Static Local Coordination Avoidance for Distributed Objects

Tim Soethout
ING Bank, Amsterdam
CWI, Amsterdam
The Netherlands
Tim.Soethout@ing.com

Tijs van der Storm
CWI, Amsterdam
University of Groningen
The Netherlands
storm@cwi.nl

Jurgen J. Vinju
CWI, Amsterdam
Eindhoven University of Technology
The Netherlands
Jurgen.Vinju@cwi.nl

Abstract

In high-throughput, distributed systems, such as large-scale banking infrastructure, synchronization between actors becomes a bottle-neck in high-contention scenarios. This results in delays for users, and reduces opportunities for scaling such systems. This paper proposes Static Local Coordination Avoidance, which analyzes application invariants at compile time to detect whether messages are independent, so that synchronization at run time is avoided, and parallelism is increased. Analysis shows that in industry scenarios up to 60% of operations are independent. Initial performance evaluation shows that, in comparison to a standard 2-phase commit baseline, throughput is increased, and latency is reduced. As a result, scalability bottlenecks in high-contention scenarios in distributed actor systems are reduced for independent messages.

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, Coordination, Atomic consistency, Distributed systems

ACM Reference Format:
Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. 2019. Static Local Coordination Avoidance for Distributed Objects. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE '19), October 22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358499.3361222

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
AGERE '19, October 22, 2019, Athens, Greece
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6982-4/19/10…$15.00
https://doi.org/10.1145/3358499.3361222

1 Introduction

Enterprise software systems are large, complex, and hard to maintain. For instance, banks such as ING Bank[1], deal with large and complex IT landscapes consisting of many heterogeneous communicating applications, under high transaction loads. There is increased demand for high throughput and scalability for such systems.

In a distributed setting, both throughput and latency are heavily influenced by the amount of synchronization that is required between distributed objects. For instance, the generic atomic commit protocol Two-Phase Commit (2PC) [9] only allows a single event to be in progress per actor, all other events are queued. For high contention objects this leads to high latency and time-outs.

Recent work [3, 4] shows that invariants to maintain program-level consistency can be leveraged to optimize the implementation of synchronization. Invariant Confluence [3] shows that for the TPC-C benchmark [18] ten of twelve invariants are invariant confluent and require no coordination.

In this paper we propose a similar, but novel concurrency mechanism, called Local Coordination Avoidance (LoCA), that allows multiple concurrent in-progress events per object, when it can be determined that such events are independent of each other. One event is independent of another if the commit or abort of the latter can never invalidate the result of the former. In that case, processing of the latter can be started without waiting.

As an example, consider the text book example of a bank account entity with withdraw and deposit events, where withdraw has a precondition that the balance should be sufficient for the withdrawal. In this case the deposit event is independent of deposit itself, because depositing money can never invalidate the requirements of a deposit. Deposit is also independent of withdraw, because depositing money is always possible, even if the in-progress withdraw would fail. A withdraw event, however, is not independent from withdraw, because the failure or success of in-progress withdraw might influence the precondition of the second withdraw.

LoCA is informed by static analysis of state machine models. In our case, we use Rebel [23], a domain specific language (DSL) to model financial products as state machines which

[1] https://www.ing.com

RIGHTSLINK

communicate using atomic, synchronized events. LoCA consists of statically analyzing application invariants declared as pre- and postconditions in the state machine models. This results in pairs of events that are independent, regardless of local state at run time.

We have implemented the independent event analysis by transforming Rebel state machine models to constraint definitions for the Z3 Satisfiability Modulo Theory (SMT) solver [6], which computes the set of independent event pairs. This set is then input to the run-time system, which safely skips the precondition check if a new event comes in that is independent of all in-progress events,—otherwise it falls back to 2PC.

We have run the analysis on state machine models manually derived from the standard TPC-C [18] benchmark, and state machine models currently being prototyped inside ING Bank. In both cases, the results show that around 60% of event combinations are independent, suggesting that the benefits of LoCA could be substantial. Initial performance evaluation shows that LoCA, or a variant of LoCA that detects independence at run time increases throughput and reduces latency compared to vanilla 2PC in high contention scenarios.

The contributions of this paper are as follows:

- We formalize the notion of Statically Independent Events (*SIE*), a characterization of state machine models that captures when an event's preconditions are always independent of in-progress event's effects, and show how an SMT solver can be used to compute independent pairs from state machine models (Section 2);
- We describe a novel run-time concurrency control mechanism, Local Coordination Avoidance (LoCA), which uses independent events to speed up synchronization on distributed objects. We present a LoCA implementation leveraging *SIE* analysis results: LoCA^S (Section 3);
- We evaluate the *SIE* analysis on two realistic examples: the TPC-C benchmark and Rebel specifications developed at ING. We evaluate the performance of both 2PC and LoCA variants, and show that LoCA^S outperforms 2PC in high contention scenarios (Section 4).

Source code of the *SIE* analysis and LoCA implementation, together with result data is found in [21].

We conclude with a discussion and limitations (Section 5); related work (Section 6); further directions for research (Section 7); and a conclusion (Section 8).

