Check-Wait-Pounce: Increasing Transactional Data Structure Throughput by Delaying Transactions

Lance Lebanoff, Christina Peterson(✉), and Damian Dechev

University of Central Florida, 4000 Central Florida Blvd, Orlando, FL, USA
{lancelebanoff,clp8199}@knights.ucf.edu, dechev@cs.ucf.edu

Abstract. Transactional data structures allow data structures to support transactional execution, in which a sequence of operations appears to execute atomically. We consider a paradigm in which a transaction commits its changes to the data structure only if all of its operations succeed; if one operation fails, then the transaction aborts. In this work, we introduce an optimization technique called *Check-Wait-Pounce* that increases performance by avoiding aborts that occur due to failed operations. Check-Wait-Pounce improves upon existing methodologies by delaying the execution of transactions until they are expected to succeed, using a thread-unsafe representation of the data structure as a heuristic. Our evaluation reveals that Check-Wait-Pounce reduces the number of aborts by an average of 49.0%. Because of this reduction in aborts, the tested transactional linked lists achieve average gains in throughput of 2.5x, while some achieve gains as high as 4x.

1 Introduction

As multi-core machines are becoming the norm, many software developers turn to multi-threaded solutions to increase the execution speed of their applications. Building concurrent programs is difficult, because the programmer needs to have in-depth knowledge of the pitfalls of multi-threaded programming. Concurrent programs are prone to semantic errors, performance bottlenecks, and progress issues such as deadlock and starvation. Therefore, simplifying the task of concurrent programming has become an important challenge.

Concurrent data structures allow users to reap the benefits of concurrency while avoiding the dangers of multi-threaded programming [5,13]. These data structures support a predefined set of operations (e.g. insert, delete, find) such that any execution of concurrent operations is guaranteed to behave as if those operations were executed atomically. While concurrent data structures provide this guarantee for individual operations, the same guarantee does not hold for sequences of operations known as *transactions*. To overcome this issue, programmers often resort to coarse-grained locking, which hinders parallelism.

Published by Springer Nature Switzerland AG 2019
J. Pereira and L. Ricci (Eds.): DAIS 2019, LNCS 11534, pp. 19–35, 2019.
https://doi.org/10.1007/978-3-030-22496-7_2

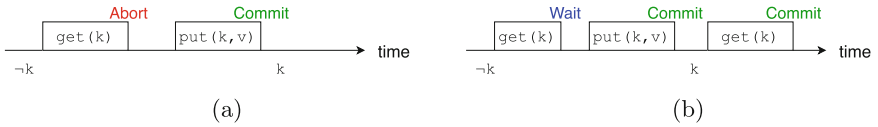


Fig. 1. A scenario in which a transaction experiences a self-abort that is avoidable. In (a), Transaction 1 executes `get(k)` which fails because k does not exist in the data structure. In (b), Transaction 1 avoids this scenario by waiting until Transaction 2 has inserted k into the data structure.

This issue has motivated the development of transactional data structures [7–10,12]. We recognize a data structure to be transactional if it supports *atomicity* and *isolation*. Atomicity means that a transaction may commit its changes to the data structure only if all its operations succeed. Isolation guarantees that transactions executed concurrently will appear to execute in some sequential order. There are several techniques that can be used to create transactional data structures, and we refer to these techniques as *transactional data structure methodolgies* (TDSMs).

We consider a paradigm of transactional data structures in which each operation in a transaction has a defined precondition [18]. For example, in a reservation system, an operation to reserve an item might require that the item has not already been reserved. If an operation's precondition is not satisfied at the beginning of the operation's execution, then the operation fails. A TDSM must abort a transaction if one of its operations fails, in order to preserve atomicity. We refer to this type of abort as a *self-abort*. Self-aborts waste computation time, as time spent executing transactions that will ultimately abort does not contribute to the overall throughput.

By reordering the execution of transactions, we can reduce the number of self-aborts and improve performance. For example, consider the scenario in Fig. 1. In Part (a), two transactions perform operations on a key-value map M. Transaction 1 consists of one operation `get(k)` that reads the value associated with a key k, and the operation's precondition requires that the $k \in M$. When the operation executes, it observes that $k \notin M$, so it fails, and the transaction self-aborts. Then, Transaction 2 executes `put(k,v)` to insert k into M. Part (b) shows that if Transaction 1 had waited until Transaction 2's completion, then its precondition would have been met, and Transaction 1 would avoid a self-abort.

Finding the optimal ordering of transactions would minimize the number of self-aborts. Although this approach might be possible in some cases, the large search space of potential orderings makes this approach computationally expensive [1], possibly more so than the self-aborts themselves.

In this paper, we present an optimization technique called *Check-Wait-Pounce* that reduces the number of self-aborts by delaying transactions. Transactions are analyzed before they execute, and if they are predicted to abort, then they are delayed instead. This results in an ordering of transactions with fewer self-aborts.

