Exploring Design Alternatives for RAMP Transactions through Statistical Model Checking

Si Liu[1], Peter Csaba Ölveczky[2,1], Jatin Ganhotra[3], Indranil Gupta[1], and José Meseguer[1]

[1] University of Illinois at Urbana-Champaign
[2] University of Oslo
[3] IBM Research, New York

Abstract. In this paper we explore and extend the design space of the recent RAMP (Read Atomic Multi-Partition) transaction system for large-scale partitioned data stores. Arriving at a mature distributed system design through implementation and experimental validation is a labor-intensive task, so that only a limited number of design alternatives can be explored in practice. The developers of RAMP did implement and validate three design alternatives for RAMP, and sketched three additional designs. This work addresses two questions: (1) How can the design space of a distributed transaction system such as RAMP be explored with modest effort, so that substantial knowledge about design alternatives can be gained *before* designs are implemented? and (2) How realistic and informative are the results of such design explorations? We answer the first question by: (i) formally modeling eight RAMP-like designs (five by the RAMP developers and three of our own) in Maude as *probabilistic rewrite theories*, and (ii) using *statistical model checking* of those models to analyze key *performance metrics* such as throughput, average latency, and degrees of strong consistency and read atomicity. We answer the second question by showing that our quantitative analyses: (i) are consistent with the experimental results obtained by the RAMP developers for their implemented designs; (ii) confirm the conjectures made by the RAMP developers for their other three unimplemented designs; and (iii) uncover some promising new designs that seem attractive for some applications.

1 Introduction

The Problem. Distributed systems are remarkably hard to get right, both in terms of their correctness and in meeting desired performance requirements. Furthermore, in cloud-based storage systems, such as the RAMP (Read Atomic Multi-Partition) transaction system for large-scale partition data stores [5,6], whose design space we systematically explore in this paper, correctness and performance properties are intimately intertwined and need to be balanced out. This is because for cloud-based systems high availability is an essential requirement; but the CAP Theorem [9] states the intrinsic impossibility of having both efficiency (low latency) and strong consistency (correctness) in a distributed storage system. Therefore, tradeoffs that combine efficiency with weaker transactional guarantees (such as Read Atomicity (RA) for RAMP), are needed.

This makes the task of arriving at a good design of a large scale distributed storage system meeting both performance and correctness requirements highly non-trivial. Building such a system is challenging. Currently, to improve its performance the only available option is making changes to a typically very large source code base. This is very labor-intensive, has a high risk of introducing new bugs, and is not repeatable. In practice, very few design alternatives, which may in the end fail to lead to a better design, can be explored this way. In the case of RAMP [5,6], three designs were explored in detail, and three more were sketched out but not implement. Even for the three implemented RAMP designs, only a limited number of properties and performance parameters were actually evaluated, due to the serious effort involved in experimental evaluation.

Our Proposed Solution. Since design errors can be orders of magnitude more costly than coding errors, the most cost-effective application of formal methods is not as a *postmortem* analysis of an already implemented system, but during the design process, to maximize the chances of arriving at a good system design *before* it is implemented. In this way, formal methods can bring the power of the *Gedankenexperiment* to system design, greatly increasing the capacity for designers to explore design alternatives and subject them to rigorous analysis before implementation. For large scale distributed storage systems, not all formal methods can support well this process. Since in such systems correctness and performance are closely intertwined, the methods in question should support: (i) *executability*, so that formal specifications are easy to understand by designers and can serve as *system prototypes*; (ii) *qualitative analysis* of correctness properties with a Yes/Counterexample answer, hopefully automatically; and (iii) *quantitative analysis* of performance properties, also automatically.

In previous work [17], we used Maude [10] to develop formal, executable specifications of several RAMP designs, some proposed by the RAMP designers and some by us; and model checked such specifications in Maude to analyze consistency properties, thus meeting above requirements (i)–(ii). In this work we meet requirement (iii) by extending our previous specifications of RAMP designs as *probabilistic rewrite theories* [3] and exploring in depth various performance and consistency properties for eight RAMP designs, six of them never implemented before, through statistical model checking [22,24] using PVeStA [4].

Main Contributions. Our first main contribution is *methodological.* It applies not just to RAMP designs, but more broadly to the design of complex distributed systems with non-trivial correctness and performance requirements. Using RAMP as a case study, we illustrate in detail a formal method by which: (i) designers can easily and quickly develop formal executable models of alternative designs for a system; (ii) specifying system behavior with *probabilistic rewrite rules*, both performance and the degree to which correctness requirements are met can be modeled; (iii) alternative system designs can then be thoroughly analyzed through *statistical model checking* to measure and compare them agains each other along various performance and correctness dimensions, maximizing the chances of arriving at a design best meeting given requirements *before* implementation; and, as we show, (iv) a thorough analysis, widely ranging in properties and parameter choices, can be easily achieved, whereas a similar experimental evaluation would require prior implementation and a large effort.

A second, key contribution is the uncovering by the above method of *several unimplemented RAMP designs* that seem highly promising alternatives to the three already implemented. According to our analysis, they outperform all other designs among the eight we analyzed in key properties. Specifically: (1) the never implemented RAMP-F+1PW design sketched in [5,6] outperforms all others if 100% degree of read atomicity is given higher priority than any other correctness or performance properties; and (2) our own RAMP-Faster design outperforms all others in degree of strong consistency, throughput, and lower average latency, while still providing read atomicity for around 97% - 99% of the transactions with typical workloads. Modeling analyses do not provide as much assurance as experimental evaluations; however, our evaluations: (i) are consistent with those in [5,6] for the properties already measured experimentally for the implemented designs; (ii) are also consistent with the properties conjectured by the RAMP designers for their unimplemented designs; and (iii) they subject the eight designs to a considerably wider range of properties —and of parameter variations for each property— than any previous experimental evaluation, thus providing further insights about both the implemented and unimplemented designs.

The rest of the paper is organized as follows. Section 2 gives some background on RAMP, Maude, and statistical model checking with PVeStA. Section 3 presents our new RAMP design alternative, RAMP-Faster. Section 4 shows how we can specify our RAMP designs as probabilistic rewrite theories. Section 5 explains how we can evaluate the performance of the designs for different performance parameters and workloads, and shows the results of these evaluations. Section 6 discusses related work, and Section 7 gives some concluding remarks.

2 Preliminaries

2.1 Read-Atomic Multi-Partition (RAMP) Transactions

To deal with ever-increasing amounts of data, distributed databases *partition* their data across multiple servers. Unfortunately, many real-world systems do not provide useful semantics for transactions accessing multiple partitions, since the latency needed to ensure correct multi-partition transactional access is often high. Therefore, trade-offs that combine efficiency with weaker transactional guarantees for operations accessing multiple partitions are needed.

In [5,6], Bailis *et al.* propose a new isolation model, *read atomic* (RA) isolation, and *Read Atomic Multi-Partition* (RAMP) transactions, that together provide efficient multi-partition operations with the following guarantee: either all or none of a transaction's updates are visible to other transactions.

RAMP transactions use metadata and multi-versioning. Metadata is attached to each write, and the reads use this metadata to get the correct version. There are three versions of RAMP, which offer different trade-offs between the size of the metadata and performance: RAMP-Fast, RAMP-Small, and RAMP-Hybrid. In this paper we focus on RAMP-F and RAMP-S, which lie at the end points. The write protocols in these algorithms only differ in the amount of attached metadata. To guarantee that all partitions perform a transaction successfully or that none do, RAMP performs two-phase writes using the two-phase commit protocol (2PC). 2PC involves two phases: In the *prepare* phase, each timestamped write is sent to its partition, which adds the write to its local database. In the *commit* phase, each such partition updates an index which contains the highest-timestamped committed version of each item stored at the partition. The RAMP algorithms in [5] only deal with read-only and write-only transactions.

RAMP-Fast (abbreviated RAMP-F). In RAMP-Fast, read operations require one round trip time delay (RTT) in the common race-free case, and two RTTs in the worst case; writes require two RTTs. Read transactions proceed by first fetching the highest-timestamped committed *version* of each requested data item from the corresponding partition, and then decide if they have missed any version that has been prepared but not yet committed. The timestamp and the metadata from each version read induce a mapping from items to timestamps that records the highest-timestamped write for each transaction, appearing in the first-round read set. If the reader has a lower timestamp version than indicated in the mapping for that item, a second-round read will be issued to fetch the missing version. Once all the missing versions have been fetched, the client can return the resulting set of versions, which include both the first-round reads as well as any missing versions fetched in the second round of reads. The detailed specification of RAMP-Fast in [5] is shown in Appendix A.

RAMP-Small (abbreviated RAMP-S). Unlike RAMP-Fast, RAMP-Small read transactions proceed by first fetching the highest committed *timestamp* of each requested data item; the readers then send the entire set of those timestamps in a second message. The highest-timestamped version that also exists in the received set will be returned to the reader by the corresponding partition. RAMP-Small transactions require constant-size metadata, but require two RTTs for reads and writes. RAMP-Small writes only store the transaction timestamp, instead of attaching the entire write set to each write.

Extensions of RAMP. The paper [5] briefly discusses the following extensions and optimizations of the basic RAMP algorithms, but without giving any details:

- RAMP with one-phase writes (RAMP-F+1PW and RAMP-S+1PW), where writes only require one *prepare* phase, as the client can execute the *commit* phase asynchronously.
- RAMP with faster commit detection (RAMP-F+FC). If a server returns a version with the timestamp fresher than the highest committed version of the item, then the server can mark the version as committed. This allows faster updates to correct versioning and thus fewer round trip time delays.

In [17] we formalized these extensions in Maude and used Maude model checking to analyze their correctness properties. In [17] we also developed two new RAMP-like designs on our own, where RAMP-F and RAMP-S are executed without two-phase commit (denoted RAMP-F-2PC and RAMP-S-2PC). This allows interleaving of the prepare phase and the commit phase (unlike RAMP where those two phases are strictly ordered). In Section 3 we explain a third design of our own called RAMP-Faster.

