Sparkle: Speculative Deterministic Concurrency Control for Partially Replicated Transactional Data Stores

Zhongmiao Li†⋆, Peter Van Roy† and Paolo Romano⋆
†Université catholique de Louvain ⋆Instituto Superior Técnico, Lisboa & INESC-ID

ABSTRACT

Modern transactional platforms strive to jointly ensure ACID consistency and high scalability. In order to pursue these antagonistic goals, several recent systems have revisited the classical State Machine Replication (SMR) approach in order to support sharding of application state across multiple data partitions and partial replication. By promoting and exploiting locality principles, these systems, which we call Partially Replicated State Machines (PRSMs), can achieve scalability levels unparalleled by classic SMR. Yet, existing PRSM systems suffer from two major limitations: 1) they rely on a single thread to execute or serialize transactions within a partition, which does not fully exploit the computation capacity of multi-core architecture, and/or 2) they rely on the ability to accurately predict the data items to be accessed by transactions, which is non-trivial for complex applications.

This paper proposes Sparkle, an innovative deterministic concurrency control that enhances the throughput of state of the art PRSM systems by more than an order of magnitude, on standard benchmarks, through the joint use of *speculative* transaction processing and *scheduling* techniques. On the one hand, speculation allows Sparkle to take full advantage of modern multi-core micro-processors, while avoiding any assumption on the a-priori knowledge of the transactions' working sets — which increases its generality and widens the scope of its scalability. Transaction scheduling techniques, on the other hand, are aimed to maximize the efficiency of speculative processing, by greatly reducing the cost of detecting possible misspeculuations.

PVLDB Reference Format:
Zhongmiao Li, Peter Van Roy and Paolo Romano. Sparkle: Scalable Speculative Replication
for Transactional Data Stores. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/TBD

1. INTRODUCTION

Nowadays, large-scale online services serve millions of clients dispersed across the globe and are faced with a number of challenging, and often antagonistic, requirements. On the one hand, to tame the ever growing complexity of modern applications, distributed data storage systems have shifted away from weak consistency models [15, 35] and embraced strong, transactional, semantics [9]. On the other hand, a number of works [24] have shown that the profitability of large-scale on-line services hinges on their ability to ensure tight requirements on user-perceived latency and high availability — an arduous goal given the sheer volume of traffic and data that modern applications need to cope with.

The above trends have fostered significant research interest in the design of high performance transactional platforms capable of ensuring strong consistency and fault-tolerance even when deployed on large scale infrastructures, e.g., [46, 8, 36]. The techniques proposed by recent works in this area extend, in various ways, the classic State-Machine Replication (SMR) approach [45], a long-studied replication technique for building strongly consistent, fault tolerant systems. In a nutshell, SMR operates according to an *order then execute* approach: replicas rely on a consensus protocol [29] to agree, in a fault-tolerant way, on a common, total order in which transactions should be executed — which we refer to as *final order*. Transactions are then executed at each replica using a *deterministic concurrency control*, which ensures that their serialization order is equivalent to the final order [23].

Several recent works [46, 8, 36] have focused on addressing what is arguably the key scalability limitation of the classic SMR approach, namely its reliance on a full replication model, by allowing to shard applications' state across multiple partitions, which are then replicated across a (typically small) number of machines. This approach, which we call *Partially Replicated State Machine* (PRSM), allows, at least theoretically, for scaling out the volume of data maintained by the platform, as well as the achievable throughput, by increasing the number of data partitions.

However, the partial replication model at the basis of the PRSM approach introduces also a major source of complexity: how to regulate the execution of transactions that access multiple partitions. In fact, single-partition transactions (SPTs) can be ordered by running consensus only among the replicas of the partitions they access, and then be processed locally at each replica using a deterministic concurrency control — just like in classic SMR. The ordering phase of multi-partition transactions (MPT), though, requires an additional inter-partition coordination phase in order to ensure that the MPTs accessing common partitions are ordered in the same way at these partitions. Further, during their execution phase, MPTs need to access data hosted at remote partitions and, as such, the deterministic concurrency control also needs to cope with distributed inter-partition conflicts, which need to be regulated so to enforce a transaction serialization order equivalent to the one specified by the (intra- and inter-partition) replica coordination phases.

The existing literature on PRSM has mostly focused how to

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 45th International Conference on Very Large Data Bases, August 2019, Los Angeles, California.
Proceedings of the VLDB Endowment, Vol. 12, No. xxx
Copyright 2018 VLDB Endowment 2150-8097/18/10... $ 10.00.
DOI: https://doi.org/TBD

minimize the performance toll imposed by the ordering phase of PRSM, by devising techniques aimed to overlap transaction ordering and execution. Thanks to these techniques, state of the art PRSM systems have shifted the limiting factor of system's throughput from the ordering phase to the execution phase, i.e., the rate at which the deterministic concurrency control allows transactions to be processed at each partition — which is the focus of this work.

A simple approach to ensure that, at each partition's replica, transactions are executed in a *serialization order* equivalent to the final order is to execute all the transactions in a partition's replica sequentially [8, 26]. Unfortunately, this solution limits the maximum throughput achievable by any partition to the processing rate of a single thread: a major limitation that prevents from exploiting the performance potential of modern multi-core systems.

Other approaches, like Calvin [46, 42], enable multiple threads to process a partition's transactions concurrently [46, 42], but the deterministic concurrency control techniques employed in these systems suffer from two crucial limitations: (i) they rely on a single thread to schedule, in a deterministic way, the execution of all transactions, which inherently limits the scalability of the solution, and (ii) they assume the ability to accurately predict the data items to be accessed by transactions, which is a non-trivial task for complex, real-life applications [4].

This work tackles the above discussed limitations by introducing Sparkle, a novel distributed deterministic concurrency control that enhances the throughput of state of the art PRSM systems by more than order of magnitude through the joint use of *speculative* transaction processing and *scheduling* techniques.

Speculation is used in Sparkle to allow transactions to be processed "out of order", i.e., to be tentatively executed in a serialization order that may potentially differ from the one established by the replica coordination phase. Thanks to speculative execution, not only can Sparkle take full advantage of modern multi-core micro-processors — by avoiding inherently non-scalable designs that rely on a single thread for executing [8] or scheduling transactions [46]. It also avoids any assumption on the a-priori knowledge of the transactions' working sets — which increases the solution's generality and widens the scope of its scalability.

The key challenge one has to cope with when designing speculative systems, like Sparkle, is to minimize the *cost and frequency of misspeculation*, which, in Sparkle occur when two conflicting transactions are speculatively executed in a serialization order that contradicts the final order dictated by the replica coordination phase. This problem is particularly exacerbated in PRSM systems, since misspeculations that affect a MPT (e.g., exposing stale versions produced by a speculative transaction executing at a remote partition) can only be detected by exchanging information among remote partitions. As such, the latency to confirm the correctness of speculative MPTs is order of magnitudes larger than for the case of SPTs, and, as we will show, can severely hinder throughput.

Sparkle tackles these challenges via two key, novel, techniques:

- The deterministic concurrency control employed by Sparkle combines optimistic techniques with a timestamp-based locking scheme. The former aims to enhance parallelism. The latter increases the chances that the spontaneous serialization order of transactions matches the one established by the replica coordination phase, and allows for detecting possible divergences in a timely fashion — thus reducing the frequency and cost of misspeculation.
- Sparkle strives to remove the inter-partition confirmation phase of MPTs from the critical path of execution of other transactions via two complementary approaches: i) controlling, in a deterministic way, the final order of transactions, so as to schedule MPTs that access the same set of partitions consecutively; ii) taking advantage of this scheduling technique to establish the correctness of MPTs via a distributed coordination phase, which we call *Speculative Confirmation* (SC). SC is designed to minimize overhead, by exploiting solely information opportunistically piggybacked on remote read messages exchanged by MPTs, and maximize parallelism, by removing the MPT coordination phase from the critical path of transaction processing.

Through an extensive experimental study, encompassing two state of the art PRSM techniques [46, 8] and based on both synthetic and standard benchmarks, we show that Sparkle can achieve more than one order of magnitude throughput gains, while ensuring robust performance even when faced with challenging workloads characterized by high contention and frequent MPTs.

The reminder of the paper is organized as follows. Sec. 2 reports the related work of Sparkle. Sec. 3 describes our system model and state-machine replication. Sec. 4 details the Sparkle protocol. Sec. 5 presents the results of the experimental evaluation of Sparkle. Finally, Sec. 6 concludes the paper.

2. RELATED WORK

A large body of works has investigated how to build strongly-consistent, yet highly scalable, transactional data stores. Existing systems can be coarsely classified based on whether they adopt the *deferred update replication* (DUR) [25] or the *state-machine replication* (SMR)[30] approaches. In systems based on the DUR approach [10, 28], transactions are first locally executed at a single replica and then globally verified, via an agreement protocol based on consensus [25] and/or Two Phase Commit [21]. Systems based on the SMR approach [46, 18], as already mentioned, first agree on the serialization order of transactions, using some form of consensus-based coordination scheme, and then execute them using a deterministic concurrency control to ensure a common serialization order at all replicas. The DUR and SMR approaches have different pros and cons, which make them fit for different types of workloads [13, 12, 47], and, in fact, various systems have proposed to dynamically use either solution based on the workload characteristics [11, 13, 27]. The focus of this work is on SMR-based systems, which excel in contention-prone workloads, where DUR systems tend to suffer of lock-convoying and high abort rates [47].

More precisely, we focus on the PRSM approach [36, 8, 46, 34], which extends the classic SRM scheme to support a more scalable partial replication model. As discussed in Sec. 1, PRSM systems need to cope with a key additional problem, not arising in SMR systems: guaranteeing that the MPTs that access a *common* set, say S, of partitions are executed in an equivalent serialization order by all the nodes that replicate partitions in S. This has an impact both on the ordering and on the execution phases of transactions. The existing literature has invested significant effort to minimize the impact of the ordering phase on throughput of PRSM systems, e.g., by only demanding coordination between partitions hosted in the same data center [46] and/or by scheduling MPT's execution in the future, so as to remove inter-partition coordination from the critical path of execution phase of transactions [34]. These techniques, which are orthogonal to the contributions of this work, allow for effectively overlapping the ordering and execution phase of transactions. As result, in state of the art PRSM systems, the system's throughout is typically limited by the efficiency of the deterministic concurrency control that is employed, at each replica, to regulate the execution of transactions.

