Bumper: Sheltering Transactions from Conflicts

Nuno Diegues and Paolo Romano
ndiegues@gsd.inesc-id.pt romano@inesc-id.pt
INESC-ID/IST, Lisbon, Portugal

***Abstract*—This paper addresses the issue of maximizing the efficiency and scalability of distributed transactional platforms, by introducing Bumper, a set of innovative techniques to minimize aborts of transactions in high-contention scenarios. At its core, Bumper relies on two key ideas: (1) sparing update transactions from spurious aborts when they access concurrently updated data, by attempting to serialize them in the past via a novel distributed concurrency control scheme that we call Distributed Time-Warping (DTW); and (2) avoiding aborts due to contention hot spots (that cannot be tackled by DTW) via a novel programming abstraction, called delayed actions, which allows to efficiently serialize, in an abort-free fashion, the execution of conflict-prone data manipulations.**

The techniques used in Bumper can be applied to a wide variety of transactional replication protocols to enhance their performance in contention intensive workloads. In this paper we show how they can be integrated with SCORe, a recent, highly-scalable genuine partial replication protocol. By means of an extensive evaluation using well-known benchmarks and a cluster of 160 nodes, we show that Bumper can boost performance up to 3x in conflict-intensive workloads, while imposing negligible (2.5%) overheads in uncontended scenarios.

I. Introduction

The advent of the cloud computing paradigm has empowered programmers with the ability to scale out their applications easily to hundreds of nodes. However, developing applications capable of effectively exploiting the computational capabilities of large scale distributed cloud platforms is far from being a trivial task. This is a consequence of the design choices characterizing the first generation of distributed data platforms (DTPs) for the cloud [1]. These typically maximize scalability by adopting very weak consistency models, such as eventual consistency [1]. By relaxing consistency, these systems have been shown to achieve impressive scalability levels. However, they also shift additional burden from the platform architects to the application developers, who are exposed to the idiosyncrasies associated with concurrency and failures. Indeed, the inherent complexity of building applications on top of weakly consistent systems has motivated a flurry of works aimed to enforce strong consistency semantics in large scale distributed platforms [2]–[5]. By relying on multi-version concurrency control algorithms, these solutions [2], [4], [5] allow for a very efficient management of read-only transactions, sparing them from the possibility of aborting as well as from the costs of any validations. Another key property ensured by these systems, and aimed to maximize their scalability is the, so called, *genuine* partial replication, according to which the execution of a transaction can only involve nodes that replicate data items it accessed [6].

Problem. As we will also show later in the paper, the actual scalability of these systems can be critically challenged by conflict-prone workloads. The key factors constraining the scalability of these systems are of a twofold nature. More precisely, they are related both to the algorithms used to regulate concurrency among transactions, as well as to the actual degree of parallelism admitted by the applications deployed over them:

- State of the art DTPs rely on overly conservative validation schemes that abort an update transaction whenever any of its reads is no longer up to date by the time it requests to commit. This mechanism gained wide adoption because it can be implemented efficiently. On the other hand, we note that it does not represent a sufficient condition to detect non-serializable histories [7], and, as we will show, it can induce a high number of spurious (i.e., unnecessary) aborts.

- It is well-understood that the maximum degree of parallelism (and hence, of scalability) admitted by any transactional system is deeply affected by the data access patterns exhibited by the applications deployed over them [8]. For instance, several standard OLTP transactional profiles are characterized by contention hot spots, i.e. frequently updated data items. Transactions accessing such data items are not only inherently non-parallelizable; they are also prone to undergo repeated aborts, which can have detrimental effects on the system's throughput and user-perceived responsiveness.

Contributions. We address the issues discussed above by introducing Bumper: a set of mechanisms aimed to shelter transactions from conflicts, thus enabling scalable performance in conflict-prone scenarios while ensuring strong-consistency (1-copy serializability). At its core, Bumper relies on two novel mechanisms to prevent different types of conflicts that would lead to transactions' abort in state of the art strongly consistent DTPs: *distributed time-warping* (time-warping for the sake of brevity) and *delayed actions*.

The idea at the basis of time-warping is to allow an update transaction T, which observes a stale read because

This work was supported by national funds through FCT (Fundação para a Ciência e Tecnologia) under project PEst-OE/EEI/LA0021/2013, and by Cloud-TM project (co-financed by the European Commission through the contract no. 57784).

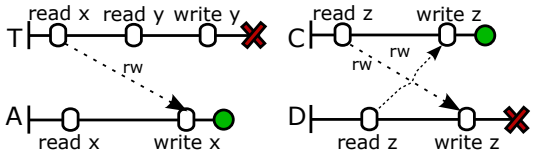

Figure 1: Executions that exemplify conflicts detected by typical strongly consistent transactional systems.

of a write issued by a concurrently committed transaction A, to be serialized before A. This is in contrast with the approach taken by strongly consistent DTPs [4], [5], [9], which only allow the commit of update transactions in the logical present, meaning the snapshot observed by A must be valid by taking into account every transaction committed before A. Using the execution in Fig. 1, we see that this conflict detection would lead to the (spurious) abort of T. Conversely, in such a scenario, we can safely serialize T before A by time-warping it, and thus sparing its abort.

A key property of time-warping consists in its efficiency. From a theoretical perspective, it is straightforward to design an algorithm capable of accepting every serializable history: it suffices to track the full graph of dependencies between *every* transaction and to ensure its acyclicity [10]. This would be an unbearably onerous approach, especially in a large scale DTP. Conversely, time-warping uses a novel, lightweight validation mechanism, which tracks only *direct* local dependencies developed by each transaction during its execution (such as the ones shown in Fig. 1). Not only does this mechanism prevent spurious aborts that would be caused by the validation schemes employed by traditional systems; it can also be implemented very efficiently and in a "genuine" fashion (by only collecting information at nodes that are involved in the distributed transaction).

