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.

I. Introduction

Nowadays, large-scale online services are faced with a number of challenging requirements. On the one hand, to tame the growing complexity of applications, distributed data storage systems have started embracing strong, transactional, semantics [7]. On the other hand, a number of works [21] have shown that the profitability of large-scale online services hinges on their ability to deliver low 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 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., [42], [5], [32]. The techniques proposed by recent works in this area extend the classic State-Machine Replication (SMR) approach [40], a long-studied 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 [27] to agree, in a fault-tolerant way, on a 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 [20].

Several recent works [42], [5], [32] have focused on addressing what is arguably the key scalability limitation of the classic SMR approach, namely its full replication model, by sharding applications' state across multiple partitions, which are then replicated across a 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 efficiently regulate the execution of transactions that access multiple partitions. While single-partition transactions (SPTs) can be processed at the partitions they access as in classic SMR systems, multi-partition transactions (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 and enforce a transaction serialization order deterministically across replicas.

A simple approach to ensure that, at each partition's replica, the transactions serialization order is equivalent to the final order is to execute all the transactions in a partition's replica sequentially [5], [24]. Unfortunately, this solution limits the maximum throughput achievable by any partition to the processing rate of a single thread, failing to fully untap the performance potential of modern multi-core systems.

Other approaches, like Calvin [42], enable multiple threads to process a partition's transactions concurrently [42], [34], but employ deterministic concurrency control techniques that 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 [1].

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 one order of magnitude through the use of *speculative* transaction processing 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 CPUs — by avoiding inherently non-scalable designs that rely on a single thread for executing [5] or scheduling transactions [42]. It also avoids any assumption on the a-priori knowledge of the transactions' working sets, thus increasing the solution's generality.

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 inconsistent data to remote partitions) 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 can severely hinder throughput.

Sparkle tackles these challenges via two key, novel, techniques, which represent the main contributions of this work: Sparkle's deterministic concurrency control, which 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 way, reducing the frequency and cost of misspeculations.

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.

Via an extensive experimental study, based on both synthetic and standard benchmarks, we show that Sparkle can achieve more than one order of magnitude throughput gains versus state of the art PRSM systems [42], [5], 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. §II discusses related work. §III defines the assumed system model and §IV describes the execution model of generic PRSM systems. §V details the Sparkle protocol, which is experimentally evaluated in §VI. §VII concludes the paper.

II. Related Work

Transactional stores. A large body of works has investigated how to build consistent, yet scalable, transactional stores. Existing systems can be coarsely classified based on whether they adopt the *deferred update replication* (DUR) [23] or the *state-machine replication* (SMR)[27] approaches. In DUR-based systems, e.g. [7], [31], [26], [44], transactions are first locally executed at a single replica and then globally verified, via an agreement protocol based on consensus [23] and/or Two Phase Commit [17]. Speculation has been employed in DUR-based solutions either at the level of the local concurrency control scheme (e.g., exposing pre-committed state rather than blocking processing [31], [36]) or at the consensus level (e.g., skipping communication steps in absence of conflicts among concurrently submitted transactions [35], [26], [44]).

Unlike DUR-systems, with SMR, e.g., [42], [12], replicas first agree on the serialization order of transactions, using consensus-based coordination schemes, and then execute them using a deterministic concurrency control. The DUR and SMR approaches have complementary pros and cons and are fit for different workloads [9], [8], [43]. The focus of this work is on SMR-based systems, which excel in contention-prone workloads, whereas DUR systems can suffer from lock-convoying and high abort rates [43].

Partially-replicated state machines. The PRSM approach [32], [5], [42], [30] extends the classic SMR scheme to support a more scalable partial replication model. Existing PRSM systems rely on diverse techniques to implement a deterministic concurrency control.

Some approaches eliminate the possibility of non-deterministic execution [5], [24] by allowing the execution of only a single thread per partition. This approach spares from the use (and cost) of any concurrency control, but it also inherently limits the maximum throughput achievable by any partition to the processing rate of a single thread. Some works [28], [24] 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 machine. However, this approach can increase significantly the frequency of MPTs, since, when using smaller partitions, it is more likely for transactions to access data scattered over multiple partitions. Accesses to multiple partitions, even if maintained by the same machine, impose severe synchronization overheads 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 (see §VI).

Other systems, e.g., [42], [34], conversely, allow for concurrent execution of transactions and enforce deterministic execution by relying on a single thread to acquire, according to the final order, the locks required by transactions, before executing them. Unfortunately, as we show in Sec. VI, in typical OLTP workloads dominated by short running transactions, the scheduler thread quickly becomes a bottleneck as the degree of parallelism increases. Further, in order to acquire all the locks needed by a transaction before its execution, these solutions require mechanisms for predicting the transaction's data access pattern — a non-trivial problem in complex real-life applications [1]. The solutions proposed in the literature to cope with this issue are quite unsatisfactory: existing techniques either require programmers to conservatively over-estimate the transaction's working set [34] (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* [42], i.e., whose set of accessed data items is influenced by the snapshot they observe.

Sparkle tackles these limitations by combining speculative transaction processing techniques — which exploit out of order processing techniques to enhance parallelism with no 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.

Deterministic execution. The problem of designing efficient deterministic concurrency controls has also been studied for classical SMR systems adopting a full replication model [19], [20], [33], [36], [25]. Some of these works, e.g., [20], [36], [25], employ speculative transaction processing techniques, as in Sparkle. Though, unlike these solutions, Sparkle targets a partial replication model, which, as already discussed, raises additional challenges related to the processing of MPTs. Analogously to Sparkle, Eve [25] incorporates scheduling techniques to maximize the efficiency of speculation. However, unlike Sparkle, Eve's scheduling mechanism requires a priori knowledge on transactions' conflict patterns.

The deterministic concurrency control of Sparkle has relations also with the works on deterministic execution of multi-threaded applications, typically aimed at debugging and testing [2], [3], [10], [11], [39]. These mechanisms intercept *all* non-deterministic events affecting threads' execution (to be later replayed). In the context of SMR/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.

III. System and transaction model

System model. We consider the typical system model assumed by PRSM approaches, e.g., [42], [5], [30], in which application data is sharded across a predetermined number of partitions, each of which is replicated over a set of servers, which we refer to as replication group. 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 provides disaster tolerance, while allowing MPTs to be ordered and executed without requiring communication across data centers [42]. However, our model is generic enough to support scenarios in which certain data partitions may be replicated only in a sub-set of the available data centers.

