is acyclic.

Proof. If $T_1 \rightarrow T_2$ in the waitpredecessor graph, then $\mathit{maxrank}(T_1) \geq \mathit{maxrank}(T_2)$ or T_1 acquired a lock on the item which is $\mathit{maxrank}(T_2)$ before T_2 did. If $T_1 \rightarrow T_2$ is in the waitlegacy graph, then T_1 acquired a lock on the item which is $\mathit{maxrank}(T_2)$ before T_2 did. So, for either predecessor or waitlegacy edges, T_1 acquires a lock on an item x_1 before T_2 acquires a lock on T_2's highest ranking item x_2 and $\mathit{rank}(x_1) \geq \mathit{rank}(x_2)$. This is an acyclic relationship. □

Lemma 2. For transactions obeying the Tame-RP algorithm, if T_j waits for T_i, then $T_i \rightarrow T_j$ in the waitpredecessor graph. Therefore the waits-for relationship is acyclic.

Proof. Suppose T_j is waiting for a lock held by T_i. There are two cases: (i) If T_i and T_j have never directly conflicted before, then $T_i \rightarrow T_j$ is added to the predecessor or waitlegacy graph by construction. Thus, the waits-for edge would belong to the waitpredecessor graph. (ii) If T_i and T_j have directly conflicted before, then either $T_j \rightarrow T_i$ or $T_i \rightarrow T_j$ in the predecessor graph. But the first is impossible since that would mean that T_i had locked an item with rank higher than T_j which violates the construction. So, $T_i \rightarrow T_j$. Thus, every edge in the waits-for graph corresponds to an edge in the waitprecedence graph and is thus a sub-graph of the waitprecedence graph. □

[2]If there are no shared locks, Tame-RP can place into $\mathit{predecessor}(T)$ only the last uncommitted (if any) transaction T' that held a mutex lock on x. The relationship between conflict order and commit order still holds because the transitive closure of the predecessor relationship enforces a commit order.

Theorem 2. Tame-RP is deadlock-free and guarantees opacity provided no tame transaction has user-defined aborts.

Proof. By Lemma 2, the waits-for graph is acyclic. There can be no cycles among waits-for edges and consequently no deadlock. Because Tame-RP also guarantees strict serializability (by Theorem 1), every read of any tame transaction will be of a transaction consistent state (viz. the state in the strict serializable order), hence Tame-RP guarantees opacity. □

To establish the good behavior of Wild-RP, we need to define the notion of the effective commit time of a transaction. For tame transactions, that is the commit time itself. But for a wild transaction T_W, $\mathit{effectivecommit}(T_W)$ is the moment after which T_W obtains all its write locks but before its check that the read-set of T_W still have version numbers less than $\mathit{begin}(T)$ (just before Step 4).

Theorem 3. A combination of Tame-RP and Wild-RP is strict serializable, deadlock-free, and guarantees opacity provided no tame or wild transaction has a user-defined abort statement.

Proof. In a pure tame transaction setting, we know the theorem holds by Theorem 2.

Consider a wild transaction T_W that commits. First, note that all reads of T_W are of a transaction consistent state. Wild-RP achieves this by ensuring that T_W reads data items based on their values when T_W begins (that time is denoted v in the algorithm). This is achieved by the version check (viz. make sure that no item in the read-set of T_W has a version greater than that of v) in Step 2 and Step 4.

Second, deadlock-freedom follows because, while T_W can abort if it detects locks in Step 2 or changes in data in its Step 4, its write-lock acquisition follows the Tame-RP protocol, so the conditions for Theorem 2 still hold.

To establish the strict serializability of wild and tame transactions, we need to make use of the notion of effective commit time. Intuitively, if T_1 precedes and conflicts with T_2 on some data item, then $\mathit{effectivecommit}(T_1)$ precedes $\mathit{effectivecommit}(T_2)$. As in Theorem 1, a cycle in the serialization graph would then imply $\mathit{effectivecommit}(T_1)$ precedes $\mathit{effectivecommit}(T_1)$, which is a contradiction.

So, we must prove that a conflict edge $T_1 \rightarrow T_2$ implies that $\mathit{effectivecommit}(T_1)$ precedes $\mathit{effectivecommit}(T_2)$. If both transactions are tame, we proved that in Theorem 1.

So, let at least one transaction T_W be wild and another transaction T be wild or tame. Assume further that both commit (otherwise they leave no effect on the data).

Case 1: $T \rightarrow T_W$**.**

(i) T writes x and later T_W writes x. So, by the precedence rules of wild transactions, *effectivecommit*(T) (whether or not T is wild) will occur before T_W obtains all its write locks on the data items T_W needs, which precedes *effectivecommit*(T_W).

(ii) T is tame and reads x and later T_W writes x. T retains at least a relaxed lock, so T_W cannot obtain a write lock on x before T commits. So *effectivecommit*(T) precedes *effectivecommit*(T_W).

(iii) T is a wild transaction and reads x before T_W writes x. Because T reads x before *effectivecommit*(T) and checks that the version of x has not changed and that there is no write lock on x in T_W's Step 4, Step 4 of T must precede *effectivecommit*(T_W). Because *effectivecommit*(T) precedes Step 4 of T, we have that *effectivecommit*(T) precedes *effectivecommit*(T_W).

(iv) T writes x before T_W reads x in which case T_W will read x after T no longer has a lock on x (even a relaxed lock). So *effectivecommit*(T) precedes the write of T on x which precedes the read of T_W on x which precedes *effectivecommit*(T_W).