The Check-Wait-Pounce algorithm follows three steps. (1) In the *check* step, we heuristically determine the expected chance that a transaction will succeed. We make this prediction based on the transactional data structure's *Likely Abstract State Array* (LASA), which is an auxiliary array introduced in this paper. LASA is a heuristic representation of the transactional data structure's abstract state. A data structure's abstract state refers to the underlying meaning of the data structure. For example, the abstract state of a skip list-based set is a function across the possible range of keys, indicating whether or not the key exists in the data structure. In this case, LASA is an array of bits, where the index of each bit represents a key and the value of each bit represents the presence of that key in the set. When a transaction commits, LASA is updated to match the abstract state of the data structure, but it does so without using costly synchronization mechanisms. Consequently, the operations performed on LASA are not atomic, so it trades accuracy for performance. In Check-Wait-Pounce, the expected chance that a transaction will succeed corresponds to the percentage of operations whose preconditions are satisfied, according to LASA. (2) In the *wait* step, we periodically perform this check until the expected chance of success exceeds a given threshold. While the transaction waits, the thread executes other transactions. (3) When the threshold is reached, we proceed to the *pounce* step, in which we use an underlying TDSM (e.g. Transactional Boosting) to execute the transaction. Check-Wait-Pounce treats the underlying TDSM as a black box.

We employ micro-benchmarks in a variety of test cases to evaluate the effects of our optimization on several transactional data structures, created by four state-of-the-art TDSMs: Lock-free Transactional Transformation (LFTT) [18], Transactional Boosting (TB) [10], Transactional Data Structure Libraries (TDSL) [16], and Software Transactional Memory (STM) [3,6]. The data structures we evaluate are transactional versions of linked lists, skip lists, and multi-dimensional lists created by these TDSMs. With our optimization, the number of self-aborts is reduced by an average of 49.0%. As a result, the transactional linked lists based on LFTT, TB, TDSL, and STM achieve 3.3x, 4.6x, 1.8x, and 41.6% gains in throughput, respectively.

This paper makes the following contributions:

- We present Check-Wait-Pounce, an optimization approach to transactional data structures that reduces the number of aborted transactions. This is achieved by delaying transactions until they are expected to succeed.
- We introduce a new auxiliary data structure called LASA that is used by the Check-Wait-Pounce scheme to heuristically determine a transaction's chance of success.
- Check-Wait-Pounce can be applied to any TDSM. It controls when each transaction should be executed, and then it treats the underlying TDSM as a black box to execute the transaction.

2 Related Work

We use the *strict serializability* correctness condition to verify the correctness of transactional data structures. Strict serializability requires that for each completed transaction, a *serialization point* can be placed between the transaction's first and last operations. A transaction's serialization point is an instantaneous point in time that marks when the transaction effectively occurred. A history of concurrent transactions is strictly serializable if a serialization point can be placed for each transaction to create a sequential history, such that the outcome of the sequential history matches the outcome of the concurrent history.

We present a brief survey of fundamental TDSMs and how they support strict serializability. Then we describe existing techniques that relate to Check-Wait-Pounce because they reorder transactions.

2.1 Transactional Data Structure Methodologies

To guarantee strict serializability, each TDSM employs a unique approach that prevents certain transactions from committing, specifically pairs of transactions that concurrently access the same nodes in the data structure.

Software transactional memory (STM) is a methodology in which each transaction maintains all the memory locations it reads in a *read set* and all the locations that it writes to in a *write set*. If one transaction's write set intersects another transaction's read set or write set, then the transactions *conflict*. One of the conflicting transactions must abort, undoing the changes that it made to the data structure. We refer to this type of abort as a *spurious abort*. Spurious aborts occur as a result of multiple threads that concurrently access the same nodes in the data structure. Transactional Boosting (TB) proposed by Herlihy and Koskinen [10] associates each key in the data structure with a lock. A transaction that performs an operation on a key must first acquire the lock associated with that key. If a transaction fails to acquire a lock, then the transaction spuriously aborts and rolls back completed operations. Lock-free Transactional Transformation (LFTT) proposed by Zhang and Dechev [18] makes each node in the data structure point to a *transaction descriptor object*, which is an object that represents the transaction that last accessed that node. Before a transaction performs an operation on a node, it must help to complete any unfinished transactions that have already accessed that node. Transactional data structure libraries (TDSL) was proposed by Spiegelman et al. [16]. A thread collects a read set and write set, and assigns each node in the read set with a version number. At the end of the transaction, the thread locks all of the nodes in the write set, then checks the version numbers of all the nodes in the read set to validate that they have not been changed. If the thread fails to acquire a lock or the validation fails, then the transaction spuriously aborts.

These TDSMs focus on reducing overhead and spurious aborts, but they do not optimize for use cases in which self-aborts are common. We present Check-Wait-Pounce to optimize these algorithms by reducing the number of self-aborts.

Reducing the number of self-aborts is important because self-aborts waste computation time and do not contribute to the application's overall throughput, similarly to spurious aborts.

