Design and Validation of Cloud Storage Systems Using Formal Methods

Peter Csaba Ölveczky(✉)

University of Oslo, Oslo, Norway
peterol@ifi.uio.no

Abstract. To deal with large amounts of data while offering high availability and throughput and low latency, cloud computing systems rely on distributed, partitioned, and replicated data stores. Such cloud storage systems are complex software artifacts that are very hard to design and analyze. Formal specification and model checking should therefore be beneficial during their design and validation. In particular, I propose rewriting logic and its accompanying Maude tools as a suitable framework for formally specifying and analyzing both the correctness and the performance of cloud storage systems. This abstract of an invited talk gives a short overview of the use of rewriting logic at the University of Illinois Assured Cloud Computing center on industrial data stores such as Google's Megastore and Facebook/Apache's Cassandra. I also briefly summarize the experiences of the use of a different formal method for similar purposes by engineers at Amazon Web Services.

1 Introduction

Cloud computing relies on dealing with large amounts of data safely and efficiently. To ensure that data are always available—even when parts of the network are down—data should be *replicated* across widely distributed data centers. Data may also have to be *partitioned* to obtain the elasticity expected from cloud computing. However, given the cost of the communication needed to coordinate the different replicas in a replicated and possibly partitioned distributed data store, there is a trade-off between efficiency on the one hand, and maintaining consistency across the different replicas and the transactional guarantees provided on the other hand. Many data stores therefore provide weaker forms of consistency and weaker transactional guarantees than the traditional ACID guarantees.

Designing cloud data stores that satisfy certain performance and correctness requirements is a highly nontrivial task, and so is the validation that the design actually meets its requirements. In addition, although cloud storage systems are not traditionally considered to be "safety-critical" systems, as more and more applications migrate to the cloud, it becomes increasingly crucial that storage systems do not lose potentially critical user data. However, as argued in, e.g., [4,15], standard system development and validation techniques are not

M.R. Mousavi and J. Sgall (Eds.): TTCS 2017, LNCS 10608, pp. 3–8, 2017.
DOI: 10.1007/978-3-319-68953-1_1

well suited for designing data stores with high assurance that they satisfy their correctness and quality-of-service requirements: Executing and simulating new designs may require understanding and modifying large code bases; furthermore, although system executions and simulations can give an idea of the performance of a design, they cannot give any (quantified) assurance about the system performance, and they cannot be used to verify correctness properties.

In [4], colleagues at the University of Illinois at Urbana-Champaign (UIUC) and I argue for the use of executable formal methods during the design of cloud storage system, and to provide high levels of assurance that the designs satisfy desired correctness and performance requirements. The key thing is that an executable formal model can be directly simulated; it can be also be subjected to various model checking analyses that automatically explore all possible system behaviors from a given initial system configuration. From a system developer's perspective, such model checking can be seen as a powerful debugging and testing method that automatically executes a comprehensive "test suite" for complex fault-tolerant systems. Having an abstract executable formal system model also allow us to quickly and easily explore many design options and to validate designs as early as possible.

However, finding an executable formal method that can handle large and complex distributed systems and that supports reasoning about both the system's *correctness* and its *performance* is not an easy task. *Rewriting logic* [14] and its associated Maude tool [5] and their extensions should be a promising candidate. Rewriting logic is a simple, intuitive, and expressive executable specification formalism for distributed systems. In rewriting logic, data types are defined using algebraic equational specifications and the dynamic behavior of a system is defined by conditional rewrite rules $t \longrightarrow u$ **if** *cond*, where the terms t and u denote state fragments. Such a rewriting logic specification can be directly simulated, from a given initial system state, in Maude. However, such a simulation only covers one possible system behavior. Reachability analysis and LTL model checking can then be used to analyze all possible behaviors from a given initial system state to check, respectively, whether a certain state pattern is reachable from the initial state and whether all possible behaviors from the initial state satisfy a linear temporal logic (LTL) property.

Cloud storage systems are often real-time systems; in particular, to analyze their performance we need timed models. The specification and analysis of real-time systems in rewriting logic are supported by the Real-Time Maude tool [17,19]. In particular, randomized Real-Time Maude simulations have been shown to predict system performance as well as domain-specific simulation tools [18]. Nevertheless, such ad hoc randomized simulations cannot give a quantified measure of confidence in the accuracy of the performance estimations. To achieve such guarantees about the performance of a design, we can specify our design as a *probabilistic rewrite theory* and subject it to *statistical model checking* using the PVeStA tool [1]. Such statistical model checking performs randomized simulations to estimate the expected average value of a given expression, until the desired level of statistical confidence in the outcome has been reached.