Existing PRSM systems rely on diverse techniques to implement a deterministic concurrency control. Some approaches avoid a priori the possibility of non-deterministic execution at different repli-

cas by forcing the execution of only a single thread per partition [8, 26]. This approach has its main advantage in sparing partitions from the use (and cost) of any concurrency control. Unfortunately, though, this solution also inherently limits the maximum throughput achievable by any partition to the processing rate of a single thread, preventing to scale up performance of partition's replicas by exploiting the parallelism offered by modern multi-core architectures. Some works [31, 26] have argued that this limitation can be circumvented by using a larger number of smaller partitions, delegating each partition to a different thread of the same parallel machine. However, this approach can increase significantly the frequency of MPTs, since, when using smaller partitions, it is more likely for transactions to have to access data scattered over multiple partitions. Accesses to different data partitions, even if maintained by the same local machine, impose implcit synchronization barriers among the different instances of the same MPT running at different partitions, which need to block until the corresponding remote instances execute and disseminate data to other partitions. As we will show in Sec. 5, this can impose a severe overhead and cripple system's throughput. Other systems, like Calvin [46] (and, previously, by Nodo [42], which assumed an SMR-based, full replication model), conversely, allow for the concurrent execution of transactions. To enforce deterministic transaction scheduling, these approaches rely on a single thread to acquire, in the order established by the replica coordination phase, the locks required by transactions, prior to executing them. As we show in Sec. 5, in typical OLTP workloads dominated by short running transactions, the scheduler thread turns quickly into a critical bottleneck as the degree of parallelism of the underlying computing platform increases. Further, in order to acquire all the locks needed by a transaction before it is executed, these solutions require mechanisms for predicting the transaction's data access pattern — a non-trivial problem in complex benchmarks [3] and real-life applications [4]. The solutions proposed in the literature to cope with this issue are, unfortunately, quite unsatisfactory: existing techniques either require programmers to conservatively over-estimate the transaction's working set [42] (e.g., at the granularity of transaction tables, even though transactions need to access just a few tuples), or they estimate it by simulating the transactions execution, and then abort them if the working set's estimation turns out to be inaccurate during (real) execution. The former approach can severely hinder parallelism. The latter can impair performance in workloads that contain even a small fraction of, so called, *dependent transactions* [46], i.e., whose execution (and set of accessed data items) is influenced by the snapshot of data they observe (Sec. 5).

Sparkle tackles these limitations by combining speculative transaction processing techniques — which exploit out of order processing techniques to fully untap the parallelism of modern multi-core processors without requiring a priori knowledge of transactions' working sets — and scheduling mechanisms — which redefine, in a deterministic way, the serialization order of transactions established by the ordering phase to minimize the cost of detecting misspeculations.

The problem of designing high performance deterministic concurrency controls has also been studied for the simpler case of (less scalable) SMR systems adopting a full replication model. Rex [22], for instance, first executes operations concurrently at a primary replica, where lock acquisition and release actions are recorded. Then the recorded trace is replicated, using consensus, to all replicas, which replay the execution by acquiring and releasing locks according to the recorded order. Both logging the trace of lock accesses and replaying it deterministically introduce non-negligible overheads on the critical path of execution of transactions, which Sparkle avoids thanks to the use of speculation. Speculative/out of order transaction processing techniques were already used in SMR-based (i.e., fully replicated) transactional platforms [23, 39, 40]. Unlike these solutions, Sparkle is a PRSM approach, i.e., it targets a partial replication model, which, as already discussed, raises a number of additional challenges related to the processing of MPTs.

The deterministic concurrency control proposed in this work has relations also with the large body of works on deterministic execution of multi-threaded applications, typically aimed at debugging and testing [6, 37, 16, 17, 41]. However, these mechanisms assume to have the possibility to first record a non-deterministic execution, during which *all* events affecting thread scheduling, including, e.g., any lock acquisition, are logged (to be later replayed deterministically). In the context of PRSM systems, though, a deterministic concurrency control scheme has to tackle a different problem: ensuring that the serialization order of transactions is equivalent to the one established by the replica coordination phase, without having a priori knowledge of which data items, and corresponding locks, transactions will require. Furthermore, these mechanisms often require hardware support [16, 17] or have significant overhead [6, 5] for use in production systems.

3. SYSTEM MODEL

Architecture. We consider the typical system architecture targeted by PRSM systems, such as [46, 8, 34], (see Fig. 1), in which application's data is sharded across a predetermined number of partitions, each of which is replicated over a (logically disjoint) set of servers, which we refer to as replication group. For the sake of brevity, in the following we use the terms *partition's replica* and *server*, interchangeably. The architecture illustrated in Fig. 1 depicts a possible scenario, in which every partition is replicated in every data center: this deployment allows to ensure disaster recovery, while allowing MPTs to be ordered and executed without requiring communication across data centers [46], although our model is generic enough to support scenarios in which certain data partitions may have to be replicated only in a specific sub-set of the available data centers.

As for the fault model, we assume that servers may be subject to crash faults and that, even though entire data centers may fail, there exists a majority of correct replicas of each partition. We assume also communication channels to be reliable, in the sense that if the sender and receiver processes do not crash, every message they exchange is eventually delivered. As mentioned, existing PRSM systems adopt various techniques during the ordering phase of transactions. As a matter of fact, these techniques are orthogonal to the solution proposed in this work, which focuses on how to enhance the efficiency of the execution phase and can be used in combination with different approaches for implementing the ordering phase. Yet, since the coordination techniques employed in the ordering phase are normally based on consensus protocols, we do need to assume that the synchrony level in the system is sufficient (e.g., eventual synchrony [19]) to allow implementing consensus [20].

Transaction and data model. Sparkle provides a basic CRUD transactional interface (create/insert, read, update and delete), and ensures *serializability* semantics [7]. Sparkle is a deterministic concurrency control that can be plugged into a generic PRSM system that abides by the order-then-execute model defined below. If the protocol used during the ordering phase respects real-time ordering between transactions (i.e., given two transactions T1, T2, such that T1 precedes T2 according to real-time order, T1 is serialized before T2 by the ordering phase) then the resulting system ac-

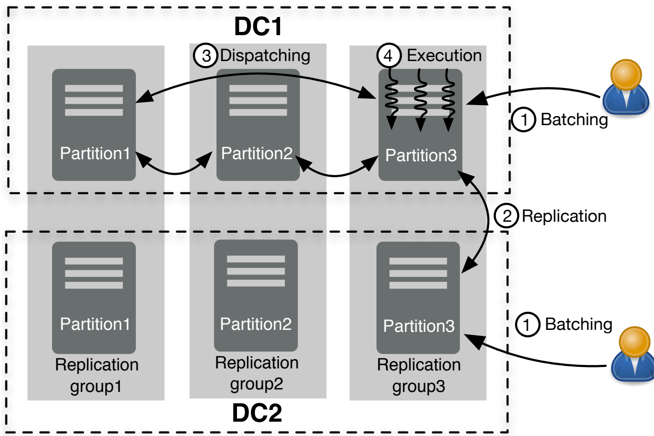


Figure 1: The System model and processing stages of Sparkle. For readability, the figure only highlights the stages happening in DC1, Partition3. Curved arrow denotes messages and spiral arrow denotes execution threads.

tually guarantees strict serializability. Transactions can be aborted and re-executed multiple times before they are committed. We call the various (re-)executions of a transaction *transaction instances*.

Like in any PRSM system, e.g. [46, 8, 34], we assume that, given a transaction and its input parameters, it is possible to identify which data partitions it accesses. This information is exploited to ensure that a transaction is deterministically ordered and executed only by the data partitions actually accessed by that transaction. Such an assumption is typically easy to meet in practice, given that data partitions are normally quite coarse grained. In fact, overestimating the set of partitions accessed by a transaction does not compromise consistency, but only impacts efficiency by causing unnecessary ordering and execution of transactions.

Unlike other PRSM solutions, e.g., [46], we do not assume any fine-grained information on the individual data items that transactions access — which is typically not easily predictable, in an accurate way, in complex applications [4]. As mentioned, we distinguish between single- and multi-partition transactions (SPTs and MPTs, respectively). We refer to the instances of an MPT at the various partitions it accesses as *sub-transactions*. Given a sub-transaction, we refer to the other sub-transactions of the same MPT as *sibling sub-transactions*, or just *siblings*. Unlike SPTs, which can execute independently at each replica, MPTs require, in the general case, communication among sibling sub-transactions, as their execution may depend on the data stored on a different partition.

PRSM model. The deterministic concurrency control proposed in this work is designed to accelerate the execution phase of a generic PRSM system, e.g., [46, 34, 8], which operates according to the order-then-execute approach. In fact, the actual protocol used during the ordering phase is irrelevant for Sparkle, which could be plugged in any PRSM system, provided that the final order they establish ensures the following three properties:

1. all the (correct[1]) replicas of the same partition deliver the same *sequence*, $B_1,...,B_n$, of transaction batches, where each batch contains the same totally ordered set of (single- or multi-partition) transactions;
2. if an MPT T is delivered in the i-th batch by a partition, then T is delivered in the $i-th$ batch of all the partitions it involves;
3. for any pair of MPTs, say T_1 and T_2, that access a set of common partitions, say $\mathcal{S}=\{P_1,...,P_n\}$, T_1 and T_2 are ordered in the same way by all the (correct) servers that replicate any partition in $\mathcal{S}$, i.e., either $\forall P_i \in \mathcal{S}\ T_1 \rightarrow T_2$ or $\forall P_i \in \mathcal{S}\ T_2 \rightarrow T_1$.

[1]Faulty servers may, of course, never deliver a batch.

It should be noted that the ordering phase establishes a total order over the transactions executing at each partition, whereas the transactions executing at different partitions are only partially ordered, i.e., no ordering restriction is imposed to transactions accessing disjoint data partitions. In the following, we will use the terms *transactions preceding/following* a transaction T at some partitions P_i to refer to the transactions ordered, respectively, before/after T according to the final order at P_i.

Once the ordering phase of a transaction is completed, a transaction is executed at all the partitions' replicas that it involves and the results it produced are delivered to the client by the data center to which the request was originally submitted. A relevant optimization employed by PRSM systems, and assumed also by Sparkle, is related to the management of read-only transactions. Since these do not alter the data store's state, they are only executed at a single replica. Further, read-only SPTs can bypass the ordering phase, and be executed in any serialization by the replica to which it is submitted. Read-only MPTs can still be executed only by a single replica, but they still need to undergo the ordering phase, since it is necessary to serialize the various sibling sub-transactions of each MPT consistently at all the partitions it involves.

Existing PRSM systems ensure the above properties using different approaches, but, for the sake of clarity and to exemplify a concrete system in which Sparkle may be integrated, below (and in Fig. 1) we overview the solution employed in Calvin [46].

Transaction submission and intra-partition replication. Clients interact with the system by submitting transactions, along with any input parameter needed for their execution, to any server of any nearby data center. Servers periodically batch, e.g., for 5-10 msecs, the transactions received from clients. After that, the replicas of each partition activate an independent intra-partition consensus instance, also called *replication phase*, which merges the transactions batched by every replica of a given partition and replicates these transactions in a fault-tolerant manner — while ensuring that all the partition's replicas agree on the set of transactions to be included in the next batch. Batching is crucial to amortize the cost of consensus in high throughput scenarios, and ensures that it does not turn into the bottleneck of the system, at the cost of a negligible increase in the latency perceived by users.