2.2 Reordering Transactions

The following techniques reorder the serialization points of conflicting transactions to avoid spurious aborts. Pedone et al. [15] introduced a reordering technique for database transactions that detects conflicts between concurrent transactions and reorders the serialization points of conflicting transactions to remove the conflicts. Chachopo and Rito-Silva [2] proposed an approach for transactional memory that avoids all spurious aborts for read-only and write-only transactions by moving serialization points. Diegues and Romano [4] extend the types of transactions that are reordered to include some read-write transactions.

The technique of reordering serialization points could possibly be applied to the problem of self-aborts. However, all of the TDSMs we study guarantee strict serializability, and to maintain this level of correctness, each transaction's serialization point may only be placed between the invocation of the transaction's first operation and response of the transaction's last operation. Consequently, a serialization point reordering technique may only reorder the serialization points of *concurrent* transactions. This restriction reduces the number of possible orderings of transactions such that the probability of a self-abort being avoided is minuscule. According to our experiments running 64 threads with a micro-benchmark, only 8.6×10^{-4} percent of self-aborts can be avoided by reordering the serialization points of concurrent transactions. On the other hand, our technique of reordering the physical execution of transactions is much more effective, avoiding 49.0% of self-aborts. The details of our experimental setup are given in Sect. 5.

The steal-on-abort technique [1] reorders the physical execution of transactions to prevent spurious aborts. Steal-on-abort has the purpose of reducing the number of spurious aborts, while Check-Wait-Pounce reduces the number of self-aborts. Also, steal-on-abort's method of reordering allows a transaction to execute and abort, *then* it forces the transaction to wait to restart. With Check-Wait-Pounce, we predict whether a transaction will abort *before* it executes in the first place.

3 Check-Wait-Pounce

We provide an overview of Check-Wait-Pounce, followed by detailed descriptions of the algorithm's steps.

3.1 Algorithm Overview

Figure 2 depicts a high-level overview of the life cycle of a transaction in Check-Wait-Pounce. After the transaction is created, we perform the *check step*: we

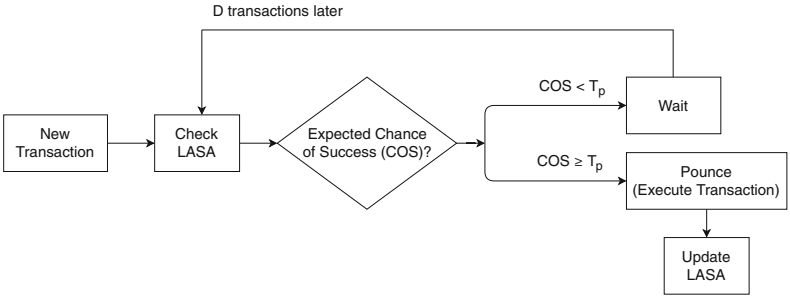


Fig. 2. Transaction life cycle in Check-Wait-Pounce.

determine the transaction's expected chance of success (COS) by checking the Likely Abstract State Array (LASA) of the data structure. LASA is a thread-unsafe representation of the data structure's abstract state. We provide details about LASA and the prediction of a transaction's COS in Sect. 3.2.

A threshold value for the COS called the pounce threshold (T_p) is a user-defined parameter. Based on the COS, we perform one of two actions:

- If the transaction's COS is less than T_p, then the transaction will likely abort if executed immediately. In this case, we proceed to the *wait step*: we delay the transaction's execution, allowing D other transactions to be processed in the meantime (D is an integer provided as an input parameter). After D other transactions complete, the waiting transaction returns to the check step to determine its new COS. We hope that after waiting, the transaction will have a higher chance of success than it had before waiting.
- If the transaction's COS is greater than or equal to T_p, then the transaction will likely commit if executed immediately. In this case, we proceed to the *pounce step*: we execute the transaction. If the transaction commits, then we update LASA to reflect the changes to the data structure's abstract state.

3.2 Algorithm Details

In this paper, we focus on applying Check-Wait-Pounce to transactional data structures that implement the set and map abstract data types. In the case of sets and maps, the available operations are INSERT, DELETE, and FIND.

For clarity, we list the data type definitions and constants of Check-Wait-Pounce in Listing 1. Note, we denote line X of Listing Y as line Y.X.

The TRANSACTION object represents a single transaction. It maintains a list of OPERATION objects, which are used in the check step to determine the transaction's chance of success. Each TRANSACTION object counts *numWaits*, the number of times the transaction has performed the wait step, as well as *waitEndTime*, which is a timestamp indicating when the transaction should stop waiting. Also, the TRANSACTION object has a reference *next* pointing to the next transaction in the wait list. These fields will be explained further in this section.

