Formal Modeling and Analysis of the Walter Transactional Data Store

Si Liu[1(✉)], Peter Csaba Ölveczky[2], Qi Wang[1], and José Meseguer[1]

[1] University of Illinois, Urbana-Champaign, USA
siliu3@illinois.edu
[2] University of Oslo, Oslo, Norway

Abstract. Walter is a distributed partially replicated data store providing Parallel Snapshot Isolation (PSI), an important consistency property that offers attractive performance while ensuring adequate guarantees for certain kinds of applications. In this work we formally model Walter's design in Maude and formally specify and verify PSI by model checking. To also analyze Walter's performance we extend the Maude specification of Walter to a probabilistic rewrite theory and perform statistical model checking analysis to evaluate Walter's throughput for a wide range of workloads. Our performance results are consistent with a previous experimental evaluation and throw new light on Walter's performance for different workloads not evaluated before.

1 Introduction

Cloud-based transaction systems provide both a challenge and an opportunity for the use of formal methods. The *challenge* has to do with the fact that the very *raison d'être* for such system is the need for a carefully chosen compromise between consistency guarantees and performance. Their massive use requires them to ensure scalability to large numbers of users with acceptable latency and throughput, while also guaranteeing the promised consistency properties. This is a challenge for formally-based design, because many formal methods tend to solely focus on correctness. Yet, correctness without due performance is useless for these systems. The *opportunities* are plentiful, including the following: (1) Many of these systems have never been formally specified, either at the *system specification* level or at the *property specification* level. (2) There is a need for *modularity* and *conceptual unification* in the design of these, currently quite ad-hoc and monolithic, systems. (3) There is also the prospect of using formal executable specifications for *code generation* purposes, achieving correct-by-construction systems that, by having been thoroughly analyzed in their correctness and performance aspects, can achieve very high quality.

This work is part of a long-term research effort in which we have been using Maude to meet the challenges and exploit the opportunities described above for cloud-based transaction systems (see [4] for a survey). Specifically, we exploit the type-(1) opportunity offered by Walter [21], a well-known cloud-based transaction system that provides an important intermediate consistency

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 136–152, 2018.
https://doi.org/10.1007/978-3-319-99840-4_8

guarantee, Parallel Snapshot Isolation (PSI). Walter is a very good type-(1) opportunity because no formal system specification exists at all; and there is no formal (or even informal) verification that it guarantees PSI. Walter is also a good stepping stone towards placing the design of cloud-based transaction systems in a formally-based modular framework. The way we are advancing this type-(2) goal is by first systematically studying system designs that cover the whole spectrum between lower-guarantees/higher-performance and higher-guarantees/lower-performance systems. We have already studied several systems in this spectrum, including RAMP [11,15], our own ROLA design [12], P-Store [18], and Megastore [9]. Walter has been a key missing design in the spectrum. The essential point is that case studies spanning the entire correctness/performance spectrum are crucial for identifying optimal decompositions of such systems into modular, reusable components. Finally, the fact that our Maude specification of Walter and of the other above-mention systems are *executable*, also helps us advance towards exploiting the type-(3) opportunity of achieving high-quality code generation for formal specifications. In this paper we focus on the type-(1) goal for Walter, but our sights are aimed at the type-(2)–(3) goals just as much.

Main Contributions and Outline. In Section 2 we give an overview of Walter, the PSI property and the stronger Snapshot Isolation (SI) property, and summarize the main features of Maude used in this paper. Section 3 provides a formal executable specification of Walter in Maude. This is a key contribution since, to the best of our knowledge, it is the first formal specification of Walter. Section 4 formalizes the SI and PSI properties and formally analyzes for the first time whether the Walter design satisfies either of these properties. This analysis is achieved by: (i) providing a parametric method to generate all initial states for given parameters; and (ii) performing model checking analysis to verify the SI and PSI properties for all initial states for various parameter choices. To analyze complex properties such as SI and PSI we propose a new general method for model checking such properties by adding a "monitor" to the state which records the global order of transaction starts and commits/aborts. In this way we can easily specify and model check SI and PSI; furthermore, this technique should also be applicable to analyze other consistency properties. Our analysis shows that the Walter design does indeed satisfy the PSI property for all our initial states but fails to satisfy the SI property. Section 5 makes four contributions. First, it extends the Maude model of Walter from a rewrite theory to a *probabilistic* rewrite theory by adding time and probability distributions for message delays to the original specification. Second, it carries out a systematic *statistical model checking* analysis of the key performance metric, transaction throughput, under a wide range of workloads. Third, it confirms that the performance estimates thus obtained are consistent with those obtained experimentally for the Walter implementation in [21]. Fourth, it provides new insights about Walter's performance beyond the limited ranges for which such information was available by experimental evaluation in [21]. Finally, related work is discussed in Section 6, and concluding remarks are given in Section 7.

2 Preliminaries

Parallel Snapshot Isolation. To deal with huge amounts of data, cloud-based applications need to *partition* their data across distributed sites, and to provide high availability and disaster tolerance, data must be *replicated* at widely distributed sites. Such *partially replicated* data stores have to: (i) maintain some consistency of replicated data, and (ii) provide some consistency for (multi-partition) transactions that access data stored at different partitions. However, ensuring high degrees of consistency for partially replicated data stores supporting multi-partition transactions requires a lot of costly coordination, which might lead to unacceptable delays and throughput for many kinds of applications. Designers of distributed data stores therefore face a trade-off between providing good consistency properties and high performance. There are a number of consistency properties, ranging from strong ones like serializability all the way to weak properties such as read atomicity and eventual consistency.