Transaction dispatching. The replication phase ensures that all replicas of a partition receive the same sequence of batch, and that each batch contains the same set of transactions. However, it is still necessary to 1) relay SPTs that were submitted by clients to wrong partitions to the ones whose data they actually need to access, and 2) ensure that MPTs are disseminated to all their involved partitions, while ensuring that MPTs are ordered consistently at all the partitions they access in common. The necessity for enforcing a consistent order for MPTs accessing common partitions is illustrated in Fig. 2, which illustrates an example scenario in which two MPTs $T1$ and $T2$ are ordered differently at two partitions Px and Py, yielding a non-serializable execution.

The above requirements are fulfilled via an additional inter-partition coordination scheme, also called *dispatching* phase, which varies from protocol to protocol. Calvin uses a deterministic dispatching scheme that is executed in parallel at each data center, in which all partitions exchange the transactions they received during the intra-partition replication stage and that involve some remote partition (either because these transactions were submitted to the wrong partition by clients or because they are MPTs that

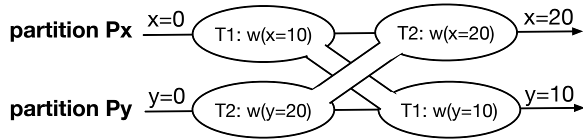


Figure 2: Two multi-partition transactions T1 and T2 are ordered differently in partitions Px and Py, violating serializability.

should execute at all its involved partitions). After a partition has received messages from all other partitions in the same data center (including those that had no transactions to disseminate), it merges and sorts the received transactions using a deterministic scheme, which ensures that different partitions will order transactions disseminated from the same partition consistently. The determinism of the dispatching mechanism ensures that different data centers will converge to the same final order, despite running the dispatching phase independently. It also simplifies fault-tolerance: if a partition p does not receive the dispatching message from some replica of partition p' in its local data center, it can simply query a correct replica of p' at a different data center to retrieve the transactions dispatched by p' for the current batch.

4. Sparkle

This section describes Sparkle's deterministic concurrency control scheme. We start by discussing the processing of SPTs (§4.1) and MPTs (§4.2). Then, we prove that Sparkle provides serializability. Finally, we discuss how to optimize the treatment of read-only transactions (§4.4).

4.1 Single partition transactions

The key idea at the basis of Sparkle is to execute transactions using a multi-versioned, optimistic concurrency control, which aims to maximize parallelism by allowing transactions to be speculatively executed according to a spontaneous, non-deterministic serialization order that can potentially contradict the final order. To ensure consistency, misspeculations, i.e., transactions speculatively executed out of order, are detected at run-time, leading to the automatic abort and restart of the affected transactions. In order to maximize efficiency, Sparkle incorporates a timestamp-based locking scheme aimed at pursuing a twofold goal: reducing the likelihood of misspeculations, by steering the spontaneous serialization order towards the final order, and minimizing their cost, by detecting possible divergences in a timely and lightweight fashion.

In a nutshell, the execution of SPTs is regulated as illustrated in the diagram in Figure 3 and overviewed in the following. In each partition's replica, Sparkle attributes to each transaction T a consecutive logical timestamp that reflects the total order established at that partition by the ordering phase. These timestamps are used to establish a total order on the data items' versions created by (possibly speculatively) committed transactions, and to define visibility of versions during read operations: a transaction only reads the latest version produced by transactions ordered before it, and when a transaction produces new item versions, it tags them with its own timestamp.

To ensure that a SPT's execution is equivalent to a sequential execution complying with the final order, Sparkle guarantees that final committed transactions read from a snapshot that includes the versions generated by all its preceding transactions. Sparkle guarantees this property by letting a transaction T *final commit* only if all its preceding transactions have *final* committed and if T did not miss any of the updates produced by these transactions — which can happen if T reads a data item before any of its preceding transactions writes to it, i.e., a write after read conflict. To minimize the occurrence of these cases, and detect them in a timely fashion, Sparkle uses a timestamp-based locking scheme, which operates as follows. When reading an item, transactions check if there exists any other *active* transaction with a lower timestamp that wrote to it: in the negative case, the reader transaction registers its timestamp to notify future writers; else, the execution of the reader transaction is suspended, so to prevent the chances of stale reads — which would happen if the writer transaction wrote a different value before committing. When writing a data item, a transaction T checks if there is any transaction with a larger timestamp that has already read that data item, and has, as such, missed the version that T is about to produce: in this case, the reader is aborted and restarted by the writer.

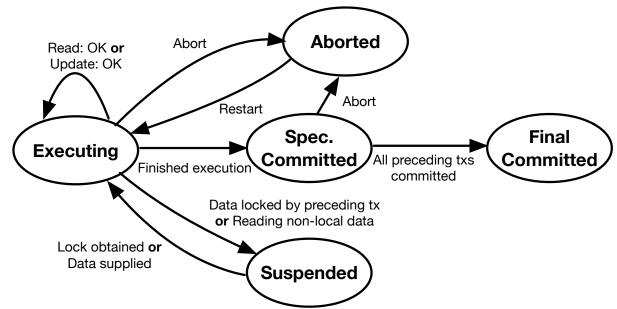


Figure 3: The execution stages of a transaction.

In the following we describe in detail the management of SPTs, along with its pseudocode. Algorithm 1 and 2 describe the behavior of transaction coordinators and the backend storage, respectively.

Main data structures. As mentioned, Sparkle adopts a multi-versioning scheme, which maintains, for each data item, a totally ordered set of versions produced by either speculatively committed or final committed transactions. The total order on the versions of a data item is established through a logical timestamp, which coincides with the logical timestamp attributed to the transaction that produced each version. Each data item maintains two additional data-structures: a *read dependency* set, which tracks the transactions that have read some version of this data item, and a timestamped lock, which is used to synchronized concurrent accesses to the data items. When each transaction's order has been determined by the dispatching phase, it is assigned a timestamp *local_id*, a monotonically increasing counter that reflects the total order established by the dispatching phase. Last but not least, to quickly check if a transaction can be committed, each partition maintains a timestamp named *to_commit_tx*, which marks the *local_id* of the next transaction that can be committed.

Start. Upon activation (Alg1, 1-3), each transaction initializes two main data structures: *workset* and $abort_flag$. The former is a private buffer that stores the data items read and updated by the transaction during its execution, and the latter is used to check whether the transaction has been aborted by some other transaction.

Execution. During its execution, a transaction T may read or update multiple data items (Alg1, 5-30). Upon executing a read/write operation, T checks its $abort_flag$ to see if, in the meanwhile, it has been flagged for abort by some other transaction (preceding T); in this case, T is aborted and re-executed. Prior to

Algorithm 1: Concurrency Control

```
1  start(Transaction tx)
2    tx.workset ← set()
3    tx.abort_flag ← false
4  execute(Transaction tx)
5    for op in tx.ops
6      if tx.abort_flag = true; return aborted
7      if op.type = update
8        if op.key is not local
9          tx.workset.add(op.key, v)
10       elif op.key ∈ tx.workset and op.key already updated
11         tx.workset.add(op.key, v)
12       else
13         result ← Lock(tx, op.key)
14         if result = ok
15           tx.workset.add(op.key, v)
16         else
17           return result //aborted or suspended
18     else //Is read operation
19       if op.key ∈ tx.workset
20         return tx.workset.get(op.key)
21       else
22         if op.key is a local key
23           {result, v} ← Read(tx, op.key)
24           if result = suspended
25             return suspended
26           else
27             tx.workset.add(op.key, v)
28         else
29           return suspended
30   return finished
31 try-commit(Transaction tx)
32   for {k, v} ∈ tx.workset s.t. k is an updated local key
33     if Put(tx, k, v) = aborted
34       remove all applied updates and return aborted
35   if to_commit_tx = tx.local_id ant tx.abort_flag = false
36     to_commit_tx ← to_commit_tx + 1
37     delete all metadata of tx
38 abort(Transaction tx)
39   for {k, v} ∈ tx.writeset s.t. k is an updated local key
40     UnlockOrRemove(tx, k)
41   reset workset and abort_flag of tx
```

its first update to a data item, T tries to obtain an exclusive lock to it (Alg2, 1-9). If the lock is held by a transaction following T in the (total) serialization order (i.e,. by a transaction whose *local_id* is larger than that of T), T preempts the locks and aborts the locking transactions by setting its *abort_flag* to *true*. Conversely, if the lock is held by a transaction T' that precedes T, T waits for T' to finish execution. Finally, once T successfully obtains the lock on the data item, it applies the update to its private *workset*.

While executing a read operation, T first reads from its *workset*, to return any version it produced in a previous write operation. Else, T redirects its read to the back-end data store and checks the state of the lock guarding the data item it intends to reads (Alg2, 10-18). Similar to the above locking procedure, T is suspended if the data item is currently being locked by any of its preceding transactions. Otherwise (the item is not locked, or locked by Tx's following transactions), Tx scans the version list and returns the version with the largest timestamp smaller than its *local_id*. Note that this may not be the version that T would observe, had it been executed according to the final order, as other transactions preceding may later produce more recent versions. Therefore, unless there are no uncommitted transactions preceding T, T appends its *local_id* to the read dependencies of this data item, to be aborted in case of a write-after-read conflict is detected.

Suspended. As mentioned, a transaction T is suspended whenever it tries to read/update a data item that is currently being locked by a preceding transaction (Alg1, 13-17 and 23-25). In that case, the thread executing T can start executing the next unprocessed transaction according to the final order, so to enhance parallelism and maximize resource utilization in parallel/multi-core systems. T will eventually be unblocked when the locking transaction releases lock on the transaction, which notifies the thread responsible of T to resume its execution.

Speculative/final commit. After completing its execution T applies all its buffered updates as follows. For each updated data item, T inserts a new version into the item's version chain, timestamped and ordered by its *local_id*, and releases the corresponding lock (Alg2, 26-34). Then, T checks the read dependencies tracked by this data item and aborts any (therein registered) transaction with a larger timestamp (as they missed T's update on this item) by setting their *abort_flag*. Additionally, T garbage collects the identifiers of any final committed transaction still tracked in read dependencies, which are, at this point, unnecessary (since they no longer risk to abort). While applying its updates, if T finds that any of its obtained locks has already been preempted (Alg2, 27-28), it aborts itself by removing all inserted versions and releasing any remaining lock. Otherwise, T is considered to be *speculatively-committed*.

Next, T checks if it can final commit, which is only possible if all its preceding transactions have already committed and if its *abort_flag* is still *false*. As T's updates have already been applied during the previous step, transactions can be final committed very quickly. To further enhance efficiency, Sparkle employs an additional optimization: it avoids pruning immediately T from the read dependency sets of the items it read, which would introduce overhead during the critical path of execution of transactions, but rather delegates (as already hinted) this task to the transactions that update those data items in the future (as they need anyway to inspect the read dependency sets to detect mispeculations).

If T can not be final committed, the thread executing T is assigned the execution of the next unprocessed transaction, and will periodically check the state of T, to final commit it, if possible.

Abort. T can only be aborted due to *data conflict with preceding transactions*, either because T missed updates from a preceding transactions, or its lock on a data item has been preempted by a preceding transaction. If either case occurs, Tx's *abort_flag* will be set to *true*. Then, Tx aborts by releasing all its obtained locks, and removing any version it had inserted in the data store (Alg2, 19-25).