We assume that servers may crash and that there exists a majority of correct replicas of each partition. While the techniques adopted by existing PRSM systems during the ordering phase are orthogonal to this work, they are normally based on consensus protocols. Therefore, we assume that the synchrony level in the system is sufficient (e.g., eventual synchrony [13]) to allow implementing consensus [14].

Transaction model. Sparkle provides a basic CRUD transactional interface (create/insert, read, update and delete). 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. [42], [5], [30], we assume that the transaction logic is deterministic and that, given a transaction and its input parameters, it is possible to identify which data partitions it accesses. This information is exploited to order and execute transactions only at the data partitions they actually access. 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 transaction execution. Unlike other PRSM solutions, e.g., [42], we do not assume any fine-grained information on the individual data items that transactions access.

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* or *siblings*, and denote the set of partitions involved by an MPT T using the notation *involved*(T). Unlike SPTs, which execute independently at each replica, MPTs require, in the general case, communication among siblings, as they may need to access data stored on remote partitions.

When a sub-transaction reads a local key for the first time, it disseminates the corresponding value to its siblings; when a sub-transaction issues a read to a remote key which has not been received yet, it blocks until the value is received. As remote keys do not need to be maintained locally, writes to remote keys are only applied to a private transaction's buffer (to be available if they are later read by the same transaction) that is discarded after the transaction's commit.

IV. PRSM model

Sparkle is a deterministic distributed concurrency control designed to accelerate the execution phase of a generic PRSM system, e.g., [42], [30], [5], which operates according to the abstract order-then-execute model defined below.

Ordering phase. The protocol used during the ordering phase is irrelevant for Sparkle, provided that the final order it establishes ensures the following properties:

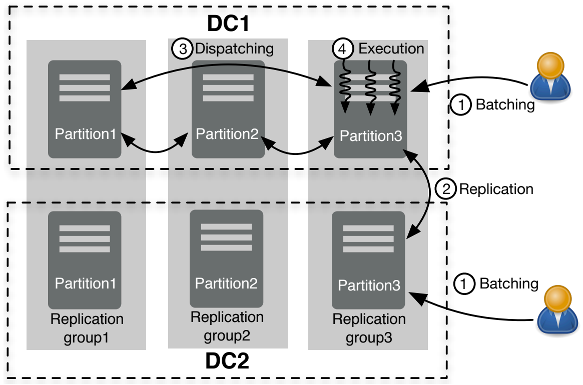


Fig. 1: Example of a typical deployment scenario for Sparkle.

1) all the (correct) replicas of the same partition deliver the same *sequence*, $B_1, \ldots, B_n$, 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 $\mathcal{S} = \{P_1, \ldots, P_n\}$, 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 $\forall P_i \in \mathcal{S}$ $T_2 \rightarrow T_1$;
4) the relation $<$ is acyclic, where $<$ is defined as follows: $T < T'$ iff any partition delivers T and T' in that order.

The ordering phase establishes a total order on the transactions executing at each partition, whereas the transactions executing at different partitions are only partially ordered. We refer to the order established by this phase as *final order*. We call the transactions ordered before/after a transaction T, T's *preceding/following transactions*, respectively.

Existing PRSM systems ensure the above properties in different ways. Calvin, for instance, relies on a two-phase scheme (see Fig. 1). In the first one, called *replication phase*, servers periodically batch, e.g., for 5-10 msecs, the transactions received from clients and submit the resulting batch to an intra-partition consensus service. This merges the transactions gathered by every replica of a given partition and replicates them in a fault-tolerant manner. In the second phase, called *dispatching phase*, all partitions within the same DC exchange the transactions they delivered during the first phase. This ensures that MPTs are delivered at all the partitions that they need to access. Finally, the transactions gathered during the dispatching phase are deterministically sorted to ensure a consistent final order across all the replicas of every partition.

Execution phase. Once the ordering phase is completed, transactions are executed at all the partitions' replicas they involve. As already mentioned, in order to ensure inter-replica consistency, the execution phase must guarantee that, at all the the replicas of a partition, the transactions delivered by the ordering phase are executed according to the same serialization order, i.e., their execution history is equivalent to a common *sequential* history. Sparkle's concurrency control ensures this guarantee, while allowing transactions to be executed concurrently. As such, it ensures *serializability* semantics [4]. Further, if the protocol used during the ordering phase ensures real-time ordering between transactions (i.e., given two transactions T1, T2, where T1 precedes T2 according to real-time order, T1 is serialized before T2 by the ordering phase) then Sparkle globally guarantees *strict serializability*.

Failure handling. Dealing with failures is relatively simple in PRSM-based systems (including Sparkle). Since all correct replicas of a partition deliver the same transactions in each batch, MPTs can fetch remote data from any available replica. In order to provide end-to-end fault-tolerance guarantees, in case the replica originally contacted by a client fails (or is suspected to have failed), the client can contact any other replica provided that some complementary mechanism is employed to ensure exactly-once semantics [16], [37].

V. SPARKLE

This section describes Sparkle's deterministic concurrency control scheme. We start by discussing the processing of SPTs (§V-A) and MPTs (§V-B). Finally, we discuss how to optimize the treatment of read-only transactions (§V-D).

A. *Single partition transactions*

Sparkle strives to achieve two seemingly antagonistic goals: maximizing the parallelism of transaction processing and ensuring that transaction execution is equivalent to a sequential execution complying with the final order.

To maximize parallelism, Sparkle employs a multi-versioned, optimistic concurrency control that imposes no constraints on the processing order of transactions. Denoting with ts the logical timestamp that reflects the final order at a partition, threads select as the next transaction to *start*, the one with the smallest ts value. However, as transactions are processed concurrently, they can be speculatively executed according to a spontaneous, non-deterministic serialization order that contradicts the final order.

To ensure consistency, Sparkle guarantees that a final committed transaction must have observed a snapshot that includes the versions produced by all its preceding transactions (according to the final order). This property is enforced 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 they produced — 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. Misspeculations are detected at run-time, leading to the automatic abort and restart of the affected transactions. Transactions are restarted with the same timestamp to ensure deterministic execution across replicas.

