Exploiting Models for Scalable and High Throughput Distributed Software

Tim Soethout
ING Bank and Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands
Tim.Soethout@ing.com

Abstract

In high-throughput distributed applications, such as large-scale banking systems, synchronization between objects becomes a bottleneck. This short paper focusses on research, in close collaboration with ING Bank, on the opportunity of leveraging application specific knowledge captured by model driven engineering approaches, to increase application performance in high-contention scenarios, while maintaining functional application-level consistency.

CCS Concepts • **Information systems** → *Distributed database transactions*; • **Software and its engineering** → *Domain specific languages*; State systems; Model-driven software engineering; • **Applied computing** → Enterprise architectures; Event-driven architectures.

Keywords Synchronization, Atomic commit protocols

ACM Reference Format:
Tim Soethout. 2019. Exploiting Models for Scalable and High Throughput Distributed Software. In *Proceedings of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '19), October 20–25, 2019, Athens, Greece.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3359061.3361073

1 Motivation

Enterprise software systems are large, complex, and hard to maintain. Many applications communicate, operate independently, and need to change frequently. Domain Specific Languages (DSLs) are an approach to control the complexity by capturing domain knowledge in a non-ambiguous, single-source, and traceable way. DSLs enables automatically generating optimized code, where domain knowledge can be used which is not available to a general purpose programming language compiler.

SPLASH Companion '19, October 20–25, 2019, Athens, Greece

ACM ISBN 978-1-4503-6992-3/19/10…$15.00
https://doi.org/10.1145/3359061.3361073

The DSL Rebel [16] describes state machines for enterprise products, which communicate using atomic synchronized actions. These specifications are generated into a horizontally scalable distributed application, built on the Akka actor toolkit. Generating code for Rebel's distributed synchronization in a generic scalable fashion is hard, because high-contention specifications result in bottlenecks in throughput and latency. Atomic synchronized actions, formalized as Atomic Commit [8], guarantee that actions on multiple objects are a single atomic step, where all or none should happen. A well-known generic blocking atomic commitment protocol is Two-Phase Commit (2PC) [8].

Improvements in scalability and throughput of Atomic Commit implementations and other optimizations related to consistency, are widely applicable to databases [1, 2, 7, 18], programming languages [3, 10–12, 17] and distributed systems in general [6, 9, 19].

2 Problem

High load on synchronization participants of distributed 2PC transactions results in high transaction latency and limits throughput, because each transaction has to wait until the previous one is finished. This high-contention is problematic in scenario's where latency and throughput requirements have to meet. For example when a single bank account has a large number of money transfers to process this becomes an issue, e.g., a tax office account paying out benefits to a lot of citizens' accounts in a small time frame.

We need approaches to improve in the high-contention bottleneck scenario. In which ways can we exploit models to safely improve synchronization performance?

3 Approach

Models of the distributed transaction's participants enable specialized implementations of safe synchronization that maintains application-level consistency, in contrast to generic atomic commitment protocols and general purpose databases that need to be overly conservative to maintain consistency. These specialized protocols can lead to higher throughput and better scalability.

An example of exploitable model information is the `Deposit` operation as found in an `Account` state machine example, shown in Figure 1. The state machine instance can stay application-level consistent, even when multiple

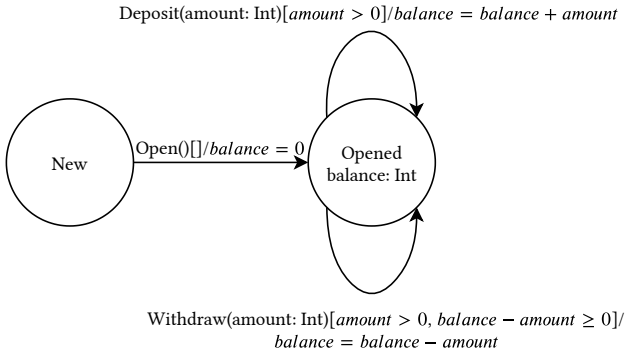

Figure 1. State machine of example simple account. Events are defined in state chart notation: Event(fields)[guard]/effect

`Deposit` operations are executed in parallel, because it's preconditions cannot be invalided.

Our research is centered around utilizing this model knowledge, which general purpose algorithms and database can not rely on.

