Design and Validation of Distributed Data Stores using Formal Methods

Peter Ölveczky

University of Oslo
University of Illinois at Urbana-Champaign

Based on joint work with Jon Grov, Indranil Gupta, Si Liu, José Meseguer, Muntasir Raihan Rahman and other members of UIUC's Center for Assured Cloud Computing

Replicated/Partitioned Data Stores

Cloud computing systems store/retrieve large amounts of data

Replicated/Partitioned Data Stores

Cloud computing systems store/retrieve large amounts of data

Availability and elasticity/scalability:

- data must be replicated and partitioned

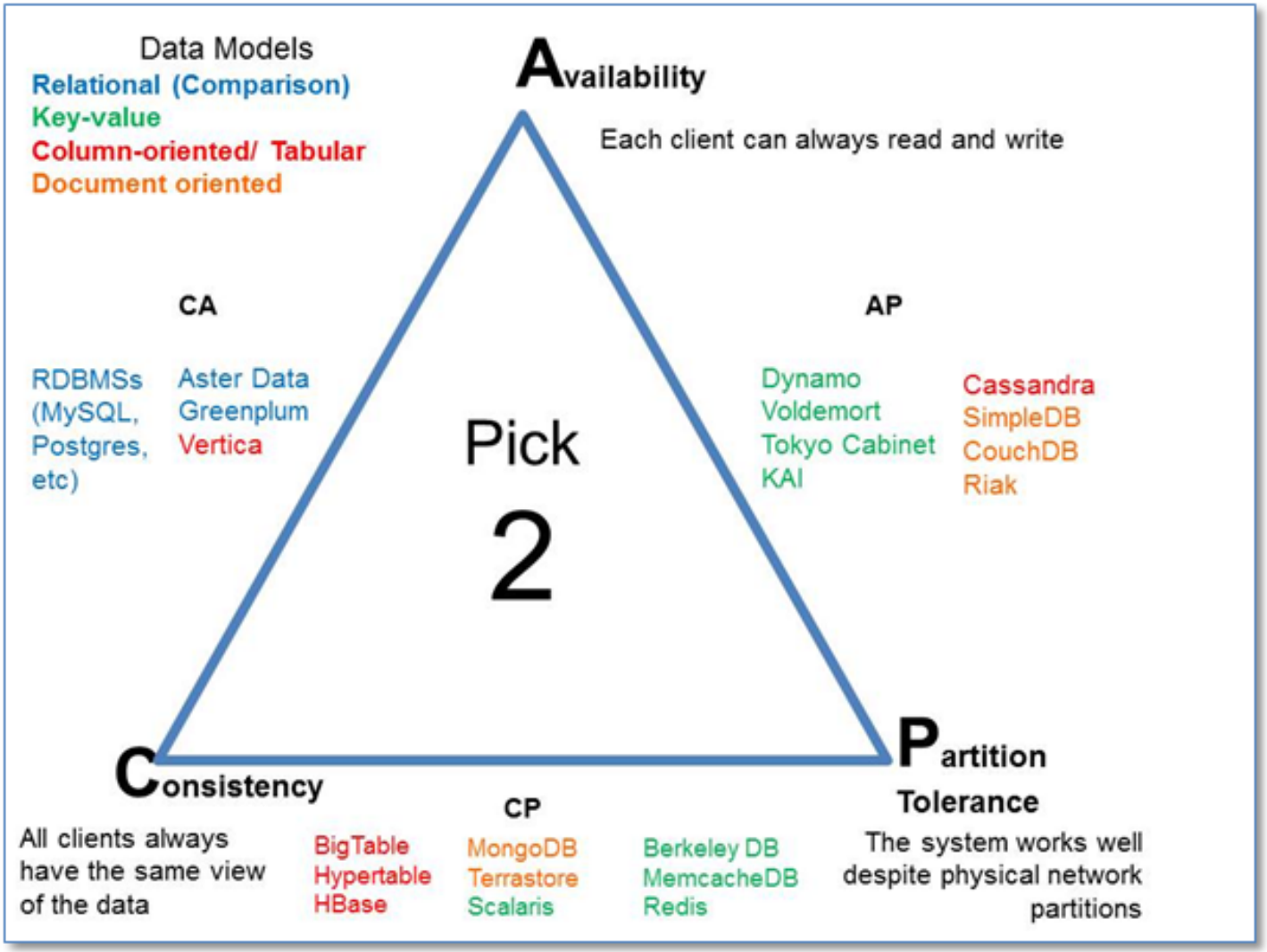
Transactions

Databases traditionally provide ACID transactions

- atomicity
- consistency
- isolation ("serializability")
- durability

"CAP Theorem"

Data consistency + partition tolerance + availability impossible



(Figure from http://flux7.com/blogs/nosql/cap-theorem-why-does-it-matter/)

Eventual Consistency

- "Eventual consistency" OK for some applications
  - Google search; newspapers; facebook

Eventual Consistency

- "Eventual consistency" OK for some applications
  - Google search; newspapers; facebook
- . . . but not others:
  - banking
  - stock exchange
  - electronic commerce
    - online auctions (eBay)
    - plane tickets . . .
  - medical (information) systems

Eventual Consistency

- "Eventual consistency" OK for some applications
  - Google search; newspapers; facebook
- . . . but not others:
  - banking
  - stock exchange
  - electronic commerce
    - online auctions (eBay)
    - plane tickets . . .
  - medical (information) systems
- In general: application developers need support for transactions with some guarantees

Transactions on Replicated/Partitioned Data Stores

- Weaker consistency models
  - serializability
  - snapshot isolation
  - prefix consistency
  - parallel snapshot isolation
  - causal+ consistency
  - causal consistency
  - update atomic
  - read atomic
  - eventual consistency
  - ...