2.2 Rewriting Logic and Maude

In rewriting logic [19] 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* [10], 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 [10] 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. An object of class C is modeled as a term `<` o `:` C `|` att_1 `:` v_1`,` att_2 `:` v_2`, ...,` att_n `:` v_n `>`, where o is 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 (with label `l`)

```
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.

2.3 Statistical Model Checking and PVeStA

Probabilistic distributed systems can be modeled as *probabilistic rewrite theories* [3] 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 additional new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ in the term t. Since for a given matching instance of the variables $\overrightarrow{x}$ there can be many (often infinite) ways to instantiate the extra variables $\overrightarrow{y}$, such a rule is *nondeterministic*. The probabilistic nature of the rule stems from the probability distribution $\pi(\overrightarrow{x})$, which depends on the matching instance of $\overrightarrow{x}$, and governs the probabilistic choice of the instance of $\overrightarrow{y}$ in the result $t'(\overrightarrow{x}, \overrightarrow{y})$.

Statistical model checking [22,24] is an attractive formal approach to analyzing probabilistic systems against temporal logic properties. 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. For example, a statistical model checking result may be "86.87% of the RAMP-F transactions satisfy read atomicity with 99% confidence." Existing statistical verification techniques assume that the system is purely probabilistic. Using the methodology in [3,11] we can eliminate nondeterminism in the choice of firing rules. We then use PVeStA [4], a parallelization of the tool VeStA [23], to statistically model check purely probabilistic systems against properties expressed by QuaTEx probabilistic temporal logic [3]. The expected value of a QuaTEx expression is iteratively evaluated w.r.t. two parameters α and δ provided as input by sampling until the size of `(1-`α`)100%` confidence interval is bounded by δ, where the result of evaluating a formula is not a Boolean value, but a real number.

3 The RAMP-Faster Design

We developed two new RAMP-like designs already in [17]. More recently, we have developed a third design, called RAMP-Faster, which also decouples two-phase commitment, but commits a write transaction in one RTT instead of the two RTTs required by writes in RAMP and RAMP without two-phase commit.

In RAMP-F, upon receiving a `prepare` message, the partition adds the timestamped write to its local database, and upon receiving the `commit` message, updates an index containing the highest-timestamped committed version of each item. In RAMP-Faster, a partition performs both operations upon receiving the `prepare` message, and hence requires only one RTT. Note that all information required to complete the two operations is provided by the `prepare` message: RAMP-Faster does not need to store more data than RAMP-F.

Since each write in RAMP-Faster needs only one RTT, it incurs lower latency per transaction and provides higher throughput. Furthermore, since writes are much faster, it seems reasonable to conjecture that there is a higher chance that reads will fetch the latest write; this means that RAMP-Faster should provide better consistency[4] than the other RAMP designs. Even though RAMP-Faster is not designed to guarantee read atomicity, as the client does not ensure that each partition has received the `prepare` message before issuing the `commit` message, it would be interesting to investigate whether RAMP-Faster provides read atomicity for a high percentage of transactions encountered in practice. If this were the case, RAMP-Faster would become an attractive option for multi-partition transactions where read atomicity, good consistency properties, and low latency are highly desired. Transaction systems are well-known to have issues w.r.t. latency to complete transactions [20]. RAMP-Faster would address this issue without compromising on the consistency. For example, in social networks we might tol-

[4] "Consistency" in such a non-replicated setting is understood as reads reading the "latest writes."

erate that a few transactions do not provide read atomicity if transactions are faster and reads show more recent writes instead of older ones.

4 Probabilistic Modeling of RAMP Designs

In [17] we describe how RAMP and its variations can be modeled in Maude for correctness analysis purposes. The state consists of a number of objects modeling partitions `<` p_i `: Partition | versions :` *ver*`, latestCommit :` *lc* `>`, with *ver* the versions of the items in the partition, and *lc* the timestamp of the latest commit of each item; and objects modeling clients `<` c_j `: Client | transac :` *txns*`, sqn :` *n*`, pendingOps :` *ops*`, pendingPrep :` *pw*`, 1stGets :` *1st*`, latest :` *latest* `>`, with *txns* a list of transactions the client wants to issue, *n* the sequence number that together with the client identifier determine timestamps, *ops* the pending reads/writes, *pw* the pending writes in the prepare phase, *1st* the pending first-round reads, and *latest* a mapping from each item to its latest committed timestamp from a client's perspective.

The models in [17] are untimed, non-probabilistic, and nondeterministic, so that Maude LTL model checking analyzes all possible interleavings. In this paper we are interested in estimating the *performance* (expected latency, percentage of transactions satisfying certain properties, etc.) of our designs. We therefore need to: (i) include time and probabilities in our models, and (ii) eliminate any nondeterminism, so that our models become purely probabilistic and can be subjected to statistical model checking.

The key idea to address both of these issues, following [11], is to *probabilistically* assign to each message a *delay*. The point regarding issue (ii) is that if: (a) each rewrite rule is triggered by the arrival of a message, and (b) the delay is sampled probabilistically from a dense/continuous time interval, then the probability that two messages have the same delay is 0, and hence no two actions could happen at the same time, eliminating nondeterminism.

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 show an example of how we have transformed the untimed non-probabilistic rewrite rules in [17] to the timed and probabilistic setting. All our models are available at `https://sites.google.com/site/siliunobi/ramp-smc`.

The following rewrites rules describe how a partition reacts when it receives a `commit` message from the client `O'` with transaction ID `TID`, operation ID `ID`, and timestamp `timestamp(O',SQN')`. The partition `O` invokes the function `cmt` to update the latest commit timestamp in the set `latestCommit` with the fresher timestamp of the incoming one and the local one; it then notifies the client to commit the write by sending the message `committed`. The difference between the untimed version (`[...-untimed]`) and the probabilistic version (`[...-prob]`) is that in the latter, the outgoing message `committed` is equipped with a delay `D` sampled from the probability distribution `distr(...)`.[5]

```
rl [on-receive-commit-untimed] :
     commit(TID, ID, ts(O', SQN')) from O' to O
     < O : Partition | versions : VS, latestCommit : LC >
  =>
     < O : Partition | versions : VS,
                       latestCommit : cmt(LC, VS, ts(O', SQN')) >
     committed(TID, ID) from O to O' .

crl [on-receive-commit-prob] :
     {T, O <- commit(TID, ID, ts(O', SQN'), O')}
     < O : Partition | versions: VS, latestCommit: LC, AS >
  =>
     < O : Partition | versions: VS,
                       latestCommit: cmt(LC, VS, ts(O', SQN')), AS >
     [D, O' <- committed(TID, ID, O)]
     with probability D := distr(...) .
```

4.1 Specifying Alternative RAMP Designs

A main advantage of the model-based approach is the ease with which new designs can be formalized and analyzed before implementation. We illustrate below how easily we can specify a number of RAMP design alternatives.

The main difference between our different versions of RAMP is how writes are committed; i.e., what happens when a node receives a `prepared` message. In the original RAMP, a client needs to check if all `prepared` messages are received (by checking if `IDS'` is `empty`) before starting to commit each write operation (using the function `startCommit` to generate all `commit` messages):[6]

```
 crl [receive-prepared-with-2PC] :
     {T, O <- prepared(TID, ID, O')}
     < O : Client | pendingPrep: IDS,  pendingOps: OI,  sqn: SQN, AS >
  =>
     < O : Client | pendingPrep: IDS', pendingOps: OI,  sqn: SQN, AS >
     (if IDS' == empty then startCommit(TID, OI, SQN, O) else null fi)
  if IDS' := delete(ID, IDS) .
```

In RAMP-F/S-2PC a client sends a `commit` message upon receiving a `prepared` message, which allows the asynchronous commitment of the write operation `ID` (no need to wait for all `prepared` messages before starting to commit):

```
 crl [receive-prepared-without-2PC] :
     {T, O <- prepared(TID, ID, O')}
```

[5] We do not show the variable declarations, but follow the Maude convention that variables are written with (all) capital letters.

[6] The variable `AS` of sort `AttributeSet` denotes the “other attributes” of the object.

```
     < O : Client | sqn: SQN, AS >
  =>
     < O : Client |  sqn: SQN, AS >
     [D, O' <- commit(TID,ID,ts(O,SQN),O)]
     with probability D := distr(...) .
```

RAMP-Faster integrates the two phases in writes: upon receiving a `prepare` message, the partition adds the incoming version to its local database `VS`, and also updates the index containing the highest-timestamped committed version of the item by invoking the function `cmt`:

```
crl [receive-prepare-faster] :
    {T, O <- prepare(TID,ID,X,V,ts(O',SQN),MD,O')}
    < O : Partition | versions: VS,  latestCommit: LC, AS >
 =>
    < O : Partition | versions: VS',
                      latestCommit: cmt(LC,VS',ts(O',SQN)), AS >
    [D, O' <- committed(TID,ID,O)]
 if VS' := (v(X,V,ts(O',SQN),MD),VS)
    with probability D := distr(...) .
```

5 Quantitative Analysis of RAMP Designs

The main difference between the RAMP designs in [5] and the three new designs we have proposed is that those in [5] guarantee read atomicity whereas ours do not. On the other hand, as mentioned in Section 3, we conjecture that our designs—in particular, RAMP-Faster—provide not only better performance (throughput, average latency, etc.) but also better "consistency" in the sense of reads more often reading the latest value written. If this is indeed the case, and, furthermore, a large fraction of transactions in representative workloads satisfy read atomicity, then our designs should be interesting for applications where read atomicity is highly desired but not an absolute requirement. For example, in a social network, read atomicity is desired (if A befriends B in a transaction, then another transaction should not observe a "fractured read" where A is a friend of B but where B is not a friend of A), but a small percentage of fractured reads might be acceptable if the performance becomes significantly better.

In this paper we compare the performance—along a number of performance parameters, including throughput, average latency, percentage of strongly consistent reads—of our own RAMP-like designs with the original RAMP designs using statistical model checking. The question is whether statistical model checking of probabilistic Maude models provides realistic performance estimates for RAMP designs. To answer this question, we compare the performance estimates obtained by our method with the implementation-based evaluations in [5].[7]

[7] Second-round reads and strong consistency are not considered in [5].

5.1 Extracting Performance Measures from Executions

For analysis purposes we add to the state an object

```
  < record : Monitor | log: log >
```

which stores crucial information about each transaction. The *log* is a list of records `record(`*tid*, *issueTime*, *commitTime*, *client*, *result*, *secRoundReads*`)`, with *tid* the transaction's ID, *issueTime* its issue time, *commitTime* its commit time, *client* the identifier of the client issuing the transaction, *result* the result of the transaction (for writes: the values written; for reads: the values read), and *secRoundReads* a flag that is `true` if the transaction required second-round reads.

We refine our models by updating the `Monitor` when needed. For example, when a client has received all `committed` messages (`allOpsCommitted(...)`), the monitor records the commit time (`T`) for that transaction. The client then also issues its next transaction, if any:

```
crl [receive-committed] :
    {T, O <- committed(TID, ID, O')}
    < M : Monitor | log: (LOG record(TID, T4, T', O, R, F) LOG') >
    < O : Client | transac: TRS,  sqn: SQN,  pendingOps: OI, AS >
  =>
    if allOpsCommitted(TID,OI')     *** commit a write txn ***
    then if TRS == nil              *** no more txns to issue ***
       then < M : Monitor | log: (LOG record(TID, T4, T, O, R, F) LOG') >
            < O : Client | transac: TRS, sqn: s SQN, pendingOps: OI', AS >
       else < M : Monitor | log: (LOG record(TID, T4, T, O, R, F) LOG') >
            < O : Client | transac: TRS, sqn: s SQN, pendingOps: OI', AS >
            [0.0, O <- next]  fi    *** issue next txn ***
    else < M : Monitor | log: (LOG record(TID, T4, T', O, R, F) LOG') >
         < O : Client | transac: TRS, sqn: SQN, pendingOps: OI', AS >  fi
  if OI' := remove(ID,OI) .
```

We can now define a number of functions on (states with) such a monitor that extract different performance parameters from the "system execution log."

Throughput. The function `throughput` computes the number of committed transactions per time unit. `size` computes the length of the `LOG`, and `totalRunTime` returns the time when all transactions are committed (i.e., the largest *commitTime* in `LOG`):

```
var C : Config .