Listing 1. Type Definitions

```
1  enum OpType
2  |   Insert;
3  |   Delete;
4  |   Find;
5  struct Operation
6  |   OpType type;
7  |   int key;
8  struct Transaction
9  |   Operation[] ops;
10 |   int numWaits ← 0;
11 |   int waitEndTime;
12 |   Transaction* next = NULL;
13 struct TxnWaitList
14 |   int timestamp ← 0;
15 |   Transaction head ← new Transaction();
16 |   Transaction tail ← new Transaction();
```

The TxnWaitList object is a thread-local queue that facilitates the wait step. It is implemented as a linked list, and it maintains an integer *timestamp* to track when transactions have finished waiting.

Each thread is given a stream of transactions to process. We show the procedure that the thread performs on each transaction in Algorithm 2. First, the thread calls the CheckWaitPounce function to process the transaction from the stream (line 2.2). We predict the transaction's COS and proceed to either the wait step or the pounce step. After processing the transaction from the stream, we re-process any transactions that have reached the end of their wait steps (line 2.3). We describe these functions in detail later in this section.

Check Step. The CheckWaitPounce method begins with the check step, in which we predict the transaction's chance of success (COS). This prediction is made based on the number of operations in the transaction that will succeed. Each operation in the transaction has a given precondition. If that precondition is satisfied, then the operation succeeds; otherwise, the operation fails. For a set S and the operation Insert(k) on a given key k, the operation's precondition is $k \notin S$. Conversely, Find(k) and Delete(k) require $k \in S$.

We predict whether or not each operation will succeed based on the LASA auxiliary data structure. LASA represents the abstract state of the transactional data structure, and its implementation may vary for different data structures. For a set, the abstract state is a list of keys that exist in the set. We implement LASA as a bitmap, where each index i represents a key k that could possibly exist in the set, and the LASA[i] is true if $k \in S$, otherwise false. This boolean array representation allows for fast constant-time traversal while keeping memory usage low. In the case of a vastly large key range, LASA can be converted from an array for bits to a hash set or bloom filter to further decrease memory usage.

First, we count the number of operations that are expected to succeed. For each operation, we compare its precondition to the data structure's abstract state, represented by LASA (line 2.14). If they match, then we predict the operation will succeed; otherwise it will fail. Next, we calculate the transaction's chance of success (COS), which is equal to the ratio of successful operations to total operations (line 2.17). The transaction's next step is chosen based on the relation between COS and the pounce threshold (T_p), as detailed in Sect. 3.1.

Algorithm 2. Check-Wait-Pounce

```
1  Function ProcessTxn(Transaction txn)
2      CheckWaitPounce(txn);
3      ProcessWaitingTxns();
4  end
5  Function CheckWaitPounce(Transaction txn)
6      //Check step
7      int successfulOps ← 0;
8      foreach op ∈ txn.ops do
9          bool precondition;
10         if op.type = Insert then
11             precondition ← False;
12         else if op.type = Delete || op.type = Find then
13             precondition ← True;
14         if LASA[op.key] = precondition then
15             successfulOps++;
16     end
17     float COS ← successfulOps / txn.ops.length;
18     if COS < T_p and txn.numWaits < MAX_WAITS then
19         //Wait step
20         if TWL.head = NULL then
21             TWL.head ← txn;
22             TWL.tail ← txn;
23         else
24             TWL.tail.next ← txn;
25         txn.waitEndTime ← TWL.timestamp + D;
26         txn.numWaits++;
27     else
28         //Pounce step
29         if TDSM.ExecuteTxn(txn) = True then
30             foreach op ∈ txn.ops do
31                 if op.type = Insert || op.type = Find then
32                     LASA[op.key] ← True;
33                 else if op.type = Delete then
34                     LASA[op.key] ← False;
35             end
36 end
37 Function ProcessWaitingTxns()
38     Transaction txn ← TWL.head.next;
39     while txn ≠ NULL and txn.waitEndTime = TWL.timestamp do
40         CheckWaitPounce(TWL.head);
41         txn ← txn.next;
42     end
43     TWL.head.next ← txn;
44     TWL.timestamp++;
45 end
```

Wait Step. If a transaction's COS is less than T_p, then it proceeds to the wait step. Two parameters are given by the user to tune the wait step. D represents the amount of time that each transaction is delayed in the wait step, measured by the number of other transactions that are processed by the calling thread during the transaction's wait step. *MAX_WAITS* places a bound on the number of times that a transaction enters the wait step to avoid situations in which a transaction waits indefinitely. If a transaction has entered the wait step more than *MAX_WAITS* times, then it proceeds to the pounce step (line 2.18).

In the common case, we add the transaction to the Transaction Wait List (TWL), at the tail of the queue (line 2.25) or the head if the queue is empty (line 2.21). The transaction waits until D other transactions have been processed. To achieve this, the transaction calculates its *wait end time*—the specific *timestamp* value in which the transaction should finish waiting, which is equal to the current *timestamp* value plus D (line 2.20).

