Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems

Yijian Liu
University of Copenhagen, Denmark
liu@di.ku.dk

Li Su∗
Alibaba Group, China
lisu.sl@alibaba-inc.com

Vivek Shah∗
Deon Digital Denmark A/S, Denmark
bonii.vivek@gmail.com

Yongluan Zhou
University of Copenhagen, Denmark
zhou@di.ku.dk

Marcos Antonio Vaz Salles
University of Copenhagen, Denmark
vmarcos@di.ku.dk

ABSTRACT
The actor model has been widely adopted in building stateful middle-tiers for large-scale interactive applications, where ACID transactions are useful to ensure application correctness. In this paper, we present Snapper, a new transaction library on top of Orleans, a popular actor system. Snapper exploits the characteristics of actor-oriented programming to improve the performance of multi-actor transactions by employing deterministic transaction execution, where pre-declared actor access information is used to generate deterministic execution schedules. The deterministic execution can potentially improve transaction throughput significantly, especially with a high contention level. Besides, Snapper can also execute actor transactions using conventional nondeterministic strategies, including S2PL, to account for scenarios where actor access information cannot be pre-declared. A salient feature of Snapper is the ability to execute concurrent hybrid workloads, where some transactions are executed deterministically while the others are executed nondeterministically. This novel hybrid execution is able to take advantage of the deterministic execution while being able to account for nondeterministic workloads.

Our experimental results on two benchmarks show that deterministic execution can achieve up to 2x higher throughput than nondeterministic execution under a skewed workload. Additionally, the hybrid execution strategy can achieve a throughput that is close to deterministic execution when there is only a small percentage of nondeterministic transactions running in the system.

CCS CONCEPTS
• Computing methodologies → Concurrent computing methodologies; • Information systems → Database transaction processing.

∗Work done mostly while the author was affiliated with the University of Copenhagen.

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 ACM 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.

ACM Reference Format:

1 INTRODUCTION
The actor model [1] is emerging as a promising concurrent and parallel programming abstraction for building stateful middle-tiers [10, 12] in large-scale interactive applications, including multi-player games such as Halo 4 [48] and League of Legends [42], telecommunication such as Ericsson [26], E-commerce such as Paypal [50] and Walmart [17], and Internet of Things [40]. There are plenty of programming languages [24, 25] as well as libraries and frameworks [2, 36, 37] that enable actor-based programming. With the actor model, applications are decomposed into concurrent actors, each encapsulating a private state and communicating with other actors via asynchronous message passing.

In the actor model, each actor processes its incoming messages sequentially. Such sequential behavior frees developers from handling concurrency issues within each actor. However, there are situations where concurrent cross-actor operations require transactional properties. For example, in an online multi-player game, player actors may exchange game equipment or purchase equipment with digital currencies. As another example, in an e-commerce application, actors maintaining product stocks and those responsible for order checkouts have to interact to complete a purchase transaction. Transactional properties are often needed to ensure application correctness in these scenarios.

Transaction management in actor-based applications is complicated by their design as stateful middle tiers, which react to changes of states in real-time and asynchronously flush accumulated states to database tiers [10]. In particular, with this architecture, transactions are executed within the middle-tier servers rather than as stored procedures in databases. Motivations of this trend include, among others, the flexibility of encoding transaction logic using programming abstractions different from database systems, and being able to use the large memory and computing power of cheap middle-tier servers to manipulate data and execute transactions instead of using more expensive [8, 9] database servers [10].
To meet these new requirements and to alleviate the burden on developers, there exist efforts in various actor systems providing high-level programming abstractions for efficient multi-actor transactions while hiding their implementation complexities from developers. Akka introduced the concept of transactors [4], which employs two-phase commit (2PC) and software transaction memory (STM) to support atomic cross-actor transactions.¹ Orleans [37] has recently made efforts to support distributed transactions [23] across multiple actors. It adopts two-phase locking (2PL) and 2PC with early lock release [7, 47], allowing for higher concurrency at the price of cascading aborts.

One way to enable actor transactions is to implement transactions on top of the actor abstraction itself without any modification to the actor runtime. For example, Orleans Transactions [23] adopt such an approach. This non-intrusive approach requires less system development and maintenance effort in comparison to alternatives with deep integration with the actor runtime. Therefore, we focus on this approach due to its low development cost. However, under this approach, multi-actor transactions are challenging. This is because the state of an actor-based application is partitioned into many fine-grained actor private states. Every multi-actor transaction, no matter if the actors are collocated on the same machine or not, is a cross-partition transaction and has to employ distributed transaction mechanisms, which are expensive.

In this paper, we argue that the existing actor transaction mechanisms have not sufficiently exploited the characteristics of actor-oriented applications to improve transaction performance, particularly transaction throughput. An interesting characteristic is that actors are accessed explicitly in an actor programming abstraction, e.g., by the names or process IDs in Erlang [25], by the paths of the actor hierarchy in Akka [2], and by the user-defined actor identities in Orleans [37]. It is often the case that the set of actors involved and the number of times that they would be accessed in a transaction are known before the transaction starts. For example, in an e-commerce system, a CheckoutOrder transaction explicitly specifies a list of product IDs, which targets a list of stock actors, each being accessed once. Another example is, in a social network application, when a user issues a JoinGroup transaction, which updates the membership data in a determined user actor and group actor, each being accessed once, respectively.

This characteristic of actor programming enables the exploration of a novel actor-based transaction abstraction, where the identities of the participating actors and the number of times that they are accessed in a transaction are pre-declared. With such apriori information, an actor system would be able to pre-schedule the transactions and execute them in a deterministic order. In comparison to nondeterministic concurrency control mechanisms adopted by existing approaches, such as 2PL, a deterministic ordering strategy would avoid transaction aborts due to conflicts [43, 52]. The latter holds the potential to significantly improve system throughput especially when the contention level is high. Furthermore, 2PC can be optimized to enhance transaction concurrency [43, 52].

Despite that we envision most actor transactions in an actor-based application can be implemented with the aforementioned transaction abstraction, there could exist transactions that do not fall into this category. For example, in a social network application, a user could issue a CleanupFriendList request, which removes friends who are in the user’s friend list but with no recent interactions, and would then trigger the removed friends to also update their friend lists. Such a transaction may need to look up a user’s friend list and the recent interaction histories to determine the set of actors that would be involved. In other words, the list of participating actors of the transaction may not be known before the transaction starts. Therefore, one may have to resort to conventional actor transaction abstractions based on nondeterministic concurrency control and 2PC, such as Akka transactor or Orleans Transactions, to execute this type of transaction. Supporting both types of abstractions in the same system is a challenge. Deterministic and nondeterministic concurrency control methods achieve serializability based on different principles, and how to reconcile these two methods in a single system is still an open problem.

