Processing Transactions in a Predefined Order

Mohamed M. Saad
Alexandria University
Alexandria, Egypt
msaad@alexu.edu.eg

Masoomeh Javidi Kishi
Lehigh University
Bethlehem, PA, USA
maj717@lehigh.edu

Shihao Jing
Lehigh University
Bethlehem, PA, USA
shj316@lehigh.edu

Sandeep Hans
IBM India Research Lab
New Delhi, Delhi, India
shans001@in.ibm.com

Roberto Palmieri
Lehigh University
Bethlehem, PA, USA
palmieri@lehigh.edu

Abstract

In this paper we provide a high performance solution to the problem of committing transactions while enforcing a predefined order. We provide the design and implementation of three algorithms, which deploy a specialized cooperative transaction execution model. This model permits the propagation of written values along the chain of ordered transactions. We show that, even in the presence of data conflicts, the proposed algorithms outperform single threaded execution, and other baseline and specialized state-of-the-art competitors (e.g., STMLite). The maximum speedup achieved in micro benchmarks, STAMP, PARSEC and SPEC200 applications is in the range of 4.3x – 16.5x.

CCS Concepts • **Theory of computation → Concurrency**; **Parallel algorithms**; • **Software and its engineering → Concurrency control**.

Keywords Transactions, Parallelization, Ordering

1 Introduction

Transaction ordering intuitively means considering not just the set of transactions as input of the problem, but also the specific commit order that must be enforced for them. Such a formulation inherently includes a fundamental trade off between the level of parallelism achievable, given the need of committing in-order, and the performance of single threaded execution without any software instrumentation.

Ordering tasks before their execution is a problem mostly relevant to contexts where producing executions equivalent to a predefined order is needed in order to satisfy certain properties (e.g., a program semantics equivalent to serial execution). Examples of these deployments include: speculative loop parallelization [13, 33, 35, 37, 38], and distributed computation using the state machine approach [18, 19, 36].

In the former, loops designed to run sequentially are parallelized by executing their iterations concurrently and guarding memory transactions (e.g., by using Transactional Memory [17] as done in [13, 33, 35]). In that case, providing an order matching the sequential one is fundamental to enforce equivalent semantics for both the parallel and sequential code. Regarding the latter, many distributed systems order tasks before executing them to guarantee that a single state machine abstraction always evolves consistently on distinct nodes. A common example of this methodology is when consensus (e.g., Paxos [20]) is employed to establish a common order among commands (or transactions) manipulating a single replicated state.

In this paper we focus on Transactional Memory (TM) as a technology to support speculative execution of tasks, and we present three algorithms to process transactions in parallel while enforcing a predefined order: *Ordered Write Back* (OWB), *Ordered Undo Logging* (OUL), and a lock-steal variant of OUL (OUL-Steal). These algorithms are based on two widely used techniques to merge transaction modifications into the shared state: write-back (in OWB) and write-through (in OUL and OUL-Steal).

All our implementations deploy a common design that uses a cooperative model where transactions exchange both data and locks to increase concurrency while preserving the predefined commit order. OWB uses data forwarding for transactions that finish their execution successfully, but are not committed yet, and OUL leverages encounter time locking with the ability to pass the lock ownership to other transactions. Our cooperative model is similar to the dependency aware transactions model (DATM) [26, 27]. However, DATM is not designed (and thus cannot be optimized) for committing transactions in a predefined order since it tracks all dependencies among transactions and analyzes them at run-time seeking for some correct serialization order instead of the predefined one.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPoPP '19, February 16–20, 2019, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295730

We implement OWB, OUL, OUL-Steal, the ordered version of four existing well-known TM designs (i.e., TL2 [10], NOrec [8], and UndoLog [12] with and without visible readers), and STMLite [21], a specialized solution that commits transactions in a predefined order. We conduct a performance evaluation using a set of micro-benchmarks, STAMP [22], and some applications from the PARSEC and SPEC2000 benchmarks. For determining the transaction order, we use either the index of the application main for-loop that generates the parallel code or a synthetic atomic integer that we inserted as transaction order. Results have been compared agains the sequential execution of the benchmarks, as well as against their parallel execution, as provided by the original version of the applications, if available.

OUL outperforms other ordered competitors consistently. The maximum speedup achieved is 4× over Ordered TL2, 4.3× over Ordered NORec, 8× over Ordered UndoLog visible, 10× over Ordered UndoLog invisible, and 5.7× over STMLite. The peak gain over the sequential non-instrumented execution in micro benchmarks is 10×, 16.5× in STAMP, more than 10× in PARSEC, and 30% in SPEC2000.

OWB ensures TMS1 [11], a weaker but safe [2] consistency condition than opacity [14, 15]. OUL achieves higher concurrency and performance, at the cost of weakening the correctness level by ensuring Strict Serializability [4].

Finally, it is worth mentioning that OWB, OUL, and OUL-Steal are TM implementations meant to be integrated into runtime systems to support their speculative execution. An example of such a system is Lerna [35]. In those systems, a sandboxing [7] mechanism prevents runtime exceptions to be propagated outside the concurrency control engine.

2 Related Work

Transactional Memory (TM) is a technology to speculative execute code [17], and has emerged as a technique to extract parallelism from sequential code [13, 21, 33, 35, 37, 38]. When a predefined order is necessary, conflicts are handled by aborting (and re-executing) whichever transaction ran code with the latest chronological ordering. The key idea is that code blocks run as transactions and commit in the program's original chronological order. The techniques for supporting the aforementioned ordering are classified as: blocking [13, 21, 39] or freezing [40]; a detailed comparison between them can be found in [34].

In the blocking approach, Mehrara et al. [21] proposed STMLite, a TM with a separate thread, Transaction Commit Manager (TCM), that detects conflicts among transactions waiting to commit. TCM orchestrates the in-order commit process with the ability to have concurrent commits. Worker threads poll and stall to wait for the TCM's permission. Gonzalez et al. [13] use a distributed approach for handling the commitment order. Each thread employs a bounded circular buffer to store its completed transactions. If all buffer slots are exhausted, the thread stalls until one of the pending executed transactions reaches the correct commit order.

Another existing solution lets threads *freeze* completed transactions and proceed to execute the transactions with later chronological order, with the disadvantage of increasing the transaction lifetime (hence, higher conflict probability). Zhang et al. [40] introduced a technique to support a predetermined total order of transactions. A *next-to-commit* shared variable is used to preserve this order. Overall, in both the blocking and freezing approach ordering transactions' commits negatively affects the overall resource utilization and may nullify any potential gain due to threads' parallelism. To overcome this restriction, OWB and OUL limit the stalling periods to only the latency of the commit.

The level of atomicity is an orthogonal classification for the aforementioned techniques. The classical TM model mandates transactions to see only *committed* values. However, concurrent transactions can construct a *dependency graph* of uncommitted values. Based on this graph, the transactions commit in the constructed order. Ramadan et al. [27] proposed a dependency aware transactions model (DASTM), in which every object keeps track of all transactional reads and writes, and transactions forward their uncommitted changes to other conflicting transactions. Based on these relations, the commit order is defined at run-time by verifying that the constructed conflict graph is acyclic. This check is very expensive, especially when executed during the transaction execution and leads to performance gain only in the presence of very high conflicts. As opposed to DASTM, OWB and OUL, and OUL-Steal do not maintain the conflicting graph because data forwarding is optimized to enforce only one predefined commit order.