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

Average Latency. The function `avgLatency` computes the average transaction latency by dividing the sum of all transaction latencies by the number of transactions. The first argument of the function `$avgLatency` computes the sum of all transaction latencies (time between the issue time and the commit time of a transaction), and the second argument computes the number of transactions:

```
op avgLatency : Config -> Float [frozen] .
op $avgLatency : Float Float Records -> Float .

eq avgLatency(< M : Monitor | log: LOG > C) = $avgLatency(0.0, 0.0, LOG) .
eq $avgLatency(N1, N2, (record(TID1, T1, T1', O1, R, F) LOG))
 = $avgLatency(N1 + (T1' - T1), N2 + 1.0, LOG) .
eq $avgLatency(N1, N2, nil) = N1 / N2 .
```

Percentage of Second-Round Reads. We complement [5] by exploring the fraction of transactions which needed second-round reads. This fraction is given by the function `2ndReadTxn`, where the function `2nd` counts the number of transactions which needed second-round reads (i.e., counts the number of flags set), and the function `sizeRO` counts the number of read-only transactions:

```
op 2ndReadTxn : Config -> Float [frozen] .
eq 2ndReadTxn(< M : Monitor | log: LOG > C) = 2nd(LOG) / sizeRO(LOG) .
```

Strong Consistency. Strong consistency means that each read transaction returns the value of the last write transaction that occurred before that read transaction. As all transactions from different clients can be totally ordered by their issuing times (stored in `Monitor`), we can define a function `sc` that computes the fraction of read-only transactions which satisfy strong consistency. Specifically, for each read transaction in *log*, we check backwards if its stored values match those of the last write transaction. If so, we count it as a transaction satisfying strong consistency. `sc` returns the number of the read transactions satisfying strong consistency divided by the number of read transactions.

The third parameter in the auxiliary function `$sc` counts the number of read-only transactions which satisfy strong consistency:

```
op sc : Config -> Float .
op $sc : Records Records Float -> Float .
eq sc(< M : Monitor | log: LOG > C) = $sc(LOG,LOG,0.0) .
```

When all records in *log* have been checked, `$sc` returns the percentage of read-only transactions which satisfy strong consistency:

```
eq $sc(nil,LOG,N) = N / sizeRO(LOG) .
```

By checking the records in *log*, we always start from the head of the list: if the head is a write transaction denoted by `null`, then we simply ignore it:

```
eq $sc((record(TID1,T1,T1',O1,(ID1 |-> v(X,V1,null,MD1),R)) LOG),LOG',N)
   = $sc(LOG,LOG',N) .
```

In the case the head is a read transaction denoted by both TS1 and TS2 not null, if it has no write transactions before, then it will satisfy strong consistency as long as it returns any of the write transactions:

```
ceq $sc((record(TID1,T1,T1',O1,(ID1 |-> v(X,V1,TS1,MD1),
                                 ID2 |-> v(Y,V2,TS2,MD2))) LOG),
       (LOG1 record(TID1,T1,T1',O1,(ID1 |-> v(X,V1,TS1,MD1),
                                   ID2 |-> v(Y,V2,TS2,MD2)))
        LOG2 record(TID2,T2,T2',O2,(ID3 |-> v(X,V1,null,MD3),
                                   ID4 |-> v(Y,V2,null,MD4))) LOG3),N)
     = $sc(LOG,(LOG1 record(TID1,T1,T1',O1,(ID1 |-> v(X,V1,TS1,MD1),
                                   ID2 |-> v(Y,V2,TS2,MD2)))
        LOG2 record(TID2,T2,T2',O2,(ID3 |-> v(X,V1,null,MD3),
                                   ID4 |-> v(Y,V2,null,MD4))) LOG3),N + 1.0)
       if noWrites?(LOG1) and TS1 =/= null and TS2 =/= null .
```

Otherwise if it has any write transaction before, then it will satisfy strong consistency as long as if it returns the last write transaction in *log*:

```
ceq $sc((record(TID1,T1,T1',O1,(ID1 |-> v(X,V1,TS1,MD1),
                                 ID2 |-> v(Y,V2,TS2,MD2))) LOG),
       (LOG1 record(TID2,T2,T2',O2,(ID3 |-> v(X,V1,null,MD3),
                                   ID4 |-> v(Y,V2,null,MD4)))
        LOG2 record(TID1,T1,T1',O1,(ID1 |-> v(X,V1,TS1,MD1),
                                   ID2 |-> v(Y,V2,TS2,MD2))) LOG3),N)
     = $sc(LOG,(LOG1 record(TID2,T2,T2',O2,(ID3 |-> v(X,V1,null,MD3),
                                            ID4 |-> v(Y,V2,null,MD4)))
        LOG2 record(TID1,T1,T1',O1,(ID1 |-> v(X,V1,TS1,MD1),
                                   ID2 |-> v(Y,V2,TS2,MD2))) LOG3),N + 1.0)
       if noWrites?(LOG2) and TS1 =/= null and TS2 =/= null .
```

The following deals with the rest cases:

```
eq $sc((record(TID1,T1,T1',O1,(ID1 |-> v(X,V1,TS1,MD1),
                                ID2 |-> v(Y,V2,TS2,MD2))) LOG),LOG',N)
     = $sc(LOG,LOG',N) [owise] .
```

Read Atomicity. A system provides *read atomic* isolation if it prevents fractured reads, and also prevents transactions from reading uncommitted, aborted, or intermediate data. A transaction T_j exhibits *fractured reads* if transaction T_i writes version x_m and y_n, T_j reads version x_m and version y_k, and $k < n$ [5].

The function `ra` computes the fraction of read transactions which satisfy read atomic isolation. For each read transaction in *log*, it checks if its stored values match those of any write transaction. If so, the transaction satisfies read atomicity. `ra` returns the number of those transactions satisfying read atomicity divided by the number of read transactions.

The third argument of the auxiliary function `$ra` counts the number of read-only transactions which satisfy read atomicity:

```
op ra : Config -> Float .
op $ra : Records Records Float -> Float .
eq ra(< M : Monitor | log: LOG > C) = $ra(LOG,LOG,0.0) .
```

When all records in *log* have been checked, `$ra` returns the fraction of read-only transactions which satisfy read atomicity:

```
eq $ra(nil,LOG,N) = N / sizeRO(LOG) .
```

When checking the records in *log*, we always start from the head of the list: if the head is a write transaction denoted by `null`, then we simply ignore it:

```
eq $ra((record(TID1,T1,T1',O1,(ID1 |-> v(X,V1,null,MD1),R)) LOG),LOG',N)
    = $ra(LOG,LOG',N) .
```

In the case the head is a read transaction denoted by both TS1 and TS2 not null, if it has no write transactions before, then it will satisfy read atomicity as long as it matches any of the write transactions:

```
ceq $ra((record(TID1,T1,T1',O1,(ID1 |-> v(X,V1,TS1,MD1),
                                ID2 |-> v(Y,V2,TS2,MD2))) LOG),
       (LOG1 record(TID2,T2,T2',O2,(ID3 |-> v(X,V1,null,MD3),
                                    ID4 |-> v(Y,V2,null,MD4))) LOG2),N)
     = $ra(LOG,(LOG1 record(TID2,T2,T2',O2,(ID3 |-> v(X,V1,null,MD3),
                        ID4 |-> v(Y,V2,null,MD4))) LOG2),N + 1.0)
      if TS1 =/= null and TS2 =/= null .
```

We also count the read-only transaction which returns the initial value:

```
ceq $ra((record(TID1,T1,T1',O1,(ID1 |-> v(X,0,TS1,MD1),
                                 ID2 |-> v(Y,0,TS2,MD2))) LOG),LOG',N)
     = $ra(LOG,LOG',N + 1.0)
      if TS1 =/= null and TS2 =/= null .
```

The following deals with the rest cases:

```
eq $ra((record(TID1,T1,T1',O1,(ID1 |-> v(X,V1,TS1,MD1),
                                ID2 |-> v(Y,V2,TS2,MD2))) LOG),LOG',N)
    = $ra(LOG,LOG',N) [owise] .
```

5.2 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 `init` to probabilistically generate initial states. `init(`*rtx*`,` *wtx*`,` *clients*`)` generates an initial state with *rtx* number of read-only transactions, *wtx* number of write-only transactions, and *clients* number of clients. We use two partitions and two data items `x` and `y`, with each partition storing one data item. The following parts of the initial states are chosen probabilistically by uniform sampling from the given distribution: (i) whether a read-only or write-only transaction is generated next, and (ii) which client is the issuer of the generated transaction. Each transaction consists of two operations, on different data items.

Each PVeStA simulation starts from `init(`*rtx*`,`*wtx*`,`*clients*`)`, which rewrites to a *different* initial state in each simulation. The reason is that this expression involves choosing certain values probabilistically. `init` is defined as follows:

```
op init : NzNat NzNat NzNat -> Config .
eq init(RTX, WTX, CLIENTS)
 = {0 | nil}  < record : Monitor | log: nil >
   < x : Partition | versions: (v(x, 0, null, empty)),
                     latestCommit: (x |-> ts(0, 0)) >
   < y : Partition | versions: (v(y, 0, null, empty)),
                     latestCommit: (y |-> ts(0, 0)) >
   generateClientsAndTranses(RTX, WTX, CLIENTS) .
```

When generating clients and transactions, we first generate the clients; then we generate the next transaction and assigns it probabilistically to some client:

```
op generateClientsAndTranses : NzNat NzNat NzNat -> Config .
op genCT : Nat Nat Nat NzNat Config -> Config .

eq generateClientsAndTranses(RTX, WTX, CLIENTS)
 = genCT(RTX, WTX, CLIENTS, CLIENTS, null) .

*** first generate clients and add then to the last parameter:
eq genCT(RTX, WTX, s CLIENTS, CLIENTS2, C)
 = genCT(RTX, WTX, CLIENTS, CLIENTS2, C
       < s CLIENTS : Client | transac: nil, sqn: 1, pendingOps: empty,
                      pendingPrep: empty, 1stGets: empty,
                      latest: empty, result: nil > {d,s CLS <- start}) .
```

When all clients have been generated, we generate transactions one by one, and assign each one to a client. The following probabilistic rule treats the case when the number of clients left to generate is 0, and the number of read (`s RTX` (= `RTX` + 1)) and write (`s WTX`) transactions to generate both are greater than 0:

```
crl [genTrans] :
    genCT(s RTX, s WTX, 0, CLIENTS, C)
 =>
    if R-OR-W < s RTX     *** new read transaction
    then genCT(RTX, s WTX, 0, CLIENTS, addReadTrans(CLIENT + 1, C))
    else genCT(s RTX, WTX, 0, CLIENTS, addWriteTrans(CLIENT + 1, C)) fi
    with probability R-OR-W := sampleUniWithInt(s RTX + s WTX) /\
                     CLIENT := sampleUniWithInt(CLIENTS) .
```

This rule first probabilistically decides whether the next transaction is a read or a write transaction. Since the probability of picking a read transaction should be $\frac{\#\mathit{readsLeft}}{\#\mathit{txnLeft}}$, it uniformly picks a value `R-OR-W` from $[0, \ldots, \#\mathit{txnLeft} - 1]$ (the number of transactions left to generate is `s RTX + s WTX`) using the expression `sampleUniWithInt(s RTX + s WTX)`. If the value picked is in $[0, \ldots, \#\mathit{readsLeft} - 1]$ (`< s RTX`), we generate a new read transaction next (`then` branch); otherwise we generate a new write transaction (`else` branch). But which client should issue the transaction? The clients have identities `1`, `2`, ..., `n`, where `n` is the number of clients (`CLIENTS`). The expression `sampleUniWithInt(CLIENTS) + 1` (i.e., `CLIENT + 1`) gives us the client, sampled uniformly from $[1, \ldots, n]$.