A popular intermediate consistency model provided by commercial database systems such as Oracle and SQL Server is *snapshot isolation* (SI) [3]. The idea is that a multi-partition transaction reads from a *snapshot* of a distributed data store that reflects a *single commit order* of transactions across sites.

In [21], the authors argue that having the same commit order across all sites is not necessary for social networks and similar applications: it does not matter much that Vlad in Moscow sees Kim's post before seeing Benny's post, whereas Don in Washington sees Benny's post before Kim's post. (Hence Benny and Kim can commit their (independent) posts without waiting for each other.) They propose a new consistency model, called *parallel snapshot isolation*, which allows different commit orders at different sites, while still guaranteeing:

- recent and "consistent" views: all operations in a transaction read the most recent version committed at the transaction execution site, as of the time when the transaction begins;
- no write-write conflicts (the write sets of committed somewhere-concurrent transactions must be disjoint); and
- preservation of *causality* across sites, which ensures that both Vlad and Benny see Kim's post before seeing Don's reply to Kim's post.

In [21] the authors specify PSI by giving an abstract pseudo-code "program" of a centralized execution that a distributed implementation must emulate.

Walter. *Walter* [21] is a partially replicated geo-distributed data store that supports multi-partition transactions and guarantees/implements PSI.

The key idea to ensure that all operations in a transaction read a consistent "snapshot" of the distributed data store is that each site s maintains a (local) *vector timestamp* $\{site_1 \mapsto k_1, \ldots, site_n \mapsto k_n\}$ representing a current snapshot of the state, as seen by site s, where $site_j \mapsto k_j$ means that the snapshot includes the first k_j transactions executed at site $site_i$. Each time a transaction starts executing at s, the transaction is assigned the current local snapshot/vector timestamp of site s. Remote reads can then be performed consistently according to

this snapshot. Another key Walter feature is that each data item has a preferred site, so that writes at preferred sites can be committed fast.

A transaction is executed as follows. When the "host" site s starts executing the transaction t, t is assigned the current snapshot of s. The site s then executes the read and write operations in t. For writes, Walter buffers the versions written in the transaction's write set. For reads, Walter fetches the latest appropriate version according to t's start snapshot, by checking any updates in the write set and its history of previous updates. If the data item is not replicated locally, Walter retrieves the right version remotely from the data item's preferred site.

When the host site has executed all the operations in a transaction, it starts committing the transaction. Read-only transactions and transactions that only write data items whose preferred site is the host site can commit locally (*fast commit*). Walter then checks whether all versions of each data item in the history of the local site are *unmodified* since the start vector timestamp, and whether all data items are *unlocked* (i.e., not being committed by another transaction). If either check fails, Walter aborts the transaction; otherwise, it can be committed. If a transaction t cannot commit locally (*slow commit*), the executing site s uses the two-phase commit (2PC) protocol to check whether t can be committed, by asking the preferred sites of data items written by t whether t can be committed. If the data items written by t are unmodified and unlocked at such a site, the site replies with a "yes" vote and locks the corresponding data items. Otherwise, the site votes "no." If the executing site receives a "no" vote, t is aborted and the other preferred sites are notified and release the appropriate locks. If all votes are "yes" votes, the transaction can be committed.

If the transaction t can be (fast or slow) committed, the site s marks t as committed, assigns it a *version* $(s, seqNo)$ (where *seqNo* is a local sequence number), updates the local history with the updates, and propagates t to other sites, which update their histories and their vector timestamps. To allow f site failures, a transaction is marked *disaster-safe durable* if its writes have been logged at $f+1$ sites. The propagation protocol first checks whether the transaction can be marked as disaster-safe durable by collecting acknowledgments from $f+1$ sites for each data item. Upon receiving the propagation of a transaction, a site acknowledges it only after it receives all transactions that causally precede the propagated transaction, and all transactions at the same executing site with a smaller sequence number. The protocol then checks whether the transaction can be marked as globally visible. This is done by committing the transaction at all sites. A transaction can be committed at a remote site when it learns that the transaction is disaster-safe durable, all transactions causally preceding the transaction have been committed locally, and all transactions at the same executing site with a smaller sequence number have been committed locally.

The paper [21] briefly discusses failure handling, but does not give much detail. The authors have implemented Walter in about 30K lines of code, and have implemented Facebook- and Twitter-like applications on top of Walter using the Amazon EC2 cloud platform to evaluate Walter's performance in a distributed setting (with nodes in US, Ireland, and Singapore). They use their distributed deployment

to estimate the transaction latency and throughput (committed transactions per second) for read-only, write-only, and 90% read workloads.

The authors to do not prove or justify that Walter actually guarantees PSI.

Rewriting Logic and Maude. In rewriting logic a concurrent system is specified a as *rewrite theory* $(\Sigma, E \cup A, R)$, where $(\Sigma, E \cup A)$ is a *membership equational logic theory* [6], with Σ an algebraic signature declaring sorts, subsorts, and function symbols, E a set of conditional equations, and A a set of equational axioms. It specifies the system's state space as an algebraic data type. R is a set of *labeled conditional rewrite rules*, specifying the system's local transitions, of the form $[l] : t \longrightarrow t'$ **if** *cond*, where *cond* is a condition and l is a label. Such a rule specifies a transition from an instance of t to the corresponding instance of t', provided the condition holds.