2 Independent Events

2.1 Bank Account Example

Consider an example of a bank account state machine as seen in Figure 1. For simplicity it has two states `New` and `Opened` with an integer data field `balance`, and three Events: `Open`, `Deposit` and `Withdraw`, the latter two with an integer

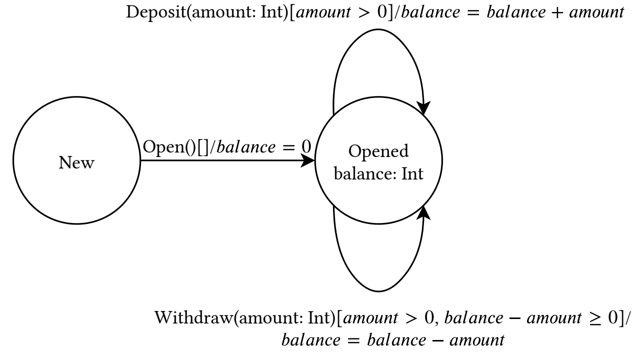


Figure 1. State machine of example simple account. Events are defined in state chart notation: Event(fields)[guard]/effect

`amount` event parameter, that should be a positive integer. The precondition on withdrawal ($balance - amount \geq 0$) makes sure that the balance does not become negative. The effect of the events are represented by labels on the edges in state chart notation. The effect of `Open` sets the balance to `0` and the state to `Opened`. The effect of `Deposit` and `Withdraw`, respectively increases or decreases the `balance` with the events' `amount` value.

Now, consider an instance of this bank account state machine running as an actor. The bank account actor handles a single event at one moment in time. A sequence of events is handled sequentially, one at a time. If an incoming event is part of a distributed 2PC-transaction with other actor participants, the actor first decides if the event is allowed by checking its preconditions, but it cannot transition to the next state yet. To maintain the serializability requirements of the distributed transaction, the actor has to wait on the whole transaction to either commit or abort the event. Other incoming events have to wait on the in-progress event.

In this particular example it becomes clear that waiting on in-progress events is not always necessary to ensure valid state machine transitions. For example, an incoming event `Deposit(10)`'s precondition check is independent of an in-progress `Withdraw(20)`'s effect in state `Opened(100)`. It does not matter if the in-progress `Withdraw(20)`'s effects are actually applied or not. The state machine stays in the `Opened` state, the only difference is the balance, which is only decremented on commit of the `Withdraw(20)` event. The new incoming `Deposit(10)` can always already start, since the `Opened` state allows it and the specific balance does not invalidate the precondition of `Deposit(10)`. The incoming `Deposit(10)` event is thus independent of the in-progress `Withdraw(20)` event.

2.2 Independent Events

An incoming event e_2 is independent of an in-progress event e_1, iff e_2 is accepted by the state machine, independent of whether e_1's effects are actually applied or not.

RIGHTSLINK

Table 1. Static independency of bank account events. E_1 in rows, E_2 in columns.

$SIE(E_1, E_2)$	`Open`	`Deposit`	`Withdraw`
`Open`	Delay	Delay	Reject
`Deposit`	Reject	Accept	Delay
`Withdraw`	Reject	Accept	Delay

In order to formalize the notion of independent event pairs, we consider a finite state machine, with states with data and events with parameters. We assume that the transition function is encoded in the preconditions $pre : Event \times State \rightarrow Boolean$. The predicate $pre(e, s)$ is true iff event e is valid in the given state s. An event is valid in a state if the transition function and its transition guards, the preconditions, allow it. The resulting outcome state of a transition, given the current state and event is encoded in $post : Event \times State \times State \rightarrow Boolean$. $post(e, s_{from}, s_{to})$ is true iff event e in state s_{from} leads to post state s_{to}.

Given an in-progress event e_1 and an incoming event e_2 in some starting state s, the independent event relation $IE(e_1, e_2, s)$ is defined as follows:

$$\forall s' \in State. \\ pre(e_1, s) \wedge post(e_1, s, s') \rightarrow (pre(e_2, s) \leftrightarrow pre(e_2, s')) \quad (1)$$

$IE(e_1, e_2, s)$ denotes that e_2's acceptance is independent of the outcome of e_1 in state s. The predicate $pre(e_1, s)$ makes sure that e_1 has valid preconditions in s, describing the situation where e_1 is already accepted by the participant, but has not yet committed nor aborted. $post(e_1, s, s')$ binds s' to the post state when e_1's effects are applied on s. An event e_2 is independent of an event e_1 iff, for all possible post states s', the evaluation of the preconditions of e_2 in both s and s' give the same result. Intuitively this means that whether e_1 eventually commits or aborts and its effects are applied or not, does not influence the precondition check of e_2. There is no possible way for the result of e_1 to influence the validity of e_2.

2.3 Statically Independent Events

The *IE* relation captures independence at run time: it considers preconditions and postconditions, given a current state machine state (e.g., balance). This however, requires runtime computation when dispatching incoming events, which could be expensive. Here we introduce statically independent events, which avoid this computation step.

Table 1 displays all statically independent event pairs of the bank account example. It shows the decision on event of type E_2 given that an event of type E_1 is in progress. For instance, `Deposit` is statically independent of both `Deposit` and `Withdraw`, because no matter the actual run-time state of the bank account, deposits can always be directly accepted. Similarly, if a `Deposit` is in progress, an `Open` event should be immediately rejected, since the state machine's transition function disallows it. For all event pairs that are not independent, the decision is delay.

SIE is a relation between two types of events, E_1 and E_2, without considering their parameter values or the run-time state of an actor.

$$SIE(E_1, E_2) = \forall s \in State, e_1 \in E_1, e_2 \in E_2.IE(e_1, e_2, s)$$

where $\forall e_i \in E_i$ means for all possible event instances of type E_i. SIE(E_1, E_2) is true when all possible instances of the event types E_1 and E_2 are independent in all possible starting states s.

2.4 Computing *SIE*

The *SIE* definition can be used to transform state machine models to first-order logic formulas as input to SMT-solvers, like Z3 [6] to find the statically independent event pairs at compile time. For this we assume a mapping of the state machine's transition relation and the pre- and postconditions of each type of event as formulas in first-order logic.

In order to let an SMT-solver find the statically independent events, we let the solver search for counter examples, the *dependent* event pairs. This means that for every combination of event types E_1 and E_2, we ask the solver whether the formula $\neg SIE(E_1, E_2)$ can be satisfied. If it is satisfiable, the resulting model represents a counter example witnessing the fact the E_2 is dependent on E_1, otherwise they are independent.