If the remaining transactions to generate are only read-only transactions, then we *uniformly* assign the transaction to a client:

```
crl genCT(s RTS,0,0,CLIENTS,C)
 =>
    genCT(RTS,0,0,CLIENTS,addReadTrans(CLIENT + 1,C))
    with probability CLIENT := sampleUniWithInt(CLIENTS) .
```

The following is the case where the remaining transactions to generate are only write-only transactions:

```
crl genCT(0,s WTX,0,CLIENTS,C)
 =>
    genCT(0,WTX,0,CLIENTS,addWriteTrans(CLIENT + 1,C))
    with probability CLIENT := sampleUniWithInt(CLIENTS) .
```

The following defines the functions `addReadTrans` which generates a new read-only transaction:

```
*** Assuming two operations per transaction with each on a different data item
op addReadTrans : Address Config -> Config .

*** If this is the first transaction to generate for a client
eq addReadTrans(O,< O : Client | transac: nil,  AS > C) =
     < O : Client | transac: ((read(O * x + 1,O * x + 1,x),
     read(O * x + 1,O * x + 2,y))), AS > C .
```

```
*** If the previous transaction is a read-only transaction
eq addReadTrans(O,< O : Client | transac: (TRS (read(ID1,ID2,x),
    read(ID1,ID3,y))), AS > C) =
    < O : Client | transac: (TRS (read(ID1,ID2,x),read(ID1,ID3,y))
    (read(ID1 + 1,ID2 + 2,x),read(ID1 + 1,ID3 + 2,y))), AS > C .

*** If the previous transaction is a write-only transaction
eq addReadTrans(O,< O : Client | transac: (TRS (write(ID1,ID2,x,ID2),
     write(ID1,ID3,y,ID3))), AS > C) =
     < O : Client | transac: (TRS (write(ID1,ID2,x,ID2),write(ID1,ID3,y,ID3))
     (read(ID1 + 1,ID2 + 2,x),read(ID1 + 1,ID3 + 2,y))), AS > C .
```

Similarly, the following is the case which deal with generating a writing-only transaction:

```
*** Assuming two operations per transaction with each on a different data item
op addWriteTrans : Address Config -> Config .

*** If this is the first transaction to generate for a client
eq addWriteTrans(O,< O : Client | transac: nil, AS > C) =
     < O : Client | transac: ((write(O * x + 1,O * x + 1,x,O * x + 1),
     write(O * x + 1,O * x + 2,y,O * x + 2))), AS > C .

*** If the previous transaction is a read-only transaction
eq addWriteTrans(O,< O : Client | transac: (TRS (read(ID1,ID2,x),
     read(ID1,ID3,y))), AS > C) =
     < O : Client | transac: (TRS (read(ID1,ID2,x),read(ID1,ID3,y))
     (write(ID1 + 1,ID2 + 2,x,ID2 + 2),write(ID1 + 1,ID3 + 2,y,ID3 + 2))), AS > C .

*** If the previous transaction is a write-only transaction
eq addWriteTrans(O,< O : Client | transac: (TRS (write(ID1,ID2,x,ID2),
     write(ID1,ID3,y,ID3))), AS > C) =
     < O : Client | transac: (TRS (write(ID1,ID2,x,ID2),write(ID1,ID3,y,ID3))
     (write(ID1 + 1,ID2 + 2,x,ID2 + 2),write(ID1 + 1,ID3 + 2,y,ID3 + 2))), AS > C .
```

When there are no more transactions or clients left to generate, `genCT` returns the generated client objects (each with a list of transactions to issue):

```
eq genCT(0, 0, 0, CLIENTS, C) = C .
```

5.3 Statistical Model Checking Results

This section shows the result of using statistical model checking from the initial states in Section 5.2 to compare all eight RAMP versions w.r.t. the performance and consistency measures defined in Section 5.1.

In our experiments we use lognormal distribution for message delay with the mean μ = `0.0` and standard deviation σ = `1.0` [8]. All properties are computed with a 99% confidence level of size at most 0.01 (Section 2.3). We could not find the distribution used in [5] for message delays, so we use those in [15]. Due to state space limitations, our analyses consider a limited number of data items (2), operations per transaction (2), clients (up to 50), and transactions (up to 400). We consider not only the 95% read transaction and 5% write transaction proportion workloads in [5], but also explore how the RAMP designs behave for different read/write proportions.

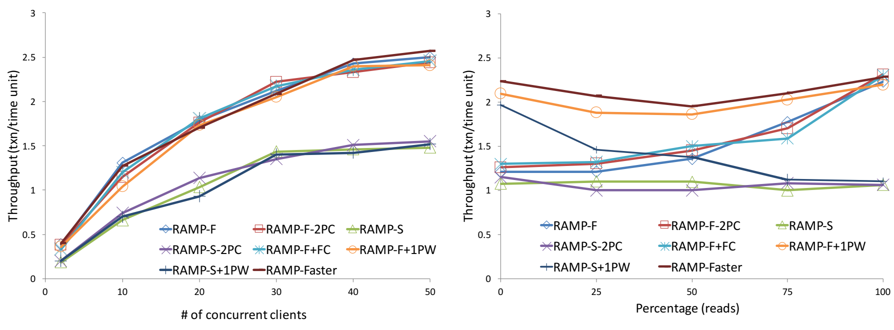


Fig. 1. Throughput under varying client and read load.

Throughput. Figure 1 shows the resulting of analyzing throughput against the number of concurrent clients (left) and percentage of read transactions (right).[8]

For the original RAMP designs, under a 95% read proportion, as the number of clients increases, both RAMP-F and RAMP-S's throughput increases, and RAMP-F provides higher throughput than RAMP-S. As the read proportion increases, RAMP-F's throughput increases, while RAMP-S's throughput keeps nearly constant; and RAMP-F also outperforms RAMP-S in throughput. These observations are consistent with the experimental results in [5].

There are no conjectures in [5] about the throughput of the designs that were only sketched in [5]. We observe that unlike other RAMP-F-like algorithms, whose throughput increases as read activities increase, RAMP-F+1PW's throughput keeps high with all reads/writes. As the right plot shows, at the beginning, when there are more writes than reads, RAMP-F+1PW and RAMP-Faster perform better than other RAMP-F-like designs. This happens because RAMP-F requires two RTTs for a write, RAMP-F+1PW needs only one RTT and RAMP-Faster, our proposed design, performs commit when the PREPARE message is received. Hence, with all write transactions, RAMP-F+1PW and RAMP-Faster will always provide higher throughput. However, as read activities increase, other RAMP-F-like designs increase their throughput, as they

[8] Larger versions of our figures can be found in the appendix.

require one RTT for all reads. Even though as the percentage of reads increases, RAMP-F+1PW and RAMP-Faster compensate the extra RTT incurred due to the races, with the RTT saved during the write operations.

The RAMP-S-like designs provide lower throughput than the RAMP-F-like designs, which is consistent with the observations in [5]. As expected, as the read percentage increases, RAMP-S+1PW's throughput converges with those of other RAMP-S-like designs, because all RAMP-S-like designs require more RTTs for reads compared to RAMP-F even when there is no race between reads and writes. In the worst case, when there is a race between read and write operations, all designs require two RTTs for reads.

Regarding our own designs, RAMP-Faster provides the highest throughput with varying read load and with larger number of concurrent clients among all RAMP versions. One reason is that RAMP-Faster's writes need only one RTT. RAMP-F-2PC (or RAMP-S-2PC) is not competitive with RAMP-F (or RAMP-S) regarding throughput. The reason is that, although they sacrifice 2PC, they still need to commit each write operation before committing the write transaction, which brings no apparent difference in throughput.

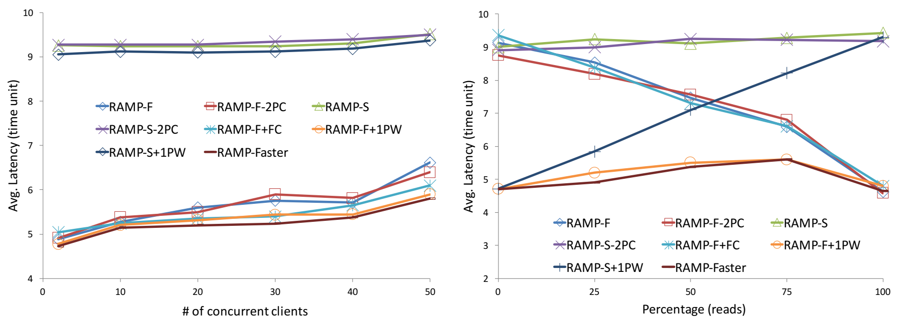


Fig. 2. Average transaction latency under varying client and read load.

Average Latency. Figure 2 shows the average transaction latency as the number of concurrent clients (left) and the proportion of read transactions (right) increases.

Under a 95% read proportion, as the number of concurrent clients increases, the RAMP-F versions' average latency increases slightly, and the RAMP-S versions are almost twice as slow as the RAMP-F variations. And although RAMP-F+1PW and RAMP-S+1PW as expected have lower latencies than RAMP-F and RAMP-S, respectively, the differences are surprisingly small. In the same way, removing 2PC does not seem to help much. Although the differences are small, RAMP-Faster is the fastest, followed by RAMP-F with one-phase writes.

In Fig. 2(right) we see that RAMP-F+1PW and RAMP-Faster significantly outperform all the other algorithms when the proportion of write transactions is between 25% and 75-80%.