There are limits to the abort scenarios that time-warping can avoid. An example is shown in Fig. 1, where transactions C and D read and write the same data item z. Since they mutually miss each other's write, neither can be time-warp committed and serialized before the other. To cope with these challenging conflict patterns, we introduce a programming abstraction, complementary to time-warping mechanism, which we name *delayed action*: this is a code fragment to be executed transactionally, but whose side-effects/outputs are not observed elsewhere in the encompassing transaction. By allowing programmers to wrap conflict-prone code within delayed action, Bumper can postpone their execution until the transaction's commit, and guarantee that the snapshot they observe can never be invalidated by a concurrent transaction. This allows ensuring that a delayed action cannot cause the abort of its encompassing transaction, while guaranteeing that it is atomically executed in the scope of the transaction that triggered it.

The key ideas at the basis of Bumper (i.e., time-warping and delayed actions) are applicable to a wide variety of DTPs, such as SCORe [5], P-Store [6], Spanner [2] or D^{2}STM [9]. In this paper, we demonstrate their practicality by integrating them with SCORe [5], a state of the art highly scalable DTP that employs genuine partial replication and a fully decentralized distributed MVCC algorithm. By means of an extensive experimental evaluation using 4 well-known benchmarks and a cluster of 160 nodes, we show that Bumper can boost performance up to $3\times$ in conflict-intensive workloads, while imposing negligible (2%) overheads in uncontended scenarios.

The structure of this paper is as follows. We start by overviewing the related work in Section II. We present our system model in Section III. Section IV presents the mechanisms at the base of Bumper. We discuss the correctness of our solution in Section V and present our evaluation in Section VI. Finally, Section VII concludes the paper.

II. Related Work

We have recently developed the idea of time-warping in the context of a shared memory (i.e., non-distributed) transactional system [11]. To achieve reduction of conflicts in that setting we exploited the availability of a shared global clock to develop an efficient validation procedure. However, applying that core idea in a partially replicated transactional system required a complete redesign of the concurrency algorithm developed for shared memory systems, to cope with two major challenges: i) the lack of a shared global clock allowing processes to agree on relative ordering of transactions, and ii) ensuring that the management of the metadata used in the time-warp abstraction respected the genuine nature of the underlying partial replication protocol, and hence preserved its scalability.

The concept of time-warping was introduced in a different context through the Virtual Time abstraction of Jefferson [12] to consistently roll-back a stale process to a safe global state. Here, instead, the time-warp commit mechanism is used to inject "back in time" the versions produced by a transaction that observed an obsolete snapshot; this is done with the ultimate goal of reducing aborts.

Distributed transactional schedulers [13] can be seen as oracles that use the current history to determine when a transaction should be allowed to execute. Broadly, a transactional scheduler determines the ordering of transactions so that conflicts are either avoided altogether or minimized. Time-warping, instead, reconciles conflicting transactions to yield serializable histories, and can be used orthogonally to any a priori scheduling strategy.

Database replication has also been widely researched. In the last decade, several works have studied full replication techniques, typically layered on top of Total-Order primitives [14], [15]. In particular, in [16], a technique for deterministically reordering totally-order delivered transactions was proposed to also reduce conflicts. More recently,

a number of works have proposed partial replication techniques that are more scalable. Walter [3] embraces a relaxed isolation level (weaker than snapshot isolation) to safely ignore serialization conflicts in a geo-replicated system. S-DUR [17] also proposes a technique to scale conflict-prone workloads — their approach minimizes the communication steps and coordination of replicas, which are similar goals of partial replication. None of these solutions overcome the inherent conflicts in the workloads, and thus their approaches can be complemented with Bumper to boost performance.

Centralized database systems have also been proposed around the work of SSI [18], with different validation schemes for conflict reduction, and assuming that the underlying system ensures Snapshot Isolation [19]. The concept of triads, presented in our work to support distributed time-warp, is similar in spirit to the dangerous structure used in SSI. Recent work has applied these ideas to a distributed database with full replication [20]. Our work is substantially different from those solutions by exploiting partial replication and striving for scalability at much larger scales.

Finally, the work of per-tx boxes [21] has also been proposed to reduce contention hot spots, and are similar in spirit to delayed actions, in that they also delay the execution of conflict-prone code until commit time. Yet, per-tx boxes were proposed for shared-memory and rely on a shared commit lock. In Bumper we consider a different, more challenging system model, i.e. a shared-nothing cluster.

III. System Model and assumptions

We consider a classical asynchronous distributed system model consisting of a set of processes (also called nodes) $\prod = p_1, ..., p_n$ that communicate via message passing and can fail according to the fail-stop (crash) model. We assume a simple key-value model for data. For each data item k there is a chain of versions as in typical MVCC schemes [10]. Each version of k is a tuple $\langle k, val, ts \rangle$, where k is the key, val is its value and ts is a scalar derived from a logical clock that totally orders the chain of k.

We also assume partial replication, in which node p_i stores only a partial copy of the key-value storage. Each data item is replicated by r processes, and we assume that among any r replicas there exists at least one that is correct (i.e., does not crash). We denote with $owners(d)$ the set of processes that replicate d. The considered model allows us to rely on consistent hashing distribution policies [22] to avoid the need for a directory lookup to compute $owners(d)$. We also denote $participants(T)$ as $\bigcup_{d \in \mathcal{F}} owners(d)$ where $\mathcal{F} = writeSet(T) \cup readSet(T)$ (the footprint of T).

We model transactions as a sequence of read and write operations on data items, preceded by a begin, and followed by a commit or abort operation. We assume no blind writes [10], i.e. every write to some k by a transaction is preceded by a read of k in that transaction. A transaction T originates on a process $p_i \in \prod$, which we call $origin(T)$, and can read/write any data item (even if not replicated locally). We additionally define two relations between transactions (similar to those in [23]). We say that transaction T $reads\text{-}from$ A when A writes to k, commits, and then T reads k and obtains the version installed by A (we denote this by $A \rightarrow T$). Transaction B $misses$ transaction T when B reads k and, during the execution of B, transaction T commits a new version for k (we denote this by $B \dashrightarrow T$). Finally, processes communicate via reliable FIFO-ordered channels, i.e. every message sent is eventually received in the send order as long as both the sender and receiver are correct.