Maude [6] is a language and tool for specifying, simulating, and model checking rewrite theories. The distributed state of an object-oriented system is formalized as a *multiset* of objects and messages. A class C with attributes att_1 to att_n of sorts s_1 to s_n is declared `class` C | $att_1 : s_1, \ldots, att_n : s_n$. An object of class C is modeled as a term `<` $o : C$ | $att_1 : v_1, \ldots, att_n : v_n$ `>`, with o its object identifier, and where the attributes att_1 to att_n have the current values v_1 to v_n, respectively. Upon receiving a message, an object can change its state and/or send messages to other objects. For example, the rewrite rule

```
rl [l] :  m(O,z)  < O : C | a1 : x, a2 : O' >
      =>          < O : C | a1 : x + z, a2 : O' >  m'(O',x + z) .
```

defines a transition where an incoming message `m`, with parameters `O` and `z`, is consumed by the target object `O` of class `C`, the attribute `a1` is updated to `x + z`, and an outgoing message `m'(O',x + z)` is generated.

Statistical Model Checking and PVeStA. Probabilistic distributed systems can be modeled as *probabilistic rewrite theories* [1] with rules of the form

$$[l] : t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \ \textbf{if} \ cond(\overrightarrow{x}) \ \textit{with probability} \ \overrightarrow{y} := \pi(\overrightarrow{x})$$

where the term t' has new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ in the term t. The concrete values of the new variables $\overrightarrow{y}$ in $t'(\overrightarrow{x}, \overrightarrow{y})$ are chosen probabilistically according to the probability distribution $\pi(\overrightarrow{x})$.

Statistical model checking [19,22] is an attractive formal approach to analyzing (purely) probabilistic systems. Instead of offering a yes/no answer, it can verify a property up to a user-specified level of confidence by running Monte-Carlo simulations of the system model. We then use PVeStA [2], a parallelization of the tool VeStA [20], to statistically model check purely probabilistic systems against properties expressed as QuaTEx expressions [1]. The expected value of a QuaTEx expression is iteratively evaluated w.r.t. two parameters α and δ by sampling, until we obtain a value v so that with $(1-\alpha)100\%$ statistical confidence, the expected value is in the interval $[v - \frac{\delta}{2}, v + \frac{\delta}{2}]$.

3 A Formal Model of Walter in Maude

This section explains how we have formalized Walter in Maude. Due to space limitations, we only show parts of our model (e.g., 4 out of 26 rewrite rules) and refer to the accompanying longer report [13] and the executable model available at https://sites.google.com/site/siliunobi/walter for more details.

We formalize Walter in an object-oriented style, where the state consists of a number of *replica* (or *site*) objects, each modeling a local database, and a number of messages traveling between the objects. A *transaction* is formalized as an object which resides inside the replica object that executes the transaction.

Some Data Types. A *version* is a pair `version(oid,sqn)` consisting of a site `oid` where the transaction is executed, and a sequence number `sqn` local to that site. A vector timestamp is a map from site identifiers to sequence numbers. The sort `OperationList` represents lists of read and write operations as terms such as `(`x `:= read` $k1$`) (`y `:= read` $k2$`) write(`$k1, x+y$`)`, where x and y are "local variables." An "operation" `waitRemote(`k, x`)` means that the transaction execution is awaiting the value of the key/data item k from a remote site to be assigned to the local variable x (see [13] for the definition of these data types).

Classes. A *transaction* is modeled as an object of the following class `Txn`:

```
class Txn | operations : OperationList,  readSet : ReadSet,
            writeSet : WriteSet,         localVars : LocalVars,
            startVTS : VectorTimestamp,  txnSQN : Nat .
```

The `operations` attribute denotes the transaction's remaining operations. The `readSet` attribute denotes the versions of data items read by the transaction as a ','-separated set of pairs `versionRead(`k, *version*`)`. `writeSet` denotes the write set of the transaction as a map `(`k_1 `|->` val_1`), ..., (`k_n `|->` val_n`)`. `localVars` maps the transaction's local variables to their values. `startVTS` is the vector timestamp assigned to the transaction when it starts to execute, and `txnSQN` is the transaction's sequence number given upon commit.

A *replica*, or *site*, stores parts of the database and executes the transactions for which it is the host, and is formalized as an object of the following class:

```
class Replica | history : Datastore,  sqn : Nat,  gotTxns : ObjectList,
                executing : ObjectList,  committed : ObjectList,
                aborted : ObjectList,  committedVTS : VectorTimestamp,
                gotVTS : VectorTimestamp,  locked : Locks,
                votes : Vote,  voteSites : TxnSites,  abortSites : TxnSites,
                dsSites : PropagateSites,  vsbSites : TxnSites,
                dsTxns : OidSet,  gvTxns : OidSet,
                recPropTxns : PropagatedTxns, recDurableTxns : DurableTxns .
```

The `history` attribute represents the site's local database, as well as propagated updates also on data items not stored at the replica, as a map from keys to *lists* of updates `<` *value*, *version* `>`. The `sqn` attribute denotes the replica's current

local sequence number. The attributes `gotTxns`, `executing`, `committed` and `aborted` denote the transaction (objects) which are, respectively, waiting to be executed, executing, committed, and aborted. The attributes `committedVTS` and `gotVTS` indicate for each site how many transactions of that site have been committed at, respectively, received by, this site. `locked` denotes the locked keys and their associated transactions at this site. The attributes `recPropTxns` and `recDurableTxns` buffer the received propagation and disaster-safe durable messages from the coordinator. See [13] for an explanation of the other attributes.