Regarding our own designs, it seems that RAMP-F-2PC (or RAMP-S-2PC) is not competitive with RAMP-F (or RAMP-S) regarding average latency. The reason is that, although they sacrifice 2PC, they still need to commit each write operation before committing the write transaction, which brings no apparent difference in latency. RAMP-Faster incurs the lowest average latency among all RAMP versions with varying client and read load.

Second-Round Reads. Figure 3 shows the percentage of transactions requiring second-round reads for different number of clients (left) and read proportions (right).

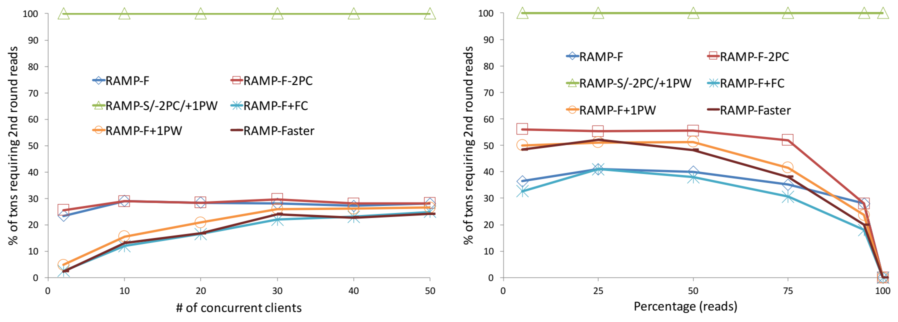


Fig. 3. Percentage of transactions requiring second-round reads under varying client and read load.

As expected, all RAMP-F versions require significantly fewer second-round reads than the RAMP-S variations, which always require second-rounds reads.

Regarding the sketched RAMP designs in [5], we observe that RAMP-F+FC requires less second-round reads than RAMP-F, since allows faster commit.

Regarding our own designs, we observe that RAMP-F-2PC requires more second-round reads than RAMP-F. The reason is that before a transaction commits, RAMP-F-2PC allows interleaving of the `prepare` phase and the `commit` phase, which increases the chances of reading from a partially-committed write transaction. Although RAMP-Faster also decouples 2PC, it requires fewer second-round reads than RAMP-F (with 95% read load; left). The reason is that it commits a write transaction in one RTT, and thus each partition adds the write to its local database, and updates an index containing the highest-timestamped committed version of each item at the same time. Furthermore, decreased write activity leads to fewer races between reads and writes. Thus, compared to RAMP-

F, RAMP-Faster decreases the chances of fetching from a partially-committed write transaction under a read-heavy workload. However, under a more write-heavy workload, RAMP-Faster need somewhat more second-round reads than RAMP-F and RAMP-F with fast commit.

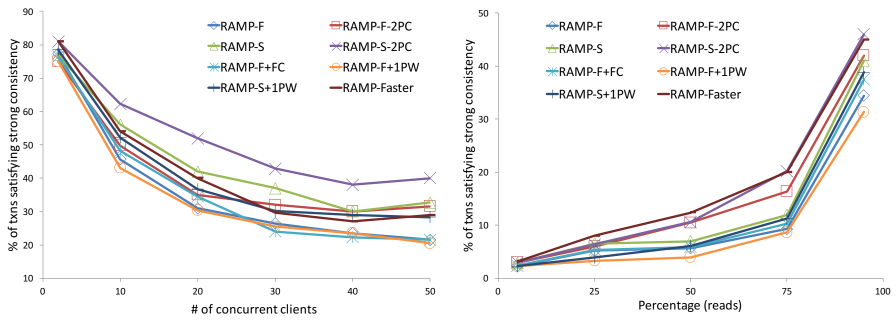


Fig. 4. Probability of satisfying strong consistency under varying client and read load.

Strong Consistency. Figure 4 shows the percentage of transactions satisfying strong consistency under varying number of clients and read/write proportions.

In all RAMP designs, the probability of satisfying strong consistency decreases as the number of clients increases, since there are more races between reads and writes, which decreases the probability of reading the preceding write.

It is natural that the percentage of transactions satisfying strong consistency increases as the reads increase: the chance of reading the latest preceding write should increase when writes are few and far between.

We also observe that RAMP-S-like designs (i.e., RAMP-S/+1PW/-2PC) provide more strong consistency than their RAMP-F counterparts. The reason is that RAMP-S-like designs always use second-round reads, which might increase the chance of reading the latest write. The only exception seems to be that RAMP-Faster outperforms all the other RAMP designs for 25-75% read workloads. The reason is that RAMP-Faster only requires one RTT for a write to commit, which increases a read transaction's chance to fetch the latest write.

Read Atomicity. Figure 5 shows the percentage of transactions satisfying read atomicity. As it should be, all designs in [5] satisfy read atomic isolation. Our own design alternatives provide 92-100% read atomicity under all scenarios considered. With 95% read transactions, they all offer 97-100% read atomicity.

Summary. Our formal model-based methodology has allowed us to quickly and easily analyze the expected performance of a large number of RAMP de-

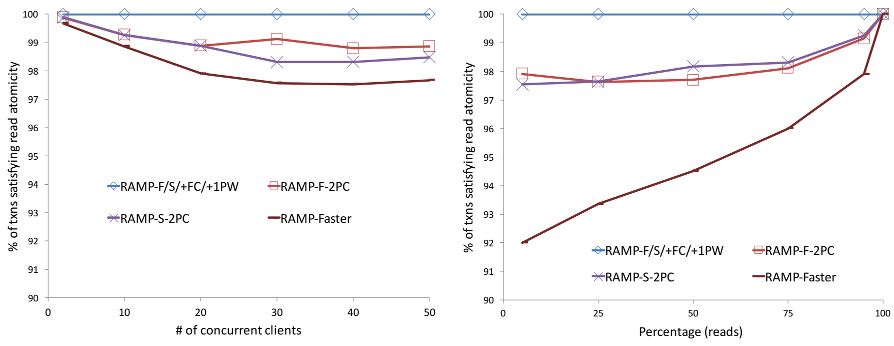


Fig. 5. Probability of satisfying read atomicity under varying client and read load.

signs along a number of performance parameters and with varying number of concurrent clients and read/write transaction proportions. This allows to predict which design is the best fit for a particular application.

Our results are consistent with the experimental results in [5]. For example: the throughput of both RAMP-F and RAMP-S increases with the number of concurrent clients, and RAMP-F provides higher throughput than RAMP-S; the latency also increases with the increase of concurrent clients (very minimally for RAMP-S, however).

Our results also confirm the conjectures about the sketched designs in [5], which were never experimentally validated by the RAMP developers. For example: RAMP-F+FC and RAMP-F+1PW have lower latency than RAMP-F (and similarly for RAMP-S). We can also compare RAMP-F-FC with RAMP-F+1PW, and see that RAMP-F+1PW typically provides better performance among these two optimizations.

We can also evaluate our own designs. It turns out that RAMP without 2PC does not improve the performance of RAMP. On the other hand, RAMP-Faster is an interesting design, as it generally provides the smallest average latency and highest throughput among all RAMP designs, in particular when there are a fair amount of write transactions, while providing more than 92% read atomicity even for very write-heavy workloads. Maybe slightly surprisingly, RAMP-Faster does not provide the highest percentage of strongly consistent reads for read-heavy workloads, but does so for workloads with 25-75% read transactions.

Note that the actual values might differ between the experiments in [5] and our statistical analysis, due to factors like hard-to-match experimental configurations, the inherent difference between statistical model checking and implementation-based evaluation[9], processing delay at client/partition side, and different distri-

[9] In general, implementation-based evaluation is based on a single trace of hundreds of thousands of transactions, while statistical model checking is based on sampling

butions of item accesses. The important observation is that the relative performance in both sides are similar.

It is also worth remarking we only use two data items, while the experiments in [5] use up to thousands. This implies that we "stress" the algorithms much more, since there are much fewer potential clashes between small transactions (typically with four operations) in a 1000-data-object setting that between our two-operation transactions on two data objects.

All the RAMP models considered in this paper consist of around 4000 lines of code, and the time to compute the probabilities for strong consistency is around 15 hours (worst-case), and for other metrics is around 8 hours (worst-case) with a workload of 400 transactions on a 2.7 GHz Intel Core i5 CPU with 8 GB memory. Each point in the plots represents the average of three statistical model checking results. The upper bound for model runtime depends on the confidence level of our statistical model checker (99% confidence level for all our experiments).

6 Related Work

Maude for Distributed Storage Systems. In [17] we formalized RAMP and some proposed extensions, and used Maude model checking to analyze their correctness properties. In contrast, this paper focuses on analyzing the performance of RAMP and its variations using statistical model checking with PVeStA. In addition, this paper also introduces our most promising RAMP design: RAMP-Faster. The papers [13,14] use Maude to formalize and analyze Google's Megastore and a proposed extension. Those papers also focus on correctness analysis, although they present some *ad hoc* performance estimation using randomized Real-Time Maude simulations. In contrast to the methodology in this paper, such ad hoc simulations cannot give any measure of statistical confidence in the results. The papers [15,16,18] describe how the Cassandra key/value store has been analyzed both for correctness and performance using Maude and PVeStA. The main differences between [15,16,18] and this paper are: Cassandra only supports single read and write operations, whereas RAMP supports transactions, which also implies that the consistency levels to analyze in RAMP are more complex; in this paper we also propose a promising variation of the system (RAMP-Faster); and, finally, we compare our performance estimates with those obtained by the RAMP developers' simulations, whereas [15,16] compares the model-based performance estimates with those obtained by running the system.

Model-Based Performance Estimation of Distributed Storage Systems. Despite the importance of globally-distributed transactional databases, we are not aware of work on (formal) model-based performance analysis of such systems. One reason might be that the most popular state-of-art formal tools supporting probabilistic/statistical model checking are mainly based on automata (e.g., Uppaal SMC [2] and Prism [1]), and it is probably just too hard, or impossible, to model

hundreds of thousands of Monte-Carlo simulations of hundreds of transactions up to a certain statistical confidence.

such complex artifacts as state-of-the-art distributed transactional systems using timed/probabilistic automata. Another reason might be that NoSQL stores became mainstream earlier than globally-distributed transactional databases and gathered attention from the research community to work on model-based performance analysis of NoSQL stores [7,12,21].

7 Concluding Remarks

We have explored eight design alternatives for RAMP transactions following a general methodology based on formal modeling with probabilistic rewrite rules and analyzing performance using statistical model checking. Substantial knowledge about both implemented and unimplemented RAMP designs has thus been gained. This knowledge can help find the best match between a given RAMP version and a class of applications. For example, we now know how the different designs behave not just for read-intensive workloads, but understand their behavior across the entire spectrum from read-intensive to write-intensive tasks.

Our work has also shown that it is possible to use this methodology to identify promising new design alternatives for given classes of applications relatively easily *before* they are implemented. This of course does not replace the need for implementation and experimental validation, but it allows us to focus implementation and validation efforts where they are most likely to pay off.