IV. Bumper

We describe Bumper in the next sections by delving into our conflict reduction mechanisms. We first discuss how Bumper can be applied to SCORe, a highly scalable state of the art protocol that provides 1-copy-serializability and abort-free read-only transactions. By integrating Bumper with SCORe, we can obtain a protocol that excels both in the management of read-dominated and write-dominated workloads. Before introducing Bumper's mechanisms, we give an overview of SCORe, which is aimed at providing a background sufficient to understand how Bumper can be applied to it. Next, Section IV-B establishes the principles of distributed time-warping transactions and how Bumper implements them. Then, we present delayed actions to bypass hot spots of contention in Section IV-C. Finally, we hint how Bumper may be integrated with alternative transactional replication protocols in Section IV-D.

A. A primer on SCORe

We briefly overview SCORe [5], the baseline protocol chosen to present Bumper. Similarly to MVCC protocols, in SCORe, each node n_i maintains multiple versions (per data item) in a chain, where each version is tagged with a scalar timestamp used to totally order the commit events of transactions that update the given data item replicated by n_i. At the core of its concurrency control, SCORe manages a distributed timestamp scheme that allows to establish which versions are visible to a transaction, and the serialization order for update transactions.

The commit procedure of a transaction T assigns to $T.ts^C$ a timestamp value that represents T's serialization point. The versions that are visible to T are determined via a timestamp associated with T, representing its snapshot (called ts^S), which is established upon its first read operation. From that moment on, any subsequent read operation by T is allowed to observe the most recently committed version of the datum having timestamp less than or equal to $T.ts^S$, as in classical MVCC algorithms. We abstract the calculation of ts^S in function $calcSnapshot_{SCORe}$, and the visibility rule for versions described above in function $localRead_{SCORe}$.

To reach consensus on the fate of T, SCORe relies on Two-Phase Commit, where we define the coordinator as $origin(T)$. Upon receiving the prepare message, each participant p_i acquires read/write locks on the keys read/written by T, which p_i replicates. Next, each participant p_i validates T and sends its vote to the coordinator. If T is successfully validated at p_i then it proposes a serialization point for T (the value of $T.ts^C$) that is later than any serialization point it has previously observed. The coordinator then merges the votes received from the participants, which for SCORe entails choosing the maximum proposed ts^C for T, or aborting T if any vote is negative. Lastly, the commit message is sent to every participant p_i, whose finalization of T we abstract in $finalize_{SCORe}$. This function eventually writes back the write-set and makes it available for new transactions to read. Every node in a SCORe cluster has a commit thread responsible for this procedure; this thread serially writes-back committed transactions by respecting their total order at the node (defined by their ts^C).

B. Distributed time-warping

The objective of time-warping is to allow an update transaction T, which $missed$ some transaction A, to be able to commit successfully — this is usually achieved for read-only transactions in MVCC. Say that $A \in \gamma$ such that T $misses$ every transaction in γ. Moreover, A is the transaction among those in γ with the earliest commit according to real time. Time-warping tries to install the writes of T so that they are visible to transactions that serialize after A — in this case we say that T time-warp commits. This way, and recalling the example in Fig. 1, Bumper would successfully time-warp commit T whereas traditional validation, checking exclusively whether the snapshot read by T is still up to date at commit time, would abort it.

In SCORe, $T.ts^C$ is used to serialize T after the transactions that T depends on. When a transaction time-warps, we additionally compute $T.ts^W$, a scalar used to order T before the transactions it $missed$. Moreover, we now tag versions by assigning $T.ts^W$ to their timestamp — if a transaction does not time-warp, then $T.ts^W = T.ts^C$.

There is a limit to which we can bypass conflicts using time-warping and still ensuring serializability. For this, we define an abort condition based on a structure created by the $misses$ relation, that we call a $triad$. A triad exists whenever $B \dashrightarrow T \dashrightarrow A$ (where, possibly, $A = B$). We call such T a $pivot$, because it links the transactions in the triad. Then, our time-warp rule for abort is that a transaction fails its validation if, by committing, it would create a triad whose pivot time-warp commits. Detailed arguments on the safety of this validation rule are provided in Section V. The underlying intuition is that the transaction that completes the triad witnesses a history in which the pivot is not contained, which may contradict the fact that the pivot time-warp commits.

We note that this condition may generate some spurious aborts (in which it would be possible to serialize the execution). However, it is well known [7] that designing a scheduler capable of accepting all and only the serializable histories is extremely expensive, especially in distributed settings. We note also that standard validation mechanisms [5], [9], [17] abort a transaction T, if by committing T, it would develop a single $miss$ relation — thus rejecting many more serializable executions than our solution. It is also important to highlight that our triad verification allows efficient implementations as we shall see in the next section.

B.1 Detailed algorithm description

In the following we describe in detail how to implement time-warping in an instantiation of Bumper that extends SCORe. For this, our key change is the computation of the timestamp $T.ts^W$. We also use a unique identifier for each transaction. Moreover, each key k has an associated stamp ($k.readStamp$) that represents the latest access of any transaction that read k; it contains two scalars: the stamp itself and the identifier of a transaction. Finally, each version tuple now also has a boolean, stating if it was installed by a time-warped transaction, and the scalar ts^C from the transaction that installed it. We will refer to Algs. 1-2, where we highlight the line numbers corresponding to the extension of Bumper over SCORe (distinguishing between time-warping and delayed action lines). We omit details that are irrelevant to the integration with Bumper.

Transaction execution. In the $begin$ operation of a transaction tx we assign a unique identifier to the transaction. In the $write$ operation we just buffer the values locally for deferred update (as in SCORe). The $read$ operation for key k first checks for a read-after-write conflict. Then, it sends a request to the nodes that replicate k and waits for a reply[1]. The only extension required by the time-warping mechanism in this phase is line 17. We first note the following: with time-warping, the prefix of versions visible to tx according to $tx.ts^S$ is not stable, because a concurrent transaction U may time-warp commit and serialize before the point in time established by $tx.ts^S$ (by having $U.ts^W \leq tx.ts^S$). Thus, that line in the algorithm serves to guarantee that tx cannot $miss$ any update transaction that time-warp commits and serializes before $tx.ts^S$, by ensuring either that: (1) tx safely $reads\text{-}from$ U (despite being a concurrent transaction); or (2) the commit procedure of U notices that $tx \dashrightarrow U$, which, as we will see, forbids U from time-warp committing. In the latter, tx witnessed a serialization order that did not include U. Hence, if we let U time-warp commit and serialize before tx, we would obtain a non-serializable history.