Transactions on Replicated/Partitioned Data Stores

- Weaker consistency models
  - serializability
  - snapshot isolation
  - prefix consistency
  - parallel snapshot isolation
  - causal+ consistency
  - causal consistency
  - update atomic
  - read atomic
  - eventual consistency
  - ...
- Plethora of transaction systems
  - Megastore, Cassandra, RAMP, DrTM, Spanner, TAPIR, FaRM, COPS, ...

Designing Data Stores

- Complex systems
  - size
  - replication
  - concurrence
  - fault tolerance

Designing Data Stores

- Complex systems
  - size
  - replication
  - concurrence
  - fault tolerance
- Many hours of "whiteboard analysis"

Validating Data Store Designs

- Correctness: "hand proofs"
  - error prone
  - informal
  - key assumptions implicit

Validating Data Store Designs

- Correctness: "hand proofs"
  - error prone
  - informal
  - key assumptions implicit
- Performance: simulation tools, real implementations
  - additional artifact
  - cannot be used to reason about correctness

Designing and Validating New Designs

Want to quickly

- develop new data store designs
- explore different design choices
- ... and their effect on correctness and performance

Using Formal Methods (I): Validation Perspective

- Formal system model S
  - precise mathematical model
  - makes assumptions precise and explicit
  - amenable to mathematical correctness analysis

Using Formal Methods (I): Validation Perspective

- Formal system model S
  - precise mathematical model
  - makes assumptions precise and explicit
  - amenable to mathematical correctness analysis
- Formal property specification P
  - precise description of consistency model
  - can (automatically?) check whether $S \models P$

Using Formal Methods (I): Validation Perspective

- Formal system model S
  - precise mathematical model
  - makes assumptions precise and explicit
  - amenable to mathematical correctness analysis
- Formal property specification P
  - precise description of consistency model
  - can (automatically?) check whether $S \models P$
- What about performance analysis?

Using Formal Methods (II): Software Engineering Perspective

Would like:

- expressive and intuitive modeling language
  - quickly develop design model

Using Formal Methods (II): Software Engineering Perspective

Would like:

- expressive and intuitive modeling language
  - quickly develop design model
- expressive and intuitive property specification language

Using Formal Methods (II): Software Engineering Perspective

Would like:

- expressive and intuitive modeling language
  - quickly develop design model
- expressive and intuitive property specification language
- automatically check whether design satisfies property
  - quick and extensive feedback
  - saves hours/days of whiteboard analysis

Using Formal Methods (II): Software Engineering Perspective

Would like:

- expressive and intuitive modeling language
  - quickly develop design model
- expressive and intuitive property specification language
- automatically check whether design satisfies property
  - quick and extensive feedback
  - saves hours/days of whiteboard analysis
- design model also for performance analysis!
  - no new artifact for performance analysis

Our Framework: Rewriting Logic

- Modeling formalism: rewriting logic
  - expressive
  - simple/intuitive
  - concurrent objects

Our Framework: Rewriting Logic

- Modeling formalism: rewriting logic
  - expressive
  - simple/intuitive
  - concurrent objects
- Maude tool:
  - simulation
  - temporal logic model checking
    - expressive property specification language

Our Framework: Rewriting Logic

- Modeling formalism: rewriting logic
  - expressive
  - simple/intuitive
  - concurrent objects
- Maude tool:
  - simulation
  - temporal logic model checking
    - expressive property specification language
- Extensions:
  - real-time systems
  - probabilistic systems