In this paper, we propose Snapper, an actor transaction library on top of Orleans that enables multi-actor transactions. Our goal with Snapper is to improve the performance of cross-actor transactions based on the fact that existing solutions such as Orleans Transaction do not perform so well especially under high contention and on a significant observation that deterministic transaction execution is well-suited to the actor model. Specifically, Snapper supports two types of actor transaction abstractions, namely PACT (ACtor Transaction) and ACT (ACtor Transaction), which employ deterministic and nondeterministic concurrency control mechanisms, respectively. A salient feature of Snapper is that it supports a novel hybrid execution strategy that enables concurrent execution of transactions specified using different actor transaction abstractions. As the first cut at the problem, this paper focuses on optimizing and evaluating the performance of single-server transactions, i.e., transactions that only involve actors located on the same server. We focus on this problem because we envision that, in order to maximize transaction performance, the allocation of actors should be optimized so that the majority of transactions are single-server transactions as in high-performance OLTP database systems [19, 39]. Besides, optimizing single-server transactions can be of great value to many applications that are able to scale vertically and are suited to exploiting locality. In summary, the main contributions of this paper include:

- We propose a novel programming abstraction for multi-actor transactions, namely PACT, which enables concurrent execution of multi-actor transactions in an actor system.
- We propose a hybrid transaction execution method that enables concurrent execution of PACTs and ACTs. To the best of our knowledge, we are the first to study how to accommodate both deterministic and nondeterministic transaction execution strategies in a single system.
- To verify practicability, we implement the proposed actor transaction abstractions and execution strategies as a library on top of Orleans, a widely adopted actor system.
- We conduct a series of experiments to evaluate the effectiveness of our transaction execution methods using SmallBank and TPC-C benchmarks. The results show that comparing to ACTs and

¹When this feature was deprecated in 2014 [3], developers have significantly complained about its absence [20, 54].
Orleans Transactions, PACTs can achieve up to 2x higher throughput. Additionally, the hybrid execution can achieve a throughput that is close to PACTs when the percentage of ACTs is small.

2 BACKGROUND

In actor systems, actors are the basic unit of programming [1]. Similar to object-oriented programming, each actor has its own private state and methods. However, unlike objects, actors do not share state in the same logical address space; rather, they communicate with each other exclusively via asynchronous messages. Conceptually, each actor is single-threaded and processes each incoming message before handling the next one. Upon receiving a message, an actor may modify its own state, create other actors, or send messages. The actor model aims at providing a coherent and simplified abstraction to build concurrent, parallel, and distributed systems.

Conventionally, actor runtimes, such as Erlang [25] and Akka [2], provide constructs for programmers to explicitly manage the lifecycle of actors, including creation/deletion, allocation, recovery, etc. Orleans [12, 16] proposes the novel concept of virtual actor, wherein the Orleans runtime system becomes responsible for actor lifecycle management. A virtual actor is conceptually in perpetual existence. The Orleans runtime automatically activates a virtual actor when it is first invoked. Similarly, the Orleans runtime automatically deactivates the actor when it is no longer under use based on configurable policies. Virtual actors offer location transparency and a basic level of fault tolerance by automatic re-activation.

In Orleans, actors always interact through strongly-typed asynchronous messaging, which is exposed to developers as asynchronous RPCs [16]. Asynchronous RPCs allow actors to interact with each other without blocking by overlapping method invocations, which is one of the key factors that makes actor systems achieve high concurrency. Orleans also allows users to explicitly wait for the result of an asynchronous call by using the keyword `await`, in a style reminiscent of promises [30]. Besides, in Orleans, message delivery timing and order are non-deterministic, i.e., messages can arrive at a destination actor in a different order than the sending order. Therefore, implementing multi-actor transactions in Orleans is non-trivial, requiring coordination under asynchrony, dealing with out-of-order messages, and still achieving high throughput.

Orleans implements turn-based scheduling [16] on each actor, which processes a request as discrete units of work called `turns`. Turns are sequentially executed. By default, turns from different requests are not allowed to interleave on the same actor activation. However, Orleans provides a mechanism called `reentrancy` [16], where an actor can switch to process the turn of another request while one request is blocked by an asynchrony operation. Reentrancy allows multiple requests to be interleaved on the same actor, which brings even higher concurrency with asynchronous RPCs.

In Orleans, actor failures are reported as exceptions thrown from actor codes or actor runtime. Orleans propagates an exception along the call chain to the caller actor. Snapper catches and handles exceptions thrown by actor codes to correctly abort a transaction.

3 SNAPPER PROGRAMMING MODEL

3.1 Conceptual Overview

Snapper is a library that supports executing transactions involving method calls over one or more actors. It provides transactional APIs to access the state of the current actor and to invoke method calls on other actors. These APIs are implemented in `TransactionalActor` which is a base class of actor and has system functionality such as persisting logs and committing/aborting transactions built on top of it. To get transactional guarantees provided by Snapper, user-defined actors must extend `TransactionalActor` and use its APIs when accessing actor state and invoking method calls.

In Snapper, each transaction is a series of method invocations performed on multiple actors. A transaction is initiated by one actor where the first method is invoked. This actor will start executing the transaction and invoke methods on other actors via asynchronous RPCs. A transaction can invoke methods on the same actor multiple times. The actor that initiates the transaction will also end the transaction when the first invoked method has finished. This actor is responsible for committing/aborting the transaction, thus there is no need to explicitly issue such requests in application codes. Snapper guarantees conflict serializability for all concurrent transactions and provides built-in durability for `TransactionalActor`.

A transaction can run under one of the following two modes:

- **Pre-declared ACtor Transaction (PACT):** Transactions in this mode must pre-declare the following information: (1) the actor that will initiate the transaction, (2) the first method that will be invoked and the corresponding input data for this method, and (3) actorAccessInfo – the set of actors the transaction will access and the number of times each of such actors will be accessed. Based on the pre-declared actorAccessInfo, Snapper performs deterministic scheduling for PACTs and each accessed actor will execute PACTs in the pre-determined order. Besides, each `TransactionalActor` has reentrancy enabled to schedule transactional method invocations since they may not arrive in order. Under the pre-scheduling strategy, Snapper guarantees no PACTs abort due to concurrency conflicts. However, users are allowed to explicitly abort a PACT by throwing an exception to Snapper.

- **ACtor Transaction (ACT):** Transactions in this mode only need to declare information (1) and (2) mentioned above. The actors accessed by an ACT are discovered when methods are invoked during the ACT’s execution. Snapper applies a conventional non-deterministic concurrency control, e.g., SzPL, for ACTs. Unlike PACTs, ACTs can be aborted due to deadlocks or read/write conflicts.

3.2 Transactional API of Snapper

Snapper exposes three APIs, `Start Txn`, `CallActor` and `Get State`. Table 1 gives the definition of each API, Figures 1 and 2 give an example of how to use those APIs.

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT</td>
<td><code>Task&lt;object&gt; StartTxn(string startFunc, object FuncInput)</code></td>
</tr>
<tr>
<td>PACT</td>
<td><code>Task&lt;object&gt; StartTxn(string startFunc, object FuncInput, Dictionary&lt;Guid, int&gt; actorAccessInfo)</code></td>
</tr>
<tr>
<td>both</td>
<td><code>Task&lt;object&gt; CallActor(string ctx, Guid actorId, FuncCall call)</code></td>
</tr>
<tr>
<td>both</td>
<td><code>Task&lt;TState&gt; GetState(TxnsContext ctx, AccessMode mode)</code></td>
</tr>
</tbody>
</table>

Table 1: Snapper’s transactional API

3.2.1 Submitting Transactions to Snapper

Fig. 1 shows how clients submit transactions to Snapper. A client submits a transaction by calling `StartTxn` on the first actor that the transaction will access (lines 9, 15). The client can choose to submit a transaction as a `PACT` or an `ACT` by passing different data to `StartTxn`. As for the `ACT` mode, the name of the first method that will be invoked and the corresponding input data should be given by the client. As opposed to `ACT`, the `PACT` mode additionally requires `actorAccessInfo` as input. Snapper distinguishes transaction modes according to the
public class Client{
    static void main(){
        // ...
        var funcInput=new Tuple<float,long>(100, toActorID);
        var actor=client.GetGrain<IAccountActor>(fromActorID);
        try{
            // submit an ACT
            var ACT_balance=await actor.StartTxn("Transfer", funcInput);
            // ...
            ACT_balance=await actor.StartTxn("Transfer", funcInput);
            var toActorID=MapAccountIDToActorID(toAccountID);
            var toAccountID=input.Item2;
            var PACT_balance=await actor.StartTxn("Transfer", funcInput, actorAccessInfo);
            catch (Exception e){
                // ...
            }
            return myBalance;
        }
    }
}

Figure 1: Submission of PACT/ACT to Snapper

input data. At last, StartTxn will return the transaction result (e.g., the balance after doing Transfer) as an object to the client. If the transaction is aborted in Snapper, the transaction is rolled back by Snapper and an exception will be thrown to the client.