Case 2: $T_W \rightarrow T$**.**

(i) T_W writes x before T reads x and T is tame. T will commit after *effectivecommit*(T_W) by the normal precedence rules of Tame DRP.

(ii) T_W writes x before T reads x and T is wild. In that case, T_W must have committed (which occurs after *effectivecommit*(T_W)) and released its locks (even its relaxed locks) before T first reads x which precedes *effectivecommit*(T) otherwise T would abort in Step 2.

(iii) T_W writes x before T writes x. In this case T obtains its lock after T_W commits which is after *effectivecommit*(T_W). Whether T is wild or tame, T will obtain its write lock on x before *effectivecommit*(T). So by transitivity, *effectivecommit*(T_W) precedes *effectivecommit*(T).

(iv) T_W reads x before T writes x. Because T_W doesn't abort, T cannot write x or even acquire a write lock on x before T_W performs Step 4 which is after *effectivecommit*(T_W). So T_W performs its Step 4 before T acquires a write lock on x which precedes *effectivecommit*(T). So, *effectivecommit*(T_W) is before *effectivecommit*(T).

In summary, (i) for committed tame and wild transactions, the serialization order will be in *effectivecommit* order, ensuring strict serializability; (ii) there will be no deadlock of either tame or wild transactions ensuring that tame transactions see a transaction-consistent state; (iii) because of the checks in Wild-RP's Step 2 and Step 4, wild transactions will all see a transaction-consistent state. Taken together this guarantees opacity for all transactions. □

Remark. In the above proof, the role of global versions is to ensure that all reads of a wild transaction T read a consistent committed version of the data, even if T ultimately aborts. This is necessary for opacity, but not for strict serializability or deadlock-freedom. Note that if, instead of a global version, each item had its own local version which was incremented every time a commit to the object was updated by a committed transaction, then all committed transactions would be strictly serializable and deadlock-free. (In such a case, a change in the version of a data item x from time t to t' would serve only to indicate that x had been modified between those two times.) All tame transactions and all committed wild transactions would also have observed consistent state. However, aborted wild transactions might have observed inconsistent states.

Acknowledgements

We thank Zhen Sun for his help porting some STAMP benchmarks to a prior version of DRP, Mihir Nanavati for providing the test machine, the IC3 authors for sharing their code, and Michael Walfish for helpful comments on a prior draft of this paper. We also thank the EuroSys anonymous reviewers for their great feedback. Finally, we are indebted to our shepherd, Eddie Kohler, whose detailed comments and suggestions significantly improved our work and the exposition of our ideas. This work was partially funded by NSF grant CNS-1514422, AFOSR FA9550-15-1-0302, and NYU Wireless.

References

[1] TL2-X86. `https://github.com/nathanielherman/sto-stamp/tree/master/tl2`, Mar. 2016.

[2] Silo: Multicore in-memory storage engine (for STO). `https://github.com/nathanielherman/silo/tree/8a63b`, Apr. 2017.

[3] Software transactional objects. `https://github.com/nathanielherman/sto/tree/fd80932`, Jan. 2018.

[4] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[5] R. Appuswamy, A. C. Anadiotis, D. Porobic, M. K. Iman, and A. Ailamaki. Analyzing the impact of system architecture on the scalability of oltp engines for high-contention workloads. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Aug. 2018.

[6] E. D. Berger, T. Yang, T. Liu, D. Krishnan, and G. Novark. Grace: Safe and efficient concurrent programming. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2008.

[7] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, SE-5(3), May 1979.

[8] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for

multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Sept. 2008.

[9] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, June 2014.

[10] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, June 2012.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, Sept. 2006.

[12] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein. High performance transactions via early write visibility. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Sept. 2017.

[13] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, June 2014.

[14] R. Guerraoui and M. Kapałka. On the correctness of transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Feb. 2008.

[15] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 2003.

[16] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Feb. 2008.

[17] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, July 2003.

[18] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2016.

[19] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2), June 1981.

[20] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, May 2017.

[21] S. Mu, S. Angel, and D. Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Mar. 2019.

[22] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.

[23] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4), Oct. 1979.

[24] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[25] T. M. Qadah and M. Sadoghi. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the ACM/IFIP International Middleware Conference*, Dec. 2018.

[26] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Feb. 2009.

[27] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Sept. 2014.

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

[29] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, Aug. 1995.

[30] A. Spiegelman, G. Golan-Gueta, and I. Keidar. Transactional data structure libraries. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2016.

[31] 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 *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Sept. 2007.

[32] The Transaction Processing Council. TPC-C benchmark (revision 5.9.0). `http://www.tpc.org/tpcc/`, June 2007.

[33] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Sept. 2010.

[34] 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 ACM International Conference on Management of Data (SIGMOD)*, May 2012.

[35] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[36] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Sept. 2017.

[37] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, June 2016.

[38] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance ACID via modular concurrency control. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[39] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Aug. 2014.

[40] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, June 2016.

[41] D. Zhang and D. Dechev. Lock-free transactions without rollbacks for linked data structures. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, July 2016.

[42] I. Zhang, N. Lebeck, P. Fonseca, B. Holt, R. Cheng, A. Norberg, A. Krishnamurthy, and H. M. Levy. Diamond: Automating data management and storage for wide-area, reactive applications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2016.

[43] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fault durability and recovery through multicore parallelism. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.