The ProcessWaitingTxns function is called each time a thread processes a transaction from the stream as in Algorithm 2 (line 2.3). This function dequeues transactions from TWL if they have reached their *wait end time* and returns them to the check step (lines 2.35−2.42)

Pounce Step. Once Check-Wait-Pounce chooses the point in time to execute the transaction, we use a transactional data structure methodology (TDSM) to actually perform the execution (line 2.28). This underlying TDSM is treated as a black box to handle the conflict management that ensures strict serializability. The TDSM returns true if the transaction commits, or false if the transaction aborts. If the transaction commits, then its operations take effect, so we must update LASA to match the data structure's new abstract state (lines 2.30–2.33).

Note that LASA is shared among all threads, yet Check-Wait-Pounce uses simple read and write instructions when dealing with LASA. Consequently, LASA is not thread-safe. As a result, multiple threads performing concurrent updates to LASA might encounter a data race and cause LASA to incorrectly reflect the data structure's abstract state. Because of the possibility of such disparities, we only use LASA as a heuristic to choose the point in time to execute a transaction, rather than using it to actually perform the execution.

4 Correctness

We use the correctness condition *strict serializability* for our correctness discussion. The four TDSMs we focus on in this paper—LFTT, STM, TDSL, and TB—all guarantee strict serializability. TDSL guarantees opacity, which is a stricter correctness condition, so our correctness proof holds for TDSL as well. First, we provide background definitions for strict serializability, then we prove that Check-Wait-Pounce does not alter the correctness of the strictly serializable TDSMs.

A *transaction* is a sequence of operations that the user desires to be executed atomically. An *event* is (1) a transaction invocation (the start of a transaction)

or response (the end of a transaction), or (2) an operation invocation or response. A *history* is a finite series of instantaneous events [11].

Definition 1. *A history h is strictly serializable if the subsequence of h consisting of all events of committed transactions is equivalent to a legal history in which these transactions execute sequentially in the order they commit [14].*

Definition 2. *Two method calls I, R and I',R' commute if for all histories h, if $h \cdot I \cdot R$ and $h \cdot I' \cdot R'$ are both legal, then $h \cdot I \cdot R \cdot I' \cdot R'$ and $h \cdot I' \cdot R' \cdot I \cdot R$ are both legal and define the same abstract state.*

Definition 3. *For a history h and any given invocation I and response R, let I^{-1} and R^{-1} be the inverse invocation and response. Then I^{-1} and R^{-1} are the inverse operations of I and R such that the state reached after the history $h \cdot I \cdot R \cdot I^{-1} \cdot R^{-1}$ is the same as the state reach after history h.*

Definition 4. *A method call denoted $I \cdot R$ is disposable if, $\forall g \in G$, if $h \cdot I \cdot R$ and $h \cdot g \cdot I \cdot R$ are legal, then $h \cdot I \cdot R \cdot g$ and $h \cdot g \cdot I \cdot R$ are legal and both define the same state.*

4.1 Rules

Any software transactional memory system that obeys the following correctness rules is strictly serializable [10].

Rule 1 ***Linearizability:*** *For any history h, two concurrent invocations I and I' must be equivalent to either the history $h \cdot I \cdot R \cdot I' \cdot R'$ or the history $h \cdot I' \cdot R' \cdot I \cdot R$.*

Rule 2 ***Commutativity Isolation:*** *For any non-commutative method calls $I_1, R_1 \in T_1$ and $I_2, R_2 \in T_2$, either T_1 commits or aborts before any additional method calls in T_2 are invoked, or vice-versa.*

Rule 3 ***Compensating Actions:*** *For any history h and transaction T, if $\langle T \text{ aborted} \rangle \in h$, then it must be the case that $h—T = \langle T \text{ init} \rangle \cdot I_0 \cdot R_0 \cdots I_i \cdot R_i \cdot \langle T \text{ aborted} \rangle \cdot I_i^{-1} \cdot R_i^{-1} \cdots I_0^{-1} \cdot R_0^{-1} \cdot \langle T \text{ aborted} \rangle$ where i indexes the last successfully completed method call.*

Rule 4 ***Disposable Methods:*** *For any history h and transaction T, any method call invoked by T that occurs after $\langle T \text{ commit} \rangle$ or after $\langle T \text{ abort} \rangle$ must be disposable.*

4.2 Strict Serializability and Recovery

We now show that Check-Wait-Pounce satisfies the correctness rules required to guarantee strict serializability. The concrete state of a map is denoted as a node set N.

Lemma 1. *The set operations* Insert, Delete, *and* Find *are linearizable, satisfying Rule 1.*

Proof. It is assumed that the underlying TDSM is strictly serializable. It is therefore guaranteed that any history generated by the TDSM is equivalent to a legal history in which these transactions execute sequentially in the order they commit, so they are linearizable.

Lemma 2. *Check-Wait-Pounce satisfies commutativity isolation as defined in Rule 2.*

Proof. Two set operations commute if they access different keys. The one-to-one mapping from nodes to keys is formally stated as $\forall n_x, n_y \in N, x \neq y \implies n_x \neq n_y \implies n_x.key \neq n_y.key$. This implies that two set operations commute if they access different nodes.