3.2.2 User-Defined Transactional Actors. Fig.2 shows how developers program user-defined actors by using Snapper’s API. AccountActor is a user-defined actor class with interface IAccountActor, same as ordinary actor definitions in Orleans. To inherit Snapper’s transactional actor features, the user-defined actor interface and actor class should derive from ITransactionalActor and TransactionalActor, respectively (lines 1-6). In addition, the type of the actor state should be explicitly declared, which can consist of primitive or user-defined types. In the example, float is the type of the AccountActor’s state, which represents the balance of the account (line 6). The interface of the user-defined actor should always contain two input parameters: TxnContext and the input data for the method involved in the transaction (e.g., Deposit or Transfer). An instance of TxnContext is an internal read-only data structure of Snapper. It contains the transaction’s context information such as tid, txnMode, etc. It is generated by Snapper after receiving the transaction request from the client and before executing the transaction. It is passed as a parameter in all three APIs, so that Snapper can schedule and execute method calls transactionally based on the context information.

As for the implementation of the user-defined actor class, the method GetState should be used to access actor state (lines 12, 18). Snapper supports two access modes: Read, which is read-only, and ReadWrite, which reads and writes the state. CallActor should be used to invoke method calls on other actors (line 25). Instead of directly calling another actor’s method in user-defined code, Snapper wraps this operation in CallActor. It is designed in this way because Snapper has to gather and propagate transaction execution information along with actor method calls, and CallActor abstracts these actions away from developers.

2.3.3 Aborting a Transaction. In Snapper, PACTs do not abort due to concurrency conflicts, but can abort due to runtime exceptions or user-defined transaction logic, e.g., a Transfer transaction might abort because of insufficient balance. By contrast, ACTs can abort due to all three reasons. Users can abort a PACT/ACT by throwing an exception to Snapper (line 19 in Fig.2). Snapper catches both internal exceptions caused by runtime issues or concurrency conflicts and external exceptions thrown by user codes. Any exceptions that are not handled by user codes will be caught by Snapper and treated by aborting and rolling back the relevant transactions. Note that submission of a PACT with user-defined aborts should not be the norm because it will lead to performance degradation (Section 4.2.3). A transaction with user-defined aborts is better submitted as an ACT. Snapper supports the aborting of PACTs mainly for the cases where unexpected runtime exceptions arise.

4 SNAPPER ARCHITECTURE

4.1 Overview

4.1.1 Components. Snapper consists of three components, including two types of actors – coordinators and transactional actors – and a group of loggers, which are in-memory C# objects shared by all actors on the machine and responsible for writing logs.

Coordinator actors are responsible for assigning a unique transaction identifier (tid) to each transaction. For PACTs, the tids should be assigned according to a global serial sequence order that determines their execution order. The sequence of PACTs is divided into batches in order to amortize the overhead of messaging and logging. Coordinator actors interact amongst themselves to reach consensus on such a global sequential order for PACTs and they also coordinate transactional actors to execute and commit batches in the pre-determined order.

Transactional actors are the base actor class provided by Snapper to program user-defined actors where applications’ transactional states are stored. Each transactional actor schedules PACTs according to the deterministic sequential order generated by the coordinator actors, and performs nondeterministic execution of ACTs. With hybrid workloads of PACTs and ACTs, transactional actors employ a novel hybrid concurrency control for them.
Figure 4: Batching

Figure 5: Content of TxnData

Figure 3: Transaction workflow

4.2.1 Ordering. To assign deterministic execution order to PACTs, one can use a single coordinator to sequentially assign a monotonically increasing tid to each PACT and use the tid to determine the order. However, using a single-threaded coordinator may not be able to scale. Instead, Snapper exploits parallelism by employing multiple coordinators and each independently receiving transaction requests. To guarantee monotonicity of tid while using multiple coordinators, we essentially need mutual exclusive access to the latest tid that has been assigned. To achieve this, Snapper adopts the classic token ring algorithm [49] for its simplicity and its natural match with the message passing abstraction of actors. More specifically, coordinators are logically placed in a ring, where each coordinator has fixed left and right neighbors. A token is circulated in a particular direction in this ring. The token carries all the information that needs to be shared among coordinators, e.g., the latest assigned transaction id (last_tid). A coordinator accumulates the PACTs that it has received while waiting for the token. When it receives the token, it allocates tids for those PACTs based on the last_tid value stored in the token. When the token carries all the information to the next coordinator immediately when the new batch is formed. Thus the token ring mechanism does not substantially increase transaction latency. Conversely, while a coordinator is waiting for the token, it can perform other tasks such as communicating with transactional actors, coordinating batch commitment, logging, etc.

4.2.2 Batching. With tid and actorAccessInfo of a PACT, the coordinator can send messages to inform each accessed actor of the existence of the PACT. However, delivering one tid per message is inefficient. Therefore, Snapper chooses to deliver information about a batch of transactions per message. This is necessary, because batches that need to be executed on an actor
may not have consecutive bids. Even if they do, the batch messages may arrive out of order due to nondeterministic message delays. With the prev_bid, an actor can know the order between batches, thus it can start executing a batch when its previous batch has completed. In Snapper, the prev_bid for each actor is stored in the token, updated by the coordinator when a new batch is created and removed if the corresponding batch has committed. After the updates of last_tid and prev_bids, the coordinator can pass on the token and emit BatchMsgs to related actors.

With batching, the overhead of sending messages is amortized over multiple PACTs in a sub-batch. The efficiency of batching can increase with skew in the workload, because more PACTs will be included in one sub-batch. To reduce overhead and improve transaction throughput, Snapper schedules, executes, and commits PACTs at the batch granularity. This accrues benefits on both the messaging ((4), (6), (7) in Fig.3a) and logging.

4.2.3 Deterministic Scheduling. Each actor maintains a local schedule to control the transaction execution order. Such a schedule is needed for two reasons. First, we cannot rely on the message arriving order to order the transactions. Second, each actor should have its own schedule because the sets of transactions to be executed on actors are different from each other. In the schedule of an actor, batches are placed in a chain according to the prev_bid relation. An actor gradually extends the schedule upon receiving batch messages and removes the batch when it is committed/aborted. If a batch arrives at an actor earlier than its previous batch, this batch creates a vacancy in the chain which is filled when the previous one arrives. For example, in Fig.4b, on A2, B2 is currently maintained separately because its previous batch B2 has not arrived yet. A batch message contains bid, prev_bid and a list of PACTs, including their tids and the number of accesses on the actor.