In order to enhance efficiency and reduce the chance of misspeculations, Sparkle incorporates a timestamp-based locking scheme. The timestamp of transactions, i.e. ts, establishes a total order on item versions created by final and speculatively committed transactions, and also defines the visibility of versions: a transaction only reads the latest version produced by speculatively or final committed transactions ordered before it. When writing a data item for the first time, a transaction T locks the data item, which prevents it from being accessed by T's following transactions before T finishes execution. Also, when writing, T inspects the data item's $read\_dependencies$. These register which transactions have already read this data item, and allow T to abort any following transaction that missed T's updates. Correspondingly, when reading, it is checked if a transaction with a lower timestamp has locked the data item: in the negative case, the reader registers its timestamp in $read\_dependencies$ to notify future writers; else, the execution of the reader transaction is suspended till the writer completes.

Next, we provide additional details on the management of SPTs. The pseudo-code, along with the used data structures, as shown in Alg. 1 and Alg. 2.

Start. Upon activation, each transaction initializes three main data structures: its $readset$, $writeset$ and $abort\_flag$. The $readset$ and $writeset$ are private buffers that store the data items read and updated by the transaction, respectively. $abort\_flag$ is used to check whether the transaction has been aborted by other transaction.

Execution. During its execution, a transaction T may read and update multiple data items. Before executing any operation, T checks its $abort\_flag$ to determine if it has been flagged for abort by some preceding transaction. In this case, T is aborted and re-executed (Alg. 1, 2 and 13). Prior to its first update to a data item, T tries to obtain an exclusive lock to it. If the lock is held by a different transaction T' that follows T in the final order (i.e., the ts of T' is larger than that of T), T ejects T' from the lock and sets the $abort\_flag$ of T' to $true$ (Alg. 2, 7-9). Conversely, if the locking transaction T' precedes T, T waits for T' to finish execution (Alg. 2, 5-6). Once T successfully obtains the lock on the data item, it applies the update to its $writeset$ (Alg. 1, 3-11).

While executing a read operation, T first attempts to read from its $writeset$ and $readset$, to return any version it has previously written or read. Else, T redirects its read to the data store and checks the state of the lock guarding the data item it intends to read (Alg. 1, 14-15). Similar to the above locking procedure, T is suspended if the data item is currently being locked by any of its preceding transactions (Alg. 2, 11-12). Otherwise (i.e., the item is not locked, or locked by T's following transactions), T scans the version list and returns the version with the largest timestamp smaller than its ts. Note that this may not be the version that T would observe, had transactions been executed serially according to the final order, as other transactions preceding T may later produce more recent versions. Thus, T appends its ts to the $read\_dependencies$ of the data item (Alg. 2, 13-15). This allows aborting T if a write-after-read conflict is later detected.

Suspended transactions. As mentioned, a transaction T is suspended if it tries to read/update a data item that is currently being locked by a preceding transaction. In that case, the thread executing T can start executing the next unprocessed transaction according to the final order, so to enhance parallelism. T will eventually be unblocked when the contending transactions release the lock requested by T. At this point, the thread responsible of T can resume its execution.

Speculative/final commit. After completing its execution, T attempts to speculatively commit, so to make its writes visible to other transactions. For each data item it updated, T inserts a new version in the item's version chain, timestamped and ordered by its ts (Alg. 1, 19 and 20). Meanwhile, T releases the corresponding lock and 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 $abort\_flag$ to $true$ (Alg. 2, 28 -32). Additionally, T prunes the identifiers of any final committed transaction still tracked in $read\_dependencies$, which are unnecessary as they no longer risk to abort (omitted in the pseudo-code). While applying its updates, if T finds that any of its obtained locks has already been preempted, it aborts itself by removing all inserted versions and releasing any remaining lock (Alg. 2, 25-26). Else, T is considered to be *speculatively-committed* (Alg. 1, 18-22).

Next, T checks if it can final commit, which is only possible if i) all its preceding transactions have already committed and ii) its $abort\_flag$ is still $false$. As T's updates have already been applied in the previous step, the final commit logic is very fast, requiring essentially to only increase the counter that tracks the timestamp of the most recent final committed transaction. Recall, in fact, that the $read\_dependencies$ of final committed transactions are pruned in an opportunistic way by transactions that update those data items in the future (Alg. 1, 24-28).

If T can not be final committed, yet, the thread processing T simply executes the next unprocessed transaction and periodically checks the state of T, to final commit it, if possible.

Abort. T can only be aborted due to *data conflicts with preceding transactions*, either because T missed updates from a preceding transaction, or because any of its locks was preempted by a preceding transaction. If either case occurs, T's $abort\_flag$ is set to $true$. Then, T aborts by releasing all its locks and removing any version it has inserted in the data store (in case T had speculatively committed) (Alg. 1, 29-32 and Alg. 2, 17-23).

B. Multi Partition Transactions

During their execution, the sub-transactions of an MPT disseminate the results of read operations on local data items to the other involved partitions (§III). By letting MPTs execute speculatively, i.e., without waiting for the final commit of their preceding transactions, then a MPT sub-transaction may miss a local data item version not yet produced by a preceding transaction and send inconsistent data to its siblings.