The state also contains an object mapping each key to the sites storing the key (these sites are also called the *replicas* of the key):

```
class Table | table : ReplicaTable .
```

Messages between sites have the form `msg` *content* `from` *sender* `to` *receiver*. We only introduce the messages appearing in the rewrite rules shown in this paper. The message content (or simply message) `request(`*key*, *txn*, *vts*`)` sends a read request for transaction `txn` to *key*'s preferred site to retrieve its state from the snapshot determined by vector timestamp *vts*. The preferred site replies with a message `reply(`*txn*, *key*, *value_version*`)`, where *value_version* is chosen based on the incoming vector timestamp. The message `ds-durable(`*txn*`)` is sent to all sites once the transaction *txn* has been marked as disaster-safe durable. The sites then reply with a message `visible(`*txn*`)` to acknowledge the notification.

Formalizing Walter's Behavior. The following three rules show how the host site `RID` executes a read operation `X :=read K` in the currently executing transaction `TID` when the transaction has not already written data item `K` (`not $hasMapping(WS,K)`) and the site `RID` does not replicate data item `K` (`not localReplica(K,RID,REPLICA-TABLE)`). In this case, the site sends a `request` message (with the transaction's start vector timestamp `VTS`) to `K`'s preferred site (`preferredSite(...)`) to fetch the version. The "next operation" of the transaction changes to `waitRemote(K,X)`:[1]

```
crl [execute-read-remote] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Replica | executing :
          < TID : Txn | operations : ((X :=read K) OPS), writeSet : WS,
                        startVTS : VTS > >
  =>
    < TABLE : Table | >
    < RID : Replica | executing :
          < TID : Txn | operations : (waitRemote(K,X) OPS) > >
    (msg request(K,TID,VTS) from RID to preferredSite(K,REPLICA-TABLE))
  if (not $hasMapping(WS,K)) /\ (not localReplica(K,RID,REPLICA-TABLE)) .
```

The remote (preferred) site responds to this request by sending the snapshot-consistent value and version (`choose(VTS, DS[K])`) of the requested key:

[1] We do not give variable declarations, but follow the convention that variables are written in (all) capital letters.

```
rl [receive-remote-request] :
   (msg request(K,TID,VTS) from RID' to RID)
   < RID : Replica | history : DS >
 =>
   < RID : Replica | >
   (msg reply(TID,K,choose(VTS,DS[K])) from RID to RID') .
```

The executing site then merges the fetched value and version in the local history, and updates the read set and local variables:

```
rl [receive-remote-reply] :
   (msg reply(TID,K,< V,VERSION >) from RID' to RID)
   < RID : Replica | history : DS, executing :
         < TID : Txn | operations : (waitRemote(K,X) OPS), readSet : RS,
                       localVars : VARS > >
 =>
   < RID : Replica | executing :
         < TID : Txn | operations : OPS,
                       readSet : (versionRead(K,VERSION),RS),
                       localVars : insert(X,V,VARS) >,
                     history : merge(K,< V,VERSION >,DS) > .
```

The last rule we show concerns the propagation of committed transactions. If a transaction TID can be committed, it is propagated to the other sites. When a receiving site has received all transactions that causally precede TID and all transactions from TID's host site with smaller sequence numbers, the transaction TID is propagated successfully and this is acknowledged to the host site. When the host has received $f + 1$ such acknowledgments it marks TID as *disaster-safe durable* and sends a ds-durable message to each site. When a remote site receives this decision, the site tries to commit the transaction locally:

```
crl [receive-ds-durable-visible] :
    (msg ds-durable(TID) from RID' to RID)
    < RID : Replica | recPropTxns : (propagatedTxns(TID,SQN,VTS) ; PTXNS),
                      recDurableTxns : DTXNS, committedVTS : VTS',
                      locked : LOCKS >
  =>
    < RID : Replica | recDurableTxns : (durableTxns(TID) ; DTXNS),
                      committedVTS : insert(RID',SQN,VTS'),
                      locked : release(TID,LOCKS) >
    (msg visible(TID) from RID to RID')
    if VTS' gt VTS /\ s(VTS'[RID']) == SQN .
```

To commit transaction TID, the remote site must check that: (i) the propagation message has been received and acknowledged (propagatedTxns(TID,SQN,VTS) shown in recPropTxns); (ii) VTS' is greater than VTS; and (iii) all transactions from TID's host site with a smaller sequence number have been received (s(VTS'[RID']) == SQN). A visible message is then sent back, all corresponding locks are released, and the transaction is committed at the remote site.

4 Correctness Analysis

In this section we use reachability analysis—from all initial system configurations up to given bounds on the number of transactions, sites, etc.—to analyze whether Walter satisfies PSI and SI. To analyze these complex properties, we use a novel technique which adds a global "logical clock" to record the global order of transaction starts and commits/aborts.