When any method invocation of a PACT (called through the CallActor API) arrives on an actor, the actor will execute it according to the local schedule. More specifically, the actor first checks the carried TxnContext of the received call. If the call comes from a PACT whose turn is yet to come according to the local schedule, the actor will suspend the execution of this method invocation by awaiting an asynchronous task which is later resolved when the scheduled previous PACT has completed. A PACT is considered completed when an actor has accessed the declared number of times. Then the actor can resume the execution of the suspended method invocation. Notice that when an execution is blocked by an asynchronous operation, the actor is free to process another request because Snapper has enabled reentrancy (Section 2) for all TransactionalActors. In Orleans, a reentrant actor is allowed to interleaver the execution of multiple requests. As for the caller actor, the invoked call is essentially an asynchronous RPC. Once the callee actor enqueues the call into its message box, a future is returned, and the corresponding promise can be fulfilled later.

Since PACTs are executed under a deterministic schedule and are guaranteed not to abort due to conflicts, a sub-batch on an actor can be speculatively executed as long as its previous sub-batches have completed their operations on this actor without waiting for them to commit. This allows for pipelined execution of batches while respecting the schedules on individual actors without waiting for coordination of batch commitment across different actors. This also brings higher concurrency comparing to conventional nondeterministic concurrency control such as S2PL where locks are only released when a transaction commits. Besides, S2PL usually introduces non-deterministic blocking due to conflicts, while PACTs can avoid this effect because pre-scheduling is applied.

However, if a PACT is aborted, the whole batch would be rolled back along with all batches that have been speculatively executed. So submitting a PACT that will eventually abort can cause performance degradation. Thus, a transaction with user-defined aborts is better submitted as an ACT.

4.2.4 Commit and Logging Protocol. Snapper applies a specialized two-phase commit protocol for PACTs, which logically includes three-round one-way messages, the BatchMsg, BatchComplete and BatchCommit messages ((4), (6) and (7) in Fig.3a). BatchMsgs are the sub-batches sent from a coordinator to participating actors, which can be analogized to the prepare message in 2PC. When an actor finishes executing the sub-batch, it acknowledges BatchComplete to the coordinator who emitted the corresponding batch, which is similar to “voting” in 2PC. When the coordinator receives BatchComplete from all participating actors and if all vote to commit, the coordinator can commit the batch and send the confirmation message BatchCommit back to the actors, which will return the final transaction results to clients for the PACTs within the batch ((8) in Fig.3a). The commitment of a batch must be done by coordinators because they are the only ones who know the list of participating actors of a batch. Besides, the commit protocol should guarantee that a batch B commits after all the batches it depends on – batches that have been scheduled before B on the actors that B accessed – have committed. To eliminate the overhead of maintaining the complex dependency graph between batches, Snapper instead tracks the logical dependency in which a batch Bj always logically depends on Bj if i > j. Further, Snapper forces all batches to commit in the order of bid. This strategy works well especially under a highly contended workload where logical dependencies reflect actual dependencies. The overhead of tracking logical dependencies between batches is negligible. Each coordinator only needs to keep track of the last assigned batch ID for the batches it generates. This batch ID is then passed along in the token.

Now we explain the process of aborting a batch. An aborted batch is detected by the actor who catches the exception thrown by a PACT. Recall that an aborted batch may cause cascading aborts of speculatively executed batches. To avoid unbounded numbers of batches being aborted in this process, Snapper stops emitting new batches whenever an aborted batch is detected and resumes when the cascading abort has completed. The classic cascading abort will abort transactions that depend on the aborted ones [15]. Again, instead of maintaining the accurate dependencies between batches, Snapper simply aborts all uncommitted batches in the system.

To ensure the durability of committed PACTs and ensure the commit process survives failures, Snapper utilizes a Write Ahead Log (WAL). When a PACT commits, it logs the commit information to the WAL. This ensures that the commit process is durable even if the server crashes. Later, when the server recovers, it can replay the committed PACTs from the WAL to ensure the consistency of the committed transactions.
Log (WAL) to store related data prior to sending out any messages such as BatchMsg, BatchComplete, and BatchCommit. Fig. 6 shows the logs written for a batch that accessed two actors. Three types of log records should be written for a batch. (1) Before emitting a batch, the coordinator persists the participating actors of the batch. (2) Before sending BatchComplete, an actor logs the updated actor state. If the batch has only read the actor, there is no need to persist the actor state. (3) Before sending BatchCommit, the coordinator logs the committed bid.

Based on the logged information, Snapper is tolerant to failures that happen to both coordinators and actors at any time while executing, committing, or aborting a batch. In the batch commit protocol, the coordinator cannot decide to commit a batch until all participating actors have voted. If an actor fails before sending the BatchComplete message, the coordinator must wait until the failed actor is recovered and the message is sent. The recovered actor will retrieve its log records. If the BatchComplete record is not found, it will tell the coordinator to abort the batch. If a coordinator fails before sending the BatchCommit message, all related actors must wait to return results to clients until the coordinator is recovered and the message is sent. Those actors can autonomously ask the coordinator about the decision to commit or abort the batch. Snapper follows the principle that the batch that has BatchComplete log records written in all participating actors can commit.

4.2.5 Recovery. Assume that the system can fail (crash) at any time and some or all actors will lose their in-memory data. Snapper relies on the failure recovery mechanism provided by Orleans that a failed actor is automatically re-instantiated when it is called again. In Snapper, failed actors are re-instantiated by loading the state of the last committed batch. A failed coordinator is re-instantiated by loading the information of emitted but uncommitted batches, which needs to be used to continue the batch commit/abort process. Besides, the token may also be lost with the failed coordinator. To make sure the system has exactly one token, a recovered coordinator must trigger a consensus protocol among all other coordinators to check if the token is lost or not and elect a coordinator to re-initiate a new token if needed. If a new token needs to be used, the system must wait to emit new batches until all existing batches have committed/aborted because the prev_bid stored in the old token are lost. When all emitted batches have committed/aborted, all actors also have their local schedules empty thus the old prev_bid is not needed.

4.3 ACT Processing

4.3.1 Transaction ID Assignment. Unlike PACTs, ACTs need to be assigned unique transaction IDs (tids), but not a deterministic execution order. To achieve this in Snapper, every time the token is received by a coordinator, it will pre-allocate a range of contiguous tids for ACTs that may arrive in the future. Those TIDs will get tids assigned immediately without having to wait for the token.

4.3.2 Nondeterministic Concurrency Control. When a method invocation of an ACT arrives on an actor, the invocation is controlled by a traditional nondeterministic concurrency control protocol. Currently, we have implemented S2PL in Snapper. Multiple ACTs can invoke method calls on an actor concurrently. The S2PL protocol is executed when an actor accesses the state using the GetState API, which grants logical read/write locks to ACTs and releases them after the second phase of 2PC. Besides, wait-die [44] is used to proactively avoid deadlocks by aborting transactions if they are suspected to be involved in a deadlock.

[Diagram: Figure 7: ACT Logging]

4.3.3 Commit and Logging Protocol. ACTs are committed via 2PC [29] and presumed abort [34] is used to save messages and logging. When an ACT completes its operations, all the actors that have been accessed within the transaction context are known. The list of participating actors of an ACT is propagated as part of the first phase of the 2PC protocol along the method call chain back to the first actor who initiates the ACT. This information is utilized to perform the 2PC protocol with all the participating actors. The actor where the ACT is initiated is designated as the coordinator of the 2PC protocol. While doing 2PC, logs are persisted before sending any messages. Fig. 7 shows logs written for an ACT that spans across actors. Again, if an actor involved in the ACT did not perform any writes, there is no need to persist the state of that actor.

4.3.4 Recovery. Upon failure, every actor finds the latest committed transaction and reloads the state of the latest committed ACT on all participating actors. And every actor reads the CoordCommit record to resume the 2PC process. Incomplete transactions will be aborted by the 2PC protocol.