We define a *global consistent* snapshot for a MPT T as the union of the *local consistent snapshots* at all the partitions involved by T, where a local consistent snapshot for T at partition X is obtained by serially committing all the transactions preceding T according to the final order at X.

```
Algorithm 1: Concurrency Control
Data structures associated to a transaction T
Int TS                    ▷ an integer denoting T's final order in its partition.
Data structures associated to an instance $T^*$ of T
Map<KeyID,Value> RS,WS    ▷ read- and write-set of $T^*$.
Bool ABORT_FLAG ▷ indicating if T has been aborted by another tx.
Data structures associated to a thread TH
set<TxInstance> SC_TXS    ▷ txs. speculatively committed by TH.
Data structures associated to each partition
Int NEXT_TX               ▷ the timestamp of the next transaction to commit.

update(TxInstance $T^*$, Key $k$, Value $v$)
  if $T^*$.ABORT_FLAG == true; localAbort($T^*$)
  if $k \in T^*$.WS
    $T^*$.WS[$k$] = $v$
  else
    result = Lock($T^*$, $k$)
    if result == OK
      $T^*$.WS[k] = v
      return OK
    else                  ▷ If lock request failed (i.e. abort returned)
      return result
read(TxInstance $T^*$, Key $k$):
  if $T^*$.ABORT_FLAG == true; localAbort($T^*$)
  if $k \in T^*$.WS return $T^*$.WS.get(k)     ▷ Check if $T^*$ wrote to k.
  if $k \notin T^*$.RS                         ▷ Check if $T^*$ already read k.
    $T^*$.RS.set(k, Read($T^*$, $k$))          ▷ Store loc. value in read-set
  return $T^*$.RS.get(k)
speculativeCommit(Thread $TH$, TxInstance $T^*$)
  for {k, v} $\in T^*$.WS
    if AddVersion($T^*$, k, v) == ABORT
      localAbort($T^*$)
  $TH$.SC_TXS .add($T^*$)
finalCommit(Thread $TH$)
  for $T^* \in TH$.SC_TXS
    if $TH$.NEXT_TX == $T$.TS ant $T^*$.ABORT_FLAG == false
      $TH$.SC_TXS.remove($T^*$)
      delete all metadata of $T^*$
      $TH$.NEXT_TX++
localAbort(TxInstance $T^*$)
  for {k, v} $\in T^*$.WS
    UnlockAndRemove($T^*$, k)
  remove all local data items from $T*$.WS and $T^*$.RS
```

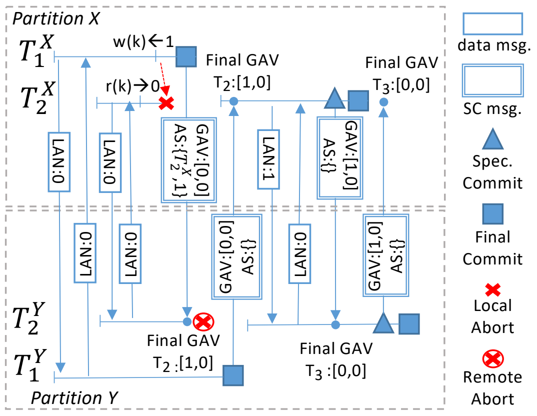


Fig. 2: Exemplifying the execution of MPTs.

The key challenge to ensure safe speculative execution of MPTs lies then in detecting if an MPT instance observed a global consistent snapshot and can, thus, be final committed.

Batches of homogeneous MPTs. To simplify presentation,

```
Algorithm 2: Backend protocol
Data structures associated to each key entry of kv
TxInstance LOCK_TX               ▷ the transaction holding lock on k.
Set<TxInstance> WAIT_TXS ▷ txs. blocked when trying to access k.
Set< {TxInstance, $Int$} > READ_DEPS ▷ txs. that have spec. read k.
List< {Int, Value} > VERSIONS         ▷ timestamped versions for k.

▷ The following operations are atomic per key entry.
Lock(TxInstance $T^*$, Key $k$)
  if kv[k].LOCK_TX.TS == ∅
    kv[k].LOCK_TX = $T^*$; return OK
  elif kv[k].LOCK_TX.TS < $T^*$.TS ▷ Locked by $T^*$'s proceeding tx..
    kv[k].WAIT_TXS.add($T^*$)
  else                              ▷ Locked by $T^*$'s following tx..
    kv[k].LOCK_TX.ABORT_FLAG = true
    kv[k].LOCK_TX = $T^*$; return OK
Read(TxInstance $T^*$, Key $k$)
  if kv[k].LOCK_TX $\neq \varnothing$ and kv[k].LOCK_TX.TS < $T^*$.TS
    kv[k].WAIT_TXS.add($T^*$)
  else                   ▷ Not locked, or locked by $T^*$'s following tx..
    $\{ts, v\}$ = the largest entry in kv[k].VERSIONS with $ts < T^*$.TS
    kv[k].READ_DEPS.add({$T^*$, $ts$})
    return $v$
UnlockAndRemove(TxInstance $T^*$, Key $k$)
  if kv[k].LOCK_TX == $T^*$
    kv[k].LOCK_TX = ∅
    Unblock following transactions of $T^*$ in kv[k].WAIT_TXS
  else
    remove $T^*$'s inserted version from kv[k].VERSIONS
    abort transactions in kv[k].READ_DEP that have read from $T^*$
AddVersion(TxInstance $T^*$, Key $k$, Value $v$)
  if kv[k].LOCK_TX != tx
    return ABORT
  else
    kv[k].LOCK_TX = ∅
    kv[k].versions.append({$T^*$, $v$})
    Abort txs. in kv[k].READ_DEP that missed $T^*$'s update
    Unblock following txs. of $T^*$ in kv[k].WAIT_TXS
    return OK
```

we describe the proposed solution by first assuming that 1) all transactions delivered during the ordering phase are MPTs that access the same set of partitions, noted $\mathcal{P}$, and 2) different batches are never concurrently executed. In the following we use the term *homogeneous MPTs* to denote a set of MPTs that access the same set of partitions. We will later discuss why this assumption is needed and how to cope with generic batches composed by SPTs and heterogeneous MPTs later.

Identifying transactions and snapshots. We define the LAN (*Local Abort Number*) of a *transaction instance* T^X executing at partition X as the number of times that T aborted and restarted at X due to *local conflicts*. Sparkle ensures that the only cause of local aborts for a transaction T_i^X (i denoting the final order of T at X) is a conflict with some local transaction that *precedes* T_i^X in the final order. It follows that when the last transaction, say T_i^X, that precedes a MPT, say T_{i+1}^X, at partition X *final* commits, any instance of T_{i+1} at X (currently active or subsequently activated) is guaranteed not to undergo any further local abort and to observe a locally consistent snapshot. We call the LAN of this instance of T_{i+1}^X the *final* LAN of transaction T_{i+1}^X.

LANs allow for tracking aborts due to local conflicts, but not aborts due to remote conflicts. These occur in case a sibling executing at a remote partition Y had previously sent inconsistent data and has to be restarted. We address this issue by associating with a sub-transaction instance T^X a vector clock, called GAV (*Global Abort Vector*). The GAV of T^X maintains an entry for each partition $Y \in \mathcal{P}$ and it stores: in the entry associated with the local partition X, the LAN of T^X; for every entry associated with a remote partition $Y \neq X$, the LAN of the transaction instance T^Y, running at partition Y, from which T^X received remote data.