4.1 Parametric Generation of Initial States

To analyze all possible initial configurations we introduce a new operator `init` so there is a one-step rewrite `init(`*parameters*`)` $\longrightarrow c_0$ for each possible initial configuration c_0. We declare a sort `ConfigSet` for *sets* of configurations, define a function `op initAux :` s_1 `...` s_n `-> ConfigSet` (see our report [13] for details) such that `initAux(`*params*,*params*′`)` generates all possible initial states for such parameters, and add the following rewrite rule to our model:

```
var C : Configuration .  var CS : ConfigSet .
crl [init] : init(params) => C if C ; CS := initAux(params,params') .
```

`init`'s parameters are the number of read-only transactions, the number of write-only transactions, the number of read-write transactions, the number of sites, the number of keys, and the replication factor. Each transaction has two operations.

One of 768 initial states generated by `init(1,1,1,2,2,2)` is

```
< 0 : Table | table : [replicatingSites(k1,1 2);;replicatingSites(k2,2 1)] >
< 1 : Replica |
       gotTxns : < 3 : Txn | operations : (k2l :=read k2) (k1l :=read k1),
                             readSet : empty, writeSet : empty,
                             localVars : (k1l |-> [0],k2l |-> [0]),
                             startVTS : empty, txnSQN : 0>,
       history : (k1 |-> <[0],version(0,0)>, k2 |-> <[0],version(0,0)>),
       sqn : 0,  ... >
< 2 : Replica |
       gotTxns : < 2 : Txn | operations : write(k2,1) write(k1,2), ... > ;;
                 < 1 : Txn | operations : (k2l :=read k2) write(k2,1), ... >,
       history : k1 |-> <[0],version(0,0)>,k2 |-> <[0],version(0,0)>), ... >
```

where data items `k1` and `k2` are replicated at sites `1` and `2`, and have preferred sites `1` and 2, respectively. Site `1` has one read-only transaction '`3`'—with operations `(k2l :=read k2) (k1l :=read k1)`—to execute, and site 2 has one write-only transaction '`2`' and one read-write transaction '`1`' to execute.

4.2 Analyzing the Correctness Properties

This section analyzes whether Walter satisfies SI and PSI using a new technique where we record relevant history during a run. The SI and PSI properties can then be easily formalized as functions on the final state of an execution.

In particular, we add to the state an object

```
< m : Monitor | clock : clock, log : log >
```

which stores crucial information about the execution. The *clock* is a kind of "logical global clock" that totally orders transaction starts and commits/aborts. This clock is incremented by one whenever a transaction starts executing or is committed or aborted somewhere. The *log* maps each transaction to a record `record(`*rid*, *issueTime*, *finishTimes*, *committed*, *reads*, *writes*`)`, with *rid* its host site, *issueTime* its issue "time" according to the logical clock, *finishTimes* its commit/abort "time" at each site, *committed* a flag that is `true` if the transaction is committed, *reads* its key/versions read, and *writes* its write set.

We modify our rewrite rules to update the `Monitor` whenever a transaction starts or is committed/aborted somewhere. For example, when a site commits a propagated transaction, the monitor records the commit time `GT` for that transaction at site `RID` and increments the logical global time by one:

```
crl [receive-ds-durable-visible] :
    (msg ds-durable(TID) from RID' to RID)
    < M : Monitor | clock : GT,
                    log : (TID |-> record(RID',T1,VTS1,true,READS,WRITES)
                                 , LOG) >
    < RID : Replica | ... >    --- as before
  =>
    < M : Monitor | clock : GT + 1 ,
                    log : (TID |-> record(RID',T1,insert(RID,GT,VTS1) ,
                                              true,READS,WRITES) , LOG) >
    < RID : Replica | ...  >   --- as before
    (msg visible(TID) from RID to RID')  if  ... .
```

Since Walter is terminating if a finite number of transactions are issued, we check the consistency properties by inspecting this monitor object in the final states, when all transactions have finished.

Formalizing Parallel Snapshot Isolation. As mentioned in Section 2, PSI is given by three properties [21]:

- PSI-1 (Site Snapshot Read): All operations read the most recent committed version at the transaction's site as of time when the transaction began.
- PSI-2 (No Write-Write Conflicts): The write sets of each pair of committed *somewhere-concurrent*[2] transactions must be disjoint.
- PSI-3 (Commit Causality Across Sites): If a transaction T_1 commits at a site A before a transaction T_2 starts at site A, then T_1 cannot commit after T_2 at any site.

[2] Two transactions are *somewhere-concurrent* if, at either host site, one of them has a commit timestamp between the start and the commit timestamp of the other.

We analyze PSI-2 (and all other properties) by searching for a reachable *final* state whose system log shows that the execution did not satisfy the property. The following function p2-psi checks whether PSI-2 holds in the execution reflected in the system log, by checking whether there is a write-write conflict between any pair of committed *somewhere-concurrent transactions* in the system log:

```
ops p1-psi p2-psi p3-psi : Log -> Bool .

ceq p2-psi(TID1 |-> record(RID1,TS1,(RID1 |-> TC , VTS1),true,RS,
                          (v(X,V),WS)) ,
           TID2 |-> record(RID2,TS2,(RID1 |-> TC' , VTS2),true,RS',
                          (v(X,V'),WS')) , LOG) = false
    if TC' > TS1 and TC' < TC .

eq p2-psi(LOG) = true [owise] .
```

In the first equation there are two committed (their "commit" flags are true) transactions TID1 and TID2—hosted, respectively, at sites RID1 and RID2—that both wrote data item X (since v(X, V) and v(X, V') are in their respective write sets). TID1 and TID2 are somewhere-concurrent, since they are concurrent at TID1's host site RID1: TID2 committed at RID1 at time TC', which is between TID1's start time TS1 and its commit time TC at RID1.