4.4 Hybrid Processing

4.4.1 Hybrid Scheduling. With hybrid workloads of PACTs and ACTs, each actor's local schedule contains PACT batches in a sorted order by bid and tid, and ACTs that are dynamically inserted between two adjacent batches. When receiving an ACT method invocation, the actor always appends the ACT to the tail of the current schedule. Fig. 8 gives an example. ACT \( T_0 \) is appended after \( B_0 \) on \( A_1 \) and after \( B_2 \) on \( A_3 \). The existence of ACTs does not affect the batch order, e.g., on \( A_3 \), \( B_0 \) is still placed after its previous batch \( B_2 \) even though \( T_0 \) and \( T_3 \) are scheduled in-between them.

With a hybrid schedule, actors need to carefully switch the execution between PACTs and ACTs. Snapper abides by the following rules: (1) an ACT can start executing when the previous batch has completed its operations, but not necessarily committed; (2) a batch can start executing when all previous ACTs have committed or aborted. By doing so, ACTs will not see the results of a half-done PACT and a PACT will not operate on data that will be aborted by an ACT. Thus, PACTs will still not abort due to concurrency control. Besides, multiple ACTs can be concurrently executed if they are placed between the same two batches on an actor. For example, in Fig. 8, on \( A_3 \), \( T_0 \) and \( T_3 \) are unblocked at the same time when \( B_2 \) completes.
Despite the fact that these two rules nicely isolate the executions of PACTs and ACTs on each individual actor, they are unfortunately insufficient to guarantee deadlock-freedom or serializability of hybrid workloads across multiple actors, which we address next.

4.4.2 Deadlock. Under hybrid execution, deadlock can happen between PACTs and ACTs due to nondeterministic scheduling of ACTs and blocking method invocations. The following two cases illustrate how such deadlocks occur: (a) an ACT $T_i$ is scheduled before a batch $B_j$ on one actor $A_1$, but scheduled after $B_j$ on another actor $A_2$, and at the same time, a PACT of $B_j$ on $A_2$ has to wait for $A_1$ to invoke the→ method of $A_1$ called (Fig.9a). (b) An ACT $T_1$ is waiting for $T_j$ to release the lock on an actor $A_3$, and $T_1$ is scheduled before a batch $B_k$ on $A_2$, $T_j$ is scheduled after a batch $B_k$ on $A_4$, and at the same time, a PACT of $B_k$ on $A_4$ has to wait for $A_1$ to invoke the method call (Fig.9b). In both cases, the global waits-for graph is cyclic. In addition to the patterns shown, a deadlock can easily involve more actors and transactions. To solve such deadlocks, we need to abort one of the transactions in the cycle. Since PACTs require more information from clients and are deterministically scheduled, Snapper always prioritises PACTs and aborts ACTs in the case of deadlocks. In our current implementation, a simple timeout mechanism is applied to detect deadlock [15].

Figure 9: Deadlock under hybrid execution

4.4.3 Serializability Check. To enforce serializability, Snapper employs deterministic transaction execution for PACTs and nondeterministic concurrency control for ACTs. However, under hybrid execution, the nondeterministic interleaving between batches and ACTs makes it challenging to achieve global serializability. Fig.10 illustrates two scenarios where the global serialization graph is cyclic: (a) An ACT is scheduled before and after the same batch on two different actors respectively. (b) Each single ACT does not manifest cyclic dependencies between any other batches, but the dependencies between ACTs make the global serialization graph cyclic. Our deadlock mechanism already aborts some ACTs that break global serializability. However, there exist cases that do not form deadlocks, but still break serializability. Such cases can happen when the dependency between two actors — any cross-actor edge, e.g., in Fig.9 — is in the opposite direction.

Figure 10: Cyclic serialization graph

Similar to the handling of deadlocks, Snapper enforces global serializability by choosing to abort ACTs that cause the problem. Snapper applies a serializability check for the ACTs that have finished execution. ACTs that fail to pass the check should thus be aborted. We rely on scheduling information defined as follows.

Definition 4.1. Given a history $H$ generated by Snapper’s hybrid processing and the corresponding serialization graph $SG(H)$, a PACT $T_i$ in $SG(H)$, its BeforeSet ($B_{S,T_i}$) and AfterSet ($A_{S,T_i}$) are defined as:

1. $B_{S,T_i} = \{B.bid| \text{there exists a path } B \rightarrow \ldots \rightarrow T_i\}$
2. $A_{S,T_i} = \{B.bid|T_i \rightarrow B\}$

In addition, max($B_{S,T_i}$) and min($A_{S,T_i}$) are the maximum and minimum numbers (bids) in $B_{S,T_i}$ and $A_{S,T_i}$, respectively.

Above, we borrow the concepts of history and serialization graph from [14]. B denotes a PACT batch that can be considered as one large transaction, while $\rightarrow$ denotes a precedence relation between two conflicting transactions.

Furthermore, we propose the following theorem as the theoretical basis of the serializability check. The detailed formalization and proof of the theorem can be found in the extended version [6].

Theorem 4.2. A history $H$ generated by Snapper’s hybrid processing is conflict serializable if:

1. $\forall B_i \rightarrow B_j, i < j$ (i, j are batch IDs);
2. the execution of all ACTs is conflict serializable;
3. $\forall$ ACT $T_i \in SG(H)$, max($B_{S,T_i}$) < min($A_{S,T_i}$).

Conditions (1) and (2) of the theorem are enforced by Snapper’s PACT and ACT concurrency control protocols, respectively, which provide serializability for either purely deterministic or purely nondeterministic processing. Condition (3) of Theorem 4.2 is the key point for enabling serializability of hybrid schedules. For each ACT $T_i$, Snapper must check if max($B_{S,T_i}$) < min($A_{S,T_i}$) holds. If not, $T_i$ should abort. To calculate max($B_{S,T_i}$), we have to consider batches that have a path to $T_i$ in the serialization graph. On each actor that is accessed by $T_i$, we consider the bid of the batch that is before $T_i$ and closest to $T_i$ in the actor’s local schedule. This batch is guaranteed to have the maximum bid among all the batches that are in the local schedule and belong to $B_{S,T_i}$.

However, considering only the actors accessed by $T_i$ is not enough. There may exist batches that belong to $B_{S,T_i}$ but do not access any actor accessed by $T_i$. For example, if $B_k \rightarrow T_j$ on actor $A_1$ and $T_j \rightarrow T_i$ on $A_2$, then $B_k$ should also be included in $B_{S,T_i}$. To take such batches into account, on an actor accessed by $T_i$, we also consider max($B_{S,T_i}$) in the calculation of max($B_{S,T_i}$) if $T_j$ to $T_i$ is true in the actor’s local schedule. Besides, as for the cases that a transaction $T_p$ transitivity precedes $T_i$, e.g., $T_p \rightarrow T_j \rightarrow T_i$, we do not need to consider max($B_{S,T_i}$) directly in the calculation of max($B_{S,T_i}$), because max($B_{S,T_i}$) ≤ max($B_{S,T_j}$).