Since code parallelization is the main application of our STM implementations, Thread-level Speculation (TLS) [23, 25] is an immediate related topic. Loop parallelization using TLS has been proposed in both hardware [25] and software [5, 30]. TLS and TM have been merged through a unified model in [3, 23, 28, 29] to get the best of the two techniques. Generally, TLS is a less flexible way to parallelize code than STM. For example, with STM only some instructions can be instrumented while the others still execute without instrumentation; on the contrary, leveraging TLS means speculating over the entire loop body. To overcome some of the well-known TLS limitations, the work in [23] proposes a software TLS implementation where write operations directly update the non-speculative memory and read-races are tracked using metadata. Some of these intuitions have been ported to STM by OWB, OUL, and OUL-Steal.

3 Execution and Memory Model

Our model assumes a set of transactions $\{T_1, T_2, \ldots, T_N\}$. Transactions access shared objects using read and write operations, with their usual meaning [17]. We denote the sets

of shared objects accessed by transaction T_i for read and write as `read-set`(T_i) and `write-set`(T_i), respectively.

A transaction execution is defined as a sequence of operations, where each operation is represented by a pair of *invoke* and *return* events. Besides the read and write operations, whose semantics is the usual one, it also includes a commit operation that starts by invoking the *try-commit* event, whose return value is either *commit* or *abort*. A transaction can also be aborted before invoking the try-commit event. A transaction that begins its execution and did not invoke the try-commit event yet is called *live*. A transaction that invoked the try-commit event but did not commit or abort yet is called *commit-pending*. When a transaction is categorized as committed, it means that all its write operations have been executed permanently on the shared state; and when it is categorized as aborted, its operations have no permanent effect. In both the cases, all metadata is cleaned before proceeding or re-executing. Figure 1 summarizes the transaction states.

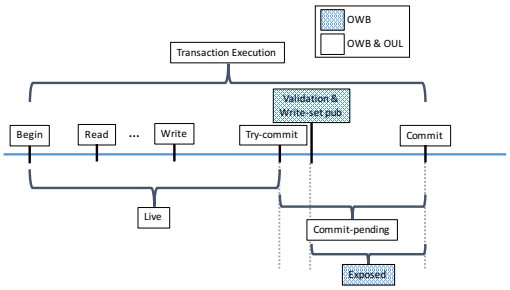


Figure 1. Transaction execution states in OWB and OUL.

A shared object has a value and a (versioned) lock associated with it. We say that a shared object is *exposed* if it is locked by some live or commit-pending transaction. Intuitively, a shared object is exposed if some transaction can already read it, although the transaction that wrote to that object is still executing. A transaction is *exposed* when it is commit-pending and has all its written objects exposed.

Two transactions are said to be *conflicting* if both are concurrent and access an object X, and at least one of them writes on X. Note that two transactions are conflicting even if both write the same object without reading it. Including such a dependency is fundamental, as motivated in the next paragraph. A conflict is handled by aborting one of the transactions, or postponing the access responsible for the conflict (if possible), until the other transaction commits.

Transaction Ordering. We focus on TM implementations providing a specific order of transaction commits, which is assumed to be known prior to the transaction execution. We denote such an order as the transaction *age*. The age is assumed to be defined before activating any transaction (e.g., an ordering layer deployed on top of the TM implementation), and must match the transactions commit order. The age is *unique*, meaning no two transactions can have the same age and once assigned to a transaction, the age does not change even if the transaction is aborted multiple times.

The age of a transaction should not be confused with the transaction timestamp taken from a global timestamp, as used by many existing TM implementations (e.g., TL2 [10], LSA [31]). The age of a transaction is externally determined (e.g., by the application) and does not depend on the execution of concurrent transactions.

Let $\prec$ be the total order relation on transaction ages, and let those ages be denoted as subscripts (e.g., T_i). If $T_i \prec T_j$ we say that T_i has a lower age than T_j; otherwise higher. A concurrency control that enforces an order of commits ensures that when two operations o_i and o_j, issued by transactions T_i and T_j, respectively, are conflicting, then o_i must happen before o_j if and only if $T_i \prec T_j$. We deploy this idea into our execution model by introducing an Age-based Commit Order (ACO). ACO mandates a customization of the classical TM model. As an example, the transaction conflict detection should guarantee that when $T_i \prec T_j$, T_i must not read a value written by T_j (intuitively, T_i should commit before T_j).

We define a transaction T_j as *reachable* if all T_j's lower age transactions are committed, which means that T_j has been reached by a serial execution where all transactions $\{T_1, \ldots, T_{j-1}\}$ committed in the order $1, \ldots, j-1$. In practice, ACO constrains the serialization orders.

In our model, when a transaction aborts, it is restarted by the TM library with the same age.

4 General Design

In this section, we present our co-operative model for supporting ACO. The core idea is to relax the common practice of letting transactions access values written by only committed or commit-pending transactions that will surely commit. In our proposed solutions, we weaken this assumption while still preserving the consistency according to ACO. Depending on the desired correctness and performance level, we permit a transaction to expose its changes either:

- after it invokes the try-commit event and performs a validation to verify execution's consistency, but still allowing it to abort later due to ACO violation (in OWB); or
- right after the write operation takes place during the execution and aborting any dependent transaction as soon as a further modification on the same early exposed value happens (in OUL).

The above idea allows transactions with higher age to use such visible changes. Although it speeds up the flow of values from lower to higher age transactions, it also creates a possible dependency chain with other live and commit-pending transactions that accessed those values. Therefore when an abort occurs, the abort event should be immediately triggered to all the dependent transactions (cascading abort).

We can classify existing transactional models according to the way they handle concurrent read and write operations as: conservative model, conflict serialization model (e.g., DASTM [27]), and cooperative ordered model (our model). The former prohibits the co-existence of read and write operations issued by concurrent transactions on the same object (this model is deployed by most concurrency controls). The conflict serialization model permits all combinations and selects the commit order based on the transaction dependency. Our cooperative ordered model restricts the memory snapshot seen by transactions to only the values exposed by transactions with lower age.

Interestingly, under the conflict serialization model transactions are aborted only when a mutual dependency (i.e., a cycle in the transaction dependency graph) exists; in our model, the graph is always acyclic. Avoiding to identify cycles in the transaction dependency chain increases the chance to achieve high performance.

4.1 Cooperative Ordered Transactional Execution

To construct our cooperative model, we start by highlighting the following two events of a transaction execution:

- a transaction is *exposed* when all its written objects are exposed, and it is in the commit-pending state by having all its read operations consistent according to the ACO, therefore no conflict with lower age transactions occurred;
- a transaction becomes *reachable*, when all the lower age transactions have been committed.

Supporting this new model requires that: *i)* aborted transactions should be able to abort other transactions that accessed their exposed updates; and *ii)* lower age transactions should enforce the abort of exposed higher age transactions. Accomplishing the above goals requires maintaining some transactional metadata (e.g., read and write sets, including acquired locks) even after a transaction is exposed. Those metadata help in identifying conflicts (or aborting) exposed transactions, and they should be kept accessible until a transaction becomes reachable. Additionally, we need to support the cascading abort of multiple live or exposed transactions that share elements in their read-sets and write-sets.

Exposing written objects before being sure that a transaction eventually commits may violate the ACO if all lower age transactions are not committed yet. Similarly, ACO might be violated when the transaction conflicts with a lower age transaction that accesses a common object that is exposed by the first. For this reason, we postpone releasing the transaction metadata until the transaction becomes reachable, thus providing a safe point to decide whether a commit or abort should be triggered. The main difference between an exposed and a reachable transaction is that: the former, although it has already published its modifications, it can still be aborted (and trigger the cascading abort of other transactions); the latter cannot be aborted anymore. It is therefore safe to release all its metadata without violating the ACO.