To achieve the previous guarantees, we use the tuple $\langle stamp, id \rangle$ from $k.readStamp$. Then, in function $updateReadStamp$, we derive a new stamp from the last

[1]If k is replicated locally, then the communication step can be avoided.

Algorithm 1 Bumper pseudo code 1/2.

```
1:  begin(Transaction tx) in node_i = origin(tx)
2|    tx.mustTW ← false
3|    tx.cannotTW ← false
4|    tx.id ← getUniqueId()
5|    tx.ts^W ← ⊤

6:  write(Transaction tx, Key k, Value v) in node_i = origin(tx)
7:    tx.writeSet ← (tx.writeSet \ ⟨k, _⟩) ∪ ⟨k, v⟩   ▷ deferred update

8:  read(Transaction tx, Key k) in node_i = origin(tx)
9:    if k ∈ tx.writeSet then return tx.writeSet.get(tx)
10:   send READREQ[k, tx] to all n_j ∈ owners(k)
11:   wait READREPLY[val, ts^S] from any n_j ∈ owners(k)
12:   if firstRead(tx) then tx.ts^S ← calcSnapshot_SCORe(tx, ts^S)
13:   return val

14○ delayAction(Transaction tx, Action code) in node_i = origin(tx)
15○   tx.delayed ← tx.delayed ∪ code

16: upon receive READREQ[k, tx] in node_j ∈ owners(k)
17|   updateReadStamp(k)
18:   wait not underExclusiveAccess(k)
19:   [val, ts^S] ← localRead_SCORe(k, tx)
20:   reply READREPLY[val, ts^S]

21| updateReadStamp(Transaction tx, Key k) in node_i
22|   atomically do {
23|     ⟨ stamp, id ⟩ ← k.readStamp
24|     newStamp ← node_i.lastCommit
25|     if newStamp > stamp
26|       k.readStamp ← ⟨ newStamp, tx.id ⟩   ▷ update the stamp
27|     else k.readStamp ← ⟨ stamp, φ ⟩ ▷ φ symbolizes several readers
28|   }
```

known commit timestamp used at $node_i$, and increase the stamp of k along with the transaction identifier (line 26). In the case that more than one transaction updates a given stamp, the corresponding identifier becomes ϕ (line 27). Together with the visibility rules for versions, these actions ensure that read-only transactions always observe a consistent (1-copy serializable) snapshot, despite time-warping transactions. Therefore, read-only transactions skip the distributed commit (as in the original SCORe [5]) that we describe next for update transactions.

Distributed commit. When an update transaction tx requests to commit, a Two-Phase Commit (2PC) is triggered by sending the prepare message to all $participants(tx)$ (line 30). The prepare phase at $node_i$ incorporates our time-warp validation (to find dangerous triads), whose outcome is sent along with the vote of the participant. We recall that tx is aborted if it completes a triad in which the pivot would time-warp commit. For this, we use the flags $tx.mustTW$ and $tx.cannotTW$ — a dangerous triad exists if both flags are true. Triads can be detected in two cases: when tx time-warp commits and it becomes a pivot, the triad is detected by the coordinator of the 2PC when merging the votes received. The case in which tx is not the pivot, but instead completes a dangerous triad, is detected in line 53.

Algorithm 2 Bumper pseudo code 2/2.

```
    ▷ attempt to commit: join the votes of the participants and decide
29: commit(Transaction tx) in node_i = origin(tx)
30:   send PREPARE[tx] to all p_j ∈ participants(tx)
31:   for all p_j ∈ participants(tx)
32:     wait VOTE[_, vote_j] from p_j
33|     tx.ts^W ← min(vote_j.ts^W, tx.ts^W)
34|     if vote_j.mustTW then tx.mustTW ← true
35|     if vote_j.cannotTW then tx.cannotTW ← true
36:   if (∃ VOTE[NO, vote_j]) ∨ (tx.mustTW ∧ tx.cannotTW)
37:     send ABORT[tx] to all p_j ∈ participants(tx)
38:     return ABORT
39:   tx.ts^C ← max(votes.ts^C)
40|   if not tx.mustTW                    ▷ if tx does not time-warp
41|     tx.ts^W = tx.ts^C                 ▷ then tx serializes at the present
42:   send COMMIT[tx] to all p_j ∈ participants(tx)

43: upon receive PREPARE[tx] in node_i ∈ participants(tx)
44:   for all k ∈ tx.writeSet : local(k)
45:     acquireLock(k,EXCLUSIVE) ▷ acquire lock in exclusive mode
46|     ⟨ stamp, id ⟩ ← k.readStamp
47|     if stamp ≥ tx.ts^S ∧ id ≠ tx.id   ▷ if concurrent T also read k
48|       tx.cannotTW ← true              ▷ then T missed tx
49:   for all ⟨k, ts⟩ ∈ tx.readSet : local(k)
50:     acquireLock(k,SHARED)            ▷ acquire lock in shared mode
        ▷ check concurrently installed versions that were missed by tx
51|     for all K ∈ k.versions() : K.ts^C > tx.ts^S ∧ ts ≠ K.ts
52|       if K.timeWarped ∨ k ∈ tx.writeSet
53|         reply VOTE[NO, tx]          ▷ tx completes a dangerous triad
54|         return
55|       tx.mustTW ← true              ▷ tx missed something
56|       tx.ts^W ← min(tx.ts^W, K.ts^C)   ▷ compute time-warp
57○ if tx.delayed not empty ∧ tx.mustTW
58○     reply VOTE[no, tx]               ▷ restart with eager delayed actions
59○     return
60○ for all action ∈ tx.delayed : local(action)
61○     for all k ∈ action.keySet()      ▷ local keys accessed
62○       acquireLock(k, DELAYED)        ▷ acquire lock in delayed mode
63:   tx.ts^C ← node_i.nextId++
64:   reply VOTE[YES, tx]

65: upon receive COMMIT[tx] in node_i ∈ participants(tx)
66:   atomically do {
67○     for all action ∈ tx.delayed : local(action)
68○       action.execute()               ▷ delayed execution without conflicts
69:     finalize_SCORe(tx)               ▷ eventually invokes writeBack for tx
70:   }

    ▷ invoked for each write of tx when it is ready to commit
71: writeBack(Transaction tx, Key k) in node_i : local(k)
72|   newVersion ← k.prependNewVersion()
73|   newVersion.ts ← tx.ts
74|   newVersion.ts^C ← tx.ts^C
75|   newVersion.timeWarped ← tx.mustTW
```