4.2 Multi Partition Transactions

Extending the above described speculative processing technique to MPTs raises non-trivial challenges. Recall that during their execution (Sec. 3), the sibling sub-transactions of a MPT need to disseminate the results of read operations issued on local data items to the other partitions involved in the transaction. By allowing MPTs to execute speculatively, i.e., without waiting for the final commit of its preceding transactions, then a MPT sub-transaction, say T, may observe an inconsistent local snapshot (e.g., as it missed a data item version produced by a preceding SPT transaction) and send inconsistent data to its siblings. Additionally, even if the local snapshot observed by T were to be correct, the correctness of the data disseminated by T can also be compromised if T, in its

```
Algorithm 2: Backend protocol
  Data variables
  kv: a map storing the data and metadata for keys.
  kv[k].lock_tx: the transaction holding lock on k.
  kv[k].wait_txs: transactions blocked when trying to access k.
  kv[k].read_deps: transactions that have speculatively read k.
  kv[k].versions: timestamped versions for k.
  //All following operations are atomic per key entry.
1 Lock(Transaction tx, Key k)
2   if kv[k].lock_tx and kv[k].lock_tx.local_id < tx.local_id
3     kv[k].wait_txs.append(tx)
4     return suspended
5   else
6     if kv[k].lock_tx.local_id > tx.local_id
7       kv[k].lock_tx.abort_flag ← true
8     kv[k].lock_tx ← tx
9     return ok
10 Read(Transaction tx, Key k)
11   if kv[k].lock_tx and kv[k].lock_tx.local_id < tx.local_id
12     kv[k].wait_txs.append(tx)
13     return {suspended, null}
14   else
15     {v, w_id} = the largest version in
       kv[k].versions whose timestamp is smaller than tx.local_id
16     if tx.local_id != to_commit_tx
17       kv[k].read_deps.add({tx, w_id})
18     return {ok, v}
19 UnlockOrRemove(Transaction tx, Key k)
20   if kv[k].lock_tx = tx
21     kv[k].lock_tx ← null
22     unblock tx's following transactions in kv[k].wait_txs
23   else
24     remove tx's inserted version in kv[k].versions
25     abort transactions read from tx in kv[k].read_dep
26 Put(Transaction tx, Key k, Value v)
27   if kv[k].lock_tx != tx
28     return aborted
29   else
30     kv[k].lock_tx ← null
31     insert {v, tx.local_id} to kv[k].versions
32     abort tx's following
       transactions in kv[k].read_dep that missed its updates
33     unblock tx's following transactions in kv[k].wait_txs
34     return ok
```

turn, is fed with inconsistent remote data by some other sibling T'.

More formally, a sibling sub-transaction at partition X, say T^X, can only be allowed to final commit if it has observed a *global consistent* snapshot, i.e., a snapshot that satisfies the following conditions:

- *C1:* T^X observed a local consistent snapshot that reflects the updates of all preceding transactions at its local partition X.
- *C2:* if T^X received remote data from a sibling transaction, say T^Y, then T^Y also observed a global consistent snapshot.

To simplify presentation, we describe the proposed solution in an incremental fashion, by first assuming that the batch of transactions delivered during the ordering phase is composed exclusively by MPTs that access the same set of partitions. In the following we use the term *homogeneous MPTs* to denote a set of MPTs that access the same set of partitions. We will relax this assumption and discuss how to cope with generic batches composed by SPTs and heterogeneous MPTs in§4.2.2.

A straw-man approach to guarantee the two properties above would consist in introducing a, so called, *conservative confirmation* phase that operates as follows. When the last transaction, say T_i, that precedes a MPT, say T_{i+1}, at partition P final commits, T_i broadcasts to the siblings of T_{i+1} a message informing whether any conflict was locally developed with T_{i+1} either by T_i or by any other transaction preceding T_i at P.

We illustrate this simplistic solution in the left diagram of Fig. 4a, which depicts two partitions X and Y, each executing 3 MPTs involving both partitions. Transaction T_2 suffers an abort at partition X, caused by missing the write to data item k by T_1 (which precedes T_2). Upon its final commit, T_1^X informs partition Y that T_2^X has been aborted once, due to a conflict with any preceding transaction (in this example only T_1^X). It is easy to see that this mechanism allows partition Y to determine that the second instance of T_2^X, denoted $T_2^X:1$, is guaranteed to have observed a locally consistent snapshot at X (i.e., that ensures property C1): T_1^X is the largest among the transactions preceding T_2 at X, and since T_1^X sends a confirmation for T_2^X upon its own final commit, i.e., when every transaction preceding T_2^X has already been final committed, it is no longer possible for T_2^X to suffer from additional aborts due to conflicts with local transactions active at partition X.

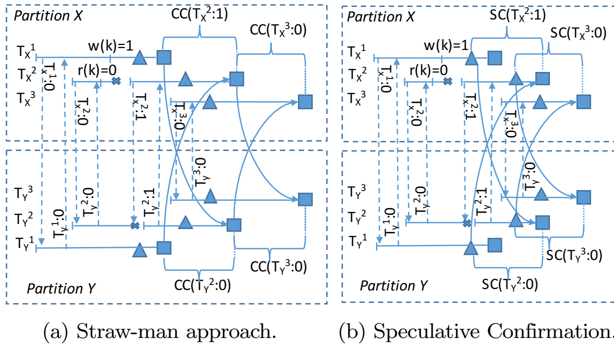
Unfortunately, though, this straw-man approach is both incorrect and inefficient.

It is incorrect, since it does not allow to detect whether T_i received inconsistent remote data from any of its sibling — i.e., violating condition C2. The Figure 5a illustrates a simple scenario in which condition C2 is violated: the first instance of T_X^1 ($T_X^1:0$) undergoes an abort due to a local conflict after having pushed its local reads to its sibling transaction on partition Y, namely $T_Y^1:0$. Having observed an inconsistent remote snapshot at partition X, the correctness of $T_Y^1:0$ execution could be compromised, e.g., by reading incorrect local data items and, consequently, sending inconsistent value to partition X. However, since in this scenario, $T_Y^1:0$ has observed a consistent local snapshot, the confirmation message generated by its preceding transaction at Y, namely T_Y^0, will (erroneously) indicate $T_Y^1:0$ as correct.As shown in the figure, this may lead to an incorrect execution in which partition X final commits an instance of T_X^1 despite this was fed with inconsistent data by $T_Y^1:0$.

It is inefficient since the conservative confirmation phase introduces the latency of an inherently sequential distributed coordination in the critical path of execution of *every* MPT transaction. As illustrated in Fig. 4a, with this mechanism, a partition can send the confirmation for the $i+1$-th transaction of a batch, only upon final committing the i-th transaction, which, in its turn, depends on the reception of the confirmation message that is only sent upon the final commit of the $i-1$-th transaction. With this approach, thus, the throughput of MPTs becomes inherently upper bounded by the rate of completion of the inter-partition confirmation phase, which involves an all-to-all synchronous communication between the involved partitions and imposes, as such, a significant performance toll.

4.2.1 *Speculative confirmation*

We discuss how Sparkle tackles the correctness and performance issues of the previously described straw-man approach in two steps. We first discuss how to parallelize the MPTs' confirmation phase, by presenting a *Speculative Confirmation* (SC) technique that ensures condition C1, but not C2. Next, we discuss how to extend this mechanism to guarantee also condition C2. The corresponding pseudocode is presented in Algorithm 3.



(a) Straw-man approach. (b) Speculative Confirmation.

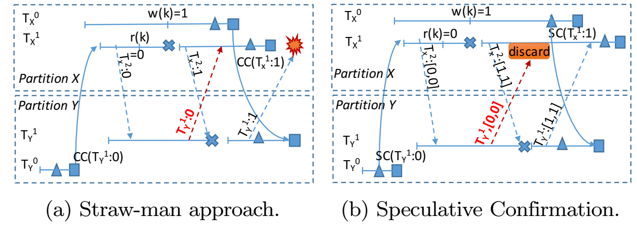
Figure 4: A straw-man approach, using a Conservative Confirmation (CC) scheme, introduces a remote coordination phase in the critical path of execution of MPTs, hindering throughput (left diagram). The Speculative Confirmation (SC) technique allows for effectively parallelizing the coordination phases of a set of homogeneous MPTs (right diagram). Dashed arrows refer to the sending of remote data; continuous arrows to confirmation messages. Triangles and squares represent speculative and final commit events, respectively.

Parallelizing the confirmation phase. The SC scheme tackles the efficiency issue illustrated in Fig. 4a by exploiting a main key idea: determining the validity of the data items sent by the i-th MPT at partition X, T_i^X, based on information disseminated by the transactions preceding T_i at X as soon as they *speculatively commit*, rather that when they *final commit*.

The SC technique is based on the following observations: i) Sparkle's concurrency control ensures that the only cause of local aborts for a transaction T_i^X is a conflict with some local transaction that *precedes* T_i^X in the final order; ii) assuming that batches are executed sequentially[2], the first transaction of a batch never aborts due to a conflict with a local transaction (as a direct consequence of the previous observation).

The SC scheme leverages these observations and disseminates, upon the *speculative commit* of a MPT T_j^X that precedes T_i^X (at partition X), a SC message that conveys information on which instances of T_i^X were locally aborted by T_j^X (if any). This information is exploited by remote partitions involved by T_i to learn which speculative instances of MPT_i^X are guaranteed to have observed a local consistent snapshot at X — thus, enforcing condition C1. In more detail, this is accomplished by associating with each instance of T_i^X a *Local Abort Number* (*lan*), which is incremented whenever this is aborted by a preceding local MPT T_j^X (Alg3, 4-5). Whenever a MPT speculatively commits, it broadcasts to all the partitions it involves the *lan* of the latest instance of any (following) local MPT that it aborted during its execution (Alg3, 14-16). Also, whenever an MPT instance sends the results of its local reads to remote partitions, it tags them with its current *lan* (Alg3, 2-3). Although SC messages are sent in parallel and in an order that does not necessarily coincide with the final order (i.e., in the order in which transactions are speculatively committed), MPTs are still final committed sequentially and in compliance with the final order. As mentioned, the first transaction of a batch, T_1, can be immediately final committed at every partition it involves. T_1 serves an "anchor" for the chain of SCs, since the SC messages generated by the execution of T_1 at its involved partitions enables also the final commit of T_2: since T_2 can only be aborted by T_1, the T_1's SC messages allow to determine which instance of T_2 is guaranteed to have observed a consistent snapshot at every partition. This logic can be extended and generalized to an arbitrary long number of homogeneous MPTs as follows: MPT T_i^Y at partition $Y \neq X$ can verify the local consistency (property C1) of remote data received from a sibling sub-transaction, T_i^X, by confirming that the *lan* of the remote data is equal to the largest *lan* for T_i^X that Y has heard of from every transaction preceding T_i^X at X, by the time in which T_{i-1}^Y has final committed (Alg3, 9-13).