We plan to look into these open questions:

- ING Bank's transaction data indicates that high contention will become a problem in practice. Do synthetic benchmarks in literature accurately exercise this behavior?
- Can we exploit run time model information to improve the performance of synchronization?
- Can we exploit compile time model information to improve the performance of synchronization?
- Using previous research results, can potential performance bottlenecks be detected at design time to support the designer?

The next section poses directions to answer these questions.

4 Evaluation Methodology

High-concurrent participants become a real problem for banking use cases when system usage keeps growing. This limits throughput and scalability of implementations. The main hypothesis is that performance can be improved by using knowledge of the application to safely parallelize transactions, where generic solutions can not. Secondary hypotheses are:

- Synthetic benchmarks from literature, such as TPC-C [13] and YCSB [4], capture high-contention use cases similarly to industrial transaction data.
- Domain-knowledge can be exploited at *run time*, to allow parallel transactions if it detects that their effects are independent at run time.
- Domain-knowledge can be exploited at *compile* time, to allow parallel transactions if it is *statically* detected that their effects are always independent.
- Similar to static compile time analyses, automated design time analyses can alert the designer of potential synchronization and performance bottlenecks.

The following paragraphs describe each secondary hypothesis approach in more detail.

Realistic Benchmark Analysis ING transaction data shows that high-contention can become a real issue in banking use cases and lead to bottlenecks for a 2PC implementation of synchronization. In order to guarantee consistency and atomicity, 2PC allows a single action per state machine instance to be in progress at the same moment, resulting in delaying other actions.

In literature, synthetic benchmarks, such as TPC-C [13] and YCSB [4], are used to evaluate database and middleware performance. These benchmarks claim to represent realistic use cases. We want to determine if these benchmarks accurately represent the high-contention use cases found in the ING transaction data set.

Path-Sensitive Atomic Commit We present a novel concurrency control mechanism, Path-Sensitive Atomic Commit [14] (PSAC), to reduce the bottlenecks of atomic commit in high-contention scenarios. Unlike 2PC, which is designed to be generic and applicable in all use possible cases, PSAC makes use of the domain knowledge of state machine actions.

PSAC uses the pre- and post-conditions of actions to detect, at run time, when actions are independent and can safely be parallelized. If an action is independent of in-progress actions' outcomes, it is safe to already start processing, while vanilla 2PC would have to delay, e.g., parallel withdrawals on bank accounts when the balance is sufficient for all. More parallel running actions in the high-contention objects reduce the delay and improve the throughput, up until the CPU is saturated. Performance evaluation shows that PSAC exhibits the same scalability characteristics as standard 2PC, but obtains up to 1.8 times median higher throughput in high-contention scenarios.

Static Independent Events Analysis PSAC reduces the bottleneck of busy objects in 2PC by preemptively calculating more to find the independent actions. This extra calculation would not be necessary when the detection of independence of actions is determined statically. Static offline analysis of the specifications can determine actions that can always be run independently, e.g., deposits on accounts without balance checks. Fewer run-time calculations and conflict checks result in lower action latency and more processing power for other actions.

An SMT-solver, such as Z3 [5], can be used to analyze all possible pairs of actions per specification to determine static independence. At run time a new concurrency control mechanism, Local Coordination Avoidance (LoCA) [15], uses the

static independent analysis output to speed up synchronization by skipping the dependency checks in the implementation. We expect that performance evaluation will show better latency and throughput than 2PC. LoCA should also perform better than PSAC in the specific cases where calculating run-time independence and new states is expensive.

The static optimization complements run-time PSAC, which still provides extra performance gains on top of this for the other actions, which are only non-conflicting when run-time data is known, e.g., in account transactions for withdrawals when enough balance is available.

Design Time Analyses Offline analysis can also be used to support specification designers by giving insight in potential performance bottlenecks at design time. The specification IDE can provide this feedback for the current in progress specification. We expect to implement at least two analyses using a similar analysis approach using SMT:

- Synchronization Bottleneck Analysis (SBA) finds actions, which potentially become synchronization bottlenecks by detecting when actions are only used in syncs and never independent.
- Synchronization Precondition Analysis (SPA) detects which precondition on an action might be weakened to reduce performance bottlenecks, by systematically removing preconditions from dependent actions, until it becomes independent.