The negation of $SIE(E_1, E_2)$, after inlining the definition of *IE* is:

$$\exists s, s' \in State, e_1 \in E_1, e_2 \in E_2. \\ pre(e_1, s) \wedge post(e_1, s, s') \wedge \neg (pre(e_2, s) \leftrightarrow pre(e_2, s'))$$

To satisfy this formula the solver needs to find a model instance with some starting state s, and two event instances e_1 and e_2, for which it holds that event e_1 is valid in s and its follow-up state is s', but event e_2 should be invalid in only one of states s or s'. Such a model instance denotes that e_2 is *dependent* on e_1's outcome. The resulting e_1 and e_2 event instances are the counter-example for the static independence of E_2 on E_1, making event type E_2 statically dependent on event type E_1.

`Withdraw` is not statically independent of `Deposit` and the first instance found by Z3 is indeed an example of this: `Withdraw(35)` is only allowed when the `Deposit(1202)` actually commits in state `Opened(34)`, otherwise the balance would not be sufficient. Another example is where `Open` is dependent on the outcome of another `Open` in state `New`, which makes sense since an account can only be opened once.

However, checking $\neg SIE$(`Withdraw`, `Deposit`) will return "unsat" which means that no counter-example could be found to show that the outcome of `Withdraw` would influence the acceptance of `Deposit`.

RIGHTSLINK

2.5 Always Accept or Always Reject?

The *SIE* relation determines whether one event is independent of the other, but does not say if an event should always be accepted or always be rejected, as shown in Table 1.

To obtain this information we partition the definition of *SIE* in two variants SIE^{Accept} and SIE^{Reject}, where the equivalence used in *IE* (Definition 1) is split in the case where the preconditions of both events are true, and the case where neither of them are true:

$$SIE^{Accept}(E_1, E_2) = \forall s, s' \in State, e_1 \in E_1, e_2 \in E_2. \; pre(e_1, s) \land post(e_1, s, s') \rightarrow (pre(e_2, s) \land pre(e_2, s'))$$

$$SIE^{Reject}(E_1, E_2) = \forall s, s' \in State, e_1 \in E_1, e_2 \in E_2. \; pre(e_1, s) \land post(e_1, s, s') \rightarrow \neg (pre(e_2, s) \lor pre(e_2, s'))$$

To ensure that the solver does not return "junk" models in which events are always invalid, regardless of the state (e.g., `Withdraw(-1000)`), we instruct the solver to only consider such events by asserting $\exists s \in State.pre(e_2, s)$. Following the same process as with *SIE* above, SIE^{Accept} and SIE^{Reject} can be used to find statically independent event pairs, knowing whether the decision should be accept or reject.

3 Local Coordination Avoidance (LoCA)

A run-time system for a state machine-based language like Rebel is typically implemented as follows. A distributed object receives a request as part of a 2PC distributed transaction. It then becomes participant in this transaction. If the request is valid according to the corresponding event's preconditions, the object is locked until the transaction completes. If the preconditions do not hold, the object declines the request immediately and does not have to lock.

In this case, for each incoming request, the participant object has to check the preconditions and act accordingly. If the request is valid, it votes to commit the transaction and the object is locked for other requests in order to maintain the consistency guarantees of 2PC. Even though the participant object has voted to commit, it cannot continue until the coordinator responds, since it does not know if other transaction participants have voted to abort. Incoming transactions are delayed in order of arrival until the transaction coordinator commits or aborts the transaction. This results in potentially high wait times and thus high transaction latency for busy objects.

Local Coordination Avoidance (LoCA) is our novel concurrency control mechanism that leverages *SIE* information at run-time to run multiple parallel 2PC requests per participant object. *IE* allows an implementation to safely start processing new events when previous events are still in progress. LoCA first checks independence according to the pre-computed results of the static independent event analysis (*SIE*). If two types of events are not statically independent, the actual event occurrences can still be dynamically independent; this is checked according to the *IE* relation at run time. If events are dynamically dependent still, LoCA falls back to vanilla 2PC.

For evaluation purposes, we distinguish variants of LoCA according to which kinds of independence checking are done at run time: LoCA^S (only checks based on *SIE*), LoCA^D (only checks based on *IE* [22]), and LoCA^{SD} (first *SIE*, then *IE*).

3.1 Static LoCA

Static LoCA (LoCA^S) first considers SIE^{Accept}, then SIE^{Reject}, and falls back to 2PC.

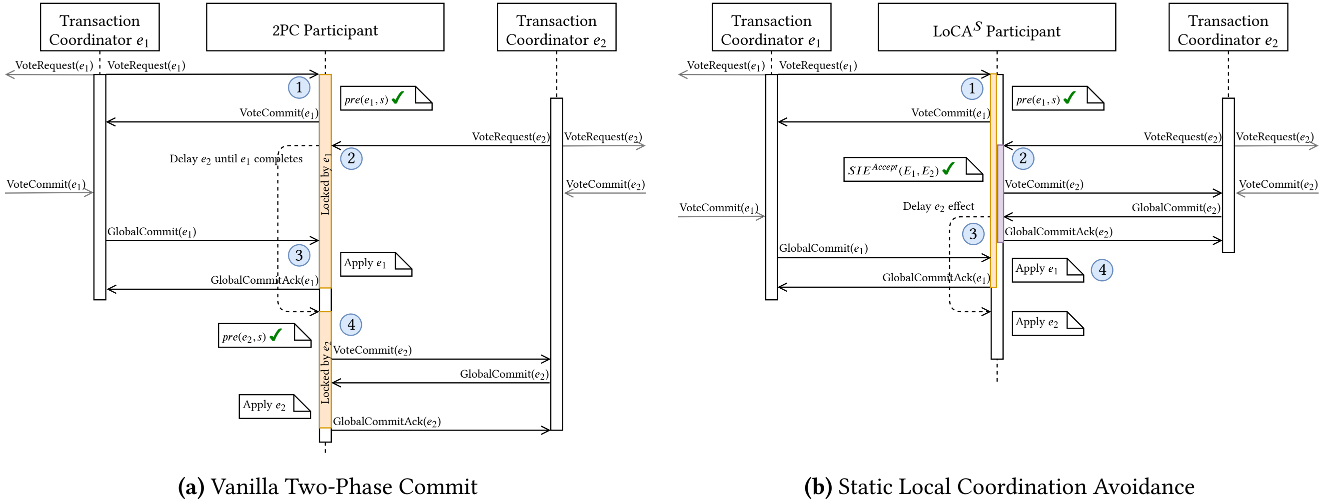
LoCA^S leverages the *SIE* analysis results. In the case where the transaction participant is waiting on a response of an in-progress transaction's coordinator, it can use the *SIE* independent event pairs to determine if it is always safe to start another incoming request in parallel.