The GAV of a transaction instance T^X serves to identify the snapshot it observed and to establish its consistency. Indeed, if the GAV of T^X contains, in each of its entry, the final LAN of every sibling, then T^X must have observed a consistent global snapshot — as this implies that, at every involved partition, T^X observed a local consistent snapshot. We call such a GAV the *final* GAV for T^X, or simply for T, as all siblings of T share the same final GAV.

Determining the final GAV. Sparkle determines the final GAV via a *speculative confirmation* (SC) scheme. When T_i^X *speculatively* commits, T_i^X broadcasts to its siblings a SC message containing its GAV, and an *abort set* which contains the identifier and LAN of every local transaction instance aborted by T_i^X.

Partition X can determine the final GAV for T_i^X only if:

C1. T_{i-1}^X has been final committed.
C2. For each partition $Y \in \mathcal{P}$, X received an SC message from T_{i-1}^Y tagged with the final GAV of T_{i-1}.

When these two conditions hold the final GAV of T_i is computed as follows: for each involved partition Y, the Y-th entry of T_i's final GAV is the largest LAN specified for T_i^Y in the abort set of *any* SC message received from Y.

The above mechanism is defined in a recursive way, as the final GAV of T_i can only be computed once T_{i-1} has final committed. This implies, in its turn, that the final GAV of T_{i-1} must also be known - as MPTs are final committed only after their final GAV is known. The base step of this recursion is the first transaction in the batch, noted T_1^X, which is guaranteed to never abort. As such, all the entries of T_1's final GAV are necessarily equal to zero and T_1^X can be used an initial "anchor" to bootstrap the SC scheme: as T_1^X speculatively commits, it can be immediately final committed; when X receives the SC messages from all the siblings of T_1, since these SC messages are tagged with T_1's final GAV, X can determine the final GAV of T_2, and so forth.

Figure 2 exemplifies a scenario in which T_1^X aborts the first instance of T_2^X due to a local conflict on data item k and notifies partition Y via an SC message (the SC message is sent by T_1^X upon final commit since, being the first transaction of the batch, it cannot abort and omits the speculative commit phase). Upon reception of T_1^X's SC message, Y establishes the final GAV for T_2, i.e., [1,0], and T_2^Y restarts with that GAV. When this instance of T_2^Y speculatively commits, it emits an SC message that is used at partition X to establish the final GAV for T_3. After speculatively committing, the instance of T_2^Y with GAV=[1,0] can be final committed, since its GAV coincides with T^2's final GAV.

Algorithm 3: MPT execution at partition X

```
Data structures associated with an MPT $T$:
Array of int[numPartitions] GAV              ▷ Current known GAV of $T$.
Array of GAV[numPartitions] SCMSG_GAV        ▷ GAVs of the last SC
                                             ▷ ... msg received from each partition
Data structures associated with every instance $T^*$ of an MPT $T$:
map<keyID,value> RS,WS                       ▷ read- and write-set of $T^*$.
map<TID,int> ABORTSET                        ▷ Map storing the LANs of any
                                             ▷...local tx. instance aborted by $T^*$.
read(TxInstance $T^*$, Key $k$):
  if $k \in T^*$.WS return $T^*$.WS.get(k)   ▷ Check if $T^*$ wrote to k.
  if $k \notin T^*$.RS                       ▷ Check if $T^*$ already read k.
    if $k$ is local
      $T^*$.RS.set(k, localRead())           ▷ Store loc. value in read-set
      send <k, $T^*$.RS.get(k), $T$.GAV[$X$]> to $T$'s siblings
    else         ▷ Remote key, wait for its value from corr. partition.
      wait receive <k, v, LAN> s.t. LAN $\geq$ $T$.GAV[k.locPart()]
      $T^*$.RS.set(k,v)                      ▷ Store remote value in read-set.
      if LAN $>$ $T$.GAV[k.locPart()]        ▷ Remote sibling has aborted.
        remoteAbort($T$, k.locPart(), LAN, <k, v>)
  return T.RS.get(k)
localAbort(TxInstance $T^*$, TxInstance $T'^*$)     ▷ $T^*$ aborts $T'^*$.
  $T'$.GAV[$X$]++
  $T^*$.ABORTSET.add($< T', T'[X] >$)
  restart a new instance of $T'$, cloning its RS from $T'^*$...
    ...and removing any local key from it
remoteAbort(Tx $T$, Partition $Y$, Int $lan$, <Key $k$, Value $v$ >):
  $T$.GAV[$Y$]=$lan$                         ▷ Update LAN of tx $T$ at partition $Y$.
  if $\exists$ an active instance $T^*$
    restart a new instance of $T'$, cloning its RS from $T'^*$...
      ...and removing from it any key received from $Y$ except for $k$
tryCommit(TxInstance $T^*$):
  speculativeCommit($T^*$)                   ▷ Spec. apply $T$'s writes to local keys.
  send:<SC, $T^*$.ABORTSET, $T$, $T$.GAV> to $T$'s siblings
  waitFinalGAV($T$)
      ▷ This tx. instance has the final GAV if it did not abort so far.
  finalCommit($T^*$)
waitFinalGAV(Tx $T$):
  $T' \leftarrow T$.getPrecedingTx()
  wait until $T'$ has final committed                 ▷ Cond. C1.
  $\forall Y \in T$.remotePartitions()
    wait until $T'$.GAV = $T'$.SCMSG_GAV.get(Y)       ▷ Cond. C2.
upon receiving <SC, ABORTSET, $T$, GAV$_{msg}$> from partition $Y$
  $T$.SCMSG_GAV[$Y$]$\leftarrow$ GAV$_{msg}$          ▷ Store GAV of last SC from $Y$.
  for each $< T'$, LAN $> \in$ ABORTSET      ▷ For each tx aborted by $T$.
    if LAN $> T'$.GAV[$Y$]:    ▷ Skip aborted txs we already know of
      remoteAbort($T'$, $Y$, LAN,$< \perp, \perp >$)
```

Pseudo-code. Alg. 3 shows the pseudo-code for managing a homogeneous batch of MPTs at partition X. To simplify presentation we assume FIFO-ordered channels. We omit discussing write operations, as these are managed as in SPTs.