5 The Ordered Write Back (OWB)

The Ordered Write Back Algorithm (*OWB*) employs a write-buffer approach; a transaction writes its modifications into a local buffer. While entering the try-commit phase, the transaction acquires a versioned-lock for each object in its write-set and writes its changes to the shared memory, and becomes exposed. To avoid concurrent writers, the locks are not released until the transaction becomes committed or is aborted. However, to allow an early propagation of the modifications, higher age transactions can read these locked objects. In case an abort is triggered, the exposed transaction is responsible to abort any dependent transaction that has read the exposed values. We use versioning to detect conflicts between concurrent transactions. The transaction performs a validation before exposing its values, and before releasing its locks to approach the final commit.

In practice, for OWB a transaction is exposed if: it is executed until the end without any conflict with other concurrent transactions; it acquired locks on its modified objects successfully; it exposed its new values to the shared memory; and it is waiting to be reachable. A transaction can commit only if it is reachable and passes the validation of its read operations. The transaction also releases its acquired locks at this stage.

A transaction keeps these metadata: *1) read-set*, which stores read objects and their read version; *2) write-set*, which stores the modified objects and their new values; and *3) dependencies list*: a list of transactions that read the changes done by this transaction after it becomes exposed. Shared objects are associated with a versioned lock. The lock stores the version number and a reference to the writer transaction (if it exists) that currently owns it. The version is incremented when a new value for the object is exposed. The pseudocode of OWB is included in the paper's technical report [32].

The `Write` operation adds the object and its new value to the write-set. The `Read` operation first checks if the object has been earlier modified by the transaction itself. If so, the new value from the write-set is returned; otherwise the object, along with its version, is fetched from the shared memory.

If the object is currently exposed, then the writer is aborted only if its age is higher ($W_2 \rightarrow R_1$), and the read operation is retried. If the transaction that holds the lock has a lower age than the reading transaction, we let the latter read the written value ($W_1 \rightarrow R_2$) and we add it to the writer's dependencies list. That way, if the writer aborts in the future, it can cascade its abort to the affected transactions who read its modifications. In order to avoid inconsistencies while reading from an exposed writer, we let the reader double check the writer state (if it is aborted) after it has been registered in the writer's dependencies list; also the dependencies list must provides a thread-safe insertion. Before a read operation returns, the read-set is validated by invoking the `Validate_Reads` procedure (see below). Missing that would

make the read-set of OWB transactions not consistent during execution.

To enter the exposed state, a validation of the read-set is executed to make sure the transaction reads a consistent view of the memory before exposing the locally buffered written objects. `Validate_Reads` compares the current versions of the read objects with the value of the corresponding versions stored in the read-set. If the current version is different, it means that the object was modified after the read, i.e., a Write after Read (WAR) conflict.

Upon passing a read-set validation, the `exposed` procedure acquires the locks and then writes the write-set to the memory. If the locks are already acquired by another concurrent exposed writer ($W_1 \rightarrow W_2$ or $W_2 \rightarrow W_1$), we handle that by favoring the lower age transaction, and aborting the other. Since exposed transactions can still be aborted by other transactions, we need to store the old value of modified objects. This is done by swapping the write-set stored values with the old objects' values at commit time.

Finally, at commit time we call `Validate_Reads` again to prevent the WAR anomaly. However, since write-set elements are already locked, we can leverage that to reduce the validation overhead. Consider T_i is executing the commit operation. Let $X \in \texttt{read-set}(T_i) \cap \texttt{write-set}(T_i)$. As T_i is still acquiring locks over its write-set (including X), T_i is sure that the value of X is unchanged since its lock acquisition, thereby it could be excluded from the commit-time read-set validation. To do so, it requires checking that read-set objects have not been changed while acquiring locks.

Keeping the commit execution time short is fundamental; the optimization just described shrinks the commit execution at the price of adding an extra check in the `Try-Commit` procedure. However, having common objects in the transaction's read-set and write-set is usual, which makes this optimization fruitful.

Finally, when a transaction becomes reachable and the re-validation succeeds, the `commit` operation releases its acquired locks and reclaims metadata.

As correctness guarantee, OWB guarantees TMS1 [11]. The intuition is that: if for a history generated by OWB, every exposed transaction is committed, then the history is opaque [14]. First of all, transactions can commit only in the ACO order, serializing all the committed transactions, which makes OWB strict serializable. Moreover, OWB allows transactions to read only from commit-pending (exposed) and committed transactions, and any time a transaction enters the exposed state, it aborts all concurrent transactions that has read a value that violates the ACO. However, exposed transactions can abort after some live transaction already read those values. This is not allowed by Opacity, but TMS1 allows that as long as the live transactions do not perform any operation after the exposed transaction is aborted. OWB implements that through an atomic cascading abort. We give more details about correctness in Section 7.

6 The Ordered Undo Log (OUL)

The Ordered Undo Log (*OUL*) Algorithm is an undo-log algorithm that preserves the ACO. Transactional updates affect the shared memory at encounter time, while the old value is kept in a local undo-log. Such a scheme implies that the transactions' order is guaranteed while operations are invoked, and not at commit time as in OWB. In order to deploy the above idea, each object is associated with a read-write lock. The transaction acquires a read or write lock according to its need, as explained later. Each lock stores the reference to the (single) writer transaction, which can be either the current transaction holding the lock or the one that committed that version, and a list of concurrent readers, meaning live or commit-pending transactions that accessed the version for reading it. Since the size of the readers list might impact the efficiency of the protocol, it should be bounded.

As in OWB, every transaction in OUL maintains a write-set, but here the write-set stores the old values of the written objects (undo-log). On the other hand, the transaction read-set is implicitly represented by the object lock's readers list.

The pseudocode of OUL is included in the paper's technical report [32]. In the `Read` procedure, we allow Read after Write (RAW) conflicts only if the writer transaction has a lower age ($W_1 \rightarrow R_2$); otherwise the speculative writer is aborted ($W_2 \rightarrow R_1$). The `Write` procedure enforces that only a single transaction can hold the write lock on the object at a time. A Write-Write conflict is solved by aborting the transaction with the highest age. As readers are visible, the writer transaction can check if there is any (wrong) speculative reader, and abort it accordingly ($R_2 \rightarrow W_1$).

One of the major benefit of a write through protocol is that the `Try-Commit` procedure is simple because the values are already in shared memory. However, in OUL exposing a transaction only means that it did not conflict with other transactions so far, but it could be still aborted to preserve ACO. In the `Commit` procedure, the transaction is marked as `Inactive` and locks are released. As said earlier, since a lock is maintained with a back-reference to the transaction that holds it, setting the transaction status is sufficient to release all the locks held by that transaction with a single step. In the `Abort` procedure the transaction restores old values from the undo-log (Rollback), and release all the locks (`Inactive`).

6.1 The OUL-Steal Algorithm

OUL-Steal is a variant of the OUL algorithm where we relax the aforementioned multiple-writers restriction and allow write-writer conflicts while guaranteeing ACO. In both OWB and OUL, conflicting transactions co-operate to commit as they are allowed to proceed without aborts even in the presence of some read-write conflict, as long as ACO is still preserved. However, a writer transaction holds the locks until reaching the *commit* state, which sometimes limits the overall concurrency.

Let T_i and T_j be two conflicting writers on object X, and $T_i \prec T_j$. In OUL, if T_i finds X locked by T_j ($W_1 \rightarrow W_2$), T_i should abort T_j. However, ACO could still be preserved if T_j overwrites the value written by T_i and there is no transaction T_k, with $T_i \prec T_k \prec T_j$, that reads X in the future.

OUL-Steal allows a transaction with higher age to overwrite the value written by a concurrent transaction with lower age ($W_1 \rightarrow W_2$), and *steal* its lock. The higher age transaction stores the stolen lock in a local list so that the lock can be returned back to the original writer (the lower age transaction) in case of abort. That way, if a mid-age reader T_k needs the value of a lower age transaction, then it can abort the higher age transaction(s) which stole the lock(s); otherwise (i.e., without T_k), the value written by the higher age transaction will be used by higher age readers. This operation could be repeated until the reader reaches the correct writer transaction.