The computation of those flags takes place after the acquisition of the locks associated with the keys (as explained for SCORe, shared mode for reads and exclusive mode for writes). In lines 44-48, we check for a possible $B \dashrightarrow tx$ by verifying if $k.readStamp$ was increased concurrently to the execution of tx (this also checks that the only reader is not tx itself). In such case, tx cannot time-warp commit because B already witnessed this execution. In lines 49-56, we check for a possible $tx \dashrightarrow A$ for every k read by tx. To

do so, we obtain the versions of k installed concurrently to the execution of tx, and verify if tx did not read that version. In this case, tx must time-warp commit to correctly serialize before A. Given this case, we note that we immediately abort tx if we can deduce the existence of a dangerous triad at that point (line 53): (1) tx would complete a dangerous triad where A is the pivot; or (2) tx and A form a cycle, as both read and write k concurrently, which is a particular case of our definition of triad. Otherwise, we update $tx.ts^W$ in line 56, which represents the point in time in which the writes of tx will be installed. To respect the fact that $tx \dashrightarrow A$, we minimize $tx.ts^W$ with $A.ts^C$ so that tx serializes before A (hence why we keep ts^C in the versions installed, in line 74): this ensures that the resulting time-warp serializes tx before the set of transactions it $missed$.

After conducting this novel validation at participant $node_i$, we let SCORe propose $tx.ts^C$ by incrementing a local scalar. Then, participants reply to the coordinator and each vote is merged in lines 32-35. The validation flags are also merged — this allows the function to check if $participants(tx)$ detected the dangerous triad that we disallow. Next, if tx did not $miss$ any transaction, we set $tx.ts^W = tx.ts^C$. This corresponds to the normal case in which typical validations (such as SCORe's) do not abort tx (so it is not necessary to time-warp commit it). Upon receiving the commit decision, a participant $node_i$ relays this event to SCORe. Eventually, this invokes the write-back mechanism that we also extended: we tag versions using $tx.ts^W$ and with the additional metadata that we described.

C. Delayed actions

A delayed action corresponds to a part of the application code that is encapsulated in a transaction, but that can be executed outside of the normal flow of execution of that code and postponed until the transaction's commit (line 15). This is possible whenever the output generated, or the state updated, by a portion of code of the transaction is not required (i.e. read) elsewhere within that transaction. As an example, consider that transactions C and D (in Fig. 1) are contending to increment z, which is the balance of an online store, upon two payments of customers. If the customers had bought different items, which is very likely in a large store, both C and D should be able to proceed independently if it was not for the contention hot spot in the balance. As a matter of fact, the balance of the store does not affect the outcome of the payments — it is only important to be updated within the transaction to ensure correct accountability checks, and possibly be made available after the transaction commits (to display it). So we address these situations by exploiting delayed actions that are executed at the end of the distributed commit, after the lock acquisition phase — this way we can guarantee that the delayed reads always observe the most recent available version at the time in which the transaction commits. In other words, no concurrent transaction can commit and invalidate those reads, thus making delayed actions abort-free.

A first challenge is how to ensure that delayed actions can be executed efficiently in a partial replicated setting. In fact, since they are executed during the distributed commit, it is desirable to ensure that, whenever a delayed action is executed at a node n_i, it only accesses data locally stored by n_i. This avoids involving additional nodes in the commit.

We avoid these issues by having delayed actions abide by the following programming paradigm. When defining a delayed action, programmers are required to identify (a possible over-approximation of) the keys to manipulate. We assume that these keys can all be read or written, and so we extract from them the $executors$ set of nodes where the delayed execution is to be performed. In the prepare phase, the delayed action is registered at those nodes by piggybacking the corresponding keys in the prepare message. During its execution at node n_i, a delayed action may only access data locally maintained by n_i. Otherwise the transaction is aborted, an error is reported to the programmer, and the transaction is restarted executing the delayed actions "eagerly", i.e. within the transaction, by not postponing them (line 58). Moreover, we assume deterministic computations, as these are being executed by different replicas. Finally, we allow each of the delayed action instances (one per each node in $executors$) to return a result, which is then reduced in the coordinator via a programmer-defined operator.

A second challenge is to regulate concurrency between transactions encompassing delayed actions and regular transactions. Suppose that k is incremented concurrently, by transactions L_1 and L_2 using delayed actions, and by another transaction T in the traditional way. Intuitively: (1) we want L_1 and L_2 to proceed concurrently (ensuring that their effects are serializable); (2) we want T to detect the concurrent read and write of k performed by the delayed actions; and finally, (3) we do not want either L_1 or L_2 to abort because T committed first and manipulated k.

To correctly address this challenge, we use the commit thread available at every node, responsible for serially writing-back updates produced by transactions in the total order defined by their commit timestamp (ts^C). This allows the distributed commit of L_1 and L_2 to progress concurrently during the execution of 2PC, until their delayed actions are executed sequentially within the commit thread, before their write-back (line 68). To guarantee correctness when some transaction T conflicts with L_1, we rely on the lock acquisition for the keys (to be manipulated in the delayed actions). Differently from normal writes, which acquire an exclusive lock at prepare time, these keys might be locked by several transactions (line 62) and written by their delayed actions (within the commit thread in line 68). Therefore we safely allow L_1 and L_2 to share locks over the keys to be manipulated by their delayed actions. For this reason, we

created a delayed mode for the locks associated with each key, which may be shared by delayed actions and is mutually exclusive with both read and write modes.