Much work remains ahead. A natural next step is to confirm experimentally our findings about some of the RAMP unimplemented designs by implementing and evaluating them to demonstrate their practical advantages. On the other hand, since our methodology can be applied not just to RAMP, but to many other distributed systems, more case studies like the one presented here should be developed to both improve the methodology, and to demonstrate its effectiveness.

References

1. PRISM, `http://www.prismmodelchecker.org/`
2. Uppaal SMC, `http://people.cs.aau.dk/~adavid/smc/`
3. Agha, G.A., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. Electr. Notes Theor. Comput. Sci. 153(2) (2006)
4. AlTurki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: CALCO'11. LNCS, vol. 6859. Springer (2011)
5. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Scalable atomic visibility with RAMP transactions. ACM Trans. Database Syst. 41(3), 15:1–15:45 (2016)
6. Bailis, P., Fekete, A., Hellerstein, J.M., Ghodsi, A., Stoica, I.: Scalable atomic visibility with RAMP transactions. In: Proc. SIGMOD'14. ACM (2014)
7. Barbierato, E., Gribaudo, M., Iacono, M.: Performance evaluation of NoSQL big-data applications using multi-formalism models. Future Generation Comp. Syst. 37, 345–353 (2014)
8. Benson, T., Akella, A., Maltz, D.A.: Network traffic characteristics of data centers in the wild. In: IMC. pp. 267–280 (2010)

9. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC. p. 7 (2000)
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
11. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Statistical model checking for composite actor systems. In: WADT'12. LNCS, vol. 7841. Springer (2013)
12. Gandini, A., Gribaudo, M., Knottenbelt, W.J., Osman, R., Piazzolla, P.: Performance evaluation of NoSQL databases. In: EPEW'14. LNCS, vol. 8721. Springer (2014)
13. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's Megastore in Real-Time Maude. In: Specification, Algebra, and Software. LNCS, vol. 8373. Springer (2014)
14. Grov, J., Ölveczky, P.C.: Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In: SEFM. LNCS, vol. 8702. Springer (2014)
15. Liu, S., Ganhotra, J., Rahman, M., Nguyen, S., Gupta, I., Meseguer, J.: Quantitative analysis of consistency in NoSQL key-value stores. Leibniz Transactions on Embedded Systems 4(1), 03:1–03:26 (2017)
16. Liu, S., Nguyen, S., Ganhotra, J., Rahman, M.R., Gupta, I., Meseguer, J.: Quantitative analysis of consistency in NoSQL key-value stores. In: QEST. pp. 228–243 (2015)
17. Liu, S., Ölveczky, P.C., Rahman, M.R., Ganhotra, J., Gupta, I., Meseguer, J.: Formal modeling and analysis of RAMP transaction systems. In: SAC'16. ACM (2016)
18. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude. In: ICFEM'14. LNCS, vol. 8829. Springer (2014)
19. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
20. Nawab, F., Arora, V., Agrawal, D., El Abbadi, A.: Minimizing commit latency of transactions in geo-replicated data stores. In: SIGMOD. pp. 1279–1294 (2015)
21. Osman, R., Piazzolla, P.: Modelling replication in NoSQL datastores. In: QEST'14. LNCS, vol. 8657. Springer (2014)
22. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: CAV'05. LNCS, vol. 3576. Springer (2005)
23. Sen, K., Viswanathan, M., Agha, G.A.: VESTA: A statistical model-checker and analyzer for probabilistic systems. In: QEST'05. IEEE Computer Society (2005)
24. 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)

A The RAMP-Fast Algorithm as Given in [5]

Figure 6 shows the RAMP-Fast algorithm as it is described in [5].

B Larger Versions of the Figures

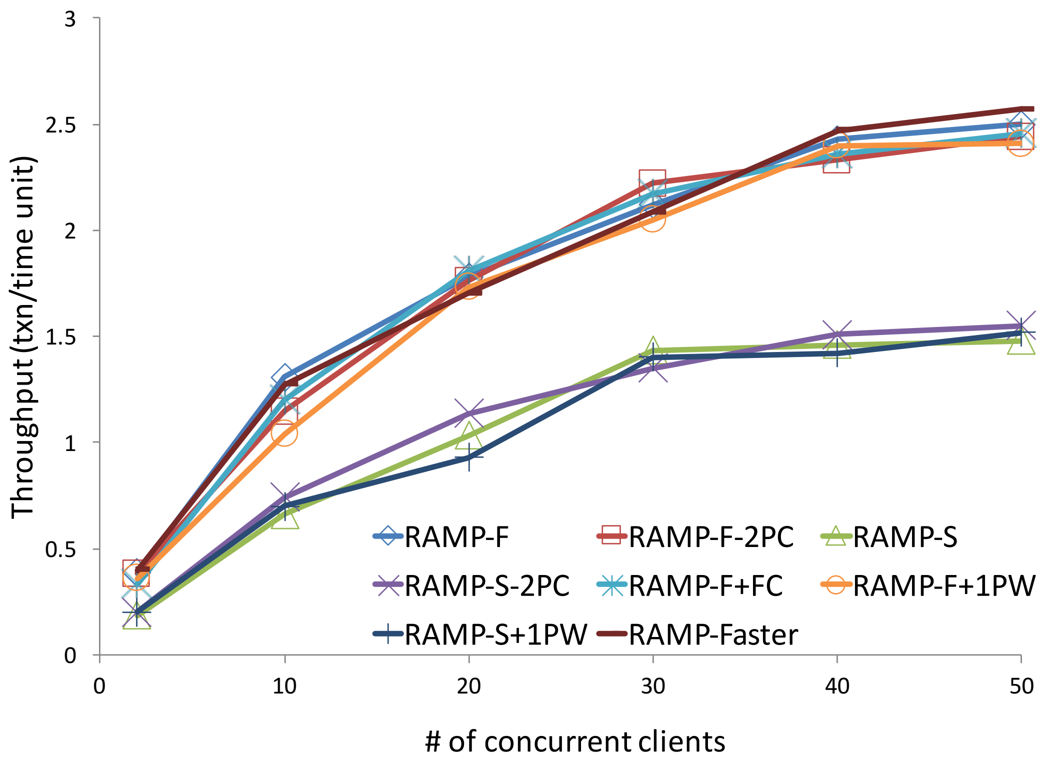
Due to space limitations, our plots have limited size in the paper. We therefore also show here larger versions of these figures.

Figure 7 shows the plots for throughput; the Fig. 8 shows the results for average transaction latency. Figure 10 shows the percentage of read transactions satisfying strong consistency (reading the latest preceding writes), and, finally, Fig. 11 shows the percentage of transactions satisfying read atomicity.

ALGORITHM 1: RAMP-Fast

```
    Server-side Data Structures
 1: $versions$: set of versions $\langle item, value$, timestamp $ts_v$, metadata $md\rangle$
 2: $lastCommit[i]$: last committed timestamp for item $i$
    Server-side Methods
 3: procedure PREPARE($v$ : version)
 4:     $versions$.add($v$)
 5:     return
 6: procedure COMMIT($ts_c$ : timestamp)
 7:     $I_{ts} \leftarrow \{w.item \mid w \in versions \wedge w.ts_v = ts_c\}$
 8:     $\forall i \in I_{ts}, lastCommit[i] \leftarrow \max(lastCommit[i], ts_c)$
 9: procedure GET($i$ : item, $ts_{req}$ : timestamp)
10:     if $ts_{req} = \emptyset$ then
11:         return $v \in versions : v.item = i \wedge v.ts_v = lastCommit[item]$
12:     else
13:         return $v \in versions : v.item = i \wedge v.ts_v = ts_{req}$
    ----------------------------------------------------------------
    Client-side Methods
14: procedure PUT_ALL($W$ : set of $\langle item, value\rangle$)
15:     $ts_{tx} \leftarrow$ generate new timestamp
16:     $I_{tx} \leftarrow$ set of items in $W$
17:     parallel-for $\langle i, v\rangle \in W$
18:         $w \leftarrow \langle item = i, value = v, ts_v = ts_{tx}, md = (I_{tx} - \{i\})\rangle$
19:         invoke PREPARE($w$) on respective server (i.e., partition)
20:     parallel-for server $s$ : $s$ contains an item in $W$
21:         invoke COMMIT($ts_{tx}$) on $s$
22: procedure GET_ALL($I$ : set of items)
23:     $ret \leftarrow \{\}$
24:     parallel-for $i \in I$
25:         $ret[i] \leftarrow$ GET($i, \emptyset$)
26:     $v_{latest} \leftarrow \{\}$ (default value: $-1$)
27:     for response $r \in ret$ do
28:         for $i_{tx} \in r.md$ do
29:             $v_{latest}[i_{tx}] \leftarrow \max(v_{latest}[i_{tx}], r.ts_v)$
30:     parallel-for item $i \in I$
31:         if $v_{latest}[i] > ret[i].ts_v$ then
32:             $ret[i] \leftarrow$ GET($i, v_{latest}[i]$)
33:     return $ret$
```

Fig. 6. The RAMP-Fast algorithm as described in [5].