In `Write`, the lock is passed to the higher age writer and is saved in its write-set. As a consequence, the written address exists in the undo-log of both the writers (the original and the one which stole the lock). During the `Abort`, the transaction uses `Rollback` to revert its changes using its undo-log. An undo-log entry can be:

- *stolen by* another writer: which means the transaction does not have the ownership record at the abort time. In this case, the transaction does not do any action, although, it keeps the undo-log entry, which contains the address value before the current transaction modifications.
- *exclusively modified* by the current transaction, reverting the old value from the undo-log, and aborting the speculative readers.
- *stolen from* another writer: in addition to the steps done in the *exclusively modified* case, the lock ownership is passed back to the old writer, and the current transaction checks the state of the old writer. If it was *not aborted*, then no further action is needed. Otherwise, the transaction calls the `Rollback` of the old owner. At this stage, the old writer will treat the entry as the cases of *exclusively modified* or *stolen from*, accordingly.

The complete pseudocode of the OUL-Steal algorithm is available in the paper's technical report [32].

As correctness guarantee, OUL guarantees Strict Serializability [24]. Unlike OWB, OUL allows reading from live transactions, which is not allowed by TMS1 (and hence opacity). However, similar to OWB, OUL restricts transactions to commit only in the ACO order, making OUL strict serializable. More details about correctness are in Section 7.

7 Correctness

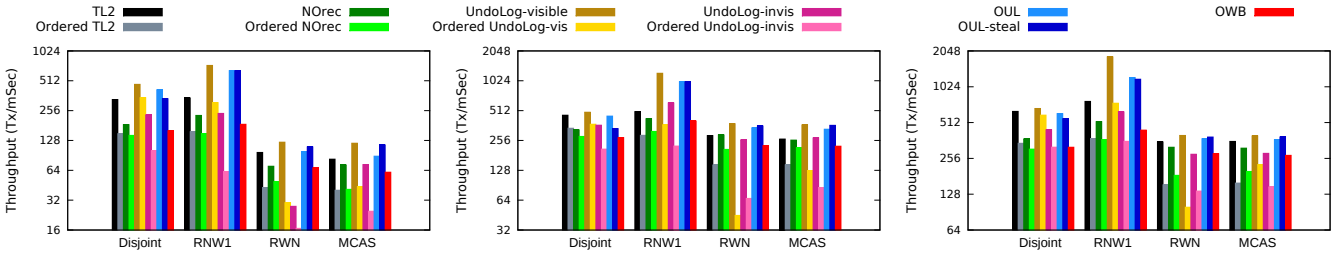
Here we discuss the correctness of the given algorithms. We do not include the case where a transaction triggers an error (e.g., division by zero) because it speculatively processes a computation that in a non-parallel execution would not happen. Such an execution might be a for-loop iteration executed speculatively and preceded by an iteration that breaks the for-loop itself. We assume a sandboxing mechanism to handle such runtime exceptions.

First, we show how our protocols preserve the ACO. Suppose by contradiction that the ACO is violated. Let T_i and T_j be two transactions such that $T_i \prec T_j$. The interesting case is if T_i successfully reads a value of an object X written by T_j. This implies that $R_i(X)$ happened after T_j exposes X's value in OWB or $write(T_j)$ in OUL. In both OWL and OUL, T_i acquires a shared lock on X at the time of the read operation, either by visible reads (OUL, OUL-Steal) or checking if there is no writer (OWB). For a successful read, the shared lock must be acquired, thus the write lock should not be already granted. This implies that T_j has released all its locks. As a transaction does not release its acquired locks until it *commits*, T_j must be necessarily committed. Therefore $R_i(X)$ must occur after $commit(T_j)$. Since a transaction cannot perform any step after it commits, $R_i(X) \rightarrow commit(T_i)$. This means $commit(T_j) \rightarrow commit(T_i)$, which cannot be the case since they must commit in order, according to their ages.

Now we prove that both OWB and OUL are serializable. In order to prove that, we define $DG(i, j)$ as a predicate defining a dependency between T_i and T_j, when T_j reads a value written by T_i, or T_j overwrites a value written by T_i. Using this definition, we can construct a dependency directed graph $DG(T, D)$, where T is the set of all committed transactions, and D is the set of dependency relations. It is easy to see that $DG \subset SG$, where SG is the conflict serialization graph [4]. A history is serializable if and only if its SG is acyclic. Note that serializability is not guaranteed if DG is acyclic.

Assume by contradiction that an execution of our algorithms produce a cyclic DG, which implies having an edge $D(i, j)$ where $i > j$. By definition of dependency, this means that either T_j reads a value written by T_i (i.e., $W_i(X) \rightarrow R_j(X)$), or T_j overwrites a T_i's written value (i.e., $W_i(X) \rightarrow W_j(X)$). In all the proposed algorithms, exclusive locks must be acquired when we expose the written values (at commit in OWB or encounter time in OUL and OUL-Steal) and released only at commit, or passed to a higher age transaction (which is not the case here). We can rewrite the previous situations as $commit(T_i) \rightarrow R_j(X)$ or $commit(T_i) \rightarrow W_j(X)$. Since a transaction cannot perform any step after it commits, $commit(T_i) \rightarrow commit(T_j)$, which cannot be the case as mentioned earlier; therefore, DG is a acyclic.

Assume $e \in E = SG \setminus DG$, this edge represents the case where $R_i(X) \rightarrow W_j(X)$, which means $R_i(X) \rightarrow commit(T_j)$. In OWB, the procedure that validates read operations captures this by comparing the read version with the current version of the accessed object; while in OUL and OUL-Steal, the readers' visibility enables T_j to detect the $R_i(X)$ and aborts it. So $E = \emptyset \implies SG = DG \implies SG$ is acyclic, making the algorithms serializable.



(a) Long Transaction **(b)** Short Transaction **(c)** Heavy Transaction

Figure 2. Peak performance of all competitors (including unordered) using all micro benchmarks (Y-axis is log scale).

The serialization point of both OUL and OWB is inside the commit procedure: for OUL is when the transaction's status is atomically set to `Inactive`; for OWB is when locks on written objects are released. As the serialization point is inside the transaction execution, all the algorithms preserve the real-time order, and are strict serializable.

In addition to being strict serializable, OWB is TMS1 [11], a stronger condition than strict serializability. Being TMS1, OWB ensures that the response of every operation on an object, even if performed by aborted and live transactions, is consistent with a serial execution. Informally, for a history to be TMS1, it must be strict serializable, and for every successful response of an object operation by a transaction T, there must exist a serialization of a subset of the transactions, justifying the response. This subset must contain T (until the response) and all the committed transactions that completed before T started. The serialization can also contain some commit-pending transactions, and some committed and even aborted transactions, that are concurrent to T.

We have shown that OWB is strict serializable. Since OWB allows a read operation to return a value written by an exposed transaction, which may get aborted later, it justifies including concurrent aborted transaction for the response. Recall that OWB allows reading values written by committed and exposed transactions only, but not from aborted transactions. The intuition is that if a transaction reads from an exposed transaction, which gets aborted later, the reading transaction is also aborted without executing any further operations. This is done using cascading mechanism in OWB.

8 Implementation and Evaluation

In our implementation locks are implemented using 32 bits. The mapping between addresses and locks is made by leveraging the least significant bits, thus a single lock might be responsible for multiple addresses. The lock is divided into two parts: the most significant bits represent the reference to the writer, and the remaining bits represent either the header address of the readers list (for OUL), or the version number (for OWB). In OUL, we use a bounded list of readers to limit the number of concurrent readers, which is set to 40.