Maude: Software Engineering Perspective I

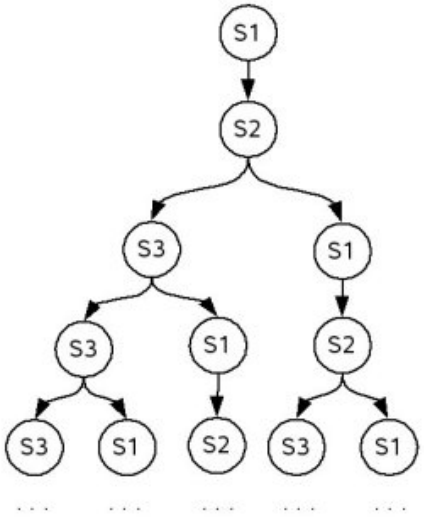
- Models can be developed quickly

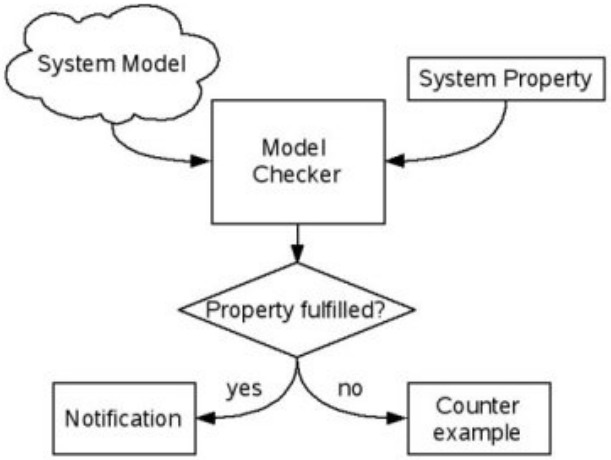
Maude: Software Engineering Perspective I

- Models can be developed quickly
- Simulation gives quick feedback (rapid prototyping)

Maude: Software Engineering Perspective I

- Models can be developed quickly
- Simulation gives quick feedback (rapid prototyping)
- Model checking: analyze all behaviors from one initial state





http://embsys.technikum-wien.at/projects/decs/verification/formalmethods.php

- formal test-driven development: "test-driven development approach where many complex scenarios can be quickly tested by model checking"

Maude: Software Engineering Perspective (cont.)

What about performance analysis?

Maude: Software Engineering Perspective (cont.)

What about performance analysis?

1. (Randomized) simulations
   - Maude performance estimation gives results close to dedicated simulation tools for wireless sensor networks

Maude: Software Engineering Perspective (cont.)

What about performance analysis?

1. (Randomized) simulations
   - Maude performance estimation gives results close to dedicated simulation tools for wireless sensor networks
2. Probabilistic analysis (using PVeStA)
   - statistical model checking
   - Monte-Carlo simulations

Maude: Software Engineering Perspective (cont.)

Same artifact used for:

- precise system description
- rapid prototyping
- extensive testing
- correctness analysis
- performance estimation

Case Study I

Modeling, Analyzing, and Extending Megastore

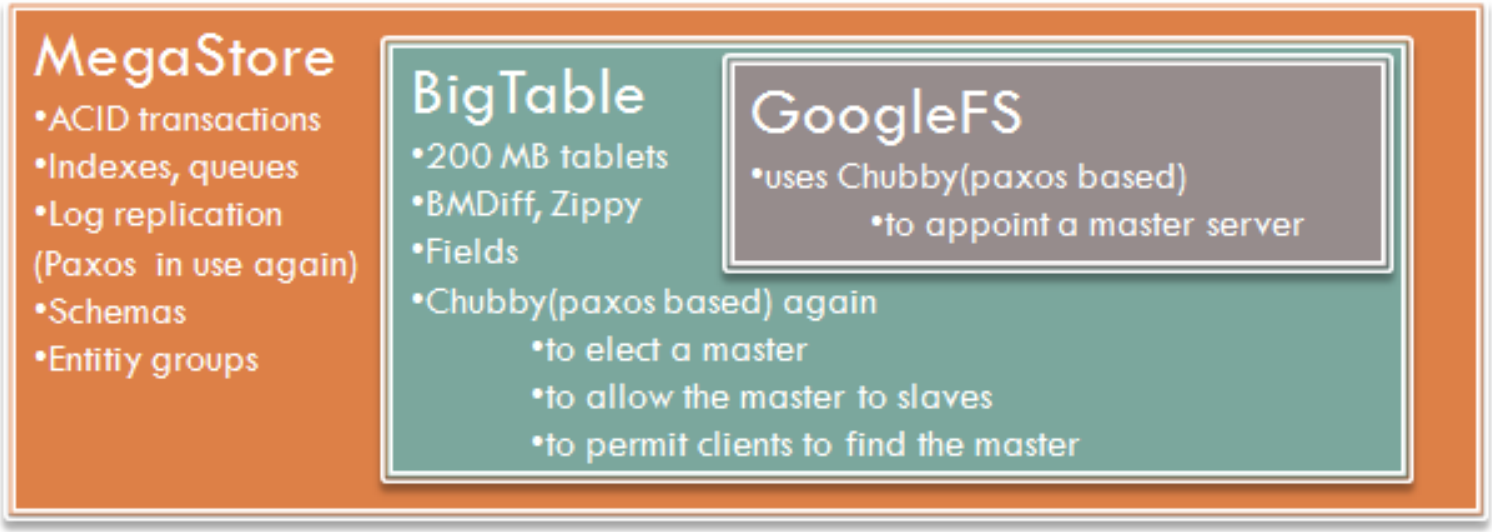
Joint work with Jon Grov (U. Oslo)

Megastore

Megastore:

- Google's wide-area replicated data store
- Key part of Google's cloud infrastructure
- 3 billion write and 20 billion read transactions daily (2011)
- Adds (limited) transactions to wide-area replicated data stores

Megastore: Key Ideas (I)



(Figure from http://cse708.blogspot.jp/2011/03/megastore-providing-scalable-highly.html)