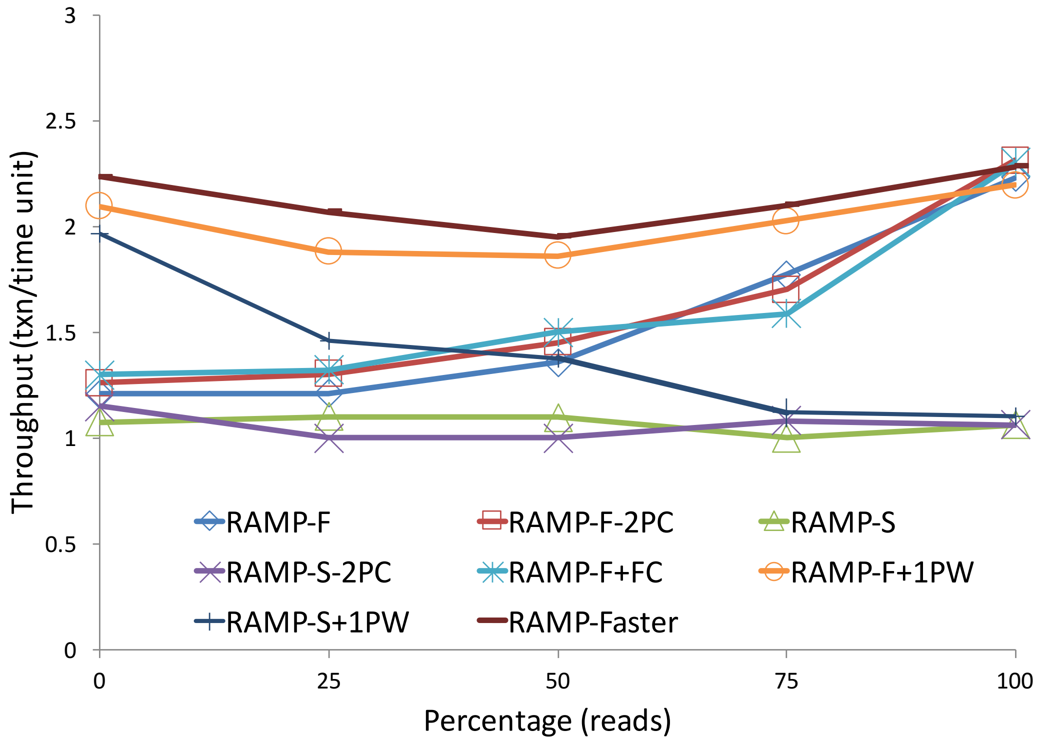
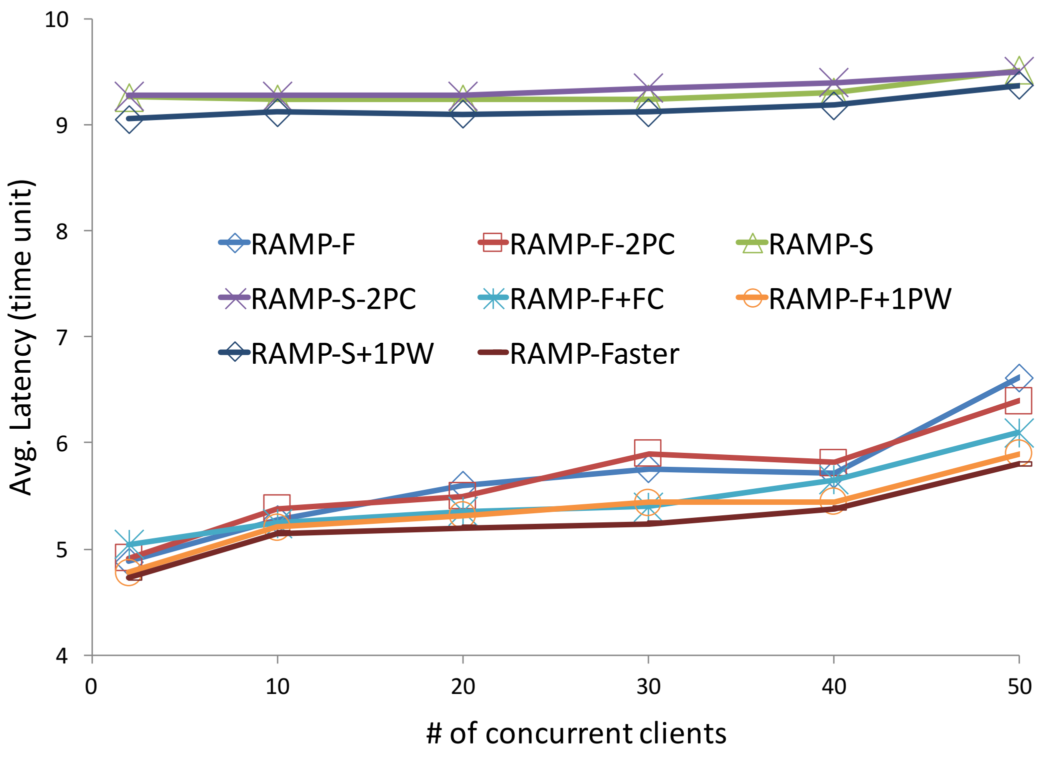


Fig. 7. Throughput under varying client and read load.



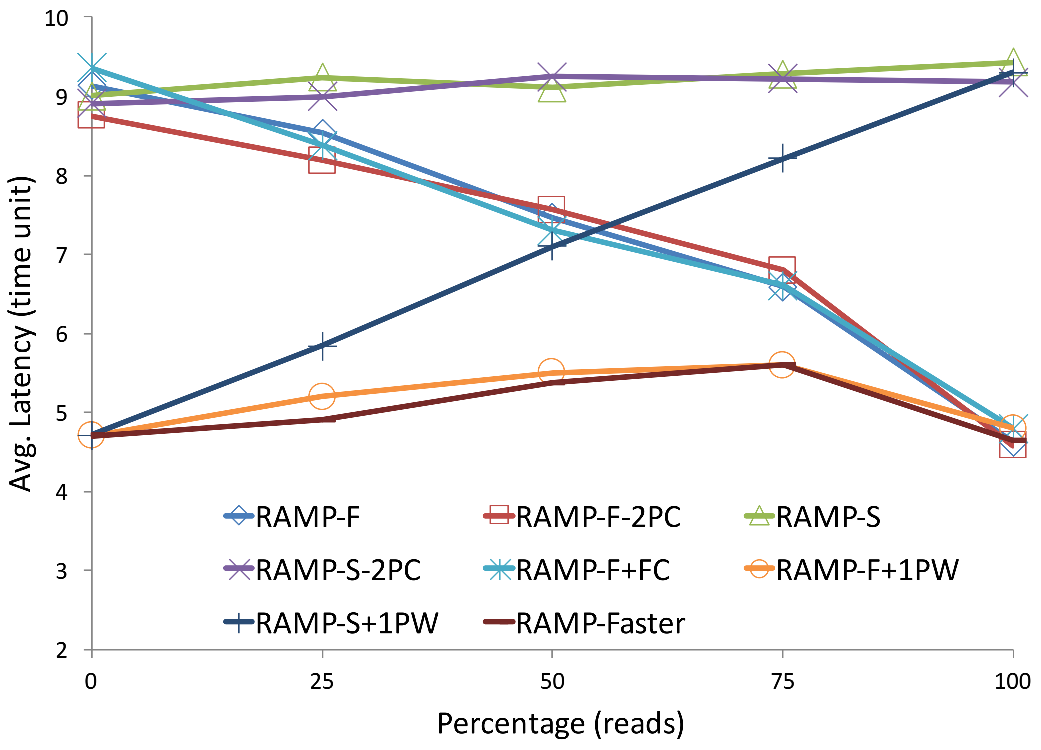
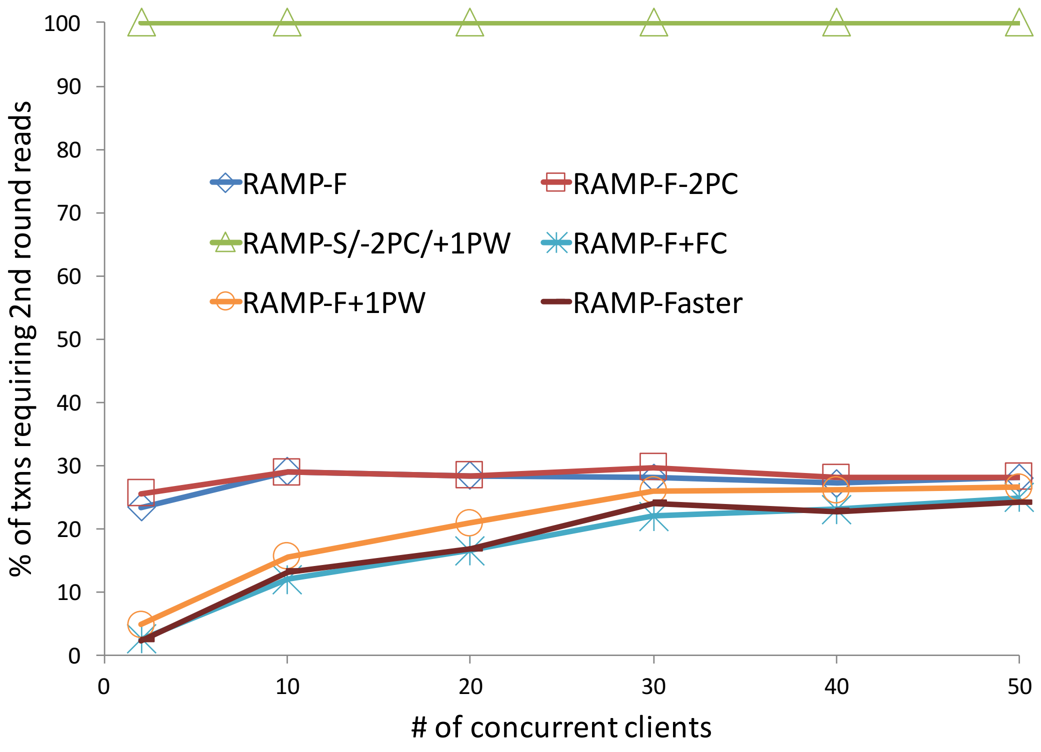


Fig. 8. Average transaction latency under varying client and read load.