Additional Data structures. For the management of MPT, each partition maintains the following data structures (in additional to the ones presented in Alg. 1) for each MPT T: i) GAV: T's currently known Global Abort Vector; ii) SCMSG_GAV: a map that stores, for every sibling sub-transaction T^Y, with $Y \in \mathcal{P}$, the GAV of the most recent SC message received at X from any instance of T^Y. Additionally, for each MPT instance T^* the following data-structures are used: i) RS/WS, which store the transaction instance's read-set and write-set, respectively; ii) the ABORTSET, a map that stores the largest LAN of any local transaction so far aborted by T^*. At any time, at partition X for an MPT T there is at most one active instance T^* that is associated with the current GAV of T at X: upon its activation, T^* is associated with the currently known GAV for T and whenever the GAV of T changes, T^* is aborted and a new instance is restarted associated with the new GAV.

Read logic. When T^* reads a key, it first checks if it previously wrote to or read it. In these two cases the value stored in T^*'s write-/read-set is returned, respectively. Else, i.e., first access to a key, if the key is local, T^* fetches its value from the local storage and broadcasts it to its siblings. This message is tagged with the transaction instance's LAN, which coincides with the local entry of the GAV of T. If the key is hosted at a remote partition, say Y, T^* waits for the key's value from Y and checks if the received LAN is larger than the Y-th entry of the GAV of T. In this case, T^* had previously received stale data from a sibling running at Y, which later aborted. Thus, T^* is aborted and restarted. If the LAN of the value received from Y coincides with the Y-th entry of the GAV of T at X, instead, the value is added to the read-set and is returned.

Handling aborts. When T^* aborts a local transaction instance T'^* (Alg. 3, 13), the local entry of the GAV of T', i.e., its LAN, is increased. Next, T'^* and its LAN are added to the ABORTSET of T^* and a new instance of T' is activated. The read-set of this new instance is initialized with a clone of the read-set of its previous "incarnation", purged of any *local* data. This ensures that the new transaction instance retains any remote data received so far, avoiding re-fetching it remotely.

When X learns about the abort at a remote partition Y of an instance of transaction T with a given LAN (Alg. 3, 18), X accordingly updates the Y-th entry of T's GAV and aborts any local instance of T. The restarted instance of T inherits, in this case, the read-set of its previous incarnation purged of any data previously received from Y, with one exception: if the remote abort is detected when receiving a remote value (Alg. 3, 11), this value belongs to a fresh remote snapshot at Y and can be retained in the read-set.

Commit logic. When T^* completes its execution (Alg. 3, 23), it speculatively commits and broadcasts SC messages to all partitions in $\mathcal{P}$. The SC messages disseminate the ABORTSET of T^*, informing remote partitions about the local transaction instances aborted by T^* at X. Upon reception of a SC message from partition Y (Alg. 3, 34): i) the SCMSG_GAV associated with partition Y is updated with the GAV$_{msg}$ of the transaction instance that sent the SC message; ii) for any transaction T' included in the SC message's ABORTSET, if the corresponding LAN is larger than the Y-th entry of the GAV of T', it means that X detected a new remote abort at Y. Thus, the remoteAbort() method is called.

For T_i^* to be final committed, its GAV must coincide with the final GAV for T_i. This is determined (Alg. 3, 29) after waiting for transaction T_{i-1} (i.e., the transaction immediately preceding T_i) to have final committed (Cond. C1), which implies that the final GAV of T_{i-1} is known. So, to determine the final GAV of T_i, it suffices to wait for the reception, from every remote partition, of an SC message tagged with the final GAV of T_{i-1} (Cond. C2, Alg. 3, 33). After this moment, in fact, no instance of T_i can any longer be aborted at any partition. Thus, if a speculatively committed transaction instance T^* returns from waitFinalGAV() without being aborted, it means that none of the instances of T's preceding transactions have invalidated T^* global snapshot. In this case, T^*'s GAV coincides with the final GAV for T and T^* can be final committed.

Implicit dissemination of SC messages. To reduce the overhead of the SC mechanism, Sparkle exploits a key optimization, not reported in the pseudo-code: instead of sending *ad hoc* SC messages, these are piggybacked on the messages used by MPTs to disseminate the results of read operations.

Dealing with heterogeneous MPTs The correctness of the SC mechanism presented above hinges on the assumption that the batch is composed solely by MPTs accessing the same partitions. This ensures that the first MPT of the batch never undergoes aborts. 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 is executed concurrently with it.

To cope with the above problem, Sparkle only allows executing an MPT, if all its preceding uncommitted transactions are either read-only transactions (ROTs) or homogeneous MPTs of this MPT. While trivially ensuring the correctness of the SC mechanism, if naively employed, this technique can also significantly hinder parallelism. For instance, if three homogeneous MPTs are interleaved by two SPTs, then these three MPTs have to be executed sequentially.

Sparkle tackles this issue via a *scheduling* mechanism, which operates as follows. First, upon delivery of a transaction batch, at the end of the ordering phase, each partition deterministically reorders the MPTs in the batch by grouping them according to the set of partitions they access[1]. The resulting final order is composed by a sequence $G_1, \ldots, G_n$ of transaction groups, where each group G_i contains the transactions that access the same set of partitions. Next, each partition deterministically re-orders its SPTs and ROTs, serializing them in between each pair of consecutive MPT groups, with the goal of "spacing them out". The number of SPTs and ROTs serialized in between two groups are calculated in a deterministic fashion, with the goal of ensuring that each group interval is filled with an even number of SPTs/ROTs. Note that since MPTs can not be executed while there are preceding active SPTs, in between two groups we always place SPTs before ROTs, to space out SPTs and the following MPT group. Note that since only transactions of the same batch can be reordered, and that these are necessarily concurrent, scheduling does not compromise real-time order.

[1]The current prototype uses a single thread to re-order transactions, as in all tested workloads the scheduling thread was never the bottleneck.

C. Correctness arguments

In this section, we discuss the correctness of Sparkle. Specifically, we intend to show that Sparkle's concurrency control enforces a serialization order, which we denote with $<_S$, which can be obtained by applying the deterministic transaction grouping and scheduling rules discussed in §V-B 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 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 or equal i and that write to K.