In this way we can obtain statistical guarantees about the expected performance of a design.

2 Applications

This section gives a brief overview of how Jon Grov, myself, and colleagues at the Assured Cloud Computing center at the UIUC have applied rewriting logic and its associated tools to model and analyze cloud storage systems. A more extensive overview of parts of this research can be found in the report [4].

Google's Megastore. Megastore [3] is a key component in Google's celebrated cloud infrastructure and is used for Gmail, Google+, Android Market, and Google AppEngine. Megastore is a fault-tolerant replicated data store where the data are divided into different *entity groups* (for example, "Peter's emails" could be one such entity group). Megastore's trade-off between consistency and performance is to provide consistency only for transactions accessing a single entity group. Jon Grov and I had some ideas on how to extend Megastore to also provide consistency for transactions accessing multiple entity groups, without sacrificing performance.

Before experimenting with extensions of Megastore, we needed to understand the Megastore design in significant detail. This was a challenging task, since Megastore is a complex system whose only publicly available description was the short overview paper [3]. We used Maude simulation and model checking extensively throughout the development of a Maude model (with 56 rewrite rules) of the Megastore design [6]. In particular, model checking from selected initial states could be seen as our "test suite" that explored all possible behaviors from those states. Our model also provided the first detailed publicly available description of the Megastore design.

We could then experiment with our design ideas for extending Megastore, until we arrived at a design with 72 rewrite rules, called Megastore-CGC, that also provided consistency for certain sets of transactions that access multiple entity groups [7]. To analyze our conjecture that the extension should have a performance similar to that of Megastore, we ran randomized Real-Time Maude simulations on both models.

An important point is that even *if* we would have had access to Megastore's code base, understanding and extending it would have been much more time-consuming than developing our own models/executable prototypes.

Apache Cassandra. Apache Cassandra [8] is an open-source key-value data store originally developed at Facebook that is currently used by, e.g., Amadeus, Apple, IBM, Netflix, Facebook/Instagram, GitHub, and Twitter. Colleagues at UIUC wanted to experiment with whether some alternative design choices would lead to better performance. In contrast to our Megastore efforts, the problem in this case was that to understand and experiment with different design choices would require understanding and modifying Cassandra's 345,000 lines of code. After studying this code base, Si Liu and others developed a 1,000-line Maude model that captured all the main design choices of Cassandra [13]. The authors used their models and Maude model checking to analyze under what circumstances Cassandra provides stronger consistency properties than "eventual consistency."

They then transformed their models into fully probabilistic rewrite theories and used statistical model checking with PVeStA to evaluate the performance of the original Cassandra design and their alternative design (where the main performance measure is how often strong consistency is satisfied in practice) [12]. To investigate whether the performance estimates thus obtained are realistic, in [12] the authors compare their model-based performance estimates with the performance obtained by actually executing the Cassandra code itself.

RAMP. RAMP [2] is a partitioned data store, developed by Peter Bailis and others at UC Berkeley, that provide efficient multi-partition transactions with a weak transactional guarantee: read atomicity (either all or none of a transaction's updates are visible to other transactions). The RAMP developers describe three main RAMP algorithms in [2]; they also sketch a number of other design alternatives without providing details or proofs about their properties. In [11], colleagues at UIUC and I develop Maude models of RAMP and its sketched designs, and use Maude model checking to verify that also the sketched designs satisfy the properties conjectured by Bailis et al.

But how efficient are the alternative designs? Bailis *et al.* only provide simulation results for their main designs, probably because of the effort required to develop simulation models of a design. Having higher-level smaller formal models allowed us to explore the design state of RAMP quite extensively. In particular, in [10] we used statistical model checking to evaluate the performance along a number of parameters, with many different distributions of transactions. In this way, we could evaluate the performance of a number RAMP designs not explored by Bailis et al., and for many more parameters and workloads than evaluated by the RAMP developers. This allow us to discover the most suitable version of RAMP for different kinds of applications with different kinds of expected workloads. We also experimented with some design ideas of our own, and discovered that one design, RAMP-Faster, has many attractive performance properties, and that, while not guaranteeing read atomicity, provides read atomicity for more than 99% of the transactions for certain workloads.

P-Store. In [16] I analyzed the partially replicated transactional data store P-Store [20] that provides some fault tolerance, serializability of transactions, and limited use of atomic multicast. Although this protocol supposedly was verified by its developers, Maude reachability analysis found a nontrivial bug in the P-Store algorithm that was confirmed by one of the P-Store developers.