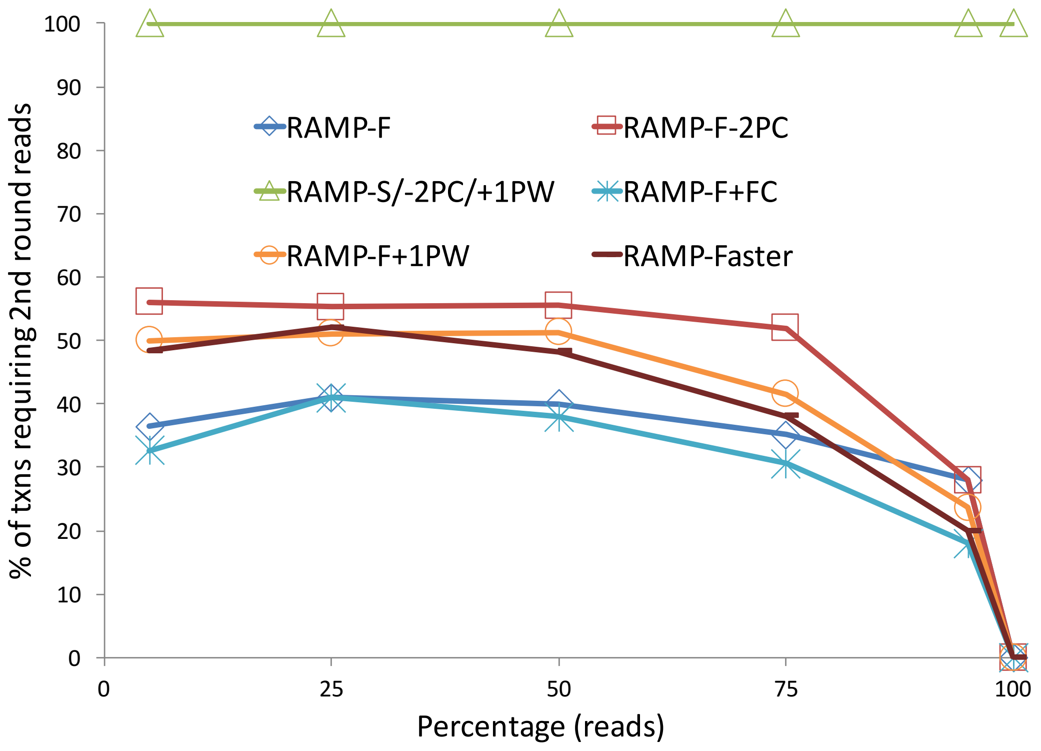
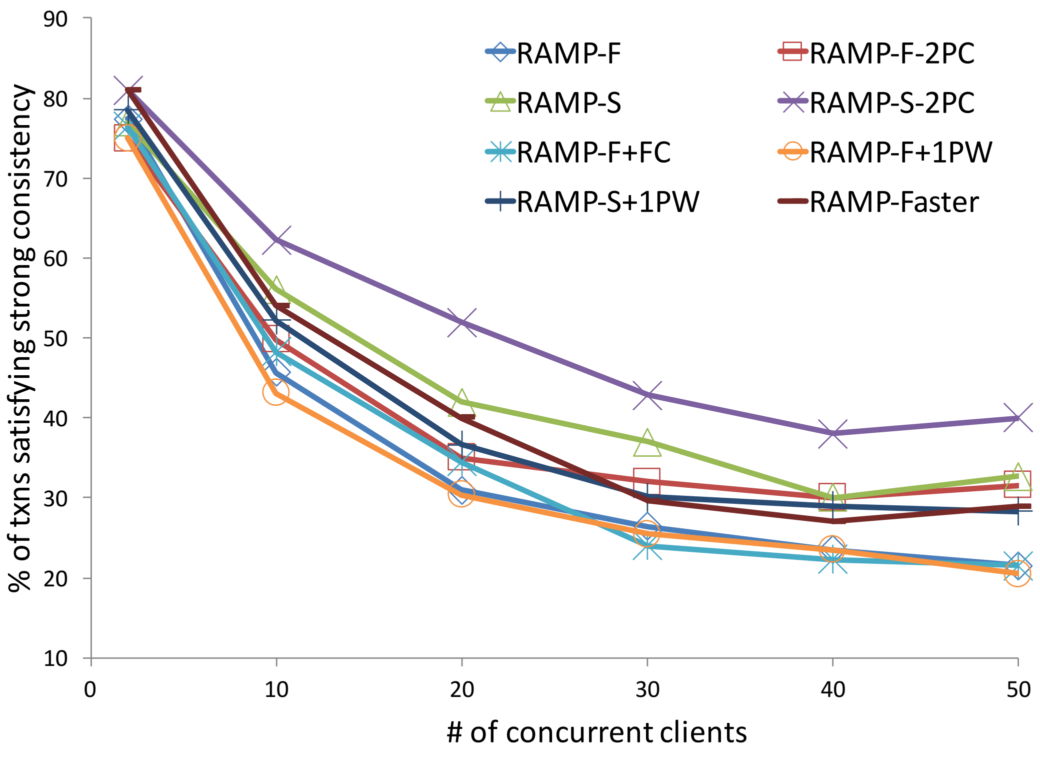


Fig. 9. Percentage of transactions requiring second-round reads under varying client and read load.



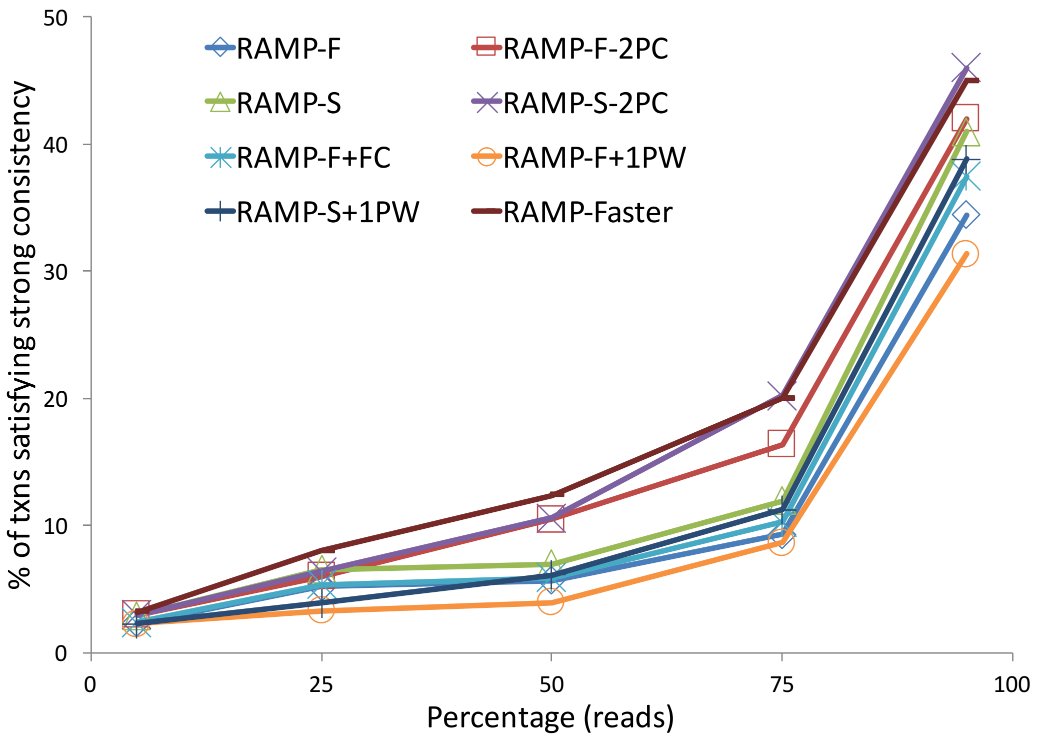
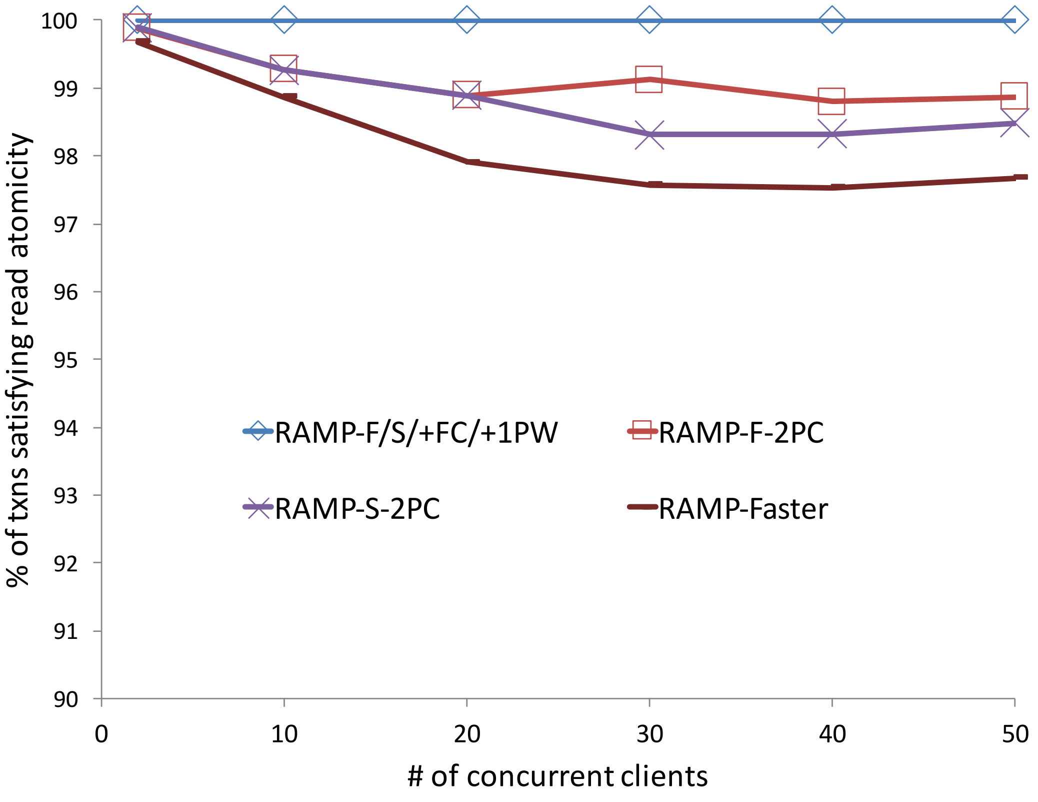


Fig. 10. Probability of satisfying strong consistency under varying client and read load.



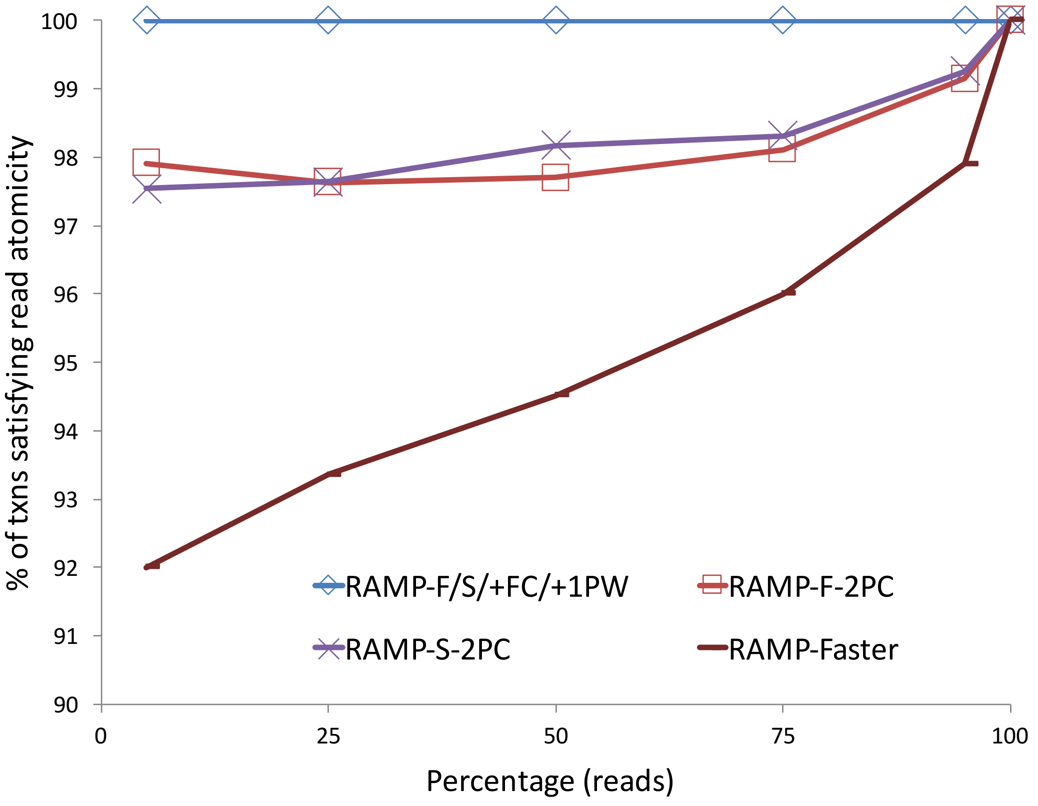


Fig. 11. Probability of satisfying read atomicity under varying client and read load.