By contradiction, let us assume that in T_i's last execution before it final commits (i.e. T_i does not get aborted during this execution), T_i reads a key K and fetches a version created by transaction T_f, where $f <_S h$ (i.e., T_i misses the version of K created by T_k and observes an earlier version created by T_f). Depending on whether T_i has updated K before reading, there are two possible scenarios:

- T_i updated K before reading: in this case, 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 before trying to read from other transactions (Alg 1, 14). In fact, in such a case $i = h$, as T_i will read its own updates. This leads to a contradiction.
- T_i did not update K before reading: there are two sub-cases in this scenario: 1) T_i reads K before T_h updates it, and 2) T_i reads K after T_h updates it.
  In the former case, T_i reads the update of T_f, or even other preceding transaction, of K, but when T_h commits (either speculatively or finally), it will abort T_i as T_i missed its update (Alg 2, 30). This contradicts our assumption that this is the last execution of T_i. Note that although T_i may miss T_h's updates and get aborted multiple times, we are only interested in the fact that T_i does not miss T_h's updates in its final execution.
  In the latter case, T_h may or may not have final committed yet. If T_h has final committed, T_i can directly read its update on K. Otherwise, since T_h has updated K but has not final committed, it must be still holding lock on K, thus T_i will be blocked until T_h final commits, and then T_i read T_h's update on K (Alg 2, 11). Therefore, in both scenarios, we have shown that in T_i's last execution, it must read from T_h instead of from T_f, which contradicts the assumption.

Multi-partition transactions of homogeneous batches. Let us now consider the scenario of batches containing homogeneous MPTs that involve the same set of partitions, and assume different batches are never concurrently executed. We intend to show that any MPT T_i is final committed, only if it observes a globally consistent snapshot.

The foundation of the proof are the C1 and C2 properties that we presented in §V-B, which are as follows: C1) T^X_{i-1} has been final committed, C2) for each partition $Y \in \mathcal{P}$, X received an SC message from T^Y_{i-1} tagged with the final GAV of T_{i-1}. Thus, we construct our proof in two parts. Nevertheless, as discussed, guaranteeing this two conditions only ensure that Sparkle can determine the final GAV of a sub-transaction T^X_i of an MPT T_i, but not directly proving that it must observe a globally consistent snapshot. Thus, we construct our proof in two parts. We first prove that if a sub-transaction has final committed with its final GAV, then it is guaranteed to observe a globally-consistent snapshot; then we prove that by enforcing C1 and C2 for a sub-transaction, we can correctly calculate its final GAV.

Observing a globally-consistent snapshot. By contradiction, we assume that a sub-transaction T^X_i final commits with its final GAV, after having observed an inconsistent snapshot. Without loss of generality, assume that for a specific key K stored in partition Y, T^X_i final commits after observing a version with value V_1, but later T^Y_i sent it with a new version with value V_2.

T^Y_i can only send V_2 to T^X_i if it gets aborted after sending V_1. Nevertheless, recall that by our assumption, T^X_i final commits with its final GAV; since its final GAV contains the final LAN of T^Y_i (according to the definition of final GAV), T^Y_i can not undergo any local abort after sending V_1, which was sent while it has received its final LAN. Therefore, T^Y_i can only send V_2 due to a remote abort, i.e., being aborted due to observing an inconsistent remote snapshot, i.e., it was executed upon a non-final, speculative snapshot sent by a sibling transaction, and then it received more recent data and got aborted.

However, as already described in §III and §V-B, a sub-transaction only disseminates the data it locally read to its remote partitions. This means that the data a sub-transaction disseminated to other partitions are always from snapshots containing some of its preceding transactions. As we assume that T^Y_i only sent V_1 after all its preceding transactions have final committed and T^Y_i, it means V_1 is already from a snapshot that contains all of T^Y_i local preceding transactions. Nevertheless, T^Y_i also sent V_2 only after all its local preceding transactions have final committed, thus V_1 and V_2 must be from the same snapshot and can not be different values. This leads to a contradiction to our assumption.

Calculating the final GAV *using C1 and C2.* Before presenting the proof, we note that T^X_1 is a special case: by assumption, transactions of different batches are never concurrently executed, thus T^X_1 and its siblings will never undergo abort. Therefore, the final LAN of T^X_1, as well as all its siblings, must be 0. In other words, all sub-transactions of T_1 can always be directly final committed after their execution. Thus, we only prove that using C1 and C2, we can calculate the final GAV for any sub-transaction T_i^X, where $i \neq 1$.

By contradiction, let us assume that for a sub-transaction T_i^Y, its final LAN is m; nevertheless, by leveraging C1 and C2, the calculated final LAN for T_i^Y (the Y's entry of T_i's calculated final GAV) is n, where $m \neq n$.

First, it is easy to see that $Y \neq X$. This is because C1 ensures that T_i^X only final commits after T_{i-1}^X final commits. By the time T_{i-1}^X final commits, all transactions that can possibly local abort T_i^X have all final committed, thus T_i^X would have known its local final LAN, meaning $m = n$. This contradicts the assumption.

Therefore, Y must be a remote partition to T_i^X. In such case, let us assume that T_h^Y is the transaction that local aborted T_i^Y and caused its LAN to be incremented from $m-1$ to m. According to Alg. 3 lines 15, one of T_h^Y's SC message must contain the final LAN of T_i^Y. It is easy to see that by combining C1, C2 and the assumed FIFO-ordered channel (§ V-B), T_i^X must know the final LAN of T_i^Y.

1) Enforcing C1 means that T_h^X must have been final committed – thus, partition X must know the final GAV of T_h before it can final commit T_i^X.
2) By applying $C2$ recursively, we know that before final committing T_{h+1}^X, partition X must have already received the SC message tagged with the final GAV of T_h^Y. As we assume that $h < i$, thus $h+1 \leq i$, it follows that before final committing T_i^X, we must have already received the SC message tagged with the final GAV.
3) The FIFO-ordered guarantee ensures that when partition X has received the SC message tagged with the final GAV of T_h^Y, it must have also received all previous SC messages from T_h^Y.

Therefore, we can see that before partition X can final commit T_i^X, it must have already received the SC message that contains the final LAN of T_i^Y and have already included it into its GAV (Alg. 3, 11 and 38). This means that $m = n$. This contradicts our assumption that $m \neq n$.