A thread plays multiple roles in our implementations: *worker*, *validator*, or *cleaner*. A *worker* executes transactions and performs the `try-commit`. A *cleaner* takes care of reclaiming metadata. Once the transactional operations are all executed, any thread can take the lead of finalizing any transaction; however, there is only one thread at a time in the *validator* role. This role is responsible for moving commit-pending transactions to the committed state and re-executes invalid transactions. We adopt the *flat combining* [16] technique to let threads take ownership of the *validator* role.

We compare our algorithms with STMLite [21]: a lightweight STM with ACO used to support code parallelization; the unordered and ordered version of three state-of-art TM algorithms: TL2 [10], NOrec [8] and UndoLog [12] (with and without visible readers).

Both TL2 and NOrec follow the write-back design strategy and validate transactions at commit time. To enforce ACO in these ordered implementations, transactions are allowed to enter the commit phase only when all transactions with lower age have been committed. In order to aid the ordering for UndoLog, we exploit an age-based contention policy (i.e., always favor transactions with lower age) to handle write-write conflicts. In the *visible readers* variant, the writer transaction aborts all active readers, while when readers are *invisible* the writer retries multiple times if the object is locked, then it backs off. STMLite uses a write-back implementation and replaces the need for constructing a read-set by leveraging signatures (Bloom Filters). There is a tradeoff in determining the effective size of signatures, but the authors recommended a range of 32 to 1024. We used a signature of size 64 with the STL hashing function because it provided the best performance. The number of threads in STMLite also includes its commit manager.

All competitors have been re-implemented atop the same baseline software framework so that all take advantage of the same low-level optimizations. It is worth noting that competitors may provide different correctness guarantees (e.g., OWB provides TMS1 while NOrec/TL2 give opacity).

In our experiments, the ACO is defined in two ways. Unless otherwise specified, the index of the dominant for-loop that each benchmark uses to generate parallel code (e.g., transactions in STAMP) is used as transaction age. In some application with more complex patterns, such as nested loops, we insert an atomic integer to define and assign ages.

Threads are pinned to cores. The scheduling policy is to use up all cores of one socket before moving to the other one. We report the throughput for micro benchmarks and the application execution time for STAMP and some applications of PARSEC and SPEC200 benchmarks by varying the number of serving threads in the thread-pool (the datapoint at 1 thread shows the performance of the single threaded transactional execution). We also compare our performance against the unordered algorithms, which do not use ACO. In this case, applications directly activate transactions in parallel because no ACO needs to be defined. Performance of the non-transactional single threaded execution (green line) is also included.

We used two different machines for our experiments: micro benchmarks and STAMP have been evaluated on an AMD machine equipped with 2 Opteron 6168 CPUs, each with 12-core running at 1.9 GHz. The total memory available is 12 GB. Evaluation of PARSEC applications and SPEC2000 has been done using a Intel server hosting 4 Intel Xeon Platinum 8160. Results are the average of five runs.

Micro Benchmark. In our first set of experiments we consider the RSTM micro-benchmarks [1] to evaluate the effect on performance of different workload characteristics, such as the amount of operations per transaction, the transaction length, and the read/write ratio. Each experiment includes running half a million transactions. For each micro benchmark, we configure three types of transactions: *short*, *long*, and *heavy*. Both short and heavy have the same number of accesses (i.e., a random between 10 and 20), but the latter adds more local computation in between them (i.e., 100 CPU-ops). Long transactions simply produce more transactional accesses (i.e., a random between 30 and 60).

Figure 2 summarizes the peak performance of all competitors. The gap in performance between the ordered and unordered versions of the same algorithm is: 26-56% for TL2, 13-41% for NOrec, 12-88% for UL-vis, and 28-74% for UL-inv.

As a general comment on these results, OUL and OUL-Steal outperform all other ordered versions of the algorithms. OUL-Steal excels for write dominant workloads and performs equally to OUL in read dominant workloads; OWB outperforms all write-back-based implementations in most benchmarks. At high thread count, STMLite suffers from false conflicts due to the use of signatures. However, at low number of threads (less than 8) and with long transactions it achieves a higher peak throughput than Ordered TL2 and Ordered NOrec, because it benefits from the quick validation using signatures. For the UL-inv algorithm, the readers' visibility is crucial; without this information, the algorithm may abort a lower age transaction (using timeout) while some higher age transaction holds the read shared lock. On the other hand, these higher age transactions cannot commit before their order comes, hence they timeout.

In configurations where the performance of the sequential (non-transactional) execution is faster than many ordered algorithms, our solutions outperform it, letting parallelism pay off. However, there are two benchmarks with long transactions where the sequential execution is still faster. These workloads represent unfavorable scenarios for processing ordered transactions because of the high cost of aborting transactions (possible repeatedly) due to ACO violation.

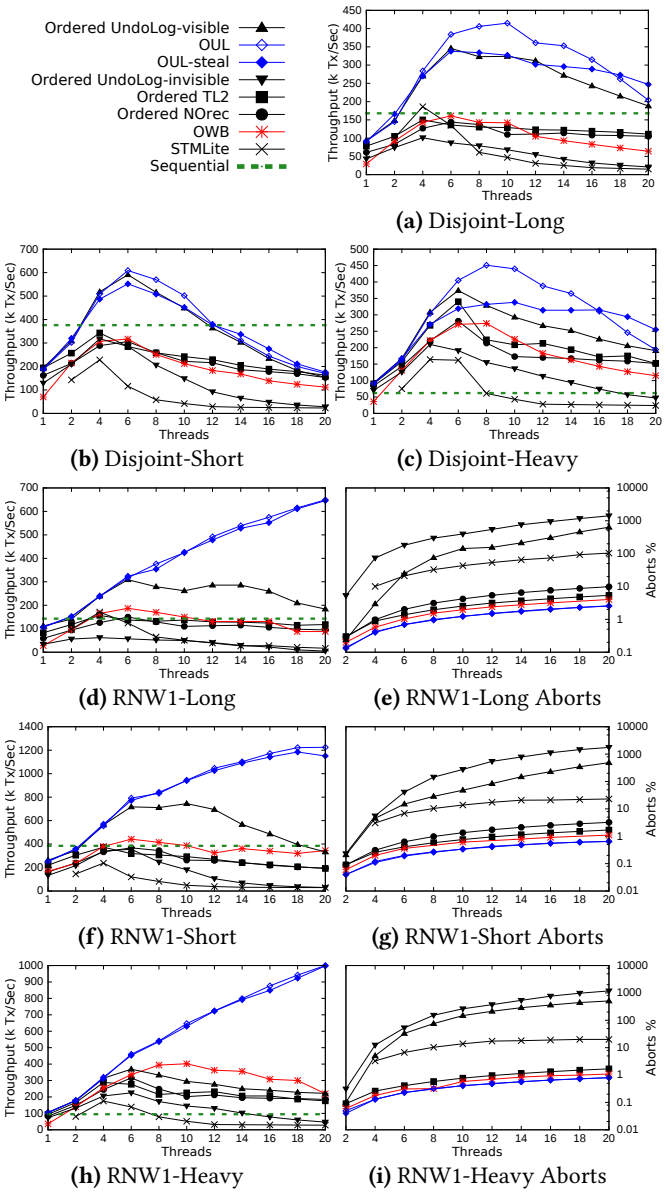


Figure 3. Disjoint 3a – 3c and ReadNWrite1 3d – 3i.