Finally, we consider a transaction T that must time-warp commit (because it has at least one $miss$). If T has any pending delayed actions to be executed, this can pose a problem: the time-warp commit serializes the transaction in a past point in time, but the delayed actions take place in the present. So there is a duality between the intent of both mechanisms, as time-warping tries to deal with stale data whereas delayed actions are designed to always operate over fresh data. We delegate further research on this duality for future work; there is room for reconciliation between the mechanisms, albeit possibly in an intricate matter. In this work we reconcile these two mechanisms by aborting transactions that must both time-warp and execute delayed actions — this causes the delayed actions to be executed in the normal way (thus eagerly) in the re-execution (line 58).

D. Integration with alternative DTP protocols

In order to highlight the generality and "portability" of the conflict reduction mechanisms of Bumper, we briefly discuss how it may be adapted for the case of full replication, or in case that the system does not use MVCC.

In fully replicated systems, all state is available in every node. In this case we note that it is possible to relax some of the constraints on the programming paradigm of delayed actions, as it would no longer be necessary to act upon just a group of keys that is known to be co-located. Hence, it would not be necessary to have any a priori knowledge of the set of keys to be accessed within a delayed action.

On the other hand, it is possible that the local concurrency control does not support MVCC, and instead maintains only a single version of data. Then, the time-warping mechanisms can still be applicable, as long as there exists a timestamping mechanism, as this allows for reasoning on the concurrency of events. This is typically available in optimistic concurrency control schemes such as those used in the DTPs that we mentioned in this paper. Also in this case, with regard to delayed actions, we consider that there may exist multiple commit threads performing the write-back phase of non-conflicting transactions in parallel. In this case one can resort to alternative locking mechanisms (not described for space constraints) to correctly serialize two delayed actions that contend (as they would no longer be executed sequentially by the same thread). In fact, delayed actions can also be used in DTPs guaranteeing lower isolation levels (e.g., Snapshot Isolation) to mitigate contention on hot spots.

V. Discussion about Correctness and Failures

We now show that applying Bumper to SCORe preserves the latter's isolation level (1-copy serializability) when considering the time-warping algorithm. In Section V-B, we argue that delayed actions do not change that result. For space constraints, we omit a full formal proof. Nevertheless, we provide informal correctness arguments. Finally, we discuss the impact of Bumper on failure recovery.

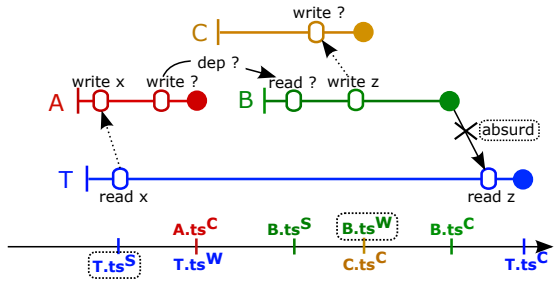

Figure 2: Execution allowed by Bumper but rejected by SCORe (by aborting some transaction).

A. Time-warping

We start by considering the fail-free executions that SCORe accepts (named $\mathcal{H}^G$), and show that Bumper also accepts them. Then we consider the fail-free executions that Bumper accepts, but that SCORe rejects (named $\mathcal{H}^B$), and show that they are necessarily serializable.

We begin by noting that SCORe always aborts a transaction T in the same conditions in which Bumper tries to time-warp commit T. Then, we can show for $\mathcal{H}^G$ that, when Bumper aborts, this implies that SCORe also aborts: lines 37 and 53, where Bumper aborts tx, correspond to the cases in which tx has at least a $miss$ and so SCORe aborts tx.

We now consider an arbitrary execution, which can be extended to match every execution in $\mathcal{H}^B$, and show that it is serializable[2]. In the following we shall derive a set of restrictions on this arbitrary execution until we reach an absurd, which we also exemplify in Fig. 2. Since we are considering the set of executions that are accepted by Bumper, but rejected by SCORe, they must contain one time-warped transaction T that $missed$ transaction A. This implies that A commits before T does so (such that T notices the $miss$ and time-warps) and also that A is concurrent with T (otherwise no $miss$ would occur). Also note that A cannot have time-warp committed as well, as that would have triggered the abort of T in line 52. The same line (but its second condition instead) would be triggered if A $missed$ T (which would have created a cycle between A and T). It is also impossible for T to $read\text{-}from$ A because they are concurrent and T has already $missed$ A; this would require $A.ts^W \leq T.ts^S$, but we already stated that A cannot time-warp commit, so we actually have $T.ts^S < A.ts^W = A.ts^C$.

Now consider some transaction B that $misses$ T: this would create a triad (detected either by B in line 52 or

[2]Recalling classic serializability theory results [7], we say that an execution is serializable if the graph of $read\text{-}from$ and $miss$ relations (edges) between transactions (nodes) is acyclic.

by T in line 37). So the arbitrary execution that we are considering can only contain a cycle if T $reads\text{-}from$ B. Consider, without loss of generality, that B $reads\text{-}from$ A (perhaps transitively), and thus $A.ts^C \leq B.ts^S$. Then, because $T.ts^S < A.ts^C$, and $A \rightarrow B$ (even if transitively), we obtain $T.ts^S < B.ts^S$. We note that these conclusions are also visible in the timeline drawn in Fig. 2. Because $T.ts^S < B.ts^C$, T can only $read\text{-}from$ B if B time-warps before $T.ts^S$ — this implies that B $misses$ some transaction C. But at this point we can summarize the restrictions devised so far and reach an absurd that forbids $B \rightarrow T$. We have that $T.ts^S < B.ts^S$, and for $B \dashrightarrow C$, it must be that $B.ts^S < C.ts^C$; but then we get $T.ts^S < B.ts^W$, so it is an absurd that $B \rightarrow T$. Thus all executions in $\mathcal{H}^B$ are serializable, and Bumper preserves 1-copy serializability.

B. Delayed Actions

When considering delayed actions, we explain why we can disregard the validation for these postponed manipulations, and what guarantees the consistent evolution of the state of replicas of the same set of keys.