If the incoming event's type is an independent accept for all the in-progress events' type, as determined by SIE^{Accept}, it can be started immediately without checking its preconditions. If not matched by SIE^{Accept}, and the incoming event is an independent reject, determined by SIE^{Reject}, for all in-progress events, it can be immediately rejected. If the request is not statically independent for both sets, the incoming event is dependent on at least one of the in-progress events' type and has to be delayed until it is finished.

Figure 2 shows sequence diagrams to compare vanilla 2PC to LoCA^S in the case that LoCA^S can directly accept an event. For 2PC an action is delayed when another action is in progress. For LoCA^S the action's transaction is started when SIE^{Accept} allows it. In Figure 2a, (1) a 2PC-participant receives a VoteRequest(e_1) message, and responds with VoteCommit(e_1) because the preconditions $pre(e_1, s)$ allow it. This locks the resource until e_1 commits or aborts. When e_1 is still in progress, the participant receives (2) a VoteRequest for another event e_2, and it is delayed until e_1 completes. On receiving of GlobalCommit(e_1) (3), the effects of e_1 are applied, the state is updated and acknowledgement is replied. Now the delayed e_2 is started (4), and a VoteCommit(e_2) is send. e_2 is eventually committed, and its effects applied.

In Figure 2b, a LoCA^S-participant receives (1) a VoteRequest(e_1) message, and similarly accepts the event because the precondition $pre(e_1, s)$ holds. Unlike 2PC the resource is not locked, but guarded by static independence guarantees. When VoteRequest(e_2) arrives (2), it now checks if it is safe to execute e_2 in parallel with e_1, by checking the event's types in $SIE^{Accept}(E_1, E_2)$. In this case $SIE^{Accept}(E_1, E_2)$ holds and VoteCommit(e_2) is readily sent. Now e_2 commits earlier (3), and in order to maintain serializability, the effects of e_2 are delayed to preserve the original order. When e_1 is allowed to commit (4), its effects are applied, and the postponed effects of e_2 as well. The case for immediate reject is analogous.

RIGHTSLINK



(a) Vanilla Two-Phase Commit **(b)** Static Local Coordination Avoidance

Figure 2. 2PC and LoCA^S, Direct Accept.

4 Evaluation

In this section we evaluate static local coordination avoidance to answer two questions:

RQ 1. *How often are events independent in realistic scenarios?*

RQ 2. *What is the effect of LoCA on performance in terms of throughput and latency?*

4.1 Independence in Realistic Scenarios (RQ 1)

In order to evaluate the relevance of *SIE* analysis we have analyzed two sets of Rebel models and computed the independence results. The first set consists of Rebel state machine models manually derived from the standard TPC-C benchmark [18], the second set consists of models of payment infrastructure developed at ING Bank.

TPC-C There is no direct mapping for TPC-C's transactions to Rebel, but we can model the tables and the transaction's operations on them. TPC-C consist of 9 database tables and 5 transactions, which we model as state machines where transactions are represented as events.

Some TPC-C transactions make decisions based on data which is read from the state machine. To avoid that in-progress events modify such data, we model this data flow using event parameters.

While this approach makes sure that exposed values are not changed by in-progress events, it is an over-approximation and leads to false negatives of dependent event pairs. Incorrectly detected dependent event pairs cannot be parallelized at run time, which is still correct, but not as efficient as when correctly identified.

Table 2 shows the *SIE* analyses' results for each table specification and a percentage describing the ratio between independent and all event pairs. As can be seen, many events are independent for this case. This is the case because often

Table 2. TPC-C *SIE* Analyses

TPC-C Table	#States / #Events	#Direct Accept / #Direct Reject	Indep. Ratio
Stock	1 / 2	4 / 0	100%
NewOrder	3 / 3	2 / 2	44%
Order	2 / 2	4 / 0	100%
District	1 / 3	6 / 0	67%
Customer	1 / 4	12 / 0	75%
OrderLine	2 / 4	6 / 3	56%
Warehouse	1 / 1	1 / 0	100%
History	2 / 1	0 / 0	0%
Item	2 / 2	0 / 1	25%
Total	15 / 22	35 / 6	64%

the specific tables' data is only read, and not used for decisions later in the transaction. Note that these results are only for local independency decisions in a state machine instance, representing a single row in the database tables.

ING Bank Account Models In order to evaluate *SIE* effectiveness on an industrial use case, we run the analysis on Rebel specifications being developed at ING Bank, containing multiple types of bank accounts and Single Euro Payments Area (SEPA) bank transactions. Our specification data set consists of 29 Rebel specifications. 7 of them used features not yet supported by our analysis tool. The remaining 22 specifications are analyzed with small changes. For some of them the data types and preconditions are simplified, in such a way that it does not influence the *SIE* analysis, for instance changing `DateTime` fields to `Integer` fields and mapping static set membership tests to string equality.