The function p3-psi that analyzes PSI-3 by checking whether there was "bad situation" in which a transaction TID1 committed at site RID2 *before* a transaction TID2 started at site RID2 (TC1 < TS2), while TID1 committed at site RID *after* TID2 committed at site RID (TC1 > TC2):

```
ceq p3-psi((TID1 |-> record(RID1,TS1,(RID2 |-> TC , RID |-> TC1 , VTS1),
                            true,RS,WS),
            TID2 |-> record(RID2,TS2,(RID1 |-> TC' ,RID |-> TC2 , VTS2),
                            true,RS',WS') , LOG)) = false
    if TC < TS2 /\ TC1 > TC2 .

eq p3-psi(LOG) = true [owise] .
```

We have equally easily defined a function p1-psi for property PSI-1, and functions p1-si and p2-si characterizing the executions where the requirements SI-1 and SI-2 for SI hold (see [13] for details).

We have analyzed Walter from *all* initial states with up to 3 transactions, 2 sites, 2 keys, and 2 replicas per key, as well as from a number of initial states with 3 transactions. The following command searches for a reachable final state where the log shows that PSI-2 is violated:

```
Maude> (search [1] init(1,0,2,2,2,2) =>!
          < M:Oid : Monitor | log : LOG:Log >  C:Configuration
             such that not p2-psi(LOG:Log) .)

No solution
```

Our analysis found that Walter may violate both SI-1 and SI-2, but did not uncover a violation of PSI. Each `search` command took about 2 h (worst-case) to execute on a 3.4 GHz $\times$ 8 Intel Core i7-2600 CPU with 11.7 GB memory.

5 Performance Estimation by Statistical Model Checking

In this section we use PVeStA statistical model checking to estimate the performance of Walter in a wider range of settings than in the experiments in [21], thereby providing further insight about Walter. For example, the experiments with fast commit in [21] assume full replication, whereas we also experiment with a partially replicated setting (which necessitates remote reads, etc.), and with workloads involving both slow and fast commits.

Probabilistic Modeling of Walter. For statistical model checking in PVeStA we need to eliminate nondeterminism in the untimed model in Section 3, and for performance estimation we need to add time and probabilities. All of this can be achieved by following the techniques in [8] and *probabilistically* assign to each message a *delay*. The idea is that if each rewrite rule is triggered by the arrival of a message (either directly, or indirectly by becoming enabled as a result of applying a rule that is triggered by the arrival of a message) and the message delay is sampled probabilistically from a dense time interval, then the probability that two messages have the same delay is 0, and hence no two actions are enabled simultaneously, eliminating nondeterminism and introducing time.

In more detail, nodes send messages of the form `[`Δ`,`*rcvr* `<-` *msg*`]`, where Δ is the message delay, *rcvr* is the recipient, and *msg* is the message content. When time Δ has elapsed, this message becomes a *ripe* message `{`T`,`*rcvr* `<-` *msg*`}`, where T is the "current global time" (used for analysis purposes only). Such a ripe message must then be consumed by the receiver *rcvr* before time advances.

We exemplify with the rule `receive-remote-request` how we have transformed the untimed non-probabilistic rewrite rules to the timed and probabilistic setting. In the probabilistic rule below, the incoming message `request` is equipped with the current global time `T`, and the outgoing message `reply` is equipped with a delay `D` sampled from the probability distribution `distr(...)`:

```
rl [receive-remote-request-prob] :
   {T, RID <- request(K,TID,VTS,RID')}
   < RID : Replica | history : DS >
 =>
   < RID : Replica | >
   [D, RID' <- reply(TID,K,choose(VTS,DS[K]),RID)]
  with probability D := distr(...) .
```

Extracting Performance Measures from Executions. This time we add to the state a monitor object `<` *m* `: Monitor | log:` *log* `>`. The logical clock is no longer needed, since now "real" time is given by the message arrival times. Furthermore, since we now analyze transaction throughput, the log is simpler: a list

of records `record(`*tid*, *issueTime*, *finishTime*, *committed*`)`, with *tid* the transaction identifier, *issueTime* its issue time, *finishTime* its commit/abort time, and *committed* a flag that is `true` if *tid* is committed.

We define a number of functions on (states with) such a monitor that extract different performance metrics from this "execution log." The function `throughput` computes the number of committed transactions per time unit:

```
op throughput : Config -> Float [frozen] .
eq throughput(< M : Monitor | log: LOG >  REST)
 = committedNumber(LOG) / totalRunTime(LOG) .
```

where `committedNumber` gives the number of committed transactions in `LOG` and `totalRunTime` returns the time when all transactions are finished (i.e., the largest *finishTime* in `LOG`) (see [13] for details).

Experimental Setup. We performed our experiments with 100 transactions, 1 or 5 operations per transaction, 100 keys, and up to 4 sites. The number of sites and the transaction size are the same as in the experiments in [21]. We used lognormal message delay distributions with parameters $\mu = 3$ and $\sigma = 1$ for local delays, and $\mu = 1$ and $\sigma = 2$ for remote delays.