We start by recalling that the delayed actions of a transaction T are executed after a commit decision for T arrives to the commit thread of some node $node_i \in participants(T)$ — thus the corresponding locks will have been acquired in the prepare phase of the commit procedure. This ensures mutual exclusion when accessing some key k between (1) the accesses inside delayed actions and (2) validations of transactions that access k normally. In addition to this, because the execution takes place in the commit thread, this implies that delayed actions cannot observe concurrent transactions. Given that we restricted T not to time-warp when it has delayed actions, then $T.ts^W = T.ts^C$. Therefore, both normal and delayed portions of T are serialized on $T.ts^C$.

Finally, the consistent evolution of the state trajectory of the various replicas of a key is ensured given that i) SCORe ensures that replicas of the same keys update this set of keys according to the same total-order [5], and ii) given that we assumed delayed actions to be deterministic.

C. Dealing with failures

Bumper, when built upon SCORe, inherits its virtues and issues for what concerns fault-tolerance. We refer to the original SCORe paper [5] for a detailed discussion on how to deal with failures. For space constraints, we will only briefly recall that, due to the reliance of SCORe on Two-phase Commit, it is necessary to adopt additional mechanisms (such as replication techniques for ensuring high availability of the coordinator's state [24]) in order to avoid blocking in spite of failures of the coordinator. Note that Bumper does not introduce any additional complexity/drawback for what concerns failure-handling: this enhances the relevance and practicality of the proposed solution.

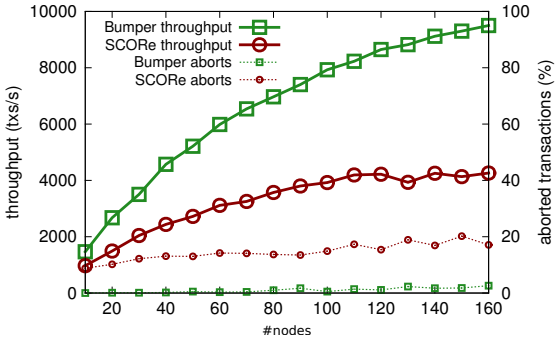

Figure 3: Benefits of time-warp in SkipList.

VI. Experimental Evaluation

We integrated Bumper into a publicly available implementation of the SCORe protocol, which is based on Infinispan, a mainstream in-memory DTP developed by Red Hat. This allows us to evaluate the benefits achievable by Bumper using as baseline a highly scalable, strongly consistent genuine partial replication protocol. We measure both overall throughput and abort probability (note that read-only transactions are abort-free in SCORe). Every run uses replication degree of two for fault-tolerance. Our experimental study aims at answering the following questions: (1) How much can Bumper enhance SCORe's scalability in a conflict-prone scenario? (2) To what extent can it reduce the transaction's abort rate? (3) What overheads does Bumper introduce in conflict-free workloads?

We used four well-known benchmark applications, which we will briefly describe while presenting the results. Each execution is the result of ten runs with exclusive access to all the machines used. We use geometric mean whenever showing averages over normalized results. We conducted our tests on top of OpenStack, a cloud computing infrastructure, deployed in a dedicated cluster. Each machine is equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors, 40 GB of RAM and interconnected via a private Gigabit Ethernet. The VMs instantiated via OpenStack were allocated 1 physical core plus 4GB Ram and the virtualization took advantage of the hardware support provided by the Intel(R) processors. For all tests we varied the number of VMs from 10 to 160, such that they were always uniformly distributed across the physical machines. Finally, the virtualized OS was Ubuntu 12.04 and our prototypes ran on Java HotSpot version 1.6.0_38.

We begin with two benchmarks and workloads without any obvious contention hot spots, and thus we avoid using any delayed action. This is done to ensure that the benefits achieved are exclusively due to time-warping mechanism. We start with a micro-benchmark originally proposed to evaluate transactional memory systems, and that exercises a skip list data-structure — a building block for many applications. A skip list is used to maintain an ordered

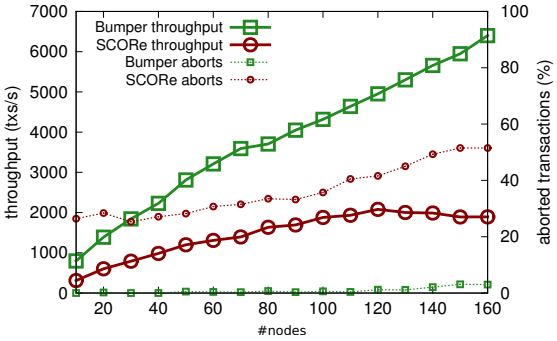

Figure 4: Benefits of delayed actions in TPC-C.

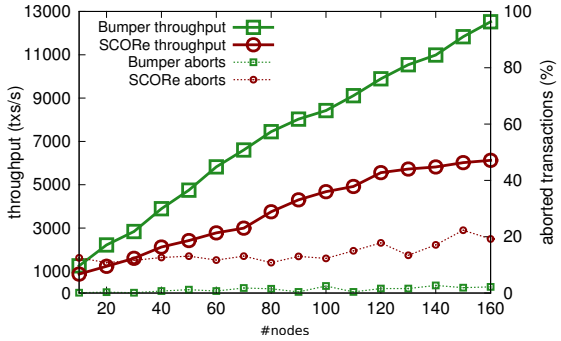

Figure 5: Vacation with time-warping and delayed actions.

set of integers with an average size of 256 elements and a range of possible keys of 65 thousand integers. This means that most of the time there might exist structural conflicts when manipulating the list, but rarely should the transactions be attempting to insert or remove the same element. Fig. 3 shows the results for a workload with 50% read-only transactions (that check the existence of a given element) and update transactions that insert and remove items. The results show a peak gain of 2.23 speedup at 160 nodes. As we will consistently witness, these gains are due to a considerable reduction of aborts. In this case, we report a reduction of average likelihood of update transactions' aborts from 15% to 0.9%.

We then consider YCSB (Yahoo! Cloud Serving Benchmark) [25], which was designed to benchmark NoSQL key-value storage systems and generates data access patterns that mimic real applications' skewed workloads. We used a workload containing 50% of update transactions, which access 16 keys and modify up to 4 keys, and 50% short read-only transactions that access a single data item. For space constraints, we report only textually the results of this experiment: once again, we obtain considerable gains due to the avoidance of many conflicts by exploiting the time-warping mechanisms; Bumper yielded an average speedup of 2.8 over SCORe, by reaching an impressive peak throughput of almost 13k txs/s against 4.8k txs/s at 160 nodes in this conflicting-prone and update intensive workload with lightweight transactions.