Data divided into entity groups

- Peter's email
- Books on rewriting logic
- Jon's documents

Megastore: Key Ideas (II)

- Consistency for transactions accessing a single entity group
  - no guarantee if transaction reads multiple entity groups

Megastore: Key Ideas (II)

- Consistency for transactions accessing a single entity group
  - no guarantee if transaction reads multiple entity groups
- ElasTraS, Spinnaker, Calvin, and Microsoft's Azure: consistency within each data partition

Our Work

- [Developed and] formalized [our version of the] Megastore [approach] in Maude

Our Work

- [Developed and] formalized [our version of the] Megastore [approach] in Maude
  - first (public) formalization/detailed description of Megastore

Our Work

- [Developed and] formalized [our version of the] Megastore [approach] in Maude
  - first (public) formalization/detailed description of Megastore
- 56 rewrite rules (37 for fault tolerance features)

Performance Estimation

- Key performance measures:
  - average transaction latency
  - number of committed/aborted transactions
- 2 entity groups
- Randomly generated transactions (rate 2.5 TPS)
- Network delays:

	30%	30%	30%	10%
London $\leftrightarrow$ Paris	10	15	20	50
London $\leftrightarrow$ New York	30	35	40	100
Paris $\leftrightarrow$ New York	30	35	40	100

Performance Estimation (cont.)

- Simulating for 200 seconds (no failures):

	Avg. latency (ms)	Commits	Aborts
London	122	149	15
New York	155	132	33
Paris	119	148	18

Performance Estimation (cont.)

- Simulating for 200 seconds (no failures):

	Avg. latency (ms)	Commits	Aborts
London	122	149	15
New York	155	132	33
Paris	119	148	18

- Site failures:
  - mean-time-to-failure 10 seconds per site
  - mean-time-to repair 2 seconds

	Avg. latency (ms)	Commits	Aborts
London	218	109	38
New York	336	129	16
Paris	331	116	21

Model Checking Serializability

Msg. delay	#Trans	Trans. start time	#Fail	Fail. time	Run (s
$\{20, 100\}$	4	$\{19, 80\}$ and $\{50, 200\}$	0	-	1367
$\{20, 100\}$	3	$\{10, 50, 200\}$	1	60	1164
$\{20, 40\}$	3	20, 30, and $\{10, 50\}$	2	$\{40, 80\}$	872
$\{20, 40\}$	4	20, 20, 60, and 110	2	70 and $\{10, 130\}$	241
$\{20, 40\}$	4	20, 20, 60, and 110	2	$\{30, 80\}$	DNF
$\{10, 30, 80\}$,and $\{30, 60, 120\}$	3	20, 30, 40	1	$\{30, 80\}$	DNF
$\{10, 30, 80\}$,and $\{30, 60, 120\}$	3	20, 30, 40	1	60	DNF

Megastore-CGC: extending Megastore

Motivation

- Sometimes transactions must access multiple entity groups

Motivation

- Sometimes transactions must access multiple entity groups
- Our work: extend Megastore with consistency for transactions accessing multiple entity groups

Motivation

- Sometimes transactions must access multiple entity groups
- Our work: extend Megastore with consistency for transactions accessing multiple entity groups
  - must maintain Megastore's
    - performance
    - strong fault tolerance

Megastore-CGC

Megastore-CGC piggybacks ordering and validation onto Megastore's coordination protocol

- no additional messages for validation/commit!
- maintains Megastore's performance and fault tolerance
- failover protocol when ordering site fails

Performance Comparison using Real-Time Maude

- Simulating for 1000 seconds (no failures)
- Megastore:

	Commits	Aborts	Avg. latency (ms)
Site 1	652	152	126
Site 2	704	100	118
RSite	640	172	151

- Megastore-CGC:

	Commits	Aborts	Val. aborts	Avg.latency (ms)
Site 1	660	144	0	123
Site 2	674	115	15	118
RSite	631	171	10	150

Model Checking Megastore-CGC

Model checking scenarios

- 5 transactions (3 fixed, 2 with 2 start times), no failures, message delay 30 ms or 80 ms
  $\longrightarrow$ 108,279 reachable states, 124 seconds
- 3 transactions (all with 2 start times), one site failure and fixed message delay
  $\longrightarrow$ 1,874,946 reachable states, 6,311 seconds
- 3 transactions (all with 2 start times), fixed message delay and one message failure
  $\longrightarrow$ 265,410 reachable states, 858 seconds