The *DisjointBench* (Figures 3a-3c) produces a workload with no conflict between concurrent transactions. Every transaction accesses a different set of addresses with read and write operations. In all configurations, OUL achieves the best throughput, while OUL-Steal suffers from the overhead of its lock management scheme without actually gaining from that, as the disjoint transactions do not have any shared accesses.

UL-vis achieves a throughput near to OUL-Steal, thanks to the simplicity of its immediate write strategy. In all the three plots is visible a peak performance around 6 threads. This shape is the consequence of NUMA latency [6, 9], which can be appreciated in this configuration more than in others due to the absence of data contention.

In addition to that, without aborts we can show the transactional access overhead for all competitors. It is intuitive that UndoLog algorithms (UL-vis, UL-inv, OUL, OUL-Steal) benefit from having the values already in memory, thus they outperform others. In fact, the UndoLog's main drawback is the costly abort, which never happens in this benchmark. With long transactions (Figure 3a), STMLite benefits from eliminating lock usage at the write-back phase and it has minimal overhead at low numbers of threads. On the other hand, for short transactions (Figures 3b and 3c) the Ordered TL2 algorithm performs better. OWB has a moderate overhead relative to the other write-back algorithms.

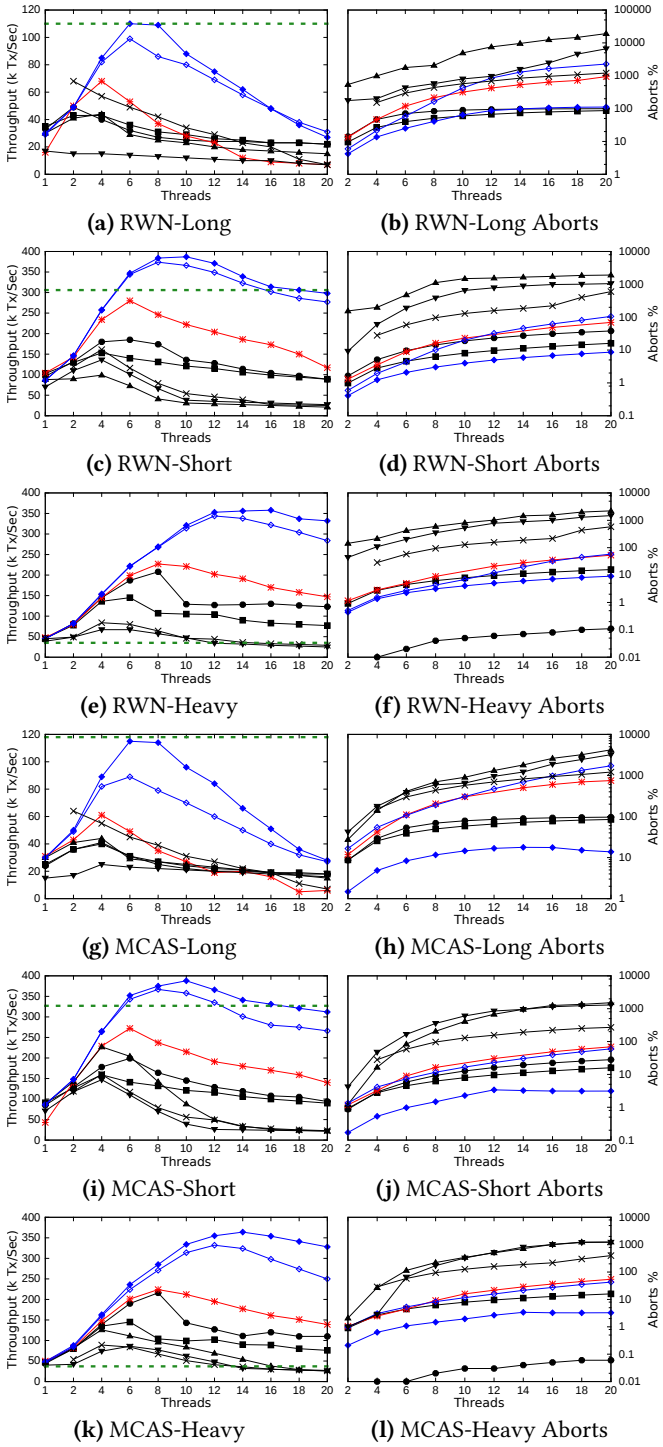
In *ReadNWrite1Bench* (Figures 3d-3i), the transaction reads N locations and writes one. Since transaction write-set is very small, the number of aborts is low. Similarly, UndoLog algorithms excel here as well. With long and heavy transactions (Figure 3d, 3h), the processing done by workers overweights the overhead due to single threaded transaction validator, so both OUL and OUL-Steal scales well with increasing the number of workers. On the other hand, the validator represents a performance bottleneck for short transactions (Figure 3f), resulting in a slightly lower scalability.

In *ReadWriteN* (Figures 4a-4f), each transaction reads N locations, and then writes to other N locations. The large transaction write-set introduces a challenge for both undo-log (increases the number of aborts) and write-buffer algorithms (delay at commit time). The cooperative execution enables OUL, OUL-Steal and OWB to outperforms all other algorithms in all workloads. OUL-Steal outperforms OUL by 10% because it significantly reduces the number of aborts (Figures 4b, 4d, and 4f).

MCASBench performs a multi-word compare-and-swap, by reading and writing N consecutive locations. Similar to ReadWriteN, the write-set is large but the abort probability is lower than before because each pair of read/write acts on the same location. Figures 4g-4l illustrate the impact of increasing workers with the different workloads. We noticed a similar trend to *ReadWriteN*.

The breakdown of the abort reasons for OWB, OUL, and OUL-Steal is shown in Figure 5. Aborts are measured for the number of workers that achieved the maximum throughput.

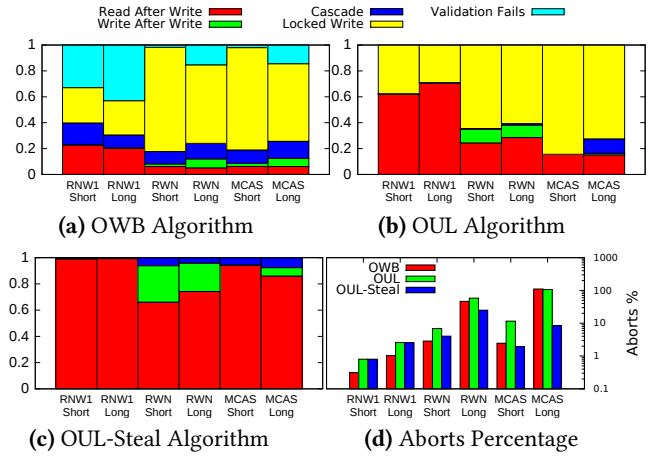
In OWB (Figure 5a), with *RNW1bench* aborts due to validation failure represent the main reason; while in write-intensive benchmarks, such as *RWNbench* and *MCASbench*, aborts are mainly (65%-82%) due to concurrent commits (*Locked Write*). However, only 3% of these cases fall in WAW, which means that OWB can benefit from the lock-steal optimization and save a considerable amount of aborts. However,



(a) RWN-Long **(b)** RWN-Long Aborts
(c) RWN-Short **(d)** RWN-Short Aborts
(e) RWN-Heavy **(f)** RWN-Heavy Aborts
(g) MCAS-Long **(h)** MCAS-Long Aborts
(i) MCAS-Short **(j)** MCAS-Short Aborts
(k) MCAS-Heavy **(l)** MCAS-Heavy Aborts

Figure 4. RWN 4a – 4f; MCAS 4g – 4l

applying lock-steal on OWB would complicate the design and the validation procedure. The reason is that transactions use commit-time locking and rely on the version number to validate their read-set. With lock-steal, multiple writers