Generating Initial States. Statistical model checking verifies a property up to a user-specified level of confidence by running Monte-Carlo simulations from a given initial state. We use an operator `probInit` to *probabilistically* generate initial states: `probInit(`*rtx*, *wtx*, *rwtx*, *sites*, *keys*, *rf*, *rops*, *wops*, *rwops*, *distr*`)` generates an initial state with *rtx* read-only transactions, *wtx* write-only transactions, *rwtx* read-write transactions, *sites* sites, *keys* keys, *rf* replication level, *rops* operations per read-only transaction, *wops* operations per write-only transaction, *rwops* operations per read-write transactions, and *distr* the key access distribution (the probability that an operation accesses a certain key). To capture the fact that some keys may be accessed more frequently than others, we also use Zipfian distributions in our experiments.

Each PVeStA simulation starts from `probInit`, which rewrites to a *different* initial state in each simulation. The reason is that this expression involves generating certain values—such as the transactions—*probabilistically.*

Statistical Model Checking Results. The plots in Fig. 1 show the *throughput* with only fast commit as a function of the number of sites, with read-only, write-only and 90% reads workload, and with uniform and Zipfian distributions. The plots show that read throughput scales nearly linearly with the number of sites; write throughput also grows with the number of sites, but not linearly. With a mixed workload, throughput is mostly determined by the transaction size. Our results are consistent with those in [21]. For uniform distribution we only plot the results with a mixed workload; for the other two workloads, the results are consistent with those using the Zipfian distribution.

The plots in Fig. 2 show the throughput with both fast and slow commit under the same experimental settings as in Fig. 1. As shown in the left plot,

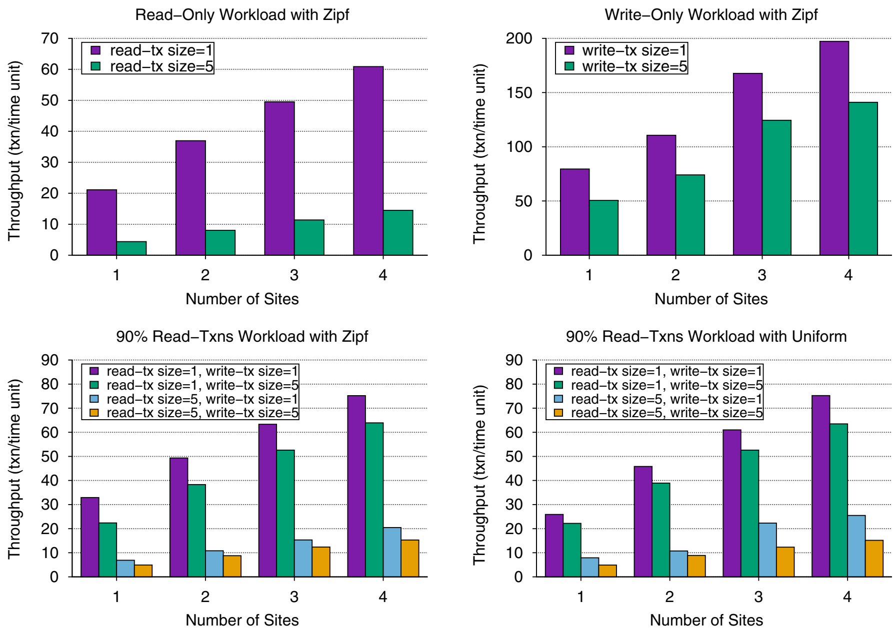

Fig. 1. Throughput with fast commit under different workloads.

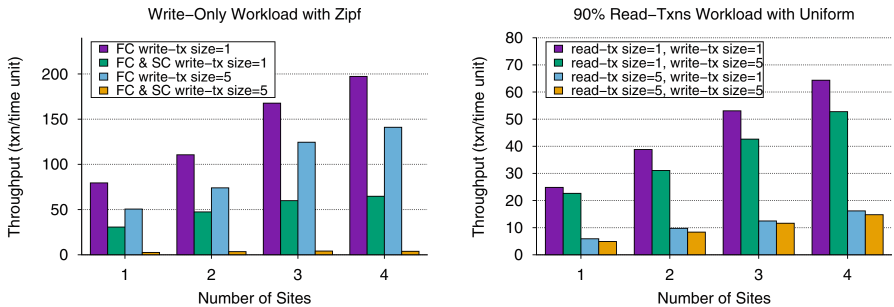

Fig. 2. Throughput with fast commit (FC) and slow commit (SC).

throughput is mostly determined by the transaction size in the mixed workload; the differences among various transaction sizes are consistent with those in Fig. 1.

Our Maude model, including the infrastructure for statistical model checking, is around 1.8K LOC. Computing the probabilities took a couple of minutes on 30 servers, each with a 64-bit Intel Quad Core Xeon E5530 CPU with 12 GB memory. Each point in the plots represents the average of 3 statistical model checking results. The confidence level for all our statistical experiments is 95%.