The analysis results of the resulting 22 specifications are presented in Table 3. Most specifications are relatively small in terms of number of states and events. Many independent

RIGHTSLINK

Table 3. *SIE* Analyses' results of ING product specifications

Specification	#States / #Events	#Direct Accept / #Direct Reject	Indep. Ratio
CreditTransfer	9 / 9	1 / 52	65%
Restriction	3 / 4	6 / 3	56%
DepositBlock	3 / 4	2 / 5	44%
DirectDebitBlock	3 / 4	2 / 5	44%
WithdrawBlock	3 / 4	2 / 5	44%
Limit	5 / 5	1 / 12	52%
NoLimit	3 / 3	1 / 2	33%
RevolvingAccount	2 / 3	4 / 2	67%
DirectDebitAccount	2 / 3	4 / 2	67%
TreasuryAccount	4 / 4	4 / 8	75%
CreditTransferBatch	5 / 6	10 / 5	42%
CreditBooking	4 / 3	0 / 2	22%
DebitCreditorBooking	3 / 2	0 / 1	25%
FromExternalDebitBooking	3 / 2	0 / 1	25%
ToExternalDebitBooking	3 / 2	0 / 1	25%
DebitBooking	13 / 17	0 / 188	65%
CurrentAccount	3 / 4	4 / 4	50%
LocalCreditTransfer	6 / 5	0 / 12	48%
SepaCreditTransfer	6 / 8	15 / 29	69%
Arrangement	4 / 4	2 / 7	56%
BankPayment	4 / 4	0 / 6	38%
ThirdPartyPayment	5 / 3	0 / 5	56%
Total	96 / 103	58 / 357	61%

events pairs are direct reject, since many events are not allowed in multiple states. More than 60% of all event pairs are independent, suggesting that *SIE* analysis would be beneficial in industrial scenarios. This analysis shows how often events are independent in realistic scenarios answering RQ 1.

4.2 Throughput and Latency (RQ 2)

LoCA^S is expected to show performance benefits when a transaction participant is involved in multiple transactions at the same moment in time and for independent events. In low-contention scenario we expect little extra performance in using LoCA^S compared to 2PC.

The goal of the performance evaluation is to find out if this expectation holds. We ran several synthetic scenarios in microbenchmarks in order to confirm these expectations.

In order to evaluate LoCA we prototyped a small accounting service providing dependent and independent events in the state machine DSL Rebel [23]. The *SIE* Analysis translates Rebel specifications to SMT using Rascal [15] and runs the analysis using the state-of-the-art SMT-solver Z3 [6]. The *SIE* results are used in a LoCA implementation [?], an actor-based runtime system, based on the Akka actor toolkit [1], on the JVM.

Akka enables fault tolerance and horizontal scalability by sharding actors over multiple servers and provides locational transparent message passing between them. These features together with persistence and state machine primitives, are used to implement LoCA and 2PC. The Rebel state machine models are translated to communicating run-time actors.

The 2PC implementation follows the description by Tanenbaum and Van Steen [25]. In order to avoid deadlocks, we make sure that all transactions participants are locked in increasing order.

For all experiments in this paper, we limit the maximum number of parallel events per actor for LoCA to a configurable limit of 8. A higher number results in reduced throughput and worse latency when contention increases. This is a problem, especially for LoCA^D and LoCA^{SD} with run-time dependent events, when the computation time grows exponentially due to more concurrently in-progress actions.

We present multiple microbenchmarks and their throughput and latency results on a single application node. Each benchmark is run using the different synchronization implementation variants, 2PC, LoCA^S, LoCA^D, LoCA^{SD}, and increasing contention rate. The benchmark scenarios are the following:

- Statically dependent events – `Withdraws` on single account
- Statically independent events – `Deposits` on single account
- Distributed transactions with statically dependent and statically independent event – Money transfers between two accounts
- Distributed transactions with high-contention statically independent events and low-contention dependent events – Tax direct debit use case: `Deposits` on a single tax account & `Withdraws` on 10 000 taxed accounts

The first two benchmarks represent statically dependent and independent events. These are baseline benchmarks to determine whether LoCA^S improves throughput when contention increases for independent events, but has to fall back to 2PC for dependent events.

The distributed transaction cases are more realistic. In the money transfer case between two accounts, the expectation is that LoCA^S improves performance only slightly over 2PC, because the whole distributed transaction has to wait on the slowest dependent event participant. On the other hand for LoCA^D and LoCA^{SD} we expect better performance, since the `Withdraws` are run-time independent, because enough balance is available for multiple parallel `Withdrawals`.

For the tax use case, the expectation is that LoCA^S performs better than 2PC and is on par with LoCA^D and LoCA^{SD}, because the statically independent `Deposits` on the tax account can be parallelized.

The microbenchmarks are run using JMH [14] and measure the maximum throughput in transactions per second and transaction latency. The hardware used is a dual core Intel i7-7567U 3.5GHz up to 4.0GHz with 32GB of ram on Linux using Java AdoptOpenJDK HotSpot 11.0.2+9 64bit. Each run consists of 5 warmup cycles and 20 measure cycles of each 10 seconds.

RIGHTSLINK

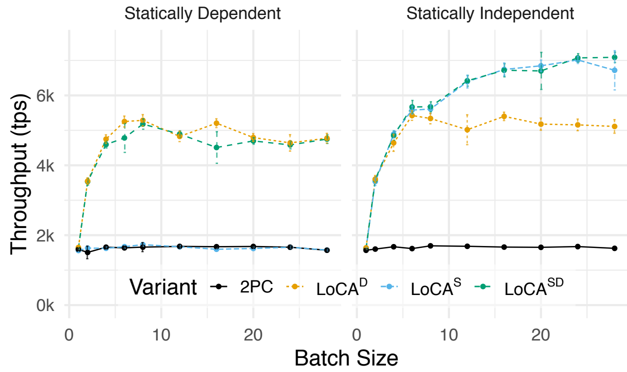


Figure 3. Statically Dependent and Independent events' throughput. Higher is better.