(a) OWB Algorithm **(b)** OUL Algorithm

(c) OUL-Steal Algorithm **(d)** Aborts Percentage

Figure 5. Aborts Breakdown

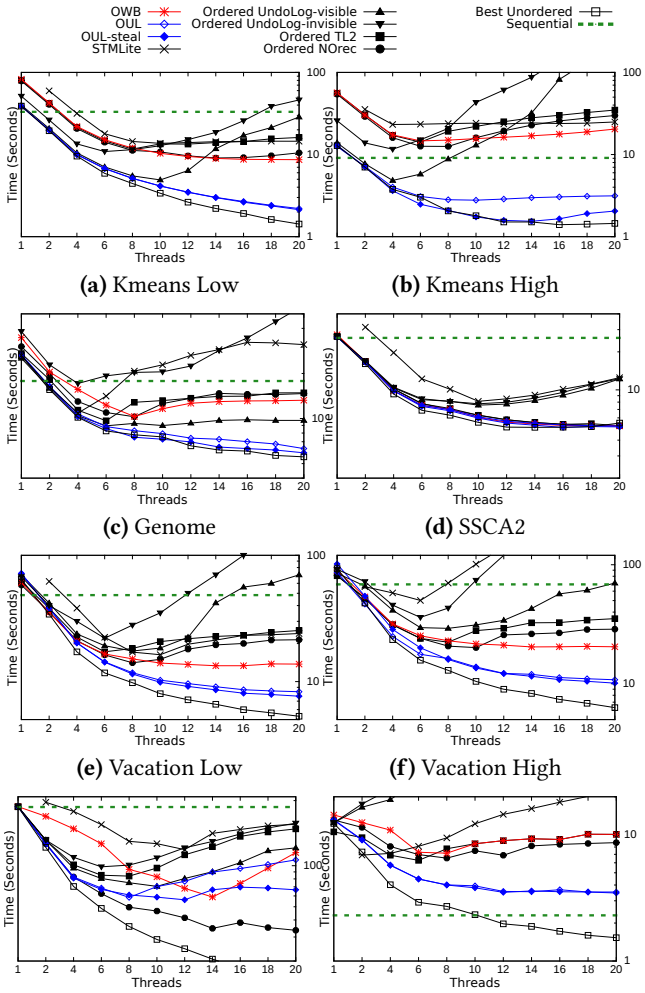
would increment the version number, thereby readers would not be able to do the validation simply.

For OUL and write-intensive benchmarks, concurrent writes generate between 70% to 85% of total aborts; a WAW represents at most 10% of them. In OUL-Steal, stealing the lock eliminates the problem of concurrent writes, and narrows write-write conflicts to only the WAW anomaly. However, it introduces several changes to the abort characteristics: a writer transaction that steals the lock becomes able to abort any invalid speculative readers earlier than before. This was reflected on increasing the number of *Read After Write* aborts; the probability of triggering cascading aborts is increased if compared to OUL (Figures 5b, 5c); and the total number of aborts of OUL is reduced by one order of magnitude (Figures 3e, 3g, 4b, 4d, 4h, 4j, 5d).

Although OUL-Steal reduces the number of aborts, the speedup is on average 20%. This is because the abort procedure of OUL-Steal involves recursive rollback for stolen locks. Also, in OUL algorithms the abort cost differs according to the transaction type. In fact, aborting a write transaction requires restoring its original value, thus forcing the transactions involved in the conflict to wait for the restoration of old written values; aborting the readers is cheaper.

Figure 5d shows the number of aborts in the maximum throughput scenario. OUL experiences more aborts than OWB because of the eager accesses; OUL-Steal avoids this drawback and experiences lesser, yet longer, aborts.

STAMP Benchmark. STAMP [22] is a benchmark suite with applications covering a variety of domains. Figure 6 shows the collected execution time (lower is better). Two applications (Yada and Bayes) have been excluded because they expose non-deterministic behaviors, thus their evolution is unpredictable. The datapoints for competitors that do not scale in some configuration are omitted to preserve the scale and readability of the plots. We also included performance



(a) Kmeans Low **(b)** Kmeans High

(c) Genome **(d)** SSCA2

(e) Vacation Low **(f)** Vacation High

(g) Labyrinth **(h)** Intruder

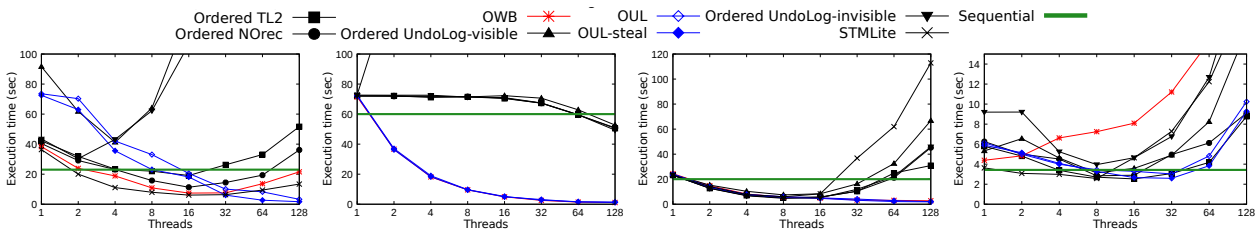
Figure 6. Execution time of STAMP (Y-axis log scale).

of the unordered STM algorithm (among those in Figure 2) that behaves best in each plot.

Kmeans, a clustering algorithm, iterates over a set of points and associates them to clusters. The main computation is in finding the nearest point, while shared data updates occur when updating the cluster centers. Both OUL and OUL-Steal scale when increasing the number of workers, while under high contention OUL-Steal performs better (Figure 6b). OWB and Ordered NOrec have similar performance, but OWB does not degrade at high thread count.

Genome reconstructs the gene sequence from segments of a larger gene. It uses a shared hash-table to organize segments, which requires synchronization over its accesses. Genome exhibits a little contention, which makes OUL and OUL-Steal perform similarly (Figure 6c).

SCAA2 is a multi-graph kernel that is commonly used in domains such as biology and security. The core of this kernel uses a shared graph structure that is updated at each



(a) PARSEC/Blacksholes **(b)** PARSEC/Swaptions **(c)** PARSEC/Fluidanimate **(d)** SPEC2000/Equake

Figure 7. Execution time using PARSEC and SPEC200 benchmarks.

iteration. The amount of contention is low as the large number of graph nodes leads to infrequent concurrent updates. Figure 6d shows that all algorithms perform almost equally and benefit from optimistic concurrency.

Vacation is a travel reservation system using an in-memory database. Each client uses a coarse-grain transaction to execute its session, consequently, aborts are costly. Again, our cooperative model boosts the performance of the proposed algorithms, and they scale well when increasing the number of workers (clients) (Figures 6e and 6f).

Labyrinth is a multi-path maze solver. The maze is represented as a three-dimensional uniform grid, and each thread tries to connect input pairs by a path of adjacent maze points. Upon finding a path, it is is highlighted at a shared output grid. Transactions conflict when their paths overlap. In Figure 6g, NOrec outdoes other algorithms because of two reasons: *1)* as Labyrinth updates adjacent addresses for the path, it is prone to produce false sharing for all other algorithms that use locks; and *2)* NOrec employs a value-based validation, thus when two conflicting transactions updating a maze point with the same value, they commit successfully.

Intruder, a network intrusion detection system using signatures. It compares the captured packets against a dictionary of intrusion signatures. Packets are processed in parallel, grouped in sessions, and stored in a self-balanced (red-black) tree. Transactions guard the tree operations; contention is high and depends on the frequency of the rebalance operation. Figure 6h shows that not all algorithms scale well. The sequential execution outperforms all of them (except the unordered).