Case Study II

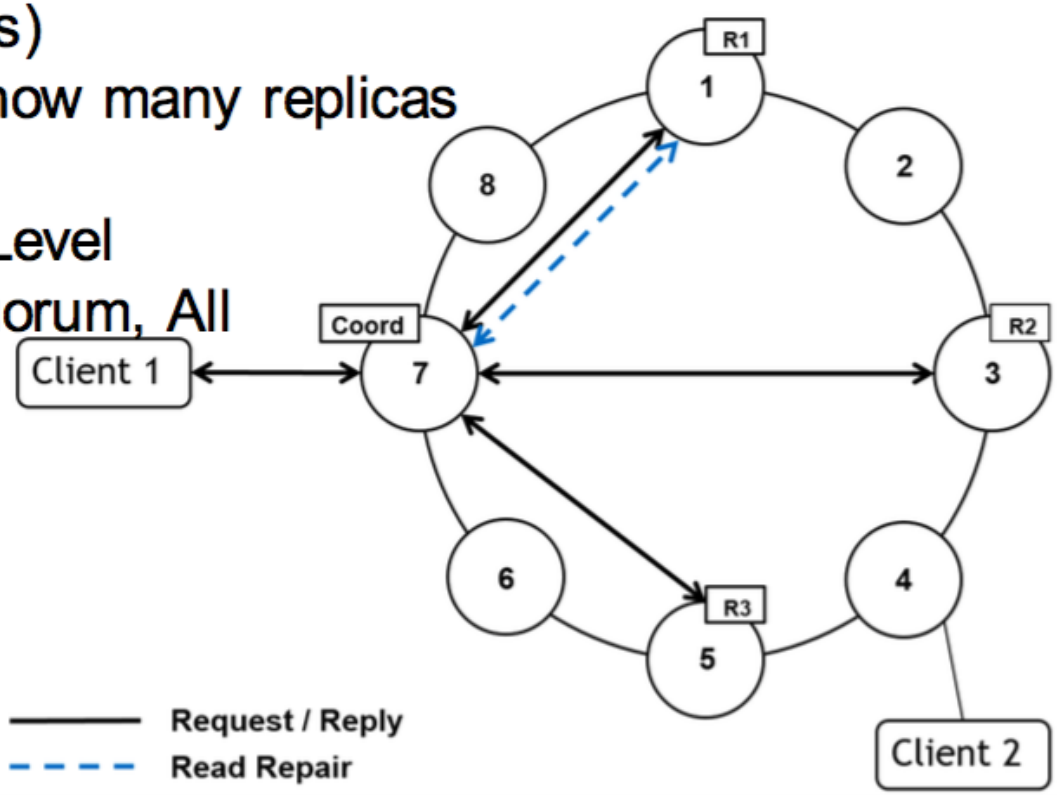
Work by Si Liu, Muntasir Raihan Rahman, Stephen Skeirik, Indranil Gupta, José Meseguer, Son Nguyen, Jatin Ganhotra (ICFEM'14, QEST'15)

Apache Cassandra

- Key-value data store
- Top-10 most popular database engine
- Originally developed at Facebook
- Used by Amadeus, Apple, CERN, IBM, Netflix, Facebook/Instagram, Twitter, ...
- Open source

Cassandra Overview

- Each key-value pair replicated at multiple servers
- Clients can read/write key-value pairs
- Read/write goes from client to Coordinator, which forwards to replica(s)
- Client can specify how many replicas need to answer
    - Consistency Level
    - E.g., One, Quorum, All



Motivation

1. High-level formal model from 345K LOC
   - captures all major design decisions
   - understand and analyze system
   - allows experimenting with different optimizations/variations
2. Analyze basic property: eventual consistency
3. When/how often does Cassandra give stronger guarantees:
   - strong consistency
   - read-your-writes
4. Performance evaluation:
   - do PVeStA analyses give results similar to real implementations?

Formal Analysis with Multiple Clients

Strong

Latency \ Consistency Lv.	ONE	QUORUM	ALL
L1 (L1<D1)	×	×	×
L2 (D1<L2<D2)	×	×	×
L3 (D2<L3)	✓	✓	✓