Since a transaction is only executed by the underlying TDSM, then Check-Wait-Pounce satisfies commutativity isolation if the underlying TDSM satisfies commutativity isolation.

Lemma 3. *When a transaction aborts, Check-Wait-Pounce ensures that the resulting history is equivalent to performing the inverse operations of all computed operations of the aborted transaction, satisfying Rule 3.*

Proof. Let T denote a transaction that executes the operations $I_0 \cdot R_0 \cdots I_i \cdot R_i$ on nodes $n_0 \cdots n_i$ and then aborts. Let S_0 denote the abstract state immediately before I_0. By Rule 3, T must execute the inverse operations of the successful method calls $I_i^{-1} \cdot R_i^{-1} \cdots I_0^{-1} \cdot R_0^{-1}$ after those method calls have succeeded. This is equivalent to requiring that the current abstract state S_i be restored to its original state S_0. We prove that the current abstract state S_i is restored to its original state S_0 following an aborted transaction.

In the pounce step, the transaction is executed by the underlying TDSM. Since the TDSM is assumed to be strictly serializable, it follows that the partial effects of an aborted transaction are rolled back to the original abstract state S_0 to guarantee that the resulting history is equivalent to a legal history. Therefore, when the TDSM aborts a transaction, $S_i = S_0$.

Lemma 4. *The LASA update operation is disposable, so Check-Wait-Pounce satisfies Rule 4.*

Proof. After a transaction executed by the underlying TDSM commits, LASA is updated using atomic reads and atomic writes to reflect the expected abstract state based on the operations performed by the transaction. We prove that the LASA update operation is disposable by showing that it does not change the abstract state of the data structure. LASA affects the outcome of the check step. Since the transactional execution by the underlying TDSM does not incorporate LASA, the LASA update operation does not change the abstract state of the data structure, making it disposable.

Theorem 1. *For a data structure that is generated using Check-Wait-Pounce, the history of committed transactions is strictly serializable.*

Proof. Follow Lemmas 1, 2, 3, 4, and the main theorem of Herlihy et al.'s work [10], the theorem holds.

Table 1. Experimental variables tested.

Variable	Values tested
Data structure	Linked list, Skip list, MDList-based dictionary [17]
TDSM	LFTT, TB, TDSL, STM
Transaction size (# operations)	1, 2, 4, 8, 12, 16
Sleep between operations	0 µs, 10 µs, 100 µs, 1 ms
T_p	0, 0.25, 0.5, 0.75, 1
D	2, 50, 100, 300
MAX_WAITS	2, 50, 100, 300
CPU architecture	Intel Xeon Platinum 8160, SMP, 24 cores @ 2.1 GHz,
	AMD Opteron 6272, NUMA, 64 cores @ 2.1 GHz

5 Evaluation

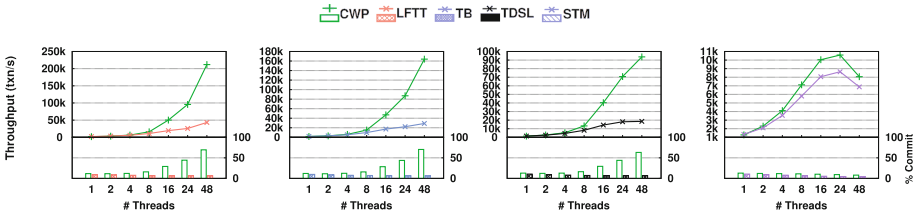
We compare the performance of several transactional data structures created using four different TDSMs, and evaluate the performance impact of Check-Wait-Pounce when applied to each data structure.

5.1 Experimental Setup

To evaluate the performance impact of Check-Wait-Pounce, we use a micro-benchmark in a similar manner to other evaluations of TDSMs [3,16]. Several threads are spawned, each one continuously executing transactions for 5 s. Each operation in a transaction is randomly assigned an operation type (INSERT, DELETE, or FIND) and a key. All code is compiled with GCC 7.3 with C++17 features and O3 optimizations.

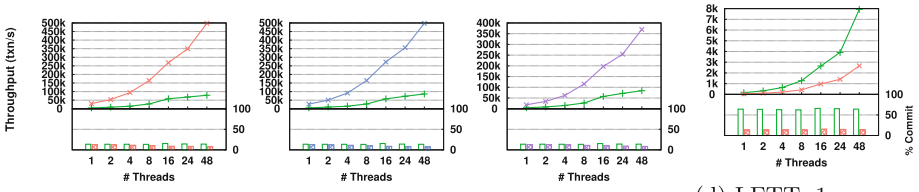
We perform our experiments in a variety of scenarios, outlined in Table 1. We compare the performance of three concurrent data structures made transactional by four TDSMs. We observe the effect of Check-Wait-Pounce on these data structures in different environments, such as the transaction size (the number of operations per transaction), user-defined parameters, and CPU architectures. We also perform tests in which threads execute transactions with different amounts of extra work in between each data structure operation. This means that the number of data structure operations remains the same (e.g. 4 operations) but the time taken to execute each transaction increases. We simulate these kinds of transactions by tasking the threads to sleep for a certain amount of time per operation.