[2]This assumption is done only to simplify presentation of the SC mechanism, and is not actually required by Sparkle.



(a) Straw-man approach. (b) Speculative Confirmation.

Figure 5: A straw-man approach tracking only conflicts with local transactions fails to ensure that the data produced by remote transactions observed a consistent remote snapshot (left diagram). Sparkle tackles this issue via the use of an additional *Remote Abort Number* (right diagram). Dashed arrows refer to the sending of remote data; continuous arrows to confirmation messages. Triangles and squares represent speculative and final commit events, respectively.

Fig. 4b depicts how the same execution illustrated in Fig. 4a is regulated via the SC mechanism. As previously discussed, the CC approach serializes the confirmation phases of T2 and T3, by allowing T3 to be confirmed (by any partition) only when T2 is (locally) final committed. Conversely, SC effectively overlaps the confirmation phases of T2 and T3 by activating their confirmation phases as soon as T1 and T2 speculatively commit, respectively. More in general, since transactions can speculatively commit at any time and in any order, they can propagate information on conflicts with any following local MPT in parallel, effectively removing the confirmation phase from the critical path of transaction processing.

It is worth mention a simple, yet effective optimization, that Sparkle exploits to minimize the overhead associated with the MPT confirmation mechanism: instead of sending *ad hoc* messages solely to disseminate the SC information, this is piggybacked on the messages used by MPTs to disseminate the results of local read operations to their sibling sub-transactions.

Detecting inconsistent remote snapshots. As illustrated by the diagram in Fig. 5a, the mechanisms described so far allow for safely detecting whether sibling sub-transactions observed a locally consistent snapshot, but fail to determine whether the execution of a sibling sub-transaction was affected by observing an inconsistent remote snapshot.

In Sparkle, we tackle this problem by extending the SC mechanism to exchange an additional metadata, called *Remote Abort Number* (*ran*), which tracks how many times a MPT is aborted as a consequence of exposing any of its sub-transactions to inconsistent remote snapshots. A sub-transaction's *ran* is initialized to 0, and it is updated if 1) it aborts after having sent a speculative version, or 2) if it receives remote data from a remote sub-transaction with a larger *ran*. In the former case, the sub-transaction increments its *ran* by one (Alg3, 7); in the latter case, the sub-transaction is notified to have received inconsistent data from some sibling transaction, so it set its *ran* to the received one and is restarted (Alg3, 24-25). The remote data disseminated by an MPT sub-transaction to its siblings is tagged with both its *lan* and *ran*. Any incoming remote data tagged with a *ran* smaller than the current sub-transaction's *ran* is discarded (Alg3, 21-23): as shown in Fig. 5b, this allows for detecting if the data

received by a sibling sub-transaction was produced on the basis of a stale remote snapshot.

It is easy to see that a sub-transaction can only complete its execution if the remote data it received from its sibling transactions is tagged with the same *ran* (as the sub-transaction blocks, resp. aborts, if it receives remote data from a partition with a *ran* smaller, resp. larger than its current one). We also note that an increase of the *ran* can always be rooted back to some local conflict occurring at some partition, which can be tracked via the *lan* metadata. This implies that if a sub-transaction completes its execution after having received from all its siblings remote data tagged with their final *lan*, then the MPT's *ran* can no longer increase — since no sibling can be subject to further local aborts. It follows that the sub-transaction must have received remote data produced by sibling transactions that have, in their turn, received remote data by sub-transactions that must have observed a consistent snapshot at their local partition — ensuring condition C2.

Algorithm 3: Speculative execution for MPTs

```
1  //Local events
2  upon reading a local key k with value v, for an MPT tx
3     send{read, k, v, tx.lan, tx.ran} to tx's siblings
4  upon aborting an MPT tx
5     tx.lan ← tx.lan + 1
6     if tx has sent remote read before aborting
7        tx.ran ← tx.ran + 1
8        send{remote-abort, tx.ran} to tx's siblings
9  upon trying to speculatively-commit an MPT tx
10    for {k, v, lan, ran, p} ∈ tx.workset s.t. k is a remote key
11       if lan != tx.lan[p] or ran != tx.ran
12          return aborted
13    try-commit(tx)
14 upon having speculatively-committed an MPT tx of group g
15    for t ∈ transactions of g aborted by tx
16       send {confirmation, t.lan} to t's siblings
17 upon having final-committed
     an MPT tx s.t. tx is the last transaction before group g'
18       t ← the first transaction of g'
19       send {confirmation, t.lan} to t's siblings
20 //Remote messages
21 upon tx receiving{read, k, v, lan, ran} from partition p
22    if ran >= tx.ran and lan >= tx.lan[p]
23          tx.workset.add({k, v, ran, lan, p});
24 upon tx receiving{remote-abort, ran} s.t. ran > tx.ran
25    tx.ran ← ran
26    remove all values in tx.workset with smaller ran
27    abort()
28 upon tx receiving{confirmation, lan} from partition p
29    if lan > tx.lan[p]
30       tx.lan[p] ← lan
```

4.2.2 Dealing with heterogeneous MPTs

The correctness of the SC mechanism presented above hinges on the assumption that the batch is composed solely by homogeneous MPTs accessing the same set of partitions. This assumption ensures that the first (multi-partition) transaction of the batch never undergoes aborts, hence the final *lan* of all its sub-transactions is necessarily equal to 0 and is *a priori* known. Clearly, this property no longer holds if batches are composed by mixes of SPT and MPTs involving heterogeneous sets of partitions. In fact, in the general case, a (multi-partition) transaction can undergo an unknown number of aborts if it is preceded even just by a single transaction (and executed concurrently with it).

As already discussed, this property is crucial to the correctness of the SC scheme, since it allows to use the first MPT of a batch as an anchor for the chain of SCs: not only can the first MPT of a batch be immediately final committed at all its partition, its SC messages also allow to determine the final *lan* of every sub-transaction of the following (i.e., the second) MPT of the batch.

Keeping this in mind, let us discuss how to extend the SC scheme presented so far to cope with generic transactions. To this end, Sparkle exploits the following three mechanisms:

Transaction grouping. Upon reception of a transaction batch, at the end of the dispatching phase, each partition deterministically reorders the transactions in the batch by grouping them according to the set of partitions they access. The resulting final order is composed by a sequence G_1,...,G_n of transaction groups, where each group G_i contains one or more transactions that access the same set of partitions.

Inter- and Intra-group confirmation. Since the transactions of a group are homogeneous, the previously described SC scheme can be seamlessly used to regulate the execution of every transaction in the group, except for the first transaction of the group (as the final *lan*s of its sub-transactions are not known in this case). To tackle this issue, whenever a partition *final commits* the last transaction of group G_i, say $T^{G_i}_{last}$, it sends a *conservative* confirmation to all the partitions involved in group G_{i+1} specifying the final *lan* of the first transaction of group G_{i+1}, say $T^{G_{i+1}}_{first}$ (Alg3, 17-19). Recall that since a partition sends the confirmation messages for $T^{G_{i+1}}_{first}$ upon final commit of $T^{G_i}_{last}$, the *lan* it specifies for $T^{G_{i+1}}_{first}$ is necessarily guaranteed to be the final one (as $T^{G_{i+1}}_{first}$ can no longer incur a local abort after the final commit of its preceding transaction).

Transaction scheduling. Realistic applications are typically designed to avoid spanning a large number of partitions, in order to enhance performance by maximizing data locality [14, 38]. As such, it is unlikely that, in practice, a batch encompasses a very large number of groups, each containing very few transactions. Yet, in workloads that do generate a large number of small transaction groups, it is possible to amortize the overhead introduced by the inter-group CC scheme by "artificially inflating" each transaction group by scheduling, in a smart way, the execution of SPTs. In particular, each partition can deterministically re-order its SPTs and serialize them in between each pairs of consecutive MPT groups with the goal of spacing out as much possible two consecutive MPT groups. Specifically, once a partition completes processing the MPTs of a given group, say G_i, it can seamlessly continue processing any following SPT and delegate to the last SPT preceding the next MPT group, say G_{i+1}, the task of sending the CC for the first transaction of G_{i+1}. This has an effect equivalent to increasing the number of transactions in each MPT group, thus amortizing the performance overhead imposed by the inter-group CC across a larger number of transactions.

4.3 Correctness arguments

In this section, we discuss the correctness of Sparkle. Specifically, we intend to show that Sparkle's concurrency control ensures *serializability* and that it enforces a serialization order, which we denote with $<_S$, which can be obtained by applying the deterministic transactions grouping and scheduling rules discussed in subsubsection 4.2.2 to the final order determined by the dispatching phase.

We start by discussing the case of batches consisting solely of SPTs. Next, we consider the case of batches composed by a single group of homogeneous MPTs. Finally, we analyze the case of generic batches composed by SPTs and heterogeneous MPTs.

Single-partition transactions Let us first examine the case of batches containing only SPTs. We show that any *final committed* SPT T_i, having serialization order (namely *local_id*) i in the batch observes a snapshot that reflects the updates produced by every transaction serialized before T_i, i.e., any read issued by T_i on a key K must return the version created by the transaction T_h, whose serialization order, h, is the largest among all transactions that precede i and that write to K.

Assume, by contradiction, that T_i reads a key K and fetches a version created by transaction T_f , where $f<h$ (i.e., T_i misses the version of K created by T_k and observes an earlier version created by T_f). If T_i has written to K before reading, then it is not possible for T_i to read T_f's version, as in Sparkle transactions first attempt to read from their own write-set. Let us consider the scenario in which T_i has not updated K before reading it. In this case, T_i can only return the version created by T_f if, when T_i reads K, T_h has not issued a write on K. In such a case, T_i registers its timestamp to the read-dependencies of K. However, given that transactions final commit according to the serialization order $<_S$ and since $h<_S i$, T_i can only final commit after T_h does. This means that, before T_i final commits, T_h must have necessarily issued a write on K, encountered T_i in the read dependencies of K and triggered the abort of T_i — contradicting the assumption that T_i final commits.

Multi-partition transactions. Let us now consider the scenario of batches containing homogeneous MPTs that involve the same set of partitions. We intend to show that any MPT T_x is final committed, only if each of its sub-transactions observes, at its own partition, a state equivalent to the one that it would observe if the transactions in the batch were executed sequentially, according to $<_S$, in a single node system. To this we prove, by induction, that every sub-transaction of T_x must have observed both a locally (condition C1) and globally (condition C2) consistent snapshot.

First we prove the base step: if T_x is the first transaction in the batch, as already discussed, all its sub-transactions must necessarily observe a locally consistent snapshot — which is sufficient to satisfy condition C1. Further, since no sub-transaction of T_x can ever observe an inconsistent local snapshot (and externalize it to its siblings), it also follows that no sub-transaction of T_x can ever receive inconsistent remote values from its sibling — proving condition C2.