We aim to create an implementation of the two design time analyses for Rebel, which feeds analysis results back into the IDE and provides infrastructure to add similar analyses.

4.1 Experimental Setup

Each hypothesis requires different experimental setup.

The synthetic benchmark evaluation requires analysis of the transactions in the benchmarks. An overview has to be created that shows which high-contention use cases are covered by the synthetic benchmarks, and how they relate to the bank transaction data.

For both PSAC and LoCA, we plan to implement variants of 2PC, PSAC and LoCA and keep the rest of the application and use case scenarios the same. This compares the difference between the mechanisms and avoids accidental implementation differences. Experiments consist of microbenchmarks and scalability benchmarks on cloud infrastructure of best and worst case scenarios, respectively low and high contention. The variants are compared in latency, maximum sustainable throughput and horizontal scalability potential.

Ideally the design time analyses are applied to real-life business use cases, to see if it detects bottlenecks correctly and is useful in industry setting.

5 Conclusion

The main goal of the thesis is to provide approaches to improve throughput and scalability of distributed software, reducing synchronization bottlenecks, by leveraging domain knowledge from models. Benchmark evaluation will show if synthetic benchmarks from literature cover the high-contention synchronization use cases. PSAC and LoCA optimize synchronization for run-time systems using dynamic and static independent events analysis, resulting in less contention and thus better latency and throughput. Problematic synchronization bottlenecks can be detected early with design-time analyses on models, such as SBA and PSA, which detects these bottlenecks and alerts the designer.

References

[1] Daniel Abadi. 2012. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *IEEE Computer* 45, 2 (2012), 37–42.

[2] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *PVLDB* 8, 3 (2014), 185–196.

[3] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno M. Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting consistency back into eventual consistency. In *EuroSys*. ACM, 6:1–6:16.

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154.

[5] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 337–340.

[6] Tamer Eldeeb and Phil Bernstein. 2016. *Transactions for Distributed Actors in the Cloud.* Technical Report.

[7] Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* Lecture Notes in Computer Science, Vol. 1032. Springer.

[8] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann.

[9] Joseph M. Hellerstein and Peter Alvaro. 2019. Keeping CALM: When Distributed Consistency is Easy. *CoRR* (2019). arXiv:1901.01930

[10] Brandon Holt, James Bornholt, Irene Zhang, Dan R. K. Ports, Mark Oskin, and Luis Ceze. 2016. Disciplined Inconsistency with Consistency Types. In *SoCC*. ACM, 279–293.

[11] Matthew Milano and Andrew C. Myers. 2018. MixT: a language for mixing consistency in geodistributed transactions. In *PLDI*. ACM.

[12] Nuno M. Preguiça, Carlos Baquero, and Marc Shapiro. 2018. Conflict-free Replicated Data Types (CRDTs). *CoRR* (2018). arXiv:1805.06358

[13] Francois Raab, Walt Kohler, and Amitabh Shah. 2013. Overview of the TPC benchmark C: The order-entry benchmark. *Transaction Processing Performance Council, Tech. Rep* (2013).

[14] Tim Soethout, Jurgen J. Vinju, and Tijs van der Storm. 2019. Path-Sensitive Atomic Commit: Local Coordination Avoidance for Distributed Transactions (Technical Report). *CoRR* abs/1908.05940 (2019).

[15] Tim Soethout, Jurgen J. Vinju, and Tijs van der Storm. 2019. Static Local Coordination Avoidance for Distributed Objects. In *AGERE!@SPLASH*. ACM. To appear.

[16] Jouke Stoel, Tijs van der Storm, Jurgen J. Vinju, and Joost Bosman. 2016. Solving the bank with Rebel: on the design of the Rebel specification language and its application inside a bank. In *ITSLE@SPLASH*. ACM.

[17] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed system development with ScalaLoci. *PACMPL* 2, OOPSLA (2018), 129:1–129:30.

[18] Michael Whittaker and Joseph M. Hellerstein. 2018. Interactive Checks for Coordination Avoidance. *PVLDB* 12, 1 (2018), 14–27.

[19] Xin Zhao and Philipp Haller. 2018. Observable atomic consistency for CvRDTs. In *AGERE!@SPLASH*. ACM, 23–32.