The intermediate results of max($B_{S,T_i}$) and min($A_{S,T_i}$) collected on each participating actor after executing $T_i$ are propagated as part of txnExeInfo (see Fig.3c) all the way to $T_i$’s local coordinator, which calculates the final values of max($B_{S,T_i}$) and min($A_{S,T_i}$) and performs the serializability check. If $T_i$ passes the check, max($B_{S,T_i}$) should be propagated together with the Commit message to all of $T_i$’s participating actors when $T_i$ commits. This value may be useful for the serializability check for the subsequent ACTs.
Note that this implementation does not guarantee that we can obtain the complete AS_T. When T_i finishes execution on an actor, there may not be any batch B such that T_i \rightarrow B. Due to asynchrony in actor systems, a batch can take an arbitrarily long time to reach an actor. It is also possible that there is no batch that is scheduled after T_i on the actor. An incomplete AS_T could result in an incorrect \min(\text{AS_T}) and a wrong decision in the serializability check.

A possible solution is that T_i’s local coordinator obtains the complete AS_T from the PACT coordinators. To achieve this, it has to contact all the PACT coordinators and obtain the schedules of all the actors involved in T_i. This is costly and would increase the commit latency of T_i significantly.

For efficiency and simplicity, Snapper only performs the serializability check based on the information available in the local coordinator. It fails an ACT T_i’s serializability check if AS_T is incomplete. AS_T is said to be incomplete if there exists an actor A involved in T_i such that no T_i \rightarrow A can be found, where A is a PACT batch. However, this approach may cause unnecessary aborts. To mitigate the problem to a certain degree, Snapper adopts an optimization in the cases where AS_T is incomplete: if BS_T is empty or all the PACT batches in BS_T have already committed, T_i can pass the serializability check. This optimization is based on the fact that all the batches in AS_T have not yet started their execution, because they must wait for T_i to commit or abort. Since PACT batches are executed in bid order, there will not exist bid \in AS_T such that bid \leq \max(BS_T).

4.4.4 Commit Protocol. Under hybrid execution, PACTs and ACTs can interleave and depend on each other. The commit protocol guarantees that a transaction commits before the transactions that depend on it. For PACTs, a batch starts executing after previous ACTs have committed or aborted, so PACTs can commit the same coordinator. It fails an ACT complete.

5 EVALUATION

In this section, we evaluate the performance of PACT, ACT and hybrid execution of Snapper. Specifically, we investigate the characteristics of PACT and ACT under different transaction sizes (Section 5.2.1) and workload skewness (Section 5.2.2). In Section 5.3, we present the performance of hybrid execution under different workload skewness and distribution of hybrid workload, and study the trade-offs of hybrid execution with regards to transaction throughput, latency, and abort rate. In Section 5.4, we study how well each concurrency control method in Snapper can scale.

5.1 Experimental Settings

5.1.1 Benchmarks. We use two benchmarks throughout the experiments: SmallBank [5] and TPC-C [53]. SmallBank is an OLTP benchmark simulating basic operations on bank accounts [5]. Each user account is implemented as an actor in the SmallBank benchmark. We employ SmallBank as it is a synthetic workload that approximates well a realistic actor-oriented workload, which is usually reactive, write-intensive, and latency-sensitive. To simulate multi-actor workloads, we implement a MultiTransfer transaction that withdraws money from one account and deposits money to multiple other accounts in parallel [45]. SmallBank is used in most of our experiments because it is easy to configure and has predictable behavior. A similar choice for a synthetic workload that can be run under different access distributions was also used to evaluate other actor-based transactional implementations [23, 45]. TPC-C is an industrial-standard OLTP benchmark. Similar to previous work [52], we only use the NewOrder transaction of TPC-C in our evaluation, as the NewOrder transaction accesses products stored in different warehouses and thus is naturally distributed. In our experiments, we can flexibly control the distribution and the size of NewOrder transaction by modeling a warehouse as an actor and partitioning the stock table into multiple actors [52].

5.1.2 Deployment. We run Snapper on Orleans 3.4.3. We deploy Orleans clients and server (the latter called silo in Orleans [37]) on two AWS EC2 instances (c5n), respectively. Each instance has a 4-core 3.0GHz processor, 10.5GB memory and the silo instance is attached with a 16GB io2 SSD volume with 8K IOPS. All the instances are located in the same region and availability zone. In the scalability experiment, as is shown in Fig.11a, all the computing resources scale proportionally with the 4-core setting as a base unit.

On the client side, we implement a push-pull queue, where a producer thread keeps generating transactions and pushing transaction requests to the queue, and multiple client threads pull from the queue concurrently. Each client thread simulates an Orleans client. Instead of spawning a large number of threads per silo with each sending one request at a time, a single client thread is used to asynchronously emit a pipeline of transactions. Whenever a transaction result returns, the client pulls a new transaction from the queue and issues it to replenish the pipeline. The number of client threads and their pipeline size decide the maximum number of concurrent transactions running in the system. Fig.11b presents the pipeline size we set for different workloads and concurrency control methods. The pipeline size is tuned such that PACT/ACT can reach a good performance while the system is not over-saturated.
is emitted by the client thread to the time that the client receives the transaction result. Note that we only report the processing latency, but not the queuing latency, i.e., the time that a transaction is buffered in the push-pull queue. The latter heavily depends on the input rate, and it increases exponentially when the input rate is getting closer to the system throughput.

Besides comparing the different execution strategies provided by Snapper, we also compare Snapper with non-transactional execution (NT) on Orleans and Orleans Transaction (OrleansTxn) shipped with Orleans 3.4.3.

5.2 PACT vs. ACT Execution

To examine the performance of PACT and ACT with different degrees of contention, we compare PACT and ACT under various transaction sizes and workload skewness. This group of experiments are run with MultiTransfer transactions on a 4-core silo with 10K transactional actors.

5.2.1 Effect of Transaction Size. We define transaction size (\(txnSize\)) as the number of actors accessed by a transaction, which reflects the transaction complexity. In this experiment, we vary \(txnSize\) to measure the overhead of the transactional support provided by Snapper. Transaction overhead is measured as the relative throughput of PACT and ACT with regards to the throughput of a non-transactional (NT) implementation. As NT only processes actor calls without any logic of concurrency control, its throughput comprises an upper bound for executing transactions on Orleans. In this section, we set the workload skewness to be uniform and fix the pipeline size to 64.

Fig.12 shows that, compared to NT, when \(txnSize = 2, 4, 8,\) concurrency control (CC) brings more throughput degradation for PACT than ACT. It is because PACT costs more messages per transaction under low contention. In this case, each BatchMsg can only deliver one transaction to an actor. For example, when \(txnSize = 2\), each PACT costs 6 one-way messages (2 BatchMsg + 2 BatchCommit + 2 BatchCommit) while each ACT only costs 2 double-way messages (Prepare + Commit) to commit a transaction (Fig.3). When \(txnSize\) increases, the throughput of ACT decreases much faster than PACT because ACT suffers a lot from workload contention. When \(txnSize\) grows, more conflicts arise, and thus more transactions will be blocked during execution and possibly aborted to avoid deadlock. As is shown in Fig.12, the abort rate of ACT reaches 90% when \(txnSize = 64\). By contrast, PACT guarantees no transaction abort due to conflicts by pre-scheduling and PACT benefits more and more from batching because it can amortize the messaging overhead (Section 4.2.2).

As for the overhead of logging, the throughput of PACT (CC + Logging) and ACT (CC + Logging) are 70% and 50% compared with the case without logging, respectively. PACT always has lower logging overhead than ACT because PACT writes less to the log than ACT. PACT can amortize the logging overhead by batching even under low contention because the coordinator always writes BatchInfo and BatchCommit log records for a batch of PACTs no matter which actors they access. When the contention level grows, the BatchComplete log record benefits more and more from batching (Fig.6). Instead, an ACT always writes two times to logs on the ACT coordinator (CoordPrepare and CoordCommit) and two times to logs (Prepare and Commit) per accessed actor (Fig.7). With the combined effects of CC and logging, PACT outperforms ACT under all contention levels.