Next, we show that if all the MPTs preceding T_x^{Pi} ($T_1^{Pi}...T_{x-1}^{Pi}$) observe a globally consistent snapshot, then also T_x^{Pi} does. By contradiction, let us assume that T_x^{Pi} (T_x's sub-transaction at partition P_i) final commits after observing an inconsistent snapshot. Using the same arguments used for the case of SPTs, it is easy to show that T_x^{Pi} cannot final commit if it observes a locally inconsistent snapshot (condition C1). We focus, therefore, on proving that T_x^{Pi} cannot final commit after having received data T_x^{Pk}, active at partition P_k, which has observed a non-globally consistent (condition C2) snapshot. There are two cases to consider:

1. T_x^{Pi} received remote data from an instance of T_x^{Pk}, having *lan* l, which observed a non-locally consistent snapshot at Pk, i.e., that instance of T_x^{Pk} missed the write of some local MPT, T_w^{Pk}, where $w<x$. In this case, though, T_w^{Pk}, before final committing, must have cause the abort of the instance of T_x^{Pk} with $lan=l$. Also, upon its speculative commit, T_w^{Pk} must have sent a SC message to Pi, advertising a LAN that must be necessarily larger than l. Also, T_x^{Pi} can only final commit at Pi after having received the SC of T_w^{Pk}. Upon its reception, though, T_x^{Pi} would detect that it received data from an instance of T_x^{Pk} that has observed an inconsistent local snapshot and abort.

2. T_x^{Pk} observed a locally consistent data at Pk, but received remote data from some sibling T_x^{Pj} that observed either a locally or a globally inconsistent snapshot. If T_x^{Pj} observed a globally inconsistent snapshot, it means that there must exist some sub-transaction, say T_x^{Pm}, which must have externalized a locally inconsistent snapshot to T_x^{Pj} either directly or, indirectly, i.e., T_x^{Pm} must have initiate the propagation of an inconsistent snapshot that affected a chain of sibling sub-transactions that eventually led to the reception of stale data by T_x^{Pj}. Let us assume that the inconsistent local snapshot was externalized by an instance of T_x^{Pm} with $lan=l$ and a $ran=r$, and that the final *lan* and *ran* of T_x^{Pm} are l' and r', respectively. Note that r must be the same *ran* with which T_x^{Pi} committed, as otherwise, this version would not be accepted by T_x^{Pi}. But for T_x^{Pi} to commit, by the inductive assumption, all the transactions preceding Tx at Pi must have already committed on a globally consistent snapshot and must have updated the *lan* of T_x^{Pm} at partition Pi to l'. So, for T_x^{Pi} to commit it must have received remote data from T_x^{Pm} with $lan=l'$ and $ran=r'$. In this case, however, T_x^{Pi} could not have accepted any read from T_x^{Pk} with ran r — a contradiction.

Heterogeneous batches. Recall that, Sparkle deterministically reorders, at each partition, the transaction batch delivered by the dispatching phase into a sequence of groups, $G_1,...,G_n$, where each group is composed by homogeneous MPTs, interleaving in between two consecutive MPT groups, G_i, G_{i+1}, the execution of a number of SPTs. Further, in order to establish the final *lan* of the first transaction of a group G_i (with $i>1$), Sparkle relies on an CC generated by the last transaction serialized before G_i.

We note that the transition from processing the MPTs in the first group, G_1, to processing the following SPTs does not raise any correctness issue: in Sparkle, in fact, a SPT is final committed only if all its preceding transactions (i.e., the MPTs in G_1 in this case) are final committed and processing the first group of MPT in a batch is equivalent to processing a batch of homogeneous MPTs — whose correctness we have discussed above.

The correctness of the transactions in G_i, with $i>1$, can be proved, by induction, using analogous arguments to the ones used above when considering a batch composed solely by homogeneous MPTs. The proof of the base inductive step, though, needs to be revised, since only the first transaction of G_1 is guaranteed to be spared from any abort, but this is false, in general, for G_i if $i>1$.

Let us denote the first transaction of G_i ($i>1$) as T and the set of partitions accessed by T as $P=\{P_1,...,P_n\}$. For any of the sub-transactions T^{P_j} to be final committed at partition $P_j \in P$, P_j must have received a CC from every partition in P, which informs P_j of the final *lan* of every sub-transaction of T^P. We denote the final *lan* of T^P notified by the CC sent by partition P as lan^P. For T^{P_j} to be final committed, not only the *lan* of T_{P_j} must coincide with lan^{P_j}. For each $P_k \neq P_j \in P$, T^{P_j} must have received remote data from an instance of T^{P_k} having $lan=lan^{P_k}$ — which guarantees that T^{P_k} must have observed a locally consistent snapshot. Additionally, the remote data received by T^{P_j} from every sibling must be tagged with the same *ran* — which, as discussed above, ensures that T^{P_k} must have also observed a globally consistent snapshot. This ensures the correctness of T.

4.4 Read-only Transactions

Since read-only transactions do not alter the state of the data store, they can be executed at a single partition's replica and serialized in an arbitrary order with respect to the other transactions,

provided that they observe a consistent snapshot of the data store. As such, in order to minimize overheads, in Sparkle read-only transactions are executed concurrently with the remaining update transactions, but in a non-speculative fashion, i.e., by assigning them a timestamp (*local_id*) associated with a final committed transaction. This allows sparing read-only transactions from the overheads associated with registering themselves among the read dependencies of the keys they read — which becomes unnecessary since, being serialized after a final committed update transaction, read-only transactions are guaranteed to observe a stable snapshot.

It should be noted that while read-only transactions that access a single partition can be freely assigned any serialization order by their local partition, this is not the case for read-only MPTs. In this case, it is in fact necessary to ensure that a read-only MPT is assigned the same serialization order at all the partitions it involves. Sparkle tackles this problem by using a deterministic scheduling policy, which serializes every read-only MPT before any other transaction of its batch — this ensures the stability of the snapshot over which they are executed and allows them to be executed in a non-speculative fashion, analogously to read-only SPTs.

5. EVALUATION

This section is devoted to experimentally evaluate Sparkle, by comparing it with two state of the art PRSM systems, namely S-SMR [8] and Calvin [46], which, as discussed in Sec. 2, adopt radically different techniques to implement a deterministic concurrency control scheme.

S-SMR, analogously to other recent systems [32] essentially circumvents the problem by processing transactions sequentially (using a single thread) at each partition: this avoids any overhead due to concurrency control, but can only exploit the parallelism potential of multi-core architectures by deploying multiple logical partitions (one per available core) at each server. As we will see, this leads to an increase of the frequency of MPTs, which impose a much larger inter-thread synchronization overhead than SPTs.

Calvin implements a deterministic concurrency control by relying on a single scheduling thread to acquire, in a deterministic order, the locks requested by transactions (at the granularity of a single data item) before their execution. As already mentioned, this approach has two fundamental limitations. 1) Its throughput is inherently upper bounded by the rate at which the scheduling thread can acquire lock on behalf of transactions. 2) Its effectiveness is strictly dependant on the ability to be able to predict accurately the transactions' access patterns: failure to do so, causes transactions to be aborted, undergo a new ordering phase and re-executed (until the prediction turns out to be exact). Using the same terminology as in Calvin's paper, we refer, in the following, to transactions whose data accesses cannot be determined statically (e.g., using code analysis techniques [43]), but which depend on the current state of the system, as *dependent transactions*.

Our study aims to answer the following main questions:

- How effective is Sparkle in exploiting the parallelism potential of large multi-core architectures? (Sec. 5.3)
- What throughput gains does Sparkle achieve, with respect to the state of the art techniques, in a typical deployment scenario, like the one illustrated in Fig. 1, where data is fully replicated across data centers, and, within a data center, data is sharded across the nodes of a medium size cluster (composed by 8 servers) (Sec. 5.4)
- How relevant are Sparkle's SC and scheduling techniques to regulate the execution of MPTs? And do the performance gains brought about by processing MPTs in a speculative fashion justify the additional complexity that speculation introduce? (Sec. 5.4.1)
- How does Sparkle's performance scale when deployed over large scale clusters (composed by up to 40 servers) (Sec. 5.4)

5.1 Implementation and setup

We implemented Sparkle and S-SMR, based on Calvin's publicly available code base [3], which was implemented in C++. The original code base uses STL's *unordered_map* as the in-memory backend to store data, which we found out to become the system's bottleneck at high thread counts (i.e., above 20 threads). Therefore, in our implementation, we replaced it with *concurrent_hash* $_{map}$ from Intel's Threading Building Blocks library [2], a highly-scalable library designed for multicore settings. The repository containing the implementation of all protocols and benchmarks are publicly available at `github.com/marsleezm/spec_calvin`.

Except for the experiments reported in Sec. 5.3 and Sec. 5.4.2, all our tests were conducted on the Grid'5000 cluster [1] using 8 *genepi* machines. Each machine has two 4-cores Intel Xeon E5420 QC CPUs, and is equipped with one gigabit ethernet card. Unless otherwise noted, all protocols use three cores for auxiliary tasks needed for the evaluation (e.g., network communication and workload generation); other than that, Calvin dedicates one core for serializing lock requests and four other cores to execute transactions, Sparkle uses five cores to execute transactions, and S-SMR only uses one core to execute transactions. The large scale deployment tests (Sec. 5.3) were conducted using Grid'5000's *paravance* machines, each equipped with two 6-cores Intel Xeon E5-2630 v3 CPUs. To quantify the scalability of Sparkle on large multi-core architectures (Sec. 5.3) we use a machine equipped with two Intel Xeon E5-2648L v4 CPUs, consisting in total of 28 cores (56 hardware threads).

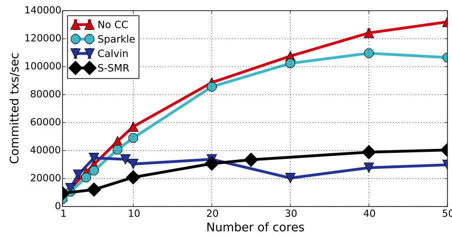
The presented results are the average of three runs. We also report the results' standard deviation, but since the performance of different runs are overall consistent (usually less then 5% of difference), many experiments' standard deviation are not readable. As shown in previous works [46, 44], the use of batching allows to greatly alleviate the load pressure on the (consensus-based) mechanisms used to implement the replication phase (Sec. 3). Thus, in practical settings, the system's throughput is not bottlenecked by the replication phase, but rather by the execution phase — which represents the focus of this work. Therefore, as in previous studies, e.g., [46], to ease deployment, we omit replicating partitions and emulate the replication phase by injecting 200ms delay (to mimic the latency for the execution of consensus across geogrphicaly distributed data-centers).

5.2 Benchmarks

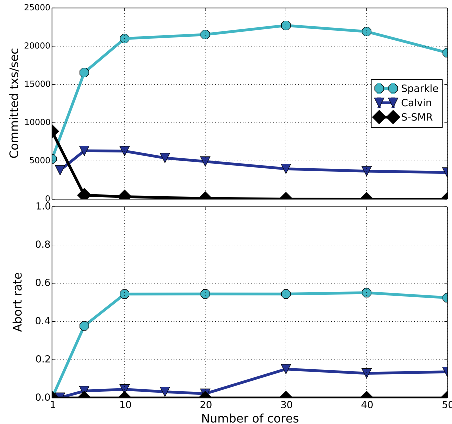
In our study, we use both a synthetic benchmark, which allows us to explicitly control key workload characteristics (e.g., the rate of dependent and distributed transactions), as well as TPC-C [3], a well-known OLTP benchmark that generates more complex and realistic workloads.