When evaluating STM, we test Fraser STM [6] for the skiplist and NOrec [3] for the other data structures. We denote Check-Wait-Pounce as CWP for the remainder of this section.



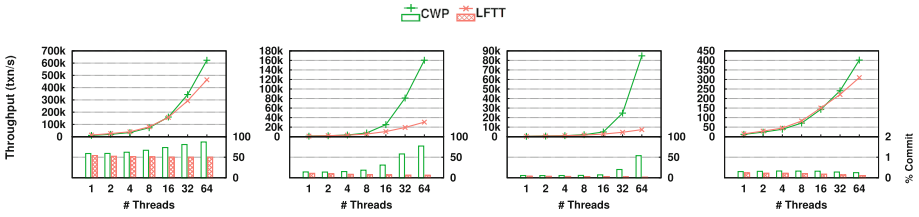
(a) LFTT (b) TB (c) TDSL (d) STM

Fig. 3. Linked list performance: throughput and commit rate.



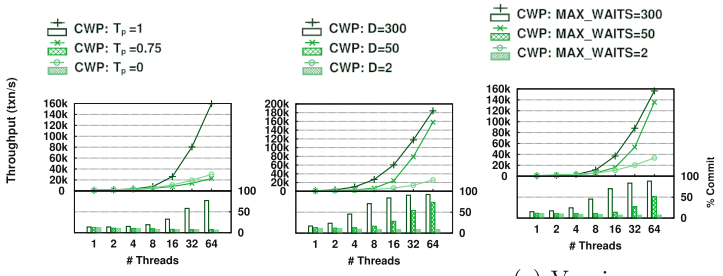
(a) LFTT (b) TB (c) STM (d) LFTT: 1 ms sleep per operation

Fig. 4. Skip list performance: throughput and commit rate.



(a) 1 operation (b) 4 operations (c) 8 operations (d) 16 operations

Fig. 5. Comparison using different transaction sizes (number of operations per transaction).



(a) Varying T_p (b) Varying D (c) Varying *MAX_WAITS*

Fig. 6. Linked list performance varying user-defined parameters.

5.2 Linked List

We show the throughput and commit rate for the transactional linked lists in a standard environment in Fig. 3. Each graph displays the performance of one TDSM compared to the performance of that TDSM when optimized by Check-Wait-Pounce. The throughput is the number of committed transactions per second. The commit rate is the percentage of transactions that commit.

For the linked lists, the key range is set to 10^4. The tests are run on the Intel Xeon Platinum 8160 using a uniform distribution, 20% write operations, transaction size of 4, T_p set to 1, D set to 50, and MAX_WAITS set to 100.

In almost every scenario across these test cases, CWP significantly increases the percentage of committed transactions of the original TDSM and enhances the throughput. On average, CWP improves the commit rate of LFTT by an average of 62.5%, TB by 61.9%, TDSL by 57.8%, and STM by 14.1%. CWP improves the throughput of LFTT by an average of 3.3x, TB by 4.6x, TDSL by 1.8x, and STM by 41.6%.

STM does not experience such a large gain in throughput as the other TDSMs. This can be explained by the percentage of commits; with CWP, STM increases its commit rate by 14.1%, while the other three TDSMs increase their commit rates by an average of 60.7%. CWP helps to avoid self-aborts, but one of the main disadvantages of STM is its high number of spurious aborts. Consequently, CWP achieves smaller gains in commit rate, resulting in smaller gains in throughput.

CWP improves both the throughput and commit rate when the number of threads increases, due to increased activity per node. In the case of a higher number of threads, during the wait step of a transaction T, more transactions execute, increasing the chance that T will succeed when it finishes waiting.

5.3 Skip List

The performance of the transactional skip lists in a standard environment is shown in Fig. 4. For the skip lists, we set the key range to 10^6, as the logarithmic nature of traversal for skip lists allows them to handle larger key ranges than linked lists. All other variables are set in the same manner as the linked lists in Sect. 5.2. The performance results of the MDList-based dictionary is similar to those of the skip list, so we do not show its results.

In every scenario across these test cases, CWP severely degrades the throughput of the skip list. On average for all the TDSMs, CWP reduces the throughput of skip lists by 79.4%. Although CWP increases the commit rate by an average of 9.8%, this increase is offset by the overhead of reading and writing to LASA. Because traversal in a skip list takes logarithmic time in comparison to the number of nodes rather than linear time, each operation completes much faster and is more harshly affected by the overhead of LASA. This finding leads us to postulate that CWP is more effective at increasing the throughput of transactions that take more time to execute.