PARSEC Benchmark. PARSEC is a benchmark suite for shared memory chip-multiprocessors architectures.

The *Black-Scholes* application calculates Black-Scholes equation for input values. Since calculations per iteration are few, each transaction involves multiple calculations to reduce the overhead of parallelization. *Swaptions* employs Monte Carlo simulation to compute prices. *Fluidanimate* is an application performing physics simulations. The main computation is spent on computing particle densities and forces, which involves six levels of loops nesting updating a shared array structure. Since it is not straightforward to assign ages based on number of iterations, a global atomic integer variable is used to assign ages to transactions.

OUL, OUL-Steal and OWB scale in these three applications; significant speedup over sequential is achieved in Swaptions. In both Black-Scholes and Fluidanimate, all other algorithms outperform sequential when contention is low. Performance drops quickly when contention increases, which is due to a high abort rate.

SPEC CPU2000 Benchmark. *Equake* is an application included in the SPEC CPU2000 benchmark and it simulates the propagation of elastic waves. The computation iterates over a number of steps and, in each time step, it iterates over a number of nodes where each performed calculation relies on the previous one. The loop-carried dependencies forces the transaction to be committed in a specific order. Each thread is assigned a consecutive region of nodes so only those in joints may abort.

When testing this benchmark, we set the input size to be 500 nodes. The results show that OUL, OUL-Steal and OTL2 scales when increasing the number of threads, up to 32 threads; the achieved peak speedup is 30%. After that, because of high contention and increasing number of aborts, all algorithms' performance drops.

9 Conclusion

In this paper, we presented three algorithms, OWB, OUL, and OUL-steal, that address the problem of committing transactions with an order defined prior to execution. Automatic code parallelization is a topic where such a requirement applies. Our results show that even if a system requires a specific commit order, it is possible to achieve high performance exploiting parallelism in the presence of data conflicts.

Acknowledgments

Authors would like to thank the anonymous reviewers for their valuable comments, Binoy Ravindran for his feedback at the early stage of this paper, and Jacob Nelson for the insightful discussion. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367 and by the National Science Foundation under Grant No. CNS-1814974.

References

[1] 2010. RSTM: The University of Rochester STM. http://www.cs.rochester.edu/research/synchronization/rstm/.

[2] Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. 2014. Safety of Live Transactions in Transactional Memory: TMS is Necessary and Sufficient. In *DISC*. 376–390.

[3] Joao Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe, and Rachid Guerraoui. 2012. Unifying thread-level speculation and transactional memory. In *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc., 187–207.

[4] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems*. Addison-Wesley.

[5] Anasua Bhowmik and Manoj Franklin. 2002. A general compiler framework for speculative multithreading. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures (SPAA '02)*. ACM, New York, NY, USA, 99–108. https://doi.org/10.1145/564870.564885

[6] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. 2016. Investigating the Performance of Hardware Transactions on a Multi-Socket Machine. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, Christian Scheideler and Seth Gilbert (Eds.). ACM, 121–132.

[7] Luke Dalessandro and Michael L. Scott. 2012. Sandboxing Transactional Memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 171–180. https://doi.org/10.1145/2370816.2370843

[8] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP*. 67–78.

[9] Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri. 2018. NUMASK: High Performance Scalable Skip List for NUMA. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018 (LIPIcs)*, Ulrich Schmid and Josef Widder (Eds.), Vol. 121. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 18:1–18:19.

[10] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *DISC*. 194–208.

[11] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2013. Towards formally specifying and verifying Transactional Memory. *Formal Aspects of Computing* 25, 5 (2013), 769–799. https://doi.org/10.1007/s00165-012-0225-8

[12] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*. 237–246.

[13] MA Gonzalez-Mesa, Eladio Gutierrez, Emilio L Zapata, and Oscar Plata. 2014. Effective Transactional Memory Execution Management for Improved Concurrency. *ACM TACO* 11, 3 (2014), 24.

[14] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *PPoPP*. 175–184.

[15] R. Guerraoui and M. Kapalka. 2011. *Principles of Transactional Memory*. Morgan & Claypool.

[16] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*. 355–364.

[17] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*. 289–300.

[18] Sachin Hirve, Roberto Palmieri, and Binoy Ravindran. 2014. Archie: a speculative replicated transactional system. In *Proceedings of the 15th International Middleware Conference, Bordeaux, France, December 8-12, 2014*. 265–276.

[19] Manos Kapritsos, Yang Wang, Vivien Quéma, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. 237–250.

[20] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.

[21] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI*. 166–176.

[22] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*. 35–46.

[23] Cosmin E. Oancea, Alan Mycroft, and Tim Harris. 2009. A Lightweight In-place Implementation for Software Thread-level Speculation. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*. ACM, New York, NY, USA, 223–232. https://doi.org/10.1145/1583991.1584050

[24] Christos H. Papadimitriou. 1979. The serializability of concurrent database updates. *J. ACM* 26 (October 1979), 631–653. Issue 4.

[25] Manohar K. Prabhu and Kunle Olukotun. 2003. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/781498.781500

[26] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. 2008. Dependence-aware transactional memory for increased concurrency. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41 2008), November 8-12, 2008, Lake Como, Italy*. 246–257.

[27] Hany E Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. 2009. Committing conflicting transactions in an STM. In *ACM Sigplan Notices*, Vol. 44. ACM, 163–172.

[28] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. 2010. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS '10)*. ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/1736020.1736030

[29] Ravi Ramaseshan and Frank Mueller. 2008. Toward thread-level speculation for coarse-grained parallelism of regular access patterns. In *Workshop on Programmability Issues for Multi-Core Computers*. 12.

[30] Lawrence Rauchwerger and David Padua. 1995. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *SIGPLAN Not.* 30 (June 1995), 218–232. Issue 6. https://doi.org/10.1145/223428.207148

[31] Torvald Riegel, Pascal Felber, and Christof Fetzer. 2006. A Lazy Snapshot Algorithm with Eager Validation. In *DISC 2006*. Lecture Notes in Computer Science, Vol. 4167. Springer, 284–298.

[32] Mohamed M. Saad, Masoomeh Javidi Kishi, Shihao Jing, and Roberto Palmieri. 2018. Processing Transactions in a Predefined Order. *CoRR* abs/1812.05727 (2018). arXiv:1812.05727 https://arxiv.org/abs/1812.05727

[33] Mohamed M. Saad, Mohamed Mohamedin, and Binoy Ravindran. 2012. HydraVM: Extracting Parallelism from Legacy Sequential Code Using STM. In *HotPar*.

[34] Mohamed M. Saad, Roberto Palmieri, and Binoy Ravindran. 2016. On ordering transaction commit. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, Rafael Asenjo and Tim Harris (Eds.). ACM, 46:1–46:2. https://doi.org/10.1145/2851141.2851191

[35] Mohamed M. Saad, Roberto Palmieri, and Binoy Ravindran. 2018. Lerna: Parallelizing Dependent Loops Using Speculation. In *Proceedings of the 11th ACM International Systems and Storage Conference, SYSTOR 2018, HAIFA, Israel, June 04-07, 2018*. ACM, 37–48.

[36] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.* 22, 4 (1990), 299–319.

[37] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. 2013. Sambamba: runtime adaptive parallel execution. In *ADAPT*. 7.
[38] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. 2008. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP*. 185–196.
[39] Christoph Von Praun, Luis Ceze, and Calin Caşcaval. 2007. Implicit parallelism with ordered transactions. In *PPoPP*. 79–89.
[40] Lingli Zhang, Vinod K Grover, Michael M Magruder, David Detlefs, John Joseph Duffy, and Goetz Graefe. 2010. Software transaction commit order and conflict management. US Patent 7,711,678.