Synthetic-benchmark. In this benchmark each partition contains one million keys, split in two sets, which we call "index" and "normal" keys, respectively. All transactions start by reading and updating five index keys selected uniformly at random. If the transaction is a 'dependent transaction', it reads five additional normal keys, whose identity is determined by the values read from the five index keys (i.e., the read- and write-set of dependent transactions can only be determined during execution). Else, if

[3]https://github.com/yaledb/calvin



(a) No conflict.



(b) High conflict.

Figure 6: Single node deployment, TPC-C 90% update workload.

the transaction is non-dependent, it reads and updates five randomly selected normal keys. If a transaction accesses more than one partition, it divides equally its accesses among its involved partitions. For instance, if a dependent transaction accesses two partitions, then it accesses three index keys and three normal keys of a partition and the other two index and two normal key from the second partition. Multi-partition transactions, unless otherwise noted, always access two partitions.

We shape the workloads generated via this synthetic benchmark by varying three parameters: contention level (low and medium contention), percentage of dependent transactions (0%, 1%, 10%, 50% and 100%) and percentage of distributed transactions (0%, 1%, 10% and 50%). Specifically, we control contention by varying the number of index keys of each partition (using the remaining keys as normal keys): in the low contention scenario, each partition uses 50000 index keys; 1000 index keys per partition are used, instead, for the medium contention scenario.

TPC-C. The TPC-C benchmark has been long studied in the literature and we adopt a typical sharding policy [14], whereby data is sharded by warehouse, except for the Item table which, being read-only, is fully replicated across all partitions. This benchmark specifies five transaction profiles: NewOrder, Payment, OrderStatus, StockLevel and Delivery.

NewOrder and Payment are update transactions that access a warehouse hosted on a remote partition with probability 10% and 15%, respectively. OrderStatus and StockLevel transactions are read-only, single-partition transactions (SPTs). Delivery transactions are update SPTs. Finally, NewOrder and Payment are independent transactions, while the other three are dependent transactions.

We consider three different transaction mixes, containing 10%, 50% and 90% update transactions. All transaction mixes always contain only 4% of Delivery transactions, while the other two update and read-only transactions evenly share the rest of the proportion. For example, the 90% update workload consists of 42% of NewOrder and Payment transactions each, 4% of Delivery transactions and 5% of OrderStatus and StockLevel transactions each. Except in Sec. 5.3, we populate 12 warehouses per data partition in all TPC-C experiments.

5.3 Single node deployment

Before testing Sparkle in distributed settings, we focus on single node performance, evaluating its scalability on a large multi-core machine equipped with 56 hardware threads (28 physical cores).

When testing Sparkle and Calvin we deploy a single data partition, and increase the total number of worker threads (and scheduling thread for Calvin) up to 50, dedicating 6 threads to workload generation. Conversely, since S-SMR can only utilize one worker thread per data partition, the only way to let it exploit the parallelism of the underlying architecture is by varying the number of partitions, which we increase up to 50 (using the same amount of data). We use the 90% update TPC-C workload, and adjust the contention level by varying the number of warehouses. In more detail, we consider two extreme scenarios: a very high conflict workload, in which only a single warehouse is populated, and a no conflict workload, in which we populate a large number of warehouses (200) and alter the workload to generate no conflicts (by having concurrent requests access disjoint warehouses). For the no-conflict workload, we consider an additional *NoCC* baseline, i.e., a protocol which implements no concurrency control whatsoever. This represents an ideal baseline that allows us to better understand the scalability limit and overhead of each protocol.

Fig. 6a reports the performance of the considered protocols using the no-conflict workload. As we can see, Sparkle has almost identical throughput to the ideal *NoCC* baseline up to 30 threads, incurring less than 20% overhead with 40 and 50 threads. These results clearly highlight the efficiency and practicality of Sparkle's concurrency control. Conversely, Calvin's throughput only scales up to five threads (one locker thread and four worker threads). At higher thread counts, the scheduling thread turns into the system's bottleneck, severely hindering its scalability. Last but not least, we can see that S-SMR achieve good scalability and outperforms Calvin when using more than 25 threads. However, S-SMR achieves 2.6× lower throughput than Sparkle at 50 threads. This is due to the fact that in this workload, approximately 10% of transactions access a remote warehouse, which on S-SMR is stored on a different partition (which needs to resort to smaller data partitions to enable parallelism), and which can be instead be stored in the same (and only) partition with Sparkle and Calvin. Despite in this test, communication between the sub-transactions of a MPT take place via efficient Unix Domain sockets, MPTs impose a large overhead as the need to exchange data between sibling sub-transactions imposes not only additional load on the memory sub-system but also an inherent synchronization phase between the worker threads of different partitions (which leads to frequent stalls in the processing).

Next, we examine the high conflict workload (Fig. 6). Given the high contention probability, the absolute peak throughput achieved by Sparkle is clearly lower than in the previous scenario. Yet, we observe up to approx. 6× speed-up versus the best baseline, i.e., Calvin, which scales only up to 5 threads, as in the previous workload, before being bottlenecked by the sequencing thread. This striking performance gain is achieved despite, as expectable, Sparkle incurs a high contention rate, given its speculative nature and the high probability of conflicts between of

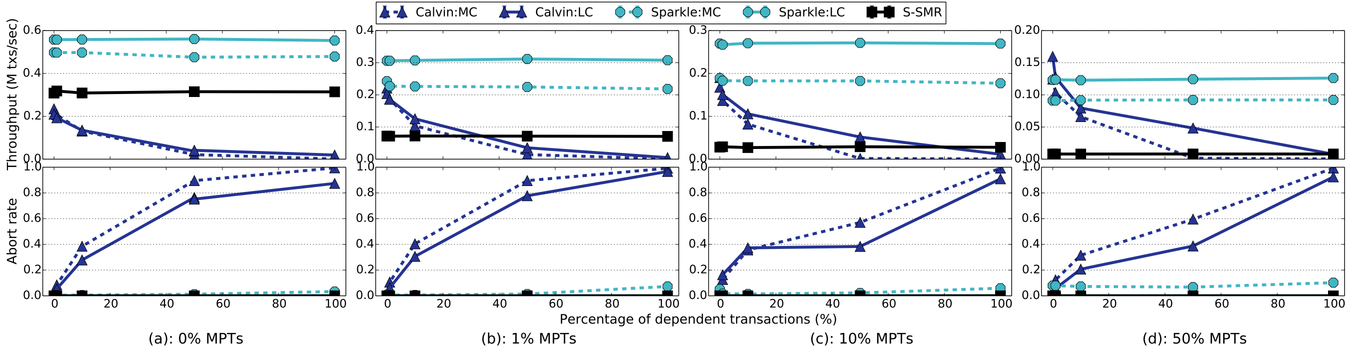


Figure 7: 8 nodes cluster, synthethic benchmark generating workloads with varying contention level, percentage of MPTs and of dependent transaction. *MC* stands for medium contention and *LC* for low contention.

transactions. The most dramatic performance drop, though, is experienced by S-SMR. In this case, in fact, when using more than a single thread, the data (which we recall is populated with a single warehouse) has to be sharded over multiple partitions (in this case we partition by district up to 10 threads, and then using random hashing), forcing most transactions to access more than a single partition. This is particularly onerous for long read-only transactions, such as OrderStatus, which access hundreds of keys and, as such, force the worker threads of different partitions to synchronize hundreds of times to process a single transaction.

5.4 Distributed deployment

Let us now analyze the performance of Sparkle when deployed over a medium scale cluster encompassing 8 machines.

We start by presenting, in Fig. 7, the results for the synthetic benchmark considering four scenarios, which differ by the percentage of MPTs they generate. In each of the 4 plots in Fig. 7 we vary, on the X-axis, the percentage of dependent transactions, and report throughput and abort rate for all the considered solutions when using a low (LC) and medium contention (MC) workload. For the case of S-SMR, since its performance is oblivious to the contention level (given that it processes transactions sequentially at each partition), we only report results for the LC workload.

First, let us discuss Fig. 7a first, which reports results for a workload that does not generate any MPT.

We can see that Sparkle overall achieves the highest throughput, and that, as expected, its performance is slightly reduced in the MC workload, but is not affected by the rate of dependent transactions. In this scenario S-SMR also achieves approx. 60% lower throughput that Sparkle. This can be explained considering that Sparkle (and Calvin) can process transactions concurrently, using all the available cores (5 in this testbed), whereas S-SMR's single thread execution model inherently spares it from the overhead of concurrency control but also intrinsically limits its scalability.

Finally, looking at Calvin's throughput, we can see that its throughput reduces dramatically as the ratio of dependent transaction increases. Nevertheless, even with 0% of dependent transaction, Calvin's throughput is throttled by its scheduling thread, which leads it to achieve lower throughput than both Sparkle and S-SMR. With 100% of dependent transaction, Calvin starts thrashing, as the likelihood for dependent transactions to be aborted (possibly several time) quickly grows even in low/medium conflict workloads. In fact, Calvin needs to execute a, so called, *reconnaissance* phase for dependent transactions to estimate their read- and write-sets, and if the prediction is detected, during transaction execution, to be wrong, these transactions have to be aborted and re-executed. It should be noted that the high frequency of abort of dependent transactions imposes overhead not only to worker threads, but also to Calvin's scheduler thread, which, as already mentioned, plays a critical role for the efficiency of Calvin. Upon each abort and restart of a (dependent) transaction, the scheduler thread has to release and acquire its locks, incurring non-negligible overhead.

Figs. 7b, 7c and 7d report the results obtained when increasing the percentage of MPTs to 1%, 10% and 50%, respectively. The first observation we make is that that S-SMR's throughput drops significantly as the rate of MPT grows. As already mentioned in Sec. 5.3, MPTs incur a large overhead with S-SMR, due to the synchronization they impose between the worker threads of different partitions. Since S-SMR uses a single worker thread per partition, whenever a MPT is forced to block waiting for remote data from a sibling partition, no other transaction can be processed at that partition — unlike in Calvin or Sparkle. In distributed settings, as the communication latency between partitions is strongly amplified (with respect to the single machine scenario considered in Sec. 5.3) the performance toll imposed by MPT also grows radically and S-SMR's throughput is severely throttled by network latency: with 50% MPTs, S-SMR's throughput drops by about 40× compared with the case of no MPTs!