Fig.13 shows the difference between PACT and ACT in terms of transaction latency when both CC and logging are enabled. When \(txnSize = 64\), PACT has almost the same median latency as ACT. When \(txnSize = 64\), however, PACT exhibits higher median latency than ACT, namely 189 vs. 125 milliseconds. The latter occurs because all PACTs are delayed to be executed and committed in batches. When \(txnSize < 64\), this impact is not very evident because PACT does logging in a more efficient way. When \(txnSize = 64\), the enforced batching dominates the influence on latency. By contrast, ACT always has much higher 90th- and 99th-percentile latencies than PACT. When \(txnSize = 64\), ACT gets almost 2x higher 99th-percentile latency than PACT. This effect emerges as ACTs that experience high contention would be blocked for a significantly long time. PACT has its tail latency lying in a moderate range (around 1.3x of 90th-percentile latency), because every actor follows a deterministic schedule without non-deterministic blocking.

In conclusion, ACT introduces more overhead than PACT because ACT suffers from high contention and its logging protocol is less efficient. In contrast, PACT reaches good throughput and predictable transaction latency under different \(txnSize\).

5.2.2 Effect of Workload Skewness. Workload skewness defines the asymmetry in the chance that each actor is accessed by a transaction. In a highly skewed workload, transactions access only a small set of actors, which causes high contention on them. We use a zipfian function implemented in the MathNet.Numerics.Distributions package [33] to generate different skewed workloads by varying the zipfian constant. Fig.11b gives the zipfian value of five skew levels we used in the experiments.

In this section, we compare the throughput of PACT and ACT under different workload skewness. We fix \(txnSize\) to 4 and enable both CC and logging. We also run the same experiment using
0rleansTxn. Both ACT and 0rleansTxn have S2PL as concurrency control method and 2PC as commit protocol [10]. The main protocol differences between them are that ACT does not perform Early Lock Release [7, 23, 47] and ACT uses wait-die to avoid deadlocks, while 0rleansTxn uses a timeout mechanism. We set the pipeline size for 0rleansTxn the same as for ACT and implement its transactional storage provider [38] by forwarding logging requests to the same number of loggers as ACT does.

Fig.14 shows that the throughput of both ACT and 0rleansTxn decreases with increasing skewness. Both ACT and 0rleansTxn suffer from high contention. We also ran 0rleansTxn with a deadlock-free workload, which is generated by accessing actors in the order of actor ID. Without deadlocks, 0rleansTxn gets 0% abort rate and relatively higher throughput compared to the one with possible deadlocks. Either with or without deadlocks, 0rleansTxn gets lower throughput than ACT.

By contrast, the throughput of PACT increases under higher skewness. PACT benefits from high skewness because batching becomes more efficient. As discussed in Section 4.2.2, one message can deliver more transactions for processing under a skewed workload, and one log record can cover committed data of more transactions.

In conclusion, ACT is more sensitive to contention, while PACT can benefit from it by batching. Changing from ACT to PACT can thus bring about performance improvements in this scenario.

5.2.3 Microbenchmarking ACT and Orleans Transaction. In Fig.14, we observe a surprisingly significant performance gap between ACT and 0rleansTxn even on the cases without deadlocks. This effect is unexpected because both of them adopt very similar mechanisms in concurrency control (2PL) and transaction commit (2PC). In this section, we microbenchmark both systems to investigate the causes of this performance gap. As 0rleansTxn adopts early lock release, which may suffer from high abort rate when the workload has high contention. We also ran ACT with a pipeline size set as 1 so that the workload has no conflicts. 0rleansTxn instead of calls with the pipeline size is set as 0 so that the workload has no conflicts.

For a varying number of actors to perform no-op grain calls in each transaction, Fig.14 shows that the throughput of both ACT and 0rleansTxn is almost the same total latency. For 0W + 4N, _t_ is the time for serially performing 3 actor calls. 0rleansTxn takes 1.6x more time on _t_ than ACT (0.32ms vs. 0.2ms). This difference indicates that actor calls under a transaction context are more expensive for 0rleansTxn.

For 1W + 3N, one RW operation is performed on the first accessed actor and the actor needs to carry out one-phase commit. 0rleansTxn takes substantially more time on _t_ than ACT (0.2ms vs. 0.01ms). This effect occurs because 0rleansTxn incurs on one Prepare message from the TA to the first accessed actor to start the commit process, while in this case ACT requires no messages for 2PC since the first accessed actor is designated as the coordinator of the 2PC protocol. Furthermore, 0rleansTxn spends significantly more time on performing 2PC than ACT does. The gap increases as more actors are involved in the commit.

ACT and 0rleansTxn have distinct codebases and they adopt disparate software stacks. So despite similar algorithms being used in both systems, we observe that dissimilarities in performance are spread over many operations and components. Thus, we ascribe their performance gap to their differences in implementations. A more detailed analysis and comparison of implementation details, e.g., data structure overheads, between OrleansTxn and ACT exceed the scope of our work.

5.3 Performance of Hybrid Execution

In this section, we investigate the performance of hybrid execution under different transaction distributions, namely the percentage of PACTs among all transactions (PACT%) and different workload skewness. We use the SmallBank benchmark for this group of experiments. We fix txnszize as 4 and enable both CC and logging. On the client side, we spawn two client threads to handle PACT and ACT requests, respectively. The settings of pipeline size are shown in Fig.11b. To vary the PACT%, we let the producer randomly generate PACT% PACTs among all transactions.

5.3.1 Throughput. Fig.16a shows the throughput of hybrid execution. Under each level of skewness, we vary PACT%. In each bar, different colors represent the part of the throughput contributed by PACT or ACT. We observe that the total throughput decreases with decreasing PACT%. Ideally, total tp = PACT% × PACT tp + ACT% × ACT tp, but the actual throughput is lower. This effect arises because: (1) the scheduling of PACTs and ACTs influences each other, i.e., PACTs force ACTs to wait for batch processing, and PACTs are blocked until the previous ACTs are committed or aborted; (2) ACTs will also be aborted due to conflicts with PACTs, in addition to conflicts between ACTs themselves.

This mutually and transitively blocking behavior between PACTs and ACTs is even more severe under high skew levels, where most transactions access the same one or two actors. That is why there is a notable throughput degradation in this case from 100% to 99% PACT and from 0% to 25% PACT under high and very high skew.
levels. So with an extremely skewed workload, we can benefit from hybrid execution only if we have a small percentage of ACTs.

In conclusion, hybrid execution can bridge the performance gap between pure PACT and pure ACT in most scenarios. Under higher skew levels, hybrid execution performs better than pure ACT if the $P\%$ is high.

5.3.2 Latency. Fig. 16b shows the latency of PACT and ACT under hybrid execution. Similarly to Fig. 13, PACT has higher 50th-percentile latency than ACT because of batching. Under hybrid execution, PACT’s 90th-percentile latency is influenced by ACT.

When the workload skewness is fixed, for both PACT and ACT, both the 50th- and 90th-percentile latencies increase first and then decrease when $P\%$ decreases.

As for PACT, when adding some ACTs to a pure PACT workload, PACTs scheduled after ACTs are blocked until the ACTs finish 2PC. When more ACTs are added, PACT latencies start to decrease. The latter is because there are less and smaller batches, which indicates a lower possibility that a batch be influenced by transitive blocking. Besides, PACT latency starts to decrease at higher $P\%$ under higher skewness. This effect arises because more ACTs are quickly aborted due to high contention.