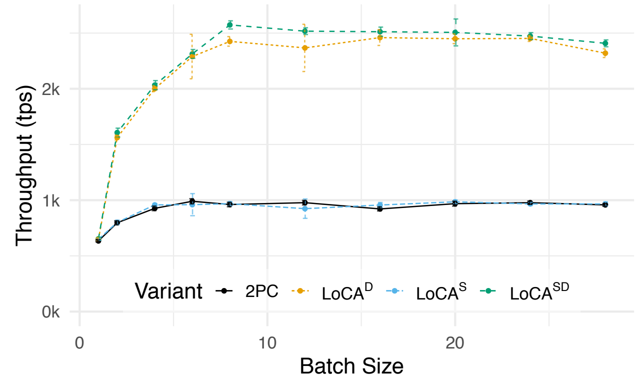


Figure 4. Money transfer microbenchmarks' throughput

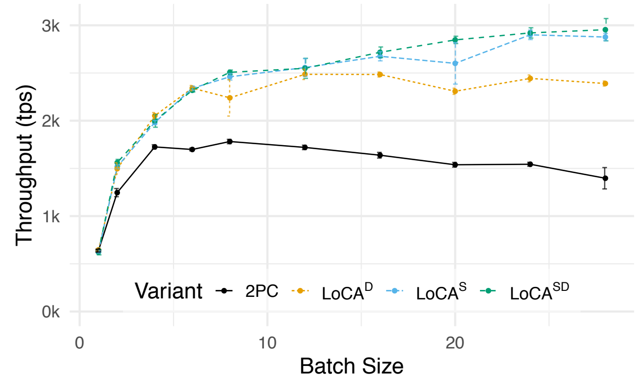


Figure 5. Tax microbenchmarks' throughput

To increase contention, multiple parallel events are requested in batches. The batch size is varied in order to find out when contention becomes a problem and determines the maximum number of events in-progress on a participant.

Microbenchmark results The throughput results for the statically dependent and independent events are shown in Figure 3. For statically dependent events, in low-contention scenarios (Batch Size = 1), all variants reach 1800 transactions per second. As expected for higher contention (Batch Size > 1), LoCA^S performs the same as 2PC, because both algorithms only allow a single event to be in progress for statically dependent events. Since LoCA^{SD} falls back to LoCA^D for run-time independent events there is a higher maximum throughput around 5000 transactions per second. At run time LoCA^D determines that the `Withdraw` events can be safely run concurrently because enough balance is available.

For statically independent events, 2PC has the same maximum throughput as the statically dependent variant, because it handles all events sequentially. LoCA^S reaches higher maximum throughput than the dependent variant, around 7000, because for independent events, it can immediately accept. LoCA^{SD} has similar performance, because for independent events it is equivalent to LoCA^S and does not have to fall back to LoCA^D. LoCA^D also detects the independence at run time, but suffers a computational overhead compared to LoCA^S, resulting in a lower maximum throughput, but still performs better than 2PC.

Interestingly, LoCA^{SD} performs the best in both cases. It leverages the static knowledge when events are independent so no precondition calculations are necessary, and in the dependent case it can profit from *IE*'s dynamic independence check.

Distributed transaction: Money transfer Figure 4 shows the throughput results for the money transfer microbenchmark. As expected, all variants perform the same in the low-contention case (Batch Size = 1). 2PC and LoCA^S have similar maximum throughput at 1000 transactions per second. This is expected because, although the `Deposit` is statically independent, the whole transaction has to wait on the statically dependent `Withdraw`, limiting the overall throughput. Both LoCA^D and LoCA^{SD} perform better at around 2500 transactions per second, which can be explained by the dynamic independence of `Withdraw`.

Distributed transaction: Tax collection For the tax collection, the throughput results shown in Figure 5 are as expected. 2PC performs up to 1750 transactions per second and for higher contentions slowly drops, which is explained by larger batch size having to wait longer for the sequential handling by the tax account. LoCA^S and LoCA^{SD} perform similar, up to 3000 tps, since throughput is limited by the tax account. LoCA^D performs slightly worse because of the computational overhead.

Latency Results For all microbenchmarks, latency results were also collected [21]. Overall, the latency percentiles follow the same curve for all algorithm variants, where higher throughput corresponds to lower latency per operation.

Overall, all LoCA variants outperform 2PC, both in throughput and latency, except for the low-contention case,

RIGHTSLINK

where it performs similar. We thus answer RQ 2 on the effect of LoCA and variants on performance.

5 Discussion

The performance evaluation has shown that *SIE* analysis may increase performance both in throughput and latency, for situations where multiple requests arrive at objects in a small amount of time. The statically independent events make sure that only the event types have to be inspected.

In distributed transactions with statically independent events, LoCA^{SD} and LoCA^{S} perform better than LoCA^{D} since LoCA^{D} is computationally more expensive when the number of parallel events increases. However, in scenarios where the independence can only be determined dynamically, the combined version LoCA^{SD} still outperforms 2PC. It turns out that all LoCA variants, LoCA^{S}, LoCA^{D} and LoCA^{SD}, perform as least as well as 2PC, and can therefore be used as a replacement in all cases where pre- and postconditions are known. The *SIE* analysis finds the independent events pairs in order of seconds per state machine model.