On the other hand, the throughput of Calvin and Sparkle throughput reduces more gradually as the MPT increases. This is due to their multi-threaded execution model, which allows them to activate the processing of different transactins, whenever an MPT is blocked waiting for remote data. Similar to what already observed in Figure 7a, also in this case, the throughput of Calvin drops dramatically in presence of even a small fraction

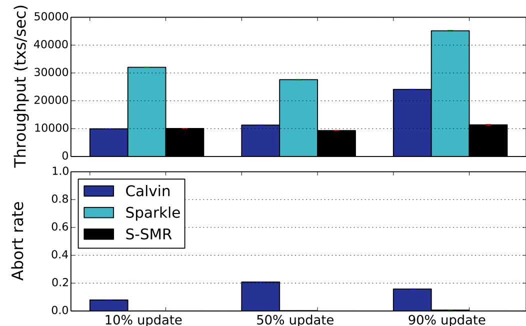


Figure 8: 8 nodes cluster, TPC-C workloads.

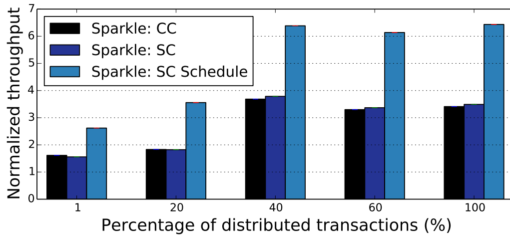


Figure 9: Throughput of $Sparkle : CC$, $Sparkle : SC$ and $Sparkle : SC{+}schedule$, normalized to that of $Sparkle : Cons$, while varying the percentage of MPTs.

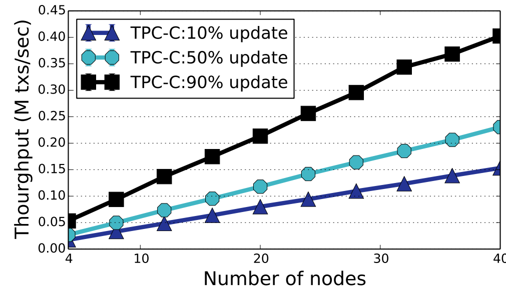


Figure 10: The performance of Sparkle for three workloads varying the scale of the system.

of dependent transactions, approximately by a factor 2× with as low as 10% dependent transactions.

By analyzing Sparkle, we see that although its throughput also reduces with distributed transactions, thanks to its SC and scheduling mechanisms (which we will evaluate more in detail in Sec. 4.2), its throughput is not affected as significantly as with S-SMR. It is also worth noting that in the 50% MPTs scenario and in absence of dependent transactions, Calvin achieves 30% higher throughput than Sparkle. This can be explained by considering that, in this workload, Calvin's throughput is upper bounded by the processing speed of MPTs (which take orders of magnitude longer than SPTs) and not by its scheduling thread. Also, due to its pessimistic/lock-based nature, Calvin does not require MPTs to undergo a confirmation phase. Despite Sparkle strives to minimize the performance impact of the MPTs' confirmation phase (via the combined use of scheduling techniques and of the SC mechanism), this still introduces additional communication overhead. Nonetheless, we highlight that, in the 50% MPT scenario, Sparkle outperforms Calvin as soon as the ratio of of dependent transactions is as large as 1%, achieving an average throughput gain (across the considered MPT ratios) of more than one order of magnitude. Analogous gains are observed also with respect to S-SMR.

Next, we present the results obtained using the TPC-C benchmark. Figure 8 shows that Sparkle outperforms Calvin and S-SMR in all workloads, with peak gains of approx. 3× and approx. 4×, respectively. The key reason why S-SMR achieves relatively poor performance is that these three TPC-C workload generate a small, but not negligible fraction (varying from approx. 1%, for the 10% update workload, to approx. 10%, for the 90% update workload) of MPT transactions. Calvin's performance, instead, can be explained considering that three out of the five transaction profiles are dependent transactions, which impose heavy load on the locking thread and are prone to incur frequent restarts.

5.4.1 Benefits of SC and scheduling

In this section, we intend to shed lights on the the performance benefits brought about by using, either jointly or in synergy, two key mechanisms used by Sparkle to regulate MPT's execution: SC and transaction scheduling. Further, we aim to quantify to what extent the use of speculative transaction processing (in particular allowing MPTs to disseminate speculative data to their siblings) can enhance the throughput of MPT transactions.

To this end, we compare the performance of four Sparkle variations. $Sparkle : Cons$: a conservative variant in which MPTs are only allowed to send remote data to their siblings if they are guaranteed to have observed a locally consistent snapshot, i.e., if their preceding transaction has final committed. This spares MPTs from the need (and cost) of any confirmation, but also throttles down throughput severely as it precludes any form of parallelism between MPTs in execution at the same partition. $Sparkle : EC$: a variant in which we use neither SC nor scheduling, and rely on an EC confirmation mechanism to final commit MPTs. ($Sparkle : SC$), which used SCs but not scheduling. ($Sparkle : SC{+}Schedule$) which uses both SC and scheduling jointly.

We use the low conflict micro benchmark configuration and generate varying ratios of MPTs. For better readability, in Fig. 9 we report the normalized throughput of the three protocols with multi- partition speculative read against $Sparkle : Cons$. The plot allows us to draw three main conclusions. First, all variants achieve significant (up to approx. 3×) with respect to $Sparkle : Cons$, confirming the relevance of using speculative processing techniques to cope with MPTs. Second, unless coupled with scheduling, SC provides no perceivable benefit with respect to a simpler CC approach: without scheduling, most MPTs need to resort to using a CC scheme, hence the throughputs of $Sparkle : EC$: and $Sparkle : SC$ is almost identical. Finally, it allows us to quantify the gains reaped through the joint use of scheduling and SC: up to 2× throughput increase when compared to $Sparkle : EC$.

5.4.2 Large scale deployment

Finally, we evaluate the scalability of Sparkle by using the three previously described TPC-C workloads and increase the number of nodes in the system from 4 to 40. Also in this case, each node hosts a data partition with 12 warehouses, so in this experiment we are scaling both the cluster size and the volume of data. As a consequence, the degree of contention between transactions also remains theoretically constant as the platform's scale grows (and similar to the levels observed for the medium scale cluster reported in Figure 8, and, hence, omitted).

Figure 10 shows that Sparkle scales linearly to 40 nodes, for all three workloads, confirming that the use of speculative transaction processing techniques employed by Sparkle are effective also in large scale data stores and that they do not compromise what is arguably one of the most relevant property of the PRSM approach: its scalability.

6. CONCLUSION

This paper introduced Sparkle, a novel distributed deterministic concurrency control that achieves more than one order of magnitude gains over state of the art PRSM systems via the joint use of *speculative* transaction processing and *scheduling* techniques.

Via an extensive experimental study encompassing both synthetic and realistic benchmarks, we show that 1) Sparkle has negligible overhead compared with a protocol implementing no concurrency control, in conflict-free workloads, 2) Sparkle can

achieve more than one order of magnitude throughput gains, comparing with state of the art PRSM systems, in workloads characterized by high conflict rates and frequent MPTs.

7. REFERENCES

[1] Grid'5000. https://www.grid5000.fr/.

[2] Threading building blocks. https://www.threadingbuildingblocks.org/.

[3] Tpc benchmark-w specification v. 1.8. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[4] A. L. P. N. Alonso. *Database replication for enterprise applications*. PhD thesis, Universidade do Minho, 2017.

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 53–64. ACM, 2010.

[6] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: How well does it actually pound nails. In *The 2nd Workshop on Determinism and Correctness in Parallel Programming (WODET?11)*, 2011.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[8] C. E. Bezerra, F. Pedone, and R. Van Renesse. Scalable state-machine replication. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 331–342. IEEE, 2014.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[11] A. Correia, J. Pereira, and R. Oliveira. AKARA: A flexible clustering protocol for demanding transactional workloads. In *On the Move to Meaningful Internet Systems: OTM 2008, OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, November 9-14, 2008, Proceedings, Part I*, pages 691–708, 2008.

[12] M. Couceiro, D. Didona, L. Rodrigues, and P. Romano. *Self-tuning in Distributed Transactional Memory*, pages 418–448. Springer International Publishing, Cham, 2015.

[13] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues. Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):2942–2955, Nov 2015.

[14] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[16] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 85–96. ACM, 2009.

[17] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. Rcdc: a relaxed consistency deterministic computer. In *ACM SIGPLAN Notices*, volume 46, pages 67–78. ACM, 2011.

[18] J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel, and F. Pedone. Clock-rsm: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 343–354. IEEE, 2014.

[19] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[20] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[21] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *SIGMOD Rec.*, 25(2):173–182, June 1996.

[22] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the Ninth European Conference on Computer Systems*, page 11. ACM, 2014.

[23] S. Hirve, R. Palmieri, and B. Ravindran. Archie: a speculative replicated transactional system. In *Proceedings of the 15th International Middleware Conference*, pages 265–276. ACM, 2014.

[24] T. Hoff. Latency is everywhere and it costs you sales - how to crush it. High Scalability, july, 25, 2009.

[25] R. Jiménez-Peris, M. Patiño Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the 22 Nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 477–, Washington, DC, USA, 2002. IEEE Computer Society.

[26] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[27] T. Kobus, M. Kokocinski, and P. T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 286–296, July 2013.

[28] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

[29] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[30] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[31] J. Li, E. Michael, and D. R. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 104–120. ACM, 2017.

[32] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 104–120, New York, NY, USA, 2017. ACM.

[33] Z. Li, P. V. Roy, and P. Romano. Sparkle: Scalable speculative replication for transactional datastores. Technical Report 4, INESC-ID, May 2018.

[34] Z. Li, P. Van Roy, and P. Romano. Enhancing throughput of partially replicated state machines via multi-partition operation scheduling. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–10. IEEE, 2017.

[35] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[36] P. J. Marandi, M. Primi, and F. Pedone. High performance state-machine replication. In *Dependable Systems and Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 454–465. IEEE, 2011.

[37] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices*, 44(3):97–108, 2009.

[38] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues. Autoplacer: Scalable self-tuning data placement in distributed key-value stores. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(4):19, 2015.

[39] R. Palmieri, F. Quaglia, and P. Romano. Aggro: Boosting stm replication via aggressively optimistic transaction processing. In *Network Computing and Applications (NCA), 2010 9th IEEE International Symposium on*, pages 20–27. IEEE, 2010.

[40] R. Palmieri, F. Quaglia, and P. Romano. Osare: Opportunistic speculation in actively replicated transactional systems. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 59–64. IEEE, 2011.

[41] S. Pande, A. Gavrilovska, and K. Ravichandran. Destm: harnessing determinism in stms for application development. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, pages 213–224. IEEE, 2014.

[42] M. Patiño-Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems (TOCS)*, 23(4):375–423, 2005.

[43] P. Raminhas, M. Matos, and P. Romano. Fine-grained transaction scheduling in replicated databases via symbolic execution. In *12th EuroSys Doctoral Workshop*, EuroDW '18, 2018.

[44] P. Romano and M. Leonetti. Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning. In *Proc. International Conference on Computing, Networking and Communications (ICNC)*, pages 786–792. IEEE, 2012.

[45] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[46] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[47] P. T. Wojciechowski, T. Kobus, and M. Kokociński. State-machine and deferred-update replication: Analysis and comparison. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):891–904, March 2017.