We support this hypothesis by performing tests in which the threads are tasked to sleep for a certain amount of time before each operation. In Fig. 4d we show the performance of the LFTT skip list with 1 ms of sleep per operation. We tested sleep times of 10 and 100 µs as well but do not display these results for space. The results show that CWP improves the performance of the data structure more for cases in which transactions take more time to execute. In the case of 1 ms of sleep, CWP improves the throughput of the skip list by 194%.

5.4 Transaction Size

In Fig. 5, we compare the effects of CWP on transactions of different sizes. Before each test run, we fill the data structure until it is 50% full of nodes, and then the number of insert and delete operations are equal during the test. Under these circumstances, the probability of success for each operation is 0.5, so an increased transaction size results in a lower commit rate. Namely, a transaction with size n has a probability of $1/2^n$ to commit.

Our results indicate that CWP performs more effectively for higher transaction sizes until a size of 8, and then its effectiveness declines for higher sizes. For sizes lower than 8, each transaction has a relatively high chance of succeeding, so CWP does not improve the commit rate drastically. For sizes higher than 8, each transaction has such a low chance of succeeding that it needs to wait a high number of wait steps before it succeeds, often greater than the value of *MAX_WAITS*, which also reduces the effectiveness of CWP.

5.5 Check-Wait-Pounce Parameters

We vary the user-defined parameters for CWP: T_p, D, and *MAX_WAITS*. The results are shown in Fig. 6. In Fig. 6a, we see that CWP is most effective when T_p is set to 1, which signifies that 100% of the operations in a transaction must be predicted to succeed in order to proceed to the pounce step. If a lower value of T_p is used, transactions with any fail-prone operations are allowed to proceed to the pounce step, and they usually abort, which hurts performance.

In Fig. 6b and c, we see that increasing the values of D and *MAX_WAITS* improves the throughput and commit rate of CWP. However, increasing these parameters leads to higher latency, as CWP allows transactions to wait for longer periods of time before executing. For applications that tolerate high latency, D and *MAX_WAITS* can be set to high values.

6 Conclusion

We present an optimization to transactional data structures called Check-Wait-Pounce that reduces the number of self-aborts by delaying transactions. In test cases with linked lists, our optimization improves the throughput of the data structure by an average of 2.5x. Our optimization uses a thread-unsafe heuristic

called a Likely Abstract State Array to predict the chance of success of a transaction. Based on our findings, the use of thread-unsafe heuristics for concurrent data structures is promising and can be the focus of future work.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. 1717515 and Grant No. 1740095. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

1. Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C., Watson, I.: Steal-on-abort: improving transactional memory performance through dynamic transaction reordering. In: Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 4–18. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-92990-1_3
2. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. **63**(2), 172–185 (2006)
3. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: streamlining STM by abolishing ownership records. In: ACM Sigplan Notices, no. 5. ACM (2010)
4. Diegues, N., Romano, P.: Time-warp: lightweight abort minimization in transactional memory. In: Symposium on Principles and Practice of Parallel Programming (2014)
5. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 131–140. ACM (2010)
6. Fraser, K.: Practical lock-freedom. Ph.D. thesis, Cambridge University Computer Laboratory (2003). Also available as Technical report UCAM-CL-TR-579 (2004)
7. Golan-Gueta, G., Ramalingam, G., Sagiv, M., Yahav, E.: Automatic scalable atomicity via semantic locking. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 31–41. ACM (2015)
8. Gramoli, V., Guerraoui, R., Letia, M.: Composing relaxed transactions. In: Parallel & Distributed Processing (IPDPS), pp. 1171–1182. IEEE (2013)
9. Hassan, A., Palmieri, R., Ravindran, B.: On developing optimistic transactional lazy set. In: Aguilera, M.K., Querzoni, L., Shapiro, M. (eds.) OPODIS 2014. LNCS, vol. 8878, pp. 437–452. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14472-6_29
10. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 207–216. ACM (2008)
11. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, Burlington (2011)
12. Koskinen, E., Parkinson, M., Herlihy, M.: Coarse-grained transactions. ACM Sigplan Not. **45**(1), 19–30 (2010)
13. Lindén, J., Jonsson, B.: A skiplist-based concurrent priority queue with minimal memory contention. In: Baldoni, R., Nisse, N., van Steen, M. (eds.) OPODIS 2013. LNCS, vol. 8304, pp. 206–220. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03850-6_15

14. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM (JACM) **26**(4), 631–653 (1979)
15. Pedone, F., Guerraoui, R., Schiper, A.: Transaction reordering in replicated databases. In: Proceedings of the IEEE Symposium on Reliable Distributed Systems, pp. 175–182 (1997)
16. Spiegelman, A., Golan-Gueta, G., Keidar, I.: Transactional data structure libraries. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 682–696. ACM (2016)
17. Zhang, D., Dechev, D.: An efficient lock-free logarithmic search data structure based on multi-dimensional list. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), pp. 281–292, June 2016. https://doi.org/10.1109/ICDCS.2016.19
18. Zhang, D., Dechev, D.: Lock-free transactions without rollbacks for linked data structures. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, pp. 325–336. ACM, New York (2016)