SIE analysis is applicable in the specific scope where requesters of operations are only interested in the success or failure of the operation. The requester receives the acknowledgement directly, but not yet the post state of an entity, when this depends on other in-progress events. The new state can be queried in a new transaction.

IE is an asymmetric relation, and also does not require events to be commutative. Even though multiple events can be in progress at the same time, their effects always are applied in order or arrival, leading to serializable behavior and commutativity is not necessary. In our specific implementation unconditional acceptance of events is already communicated to the requester, but not the outcome state.

Limitations LoCA results in performance gains in the specific scenario of independent actions and high-contention. *SIE* can find independent events, but from this information it is not directly clear if this independent result allows LoCA to speed up performance in practice, since it might not be a high-contention scenario. In practice the events could very well be low-volume or the requests are already spread out on many different specification instances.

LoCA's benefit is most visible when the following conditions hold:

- objects are involved in many synchronization steps from different objects, each with low load, but making the single objects a bottleneck for all
- objects are involved in a single type of synchronization step from the same objects

Both cases are a scalability limiting factor when request volumes continue to increase and are typical for a bank like ING Bank. If the machines are low volume and the instances spread out, LoCA's performance gain is limited, although it will never worsen the performance compared to 2PC.

6 Related Work

IE and LoCA focus on optimizing performance of a single object, running in a single location, which is highly-contended. It uses conventional actor architecture approaches to shard actors over multiple machines. Much literature focusses on how one can do parallel updates in multiple geo-distributed locations without communication overhead, and only synchronize if really necessary for application consistency. Running multiple instances of the same object in order to allow parallel operations, improves performance in high-contention scenarios as well. *SIE* analysis is related to checking which operations are commutative and violate invariants.

Many distributed databases focus on scaling by splitting data into partitions, such as Cassandra [5], H-Store/VoltDB [24], Spanner [7]. Within partitions they fall back to sequential operations, such as 2PC and Optimistic Concurrency Control. LoCA focusses on avoiding coordination locally in these partitions and could thus be implemented inside other database systems to speed up these sequential operations, when program-invariants are known.

Program-level consistency The notion of program-level consistency, instead of generic data-consistency, is a valid way to capture what a program should functionally do and also gives opportunity to improve performance while maintaining the program invariants. Work in this direction tries to find ways to characterize this notion, which in turn enables optimized implementations.

The CALM theorem [10] says that monotonic programs do not need coordination. So ideally programs should have only monotonic parts. *IE* and LoCA describe on a local object level, how one can execute events in parallel, improving performance, and make sure the object only grows monotonically in its lifecycle, by exploiting the programs functional requirements.

Coordination Avoidance [2] states that coordination can be avoided if all local commit decisions are globally valid. *IE* describes local avoidance of coordination between events on the object.

Explicit Consistency [4] also uses an SMT-solving approach to "identify which operations would be unsafe under concurrent execution". For unsafe operations, it presents approaches on changes to make concurrent execution safe or requires an explicit synchronization implementation. It focusses on parallel changes on geo-located data centers.

Observable Atomic Consistency [26], related to RedBlue Consistency[16], categorizes operations in two categories: Commutative operations on CRDTs which can be handled in any order by different replicas, and totally ordered operations, for which the replicas need to coordinate.

SIE and LoCA operate in a different design space, where all operations on an object go through a single actor, which is not designed for a geo-distributed setting. It would be interesting to explore this space and an extension of LoCA.

RIGHTSLINK Copyright Clearance Center

Conflict-free Replicated Data Types [8, 20] guarantee Strong Eventual Consistency. This means that all replica's converge to the same state if they receive the same messages, not necessarily in the same order. *IE* provides strong consistency, meaning serializability, since events are processed in the original order, but are internally processed concurrently.

Adahbi[2] [19] reasons that in many cases the high-contention bottleneck in 2PC can be avoided by making the precondition check of another participant a local decision, for example by querying the data required for its precondition check from the other participant. In that way data flows only one way, and both participants can locally decide, sometimes with data from the other, if the transactions will commit or abort without waiting on each other. *IE* focusses on local avoiding of delays, but still uses 2PC for the coordination of the transaction.

Single node optimizations Phase Reconciliation [17] is a run-time technique, that splits contended objects over multiple cores, and allows multiple commutative operations of the same type in parallel on each core. After a configurable window, the split versions are recombined in a reconciliation phase maintaining serializability. This improves throughput for contended objects. Similarly to LoCA, it thus allows safe parallel operations on an object, and the operations should not return values. Differences are that the operations are limited to commutative operations and only allow a single operation type per split phase. LoCA and *IE* do not require commutativity for operations and allow different operation types to run in parallel, as long as they are independent, which is either detected statically or dynamically, without special effort by the specification designer. Phase Reconciliation's implementation does not support durable writes yet, which LoCA explicitly supports. A difference in scope is that LoCA applies effects in the original order. Its parallelism has nothing to do with the kind of effects, but whether it influences preconditions of other events. Phase Reconciliation splits up the effects over threads, which LoCA does not do. Phase Reconciliation could be embedded within LoCA, to speed up the applying of effects within a LoCA object.

Flat Combining [11] and a distributed version of it [13] show an interesting way to speed up concurrent access to data by keeping track of concurrent operations on an object, and letting the first thread obtain the lock, batch process all the operations and notify the requesters with the result of their operation. Flat Combining focusses on reads and writes on data structures, and thus focusses on effects. LoCA differs in the sense that it is focussed on distributed transactions, where it is externally decided by other transaction participants if in-progress operations will be commit or aborted. An interesting part of Flat Combining is the canceling out of sequential operations, locally in the concurrent operations list, e.g. push and pops in a stack. This results in reduced sending of operations to the actual data structure. Interesting future work related to this, is to statically detect, using analyses similar to *SIE*, which events can safely be combined, and (partly) cancel each other out within invariant bounds.