As for ACT, when adding a few PACTs to a pure ACT workload, ACTs have their latency decreased due to the blocking caused by PACTs. Then, ACT latency decreases when adding more PACTs. The latter occurs because many long-latency ACTs were actually aborted due to deadlocks between PACTs and ACTs as well as ACTs failing the serializability checks. Recall that the aborted ACTs are not counted in latency statistics.

5.3.3 Abort Rate. In hybrid execution, an ACT can be aborted in multiple scenarios: (1) aborted due to read/write conflicts between ACTs; (2) aborted due to deadlocks between PACTs and ACTs; (3) aborted to guarantee global serializability even though we are not sure whether the ACT breaks global serializability because the ACT has an incomplete $AfterSet$; (4) aborted because the ACT surely breaks the global serializability. Fig. 16c shows the breakdown of transaction abort rate. Most of the aborts are from (1) and (3). Under higher skewness, more ACTs are aborted due to (2). When adding a few PACTs to a pure ACT workload, (3) emerges and causes the total abort rate to become higher than for a pure ACT workload.

5.4 Scalability

In this experiment, we evaluate the scalability of Snapper with both SmallBank and TPC-C benchmarks by increasing the number of cores in the silo from 4 to 32.

5.4.1 SmallBank. We set up the silo as shown in Fig. 11a. We fix $txnsize$ as 4 and enable both CC and logging. We present results under uniform and skewed workloads. By following the experiments in [52], the skewed workload is generated by a hotspot method that has 1% of the actors in the hot set and each transaction accesses three such actors in the hot set. For the skewed workload, we set pipeline size as 64 for PACT and 4 for ACT.

When adding a few PACTs to a pure ACT workload, (3) emerges and causes the total abort rate to become higher than for a pure ACT workload.

5.4.2 TPC-C. In this experiment, we model each warehouse as an actor. Within one warehouse, different actors are used to store different tables and the tables are partitioned as shown in Fig. 18. In our implementation, every NewOrder transaction accesses on average 15 actors, three of which are read-only, allowing us to control the footprint of state updates and to spread transaction processing across multiple actors. We deploy two warehouses for a 4-core silo and the number of warehouses scales with the number of CPUs.
We run the experiment with workloads under two skew levels by varying the number of partitions of the Order table. Fig.17b shows that both PACT and ACT can scale nearly linearly under low skew. Similarly to the result of SmallBank, PACT performs better than ACT under high skew.

Compared to NT, both PACT and ACT introduce around 90% throughput degradation, which is comparable with the 85% degradation observed for MultiTransfer with \( t_{\text{size}} = 16 \) (Fig.12). The degradation is mainly due to inefficient logging. In our implementation, all the actors always log the whole actor state instead of doing incremental logging. The latter is due to the fact that we have not implemented a data model for actor states in Snapper, which then treats each actor's state as a value blob. It is inefficient to log a whole table when it is insertion-only such as the Order, NewOrder and OrderLine tables. In an avenue for future work, data models can be implemented in Snapper to enhance logging performance.

6 RELATED WORK

Actor-Oriented Databases (AODBs). The concept of AODBs is to enrich actor systems with database abstractions in a pluggable fashion [15]. In recent years, several studies have contributed to enriching the features of AODBs, including indexing [13] and geodistribution [11] of actor states, and distributed transactions across actors [23]. Snapper also contributes to this direction by introducing novel transaction execution techniques to actor systems.

Deterministic Database Management Systems (DDBMS). DDBMS like Calvin [52] also apply deterministic pre-scheduling to execute transactions in a pre-determined order. Deterministic database systems primarily focus on ordered state machine replication, such that given the same sequence of transactions, replicas would end up in a consistent state [43]. Therefore, DDBMS assumes transactions being deterministic, i.e., generating the same results when executed multiple times. To process non-deterministic transactions, the system has to employ a pre-processing layer to analyze the procedure calls and substitute any non-deterministic codes with deterministic ones. By contrast, Snapper does not require the computation logic of PACTs to be deterministic, but only requires the actors that are accessed by PACTs and the number of times they are accessed to be declared upon invocation. Snapper leverages deterministic execution in order to gain performance. Besides, DDBMS usually handle failures by replaying transactions. Snapper does not follow this design. With hybrid execution, recovering by replaying may not be more efficient than loading the logged states because a PACT batch may depend on ACTs. Moreover, Snapper supports transactions implemented using the actor model, which is significantly different from the programming model of stored procedures in deterministic database systems.

In addition to the above, Snapper proposes the hybrid execution strategy to concurrently execute transactions with and without pre-declared information, thus providing developers with flexibility to execute transactions in two different modes instead of forcing the deterministic paradigm. Some DDBMS support transactions whose read/write set is unknown by inferring the read/write set through a read-only reconnaissance query [51] or an offline symbolic execution [28]. Other recent work [22, 31] applied deterministic optimistic concurrency control (DOCC), which does not require a known set of data items in the execution phase and performs a validation phase in a deterministic order. All of these existing methods only consider executing transactions deterministically.

Transaction Dependency Analysis. Transaction dependency analysis has been exploited in many studies with an aim to achieve higher throughput [27, 35, 41, 46, 55, 56] and lower latency [57]. Existing approaches usually decompose a transaction into pieces according to different rules, such as SC-cycle [46], and analyze dependencies between transaction pieces. With the dependency graph, a schedule can be generated, where independent pieces of transactions are executed in parallel and conflicting operations across transactions are serialized. By contrast, a transaction in Snapper is already naturally decomposed by developers into pieces, one per actor. Snapper analyzes transaction dependencies at actor granularity and ensures that every actor executes transactions by following the same global order. Some approaches make assumptions about the execution order of transaction pieces [46, 57]; Snapper, however, does not constrain how and how many times each actor is accessed by a transaction. Some approaches may still have transactions abort [46, 57], while Snapper guarantees PACTs do not abort due to concurrency conflicts. Some approaches combine transaction decomposition with batching and resolve dependencies between a batch of transactions [41, 56] to facilitate dynamic data partitioning at the batch level. Differently, Snapper applies batching to amortize the overhead of messaging and logging.

7 CONCLUSION

This paper presents Snapper, which is a transaction library for actor systems providing two actor transaction abstractions, namely PACT and ACT. Transactions using ACT are executed using conventional nondeterministic strategies, while those using PACT can be executed deterministically and can achieve a significantly higher transaction throughput than ACTs, especially under a highly contended workload. The hybrid execution strategy of Snapper is able to execute both types of transactions concurrently to improve system performance under a hybrid workload. It is especially beneficial when most of the transactions in the system are PACTs. Furthermore, all the execution strategies in Snapper scale well with the number of CPUs under both benchmarks used in our experiments.

As future work, we intend to extend the optimization and evaluation of Snapper in a multi-server environment, investigating the trade-offs in algorithms and mechanisms to partition and coordinate transactions across multiple servers. Deploying Snapper in a distributed environment is non-trivial. First of all, consider that in a system with both distributed and non-distributed transactions, it is obvious that non-distributed transactions do not need to be globally ordered. In this case, a hierarchical ordering service may be needed to differentiate these two types of transactions. In addition, different deployments can affect system performance differently. For example, the placement of coordinators may significantly influence the token circulation latency, which will also have impact on transaction latency. In future work, we plan to thoroughly explore different alternatives based on the current single-server design.

ACKNOWLEDGMENTS

This work was supported by Independent Research Fund Denmark under Grant 9041-00368B.