Relation between issuing latency and message delays

Eventual

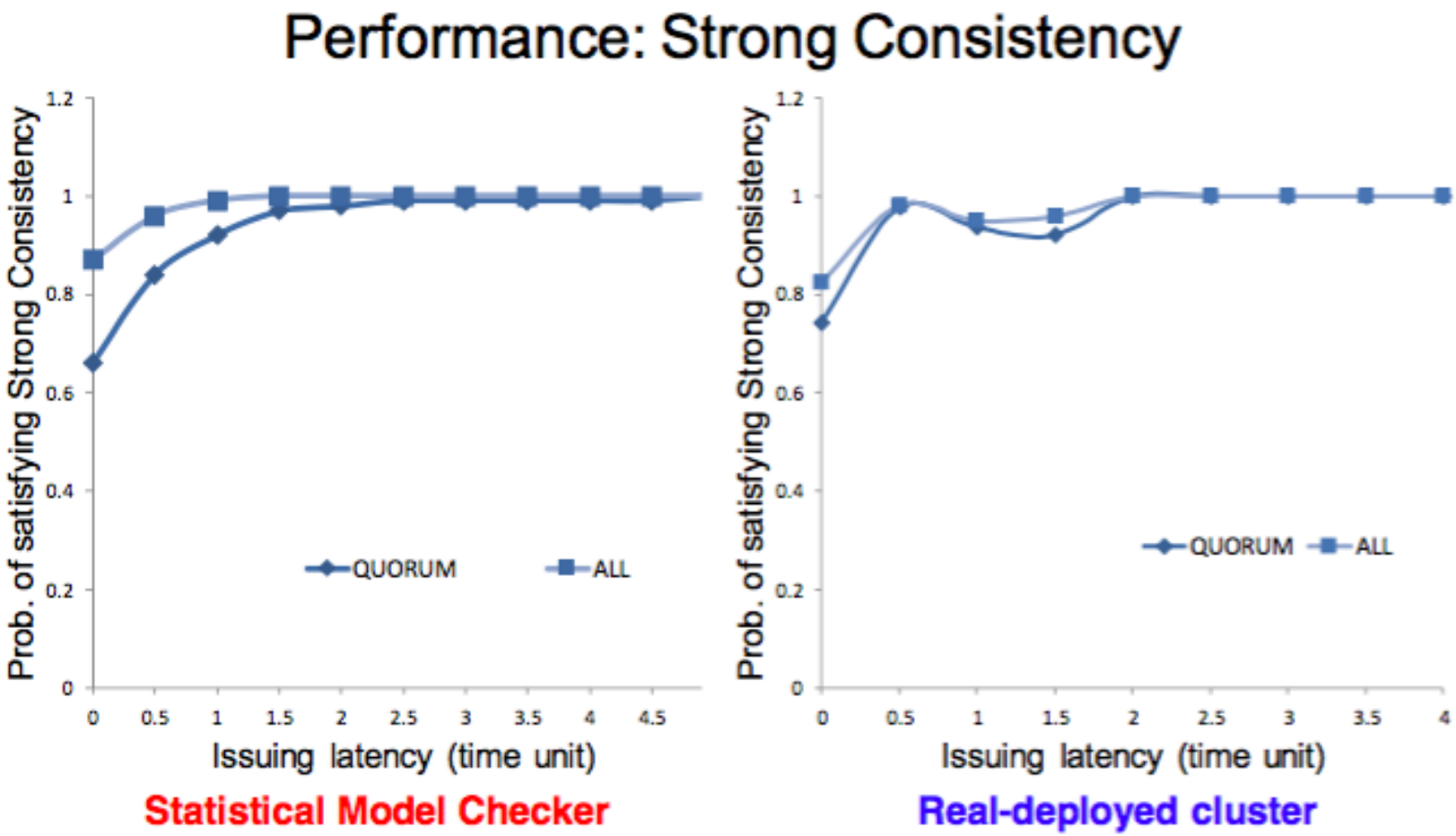
Latency \ Consistency Lv.	ONE	QUORUM	ALL
L1 (L1<D1)	✓	✓	✓
L2 (D1<L2<D2)	✓	✓	✓
L3 (D2<L3)	✓	✓	✓

- Conclusion
  - strong consistency depends on the latency between requests
  - eventual consistency is guaranteed

Performance Estimation

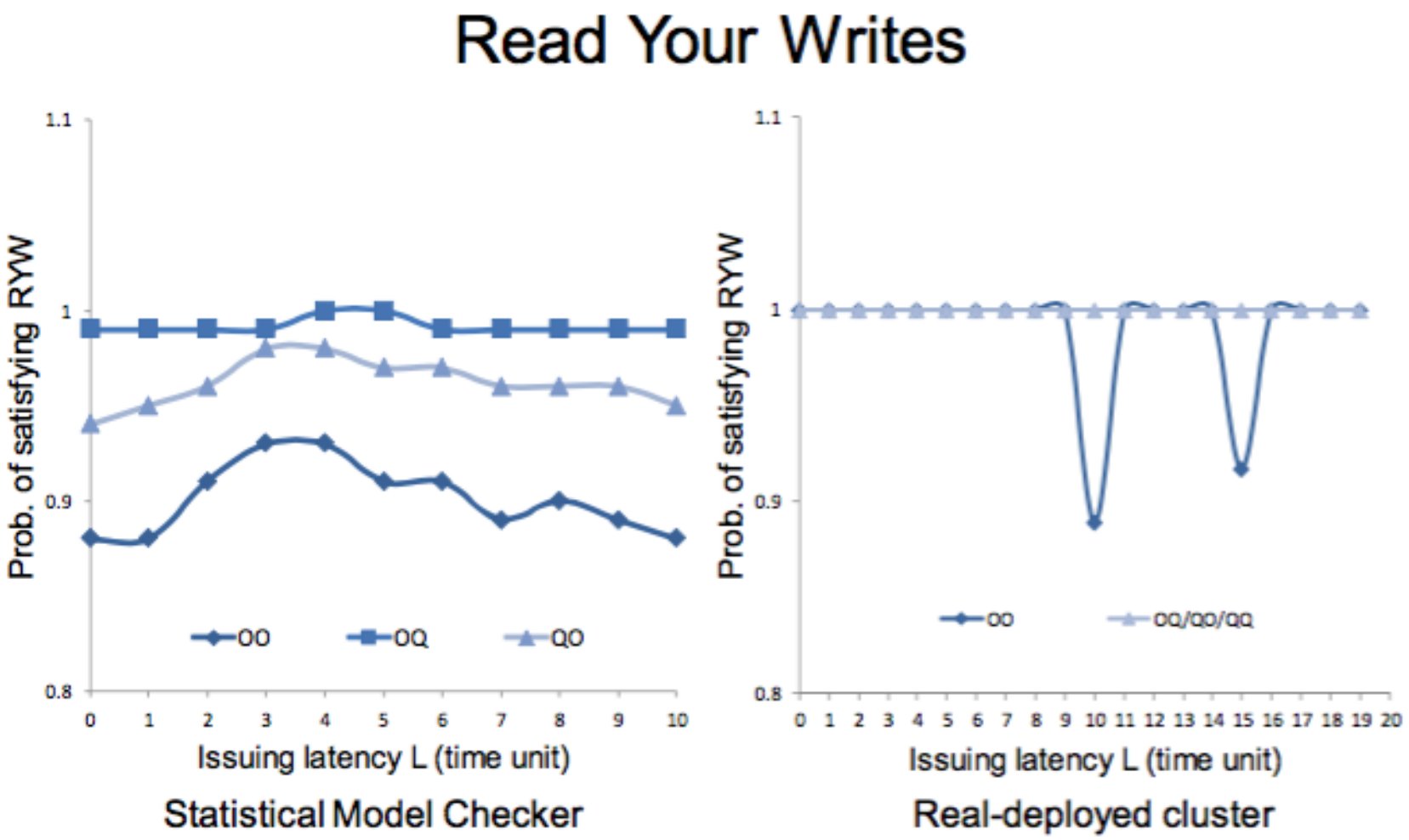
Formal model + PVeStA vs. actual implementation