[2]https://dbmsmusings.blogspot.com/2019/01/its-time-to-move-on-from-two-phase.html

7 Future Work

This paper describes Independent Events pairs on event type level, ignoring parameters. It would be interesting to have a more granular approach by including symbolic field values. The SMT-solver can synthesize computationally cheap field bound checks that LoCA can use at run time to determine when events are independent.

In order to make *IE* more generally applicable, reading of data can be added. Now, static independence is determined by event types, resulting in either accept or reject. *IE* does not support state or return values for events. This has to be simulated as seen in the TPC-C use case. If exposed (computed) values are specified, this can be taken into account for the analysis. The "exposed" effect describes which (computed) values are exposed by events next to the postconditions, which only describe internal state changes. This would result in support for SQL like transactions, which can use sub queries and can represent TPC-C's usage of data, resulting in fewer false-negative dependent events. This can also support analysis of nested synchronization in Rebel, by tracking if nested participants rely on exposed event parameter values.

IE focusses on parallel distributed transactions. It would be interesting to explore if the *IE* property can be exploited in other non-transactional cases, for example in the context of Active Objects [12].

The current *SIE* analysis can take false-positive dependent event pairs into account, because the SMT-solver is allowed to synthesize any state possible, also states that cannot occur in the normal state machine life cycle. Extra assertions could be added to make sure only reachable states are used. A drawback could be that the state machine representations becomes more involved and solve times can become higher.

In order to avoid deadlocks, transaction participants are locked in increasing order. In many cases LoCA does not need this, because it allows multiple transactions in parallel. LoCA should only need increasing order locking when deadlocks can happen for dependent events. Static analysis could detect this and switch to parallel requesting of locks, saving multiple round trip times in transaction latency, compared to increasing order locking.

This paper presents microbenchmark performance evaluations on a single application node. Since the implementation is based on actors, which can run as-is in a clustered environment, it would be interesting to do further scalability performance evaluation on a cloud environment. It would be interesting to replicate ING's workloads using LoCA to find a benefit compared to their current implementation.

RIGHTSLINK

Commutativity of statically independent events can most probably be statically determined. This would allow reordering of events at run time and would allow requesters to see outcome states earlier. Reordering would require designers of specifications to take care with pre- and postconditions to make sure that time sensitive or otherwise important event orders are captured explicitly.

Offline analysis using SMT solvers can also be used to support specification designers by giving insight in potential performance bottlenecks at design time. Research directions include detection of events which are used in multiple synchronized steps but never independent, and suggestions on how to make events independent. The latter can be done by systematically removing preconditions from dependent event pairs, until it becomes independent. This signals which preconditions might be weakened by the specification designer to reduce performance bottlenecks.

LoCA uses an atomic commitment protocol to implement the actual transaction, which is now 2PC. Optimistic concurrency control, instead of 2PC, could provide even more performance improvements, since fewer rollbacks or aborts would be required for independent events.

8 Conclusion

Atomic commitment protocols such as Two-Phase Commit (2PC) may lead to bottlenecks for high-contention objects, because requests have to wait on previous events to finish. It is possible to improve throughput and latency, by increasing parallelism of events on an object, while maintaining application consistency.

Independent Event (*IE*) pairs capture when a state machine object can safely start processing events when other events are still in progress. Statically Independent Events (*SIE*) analysis enables detection of types of event pairs that are always independent, at compile-time. We have implemented the *SIE* analysis on top of the Rebel state machine DSL by translating object invariants to SMT constraints and checking the *SIE* property. Local Coordination Avoidance (LoCA) leverages the resulting independence information to start more events per object concurrently, when it is determined that new events cannot violate the object's invariants.

We have shown that in two sets of realistic Rebel specifications, around 60% of events are always independent, which suggests that LoCA potentially increases throughput in distributed systems. Preliminary performance evaluation shows that, compared to 2PC, LoCA performs at least similar to 2PC, but LoCA does increase throughput and reduce latency in high-contention scenarios with independent events.

References

[1] Akka. 2019. . https://akka.io

[2] Peter Bailis. 2015. *Coordination Avoidance in Distributed Databases.* Ph.D. Dissertation. University of California, Berkeley, USA.

[3] 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.

[4] 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.

[5] Cassandra. 2019. . https://cassandra.apache.org/

[6] 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.

[7] James C. Corbett et al. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 8:1–8:22.

[8] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying strong eventual consistency in distributed systems. *PACMPL* 1, OOPSLA (2017), 109:1–109:28.

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

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

[11] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*. ACM, 355–364.

[12] Ludovic Henrio and Justine Rochas. 2017. Multiactive objects and their applications. *Logical Methods in Computer Science* 13, 4 (2017).

[13] Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2013. Flat combining synchronized global data structures. In *7th International Conference on PGAS Programming Models*. 76.

[14] JMH. 2019. *OpenJDK: Java Microbenchmark Harness.* https://openjdk.java.net/projects/code-tools/jmh

[15] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM*. IEEE Computer Society, 168–177.

[16] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *OSDI*. USENIX Association, 265–278.

[17] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Tappan Morris. 2014. Phase Reconciliation for Contended In-Memory Transactions. In *OSDI*. USENIX Association, 511–524.

[18] 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).

[19] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2014. An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. *PVLDB* 7, 10 (2014), 821–832.

[20] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *SSS (Lecture Notes in Computer Science)*, Vol. 6976. Springer, 386–400.

[21] Tim Soethout. 2019. Static Local Coordination Avoidance for Distributed Objects Artifacts. https://doi.org/10.5281/zenodo.3405232

[22] 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).

[23] 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.

[24] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[25] Andrew S. Tanenbaum and Maarten van Steen. 2007. *Distributed systems - principles and paradigms, 2nd Edition.* Pearson Education.

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

RIGHTSLINK