6 Related Work

Maude and PVeStA have been used to model and analyze the correctness and performance of a number of distributed data stores: the Cassandra key-value store [14,16], RAMP [11,15], Google's Megastore [9,10], and P-Store [18]. In contrast to these papers, our paper formalizes a different state-of-the-art algorithm, Walter, and, in particular, shows how the *snapshot isolation* and *parallel snapshot isolation* consistency models can be formalized and analyzed in Maude. In [12] we use PVeStA to compare the performance of our own new ROLA design with that of Walter. However, that paper focused on ROLA, and did not present the formalization of Walter or the SI and PSI properties.

In other applications of formal methods for distributed data stores, engineers at Amazon have used TLA+ and its model checker TLC to model and analyze the correctness of key parts of Amazon's celebrated cloud computing infrastructure [17]. In contrast to our work, they only use formal methods for correctness analysis. The designers of the TAPIR transaction protocol for distributed storage systems have also specified and model checked correctness (but not performance) properties of their design using TLA+ [23].

The papers [5,7] formalize a number of consistency models, including SI and PSI, but do not show how to analyze these properties.

7 Conclusions

We have formally analyzed and verified in Maude the design of Walter [21], a partially replicated distributed data store providing multi-partition transactions and guaranteeing parallel snapshot isolation (PSI), an important consistency property that offers attractive performance while providing adequate guarantees for certain kinds of applications. No formal specification of Walter existed before this work. Furthermore, PSI was only informally described by pseudo-code in [21] and no formal (or informal) verification existed. Using a logical clock to record the order of important events in an execution, we could use model checking and systematic generation of initial states to verify that Walter satisfies PSI for all such states. This technique should also make it easy to formalize and model check other consistency properties. We have also extended the Maude specification of Walter to model time and probabilistic communication delays as a probabilistic rewrite theory, and have then used statistical model checking analysis to study Walter's performance for a wide range of workloads. The results of the statistical model checking analysis are consistent with the experimental results in [21] but offer also new insights about Walter's performance for a wider range of workloads than those evaluated in [21].

Acknowledgments. We thank the anonymous reviewers for helpful comments on a previous version of this paper. This work has been partially supported by NSF Grant CNS 14-09416 and NRL under contract number N00173-17-1-G002.

References

1. Agha, G.A., Meseguer, J., Sen, K.: PMaude: rewrite-based specification language for probabilistic object systems. Electr. Notes Theor. Comput. Sci. **153**(2), 213–239 (2006)
2. AlTurki, M., Meseguer, J.: PVeStA: a parallel statistical model checking and quantitative analysis tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22944-2_28
3. Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: SIGMOD 1995. ACM (1995)
4. Bobba, R., et al.: Design, formal modeling, and validation of cloud storage systems using Maude. Technical report, University of Illinois at Urbana-Champaign (2017). http://hdl.handle.net/2142/96274
5. Cerone, A., Bernardi, G., Gotsman, A.: A framework for transactional consistency models with atomic visibility. In: CONCUR 2015. LIPIcs, vol. 42 (2015)
6. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
7. Crooks, N., Pu, Y., Alvisi, L., Clement, A.: Seeing is believing: a client-centric specification of database isolation. In: PODC 2017. ACM (2017)
8. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Statistical model checking for composite actor systems. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 143–160. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37635-1_9
9. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's Megastore in Real-Time Maude. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 494–519. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54624-2_25
10. Grov, J., Ölveczky, P.C.: Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 159–174. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_12
11. Liu, S., Ölveczky, P.C., Ganhotra, J., Gupta, I., Meseguer, J.: Exploring design alternatives for RAMP transactions through statistical model checking. In: Duan, Z., Ong, L. (eds.) ICFEM 2017. LNCS, vol. 10610, pp. 298–314. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_18
12. Liu, S., Ölveczky, P.C., Santhanam, K., Wang, Q., Gupta, I., Meseguer, J.: ROLA: a new distributed transaction protocol and its formal analysis. In: Russo, A., Schürr, A. (eds.) FASE 2018. LNCS, vol. 10802, pp. 77–93. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89363-1_5
13. Liu, S., Ölveczky, P., Wang, Q., Meseguer, J.: Formal modeling and analysis of the Walter transactional data store, report available at https://sites.google.com/site/siliunobi/walter
14. Liu, S., Ganhotra, J., Rahman, M., Nguyen, S., Gupta, I., Meseguer, J.: Quantitative analysis of consistency in NoSQL key-value stores. Leibniz Trans. Embed. Syst. **4**(1), 03:1–03:26 (2017)
15. Liu, S., Ölveczky, P.C., Rahman, M.R., Ganhotra, J., Gupta, I., Meseguer, J.: Formal modeling and analysis of RAMP transaction systems. In: SAC 2016. ACM (2016)

16. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 332–347. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11737-9_22
17. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)
18. Ölveczky, P.C.: Formalizing and validating the P-Store replicated data store in Maude. In: James, P., Roggenbach, M. (eds.) WADT 2016. LNCS, vol. 10644, pp. 189–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72044-9_13
19. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_26
20. Sen, K., Viswanathan, M., Agha, G.A.: VESTA: a statistical model-checker and analyzer for probabilistic systems. In: QEST 2005. IEEE Computer Society (2005)
21. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: SOSP 2011. ACM (2011)
22. Younes, H.L.S., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. Inf. Comput. **204**(9), 1368–1409 (2006)
23. Zhang, I., Sharma, N.K., Szekeres, A., Krishnamurthy, A., Ports, D.R.K.: Building consistent transactions with inconsistent replication. In: SOSP 2015. ACM (2015)