Performance Estimation



- (X axis =) Issuing Latency = time difference between the given read request and the latest write request
- (Y axis =) Probability of a request satisfying that model

Performance Estimation



- **Conclusion: Statistical Model Checker is reasonably accurate (to within 10-15%) in predicting consistency behaviors**

RAMP Transactions

With Si Liu, Muntasir Raihan Rahman, Jatin Ganhotra, Indranil Gupta, and José Meseguer (ACM SAC'16)

RAMP

- Read-Atomic Multi-Partition transactions
  - developed at UC Berkeley 2014 (P. Bailis et al.)
  - weak consistency guarantee: read atomicity
- Pseudo-code and implementation
- Hand proofs of key properties
- Optimizations/variations hinted at
  - not described in detail
  - properties conjectured

Our Work: Motivation

- Model checking validation of "hand proofs"
- Formalize and analyze optimizations/variations
  - faster commit; one-phase writes; with/without two-phase commit
  - analyze different properties:
    - read atomicity
    - read-your-writes

Maude Analysis

- Coverage: model checking w.r.t. all initial configurations with n operations, m clients, and k data items
- Model checked 4 properties of 7 models
- Hand proofs and conjectures validated
  - RAMP without 2PC does not satisfy read atomicity and read-your-writes
  - RAMP with one-phase writes does not satisfy read-your-writes

Concluding Remarks

Concluding Remarks I

- Developed formal models of large industrial data stores
  - Google's Megastore (from brief description)
  - Apache Cassandra (from 345K LOC and description)

  and recent state-of-the-art academic system
  - RAMP

Concluding Remarks I

- Developed formal models of large industrial data stores
  - Google's Megastore (from brief description)
  - Apache Cassandra (from 345K LOC and description)

  and recent state-of-the-art academic system
  - RAMP
- Model checking analysis of consistency properties
  - from single initial configurations
  - new techniques for increased coverage
  - state space explosion

Concluding Remarks I

- Developed formal models of large industrial data stores
  - Google's Megastore (from brief description)
  - Apache Cassandra (from 345K LOC and description)

  and recent state-of-the-art academic system
  - RAMP
- Model checking analysis of consistency properties
  - from single initial configurations
  - new techniques for increased coverage
  - state space explosion
- Designed own transactional data stores
  - Megastore-CGC
    (+ variations of RAMP)

Concluding Remarks I

- Developed formal models of large industrial data stores
  - Google's Megastore (from brief description)
  - Apache Cassandra (from 345K LOC and description)

  and recent state-of-the-art academic system
  - RAMP
- Model checking analysis of consistency properties
  - from single initial configurations
  - new techniques for increased coverage
  - state space explosion
- Designed own transactional data stores
  - Megastore-CGC
    (+ variations of RAMP)
- Maude/PVeStA performance estimation close to performance of real implementations

"Software Engineering" Conclusions

- Quickly develop formal models/prototypes of complex systems
  - quickly experiment with different design choices

"Software Engineering" Conclusions

- Quickly develop formal models/prototypes of complex systems
  - quickly experiment with different design choices

- Key: intuitive modeling language

"Software Engineering" Conclusions

- Quickly develop formal models/prototypes of complex systems
  - quickly experiment with different design choices
- Key: intuitive modeling language
- Simulation and model checking throughout design phase
  - model-checking-based-testing for subtle "corner cases"
  - replaces days of whiteboard analysis
  - too many scenarios for standard test-based development

"Software Engineering" Conclusions

- Quickly develop formal models/prototypes of complex systems
  - quickly experiment with different design choices
- Key: intuitive modeling language
- Simulation and model checking throughout design phase
  - model-checking-based-testing for subtle "corner cases"
  - replaces days of whiteboard analysis
  - too many scenarios for standard test-based development
- Single artifact for
  - system description
  - rapid prototyping
  - model checking
  - performance estimation

Discussion

First formal-methods-based development and analysis of nontrivial transactional data stores

Discussion

First formal-methods-based development and analysis of nontrivial transactional data stores

But . . .