We now move to evaluate the benefits achievable by using delayed actions. To this end we consider a porting of the TPC-C benchmark, a well-known OLTP benchmark that was adapted to run on top of transactional key-value stores (and used, in previous works [4], [5], to evaluate the performance of strongly consistent partial replication protocols). This benchmark portraits the activities of a whole-sale supplier and contains a number of easily identifiable contention hot spots. Specifically, one of its transaction profiles, the so-called Payment transaction, updates the balances of a warehouse and of its district whenever an order for an item stored in that warehouse is processed. The balances of each warehouse and its district are maintained by a distinct pair of keys, which turn quickly into contention hot spots as the scale (and consequently the parallelism level) increases. We encapsulated the update of the warehouse/district balance into a delayed action, and injected a workload containing 50% of update transactions. The results of this experiment, reported in Fig. 4, demonstrate clearly the benefits deriving from the ability of delayed actions to avoid contention over hot spots. The average abort probability of an update transaction is 38% for SCORe, whereas Bumper substantially reduces it to 0.8%. As a result Bumper scales up to 6.4k txs/s and obtains a peak speedup of 3.4 at 160 nodes.

To assess the overhead of the mechanisms at the core of Bumper we also conducted experiments in uncontended scenarios, in which there are no benefits from the usage of either time-warping or delayed actions. We resorted again to YCSB and used a workload with 50% update transactions. To guarantee absence of contention, we altered the data access pattern of transactions to update a single key selected from disjoint sets. As a result of this experiment, we measured a negligible average 2.5% overhead, which results from the additional validations that Bumper computes at commit time while holding the locks.

Finally, we also used the Vacation benchmark from the STAMP [26] suite of transactional memory applications. Vacation simulates an online travel agency in which several types of resources can be manipulated by customers or by the agency. Like for TPC-C, we used a port of this benchmark for distributed key-value stores. This benchmark showcases the benefits of both time-warping and delayed actions. The latter are used to work around contention hot spots associated with keys that maintain the number of free/used travel resources of various kinds. Several invariant checks are performed around these statistics to ensure the consistency of the application was not broken. Fig. 5 reports the results for this experiment, once again with a reduction of abort percentage yielding direct gains on the throughput and scalability of the system: the benefits of our contributions reduce the average abort rate to under 1% and lead to a speedup of 2.1 at 160 nodes.

VII. Conclusion

This paper tackled the issue of how to maximize the scalability of strongly consistent distributed transactional platforms in presence of conflict intensive workloads. We did so by introducing two innovative mechanisms aimed to reduce the transaction abort rate in two orthogonal ways, Distributed Time-Warping and Delayed Actions.

We call these mechanisms Bumper, as these techniques can be plugged on various transactional replication protocols to enhance their robustness in high contention scenarios. We discuss how to integrate Bumper in SCORe, a recent, highly scalable protocol for distributed transactional platforms, which provides genuine partial replication and relies on a multi-versioning concurrency control scheme. Finally, we evaluated the benefits achievable by the proposed solutions by conducting an experimental study using four well-known benchmarks. To this end, we integrated Bumper in Infinispan, a popular open-source transactional key-value store. Our experiments highlight that Bumper allows boosting throughput up to a $3\times$ factor in conflict intensive workloads, while imposing negligible (about 2%) overheads in the absence of contention.

References

[1] G. DeCandia et al., "Dynamo: Amazon's highly available key-value store," in *SOSP*, 2007, pp. 205–220.

[2] J. Corbett et al., "Spanner: Google's globally-distributed database," in *OSDI*, 2012, pp. 251–264.

[3] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *SOSP*, 2011, pp. 385–400.

[4] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication," in *ICDCS*, 2012, pp. 455–465.

[5] S. Peluso, P. Romano, and F. Quaglia, "SCORe: A Scalable One-Copy Serializable Partial Replication Protocol," in *Middleware*, 2012, pp. 456–475.

[6] N. Schiper, P. Sutra, and F. Pedone, "P-Store: Genuine Partial Replication in Wide Area Networks," in *SRDS*, 2010, pp. 214–224.

[7] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979.

[8] P. S. Yu, D. M. Dias, and S. S. Lavenberg, "On the analytical modeling of database concurrency control," *J. ACM*, vol. 40, 1993.

[9] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2STM: Dependable Distributed Software Transactional Memory," in *PRDC*, 2009, pp. 307–313.

[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[11] N. Diegues and P. Romano, "Brief Announcement: Enhancing Permissiveness in Transactional Memory via Time-Warping," in *DISC*, 2013.

[12] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[13] J. Kim and B. Ravindran, "On transactional scheduling in distributed transactional memory systems," in *SSS*, 2010, pp. 347–361.

[14] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris, "Middleware based data replication providing snapshot isolation," in *SIGMOD*, 2005, pp. 419–430.

[15] B. Kemme and G. Alonso, "Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication," in *VLDB*, 2000, pp. 134–143.

[16] F. Pedone, R. Guerraoui, and A. Schiper, "The Database State Machine Approach," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 71–98, 2003.

[17] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *DSN*, 2012, pp. 1–12.

[18] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," in *SIGMOD*, 2008, pp. 729–738.

[19] H. Berenson et al., "A critique of ANSI SQL isolation levels," in *SIGMOD*, 1995, pp. 1–10.

[20] H. Jung, H. Han, A. Fekete, and U. Röhm, "Serializable Snapshot Isolation for Replicated Databases in High-Update Scenarios," *PVLDB*, vol. 4, no. 11, pp. 783–794, 2011.

[21] J. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," *Science of Computer Programming*, vol. 63, no. 2, pp. 172–185, 2006. Elsevier.

[22] D. Karger et al., "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," in *STOC*, 1997, pp. 654–663.

[23] A. Adya, "Weak consistency: a generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[24] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 133–160, 2006.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010, pp. 143–154.

[26] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-processing," in *IISWC*, 2008, pp. 35–46.