3 Formal Methods at Amazon

Amazon Web Services (AWS) is the world's largest provider of cloud computing services. Key components of its cloud computing infrastructure include the DynamoDB replicated database and the Simple Storage System (S3).

In their excellent paper "How Amazon Web Services Uses Formal Methods" [15], engineers at AWS explain how they used the formal specification language TLA+ [9] and its associated model checker TLC during the development of S3, DynamoDB, and other components. Their experiences of using formal methods in an industrial setting can be briefly summarized as follows:

- Model checking finds subtle "corner case" bugs that are not found by the standard validation techniques used in industry.
- A formal specification is a valuable short, precise, and testable description of an algorithm.
- Formal methods are surprisingly feasible for mainstream software development and give good returns on investment.
- Executable formal specifications makes it quick and easy to experiment with different design choices.

The paper [15] concludes that "formal methods are a big success at AWS" and that management actively encourages engineers to use formal methods during the development of new features and design changes.

The weakness reported by the AWS engineers was that while TLA+ was effective at finding bugs, it was not (or could not be) used to analyze performance. It seems that TLC does not support well the analysis of real-time system, and neither does TLA+ come with a probabilistic or statistical model checker. This seems to be one major difference between the formal methods used at AWS and the Maude-based formal method that we propose: we have showed that the Maude tools are useful for analyzing both the correctness and the expected performance of the design.

Acknowledgments. I am grateful to Jon Grov, José Meseguer, Indranil Gupta, Si Liu, Muntasir Rahman, and Jatin Ganhotra for the collaboration on the work summarized in this abstract. I would also like to thank the organizers of TTCS 2017 for giving me the opportunity to present these results as a keynote speaker.

References

1. AlTurki, M., Meseguer, J.: PVeStA: a parallel statistical model checking and quantitative analysis tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22944-2_28
2. Bailis, P., Fekete, A., Hellerstein, J.M., Ghodsi, A., Stoica, I.: Scalable atomic visibility with RAMP transactions. In: Proceedings SIGMOD 2014. ACM (2014)
3. Baker, J., et al.: Megastore: Providing scalable, highly available storage for interactive services. In: CIDR 2011 (2011). www.cidrdb.org

4. Bobba, R., Grov, J., Gupta, I., Liu, S., Meseguer, J., Ölveczky, P.C., Skeirik, S.: Design, formal modeling, and validation of cloud storage systems using Maude. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign (2017). http://hdl.handle.net/2142/96274
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Grov, J., Ölveczky, P.C.: FormaL modeling and analysis of Google's Megastore in Real-Time Maude. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 494–519. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54624-2_25
7. Grov, J., Ölveczky, P.C.: Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 159–174. Springer, Cham (2014). doi:10.1007/978-3-319-10431-7_12
8. Hewitt, E.: Cassandra: The Definitive Guide. O'Reilly Media, Sebastopol (2010)
9. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
10. Liu, S., Ölveczky, P.C., Ganhotra, J., Gupta, I., Meseguer, J.: Exploring design alternatives for RAMP transactions through statistical model checking. In: Proceedings of ICFEM 2017. LNCS, vol. 10610. Springer (2017, to appear)
11. Liu, S., Ölveczky, P.C., Rahman, M.R., Ganhotra, J., Gupta, I., Meseguer, J.: Formal modeling and analysis of RAMP transaction systems. In: Proceedings of SAC 2016. ACM (2016)
12. Liu, S., Ganhotra, J., Rahman, M., Nguyen, S., Gupta, I., Meseguer, J.: Quantitative analysis of consistency in NoSQL key-value stores. Leibniz Trans. Embed. Syst. **4**(1), 03:1–03:26 (2017)
13. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 332–347. Springer, Cham (2014). doi:10.1007/978-3-319-11737-9_22
14. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoret. Comput. Sci. **96**, 73–155 (1992)
15. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)
16. Ölveczky, P.C.: Formalizing and validating the P-Store replicated data store in Maude. In: Proceedings of WADT 2016. LNCS. Springer (2017, to appear)
17. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order Symbolic Comput. **20**(1–2), 161–196 (2007)
18. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theoret. Comput. Sci. **410**(2–3), 254–280 (2009)
19. Ölveczky, P.C.: Real-Time Maude and its applications. In: Escobar, S. (ed.) WRLA 2014. LNCS, vol. 8663, pp. 42–79. Springer, Cham (2014). doi:10.1007/978-3-319-12904-4_3
20. Schiper, N., Sutra, P., Pedone, F.: P-Store: genuine partial replication in wide area networks. In: Proceedings of SRDS 2010. IEEE Computer Society (2010)