Discussion

First formal-methods-based development and analysis of nontrivial transactional data stores

But . . . if we are the first, is this really a promising approach?

DOI:10.1145/2699417

Engineers use TLA+ to prevent serious but subtle bugs from reaching production.

BY CHRIS NEWCOMBE, TIM RATH, FAN ZHANG, BOGDAN MUNTEANU, MARC BROOKER, AND MICHAEL DEARDEUFF

How Amazon Web Services Uses Formal Methods

SINCE 2011, ENGINEERS at Amazon Web Services (AWS) have used formal specification and model checking to help solve difficult design problems in critical systems. Here, we describe our motivation

» key insights

- Formal methods find bugs in system designs that cannot be found through any other technique we know of.
- Formal methods are surprisingly feasible for mainstream software development and give good return on investment.
- At Amazon, formal methods are routinely applied to the design of complex real-world software, including public cloud services.

growth, in 2006, AWS launched S3, its Simple Storage Service. In the following six years, S3 grew to store one trillion objects.[3] Less than a year later it had grown to two trillion objects and was regularly handling 1.1 million requests per second.[4]

ticular “rare” scenario. We have found that testing the code is inadequate as a method for finding subtle errors in design, as the number of reachable states of the code is astronomical. So we look for a better approach.

In order to find subtle bugs in a system design, it is necessary to have a precise description of that design. There are at least two major benefits to writing a precise design: the author is forced to think more clearly, helping eliminate "plausible hand waving," and tools can be applied to check for errors in the design, even while it is being written. In contrast, conventional design documents consist of prose, static diagrams, and perhaps pseudo-code in

an ad hoc untestable language. Such descriptions are far from precise; they are often ambiguous or missing critical aspects (such as partial failure or the granularity of concurrency). At the other end of the spectrum, the final executable code is unambiguous but contains an overwhelming amount of detail. We had to be able to capture the essence of a design in a few hundred lines of precise description. As our designs are unavoidably complex, we needed a highly expressive language, far above the level of code, but with precise semantics. That expressivity must cover real-world concurrency and fault tolerance. And, as we wish to build services quickly, we wanted a language that is simple to learn and apply, avoiding esoteric concepts. We

In industry, formal methods have a reputation for requiring a huge amount of training and effort to verify a tiny piece of relatively straightforward code, so the return on investment is

ics). Our experience with TLA+ shows this perception to be wrong. At the time of this writing, Amazon engineers have used TLA+ on 10 large complex real-world systems. In each, TLA+ has added significant value, either finding subtle bugs we are sure we would not have found by other means, or giving us enough understanding and confidence to make aggressive performance optimizations without sacrificing correctness. Amazon now has seven teams using TLA+, with encouragement from senior management and technical leadership. Engineers from entry level

We also find that writing a formal specification pays dividends over the lifetime of the system. All production

find a major benefit of having a precise, testable model of the core system is that we can quickly verify that even deep changes are safe or learn they are unsafe without doing harm. In several cases, we have prevented subtle but serious bugs from reaching production.

In addition, a precise, testable, well-commented description of a design is an excellent form of documentation, which is important, as AWS systems have unbounded lifetimes. Over time, teams grow as the business grows, so we regularly have to bring new people up to speed on systems. This education must be effective. To avoid creating subtle bugs, we need all engineers to have the same mental model of the system and for that shared model to be accurate, precise, and complete. Engi-

When looking for techniques to prevent bugs, C.N. did not initially consider formal methods, due to the pervasive view that they are suitable for only tiny problems and give very low return on investment. Overcoming the bias against

rect. T.R. then checked the broader fault-tolerant algorithm. This time the model checker found a bug that could lead to losing data if a particular sequence of failures and recovery steps would be interleaved with other processing. This was a very subtle bug; the

ger this bug. The bug had passed unnoticed through extensive design reviews, code reviews, and testing, and T.R. is convinced we would not have found it by doing more work in those conventional areas. The model checker later found two bugs in other algorithms, both serious and subtle. T.R. fixed all these bugs, and the model checker verified the resulting algorithms to a very high degree of confidence.

T.R. says that, had he known about TLA+ before starting work on DynamoDB he would have used it from the start. He believes the investment

Conclusion

Formal methods are a big success at AWS, helping us prevent subtle but serious bugs from reaching production, bugs we would not have found through any other technique. They have helped us devise aggressive optimizations to complex algorithms without sacrificing quality. At the time of this writing, seven Amazon teams have used TLA+, all finding value in doing so, and more Amazon teams are starting to use it. Using TLA+ will improve both time-to-market and quality of our systems. Executive management actively encourages teams to write TLA+ specs for new features and other significant design changes. In annual planning, managers now allocate engineering time to TLA+.