Heterogeneous batches. Recall that Sparkle deterministically reorders, at each partition, each delivered transaction batch into a sequence of groups each composed by homogeneous MPTs, and interleave two consecutive MPT groups with a number of SPTs followed by read-only transactions. Let us consider if the execution of transactions still enforces a serialization order determined by the scheduled order.

First, ROTs are always guaranteed to read globally-consistent snapshots, no matter whether they are re-ordered or concurrently executed with other transactions. This is because Sparkle always executes ROTs on a stable snapshot, i.e., the one containing all transactions up to the previous batch.

Secondly, the correctness of SPTs is not affected either, as we have proven that SPTs always read the snapshot containing all preceding transactions, no matter the preceding transactions are SPTs or MPTs. Thus, the above correctness argument can be directly applied to re-ordered SPTs.

Finally, as described in §V-B, each transaction group can only be concurrently executed with ROTs and following SPTs. As Sparkle's concurrency control ensures that no transaction can ever be aborted by its following transaction or ROTs, the execution of an MPT group will not be interfered by its following SPTs or ROTs. Thus, the execution of each MPT in a group can only be affected, i.e. aborted, by its preceding transaction in the same group. Therefore, we can basically regard an MPT group as a homogenous batch, and thus directly apply the above correctness of homogeneous MPT batches here.

Real-time ordering As described in §IV, if the ordering protocol ensures real-time ordering between transactions, Sparkle guarantees strict serializability. Nevertheless, Sparkle also employs deterministic scheduling mechanism that strives to reorder transactions to enhance transaction processing throughput. Let us prove that the Sparkle still guarantees real-time ordering guarantee in spite of the scheduling mechanism. By contradiction, let us assume that there are two transactions T_1 and T_2; T_2 is started after T_1 has finished, but T_2 is ordered before T_1. While there can be various definitions for the 'start' and 'end' time of transactions, here we consider them in terms of the end users: the start time of a transaction refers to the time the transaction is submitted by the client to the system, and the end time is the time when the client is notified by the system about the completion of the transaction. Since T_2 is started after T_1 has finished, T_2 must be ordered in later batches than T_1 – by the time T_1 has finished execution, the transactions to be included in T_1's batch must have been decided. Recall that Sparkle's scheduling mechanism only reorders transaction within the same batch, thus a transaction can at most be ordered in front of the batch. Therefore, T_2 is always ordered T_1, as it must be ordered after all transactions of T_1's batch. This contradicts our assumption.

D. Read-only Transactions

Since ROTs 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, provided that they observe a consistent snapshot of the data store. To minimize overheads, in Sparkle ROTs are executed concurrently with the remaining update transactions, but in a non-speculative fashion, i.e., by assigning them a timestamp associated with a final committed transaction. This allows sparing ROTs 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, ROTs are guaranteed to observe a stable snapshot.

While single partition ROTs 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 necessary to ensure that a read-only MPT is assigned the same serialization order at all the partitions it involves. Sparkle tackles this problem through a deterministic scheduling policy, which serializes every read-only MPT before any other transaction of their 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.

VI. Evaluation

This section is devoted to experimentally evaluate Sparkle, by comparing it with two state of the art PRSM systems, namely S-SMR [5] and Calvin [42].

This study aims to answer the following main questions:

- How effective is Sparkle in exploiting the parallelism potential of large multi-core architectures? (Sec. VI-C)
- What throughput gains does Sparkle achieve in a typical deployment (like the one illustrated in Fig. 1), where data is fully replicated across data centers and sharded across the nodes of a medium size
- How efficient is the management of MPTs? And do the performance gains brought about by processing MPTs in a speculative fashion justify the additional complexity that speculation introduce? (Sec. VI-D1)
- How does Sparkle's performance scale when deployed over large scale clusters (up to 40 servers) (Sec. VI-D)

A. Implementation and setup

We implemented Sparkle and S-SMR, based on Calvin's code base [41]. The original code base uses STL's *unordered_map* as the in-memory back-end to store data, which we found out to become the system's bottleneck at high thread counts. Therefore, in our implementation, we replaced it with *concurrent_hash_map* from Intel's TBB library [22]. The repository containing the code used in this study is publicly accessible [29].

To quantify the scalability of Sparkle on large multi-core architectures (§VI-C) we use a machine equipped with two Intel Xeon E5-2648L v4 CPUs, consisting in total of 28 cores (56 hardware threads). All other experiments were conducted on the Grid'5000 cluster [18] using 8 *genepi* machines, each of which has two 4-cores Intel Xeon E5420 QC CPUs. 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 presented results are the average of three runs. We also report the results' standard deviation, but since the differences in performance across different runs are usually within 5%, in various plots, standard deviations are not visible.

As in prior work [42], we emulate the ordering phase by injecting a 200ms delay and use 10 milliseconds batch time. To avoid overloading the system, we adjust the arrival rate to be 10%-20% larger than the maximum sustainable throughput (determined via a preliminary test). Therefore, batch sizes vary depending on the workload, ranging from 10s to 100s of transactions. Omitting the ordering phase allows for focusing the evaluation on scenarios where throughput is bottlenecked by the execution phase. This is typically the case when one employs batching techniques [15], [38] to increase the maximum throughput sustainable by the ordering phase. As for the choice of the delay of the ordering phase, we argue that using smaller values would reduce user perceived latency but it would not affect the throughput of the considered solutions.

B. Benchmarks

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 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%). 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, partitions use 50000 index keys; 1000 index keys per partition are used, instead, for the medium contention case.

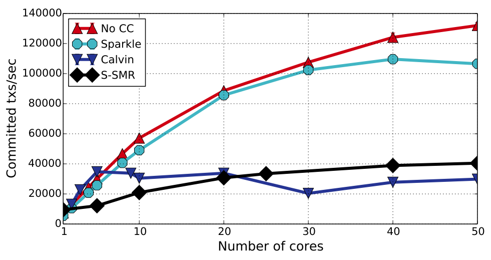
TPC-C. The TPC-C benchmark [6] has five transaction profiles: NewOrder, Payment, OrderStatus, StockLevel and Delivery. NewOrder and Payment are update transactions that access a warehouse hosted on a remote partition with probability 10% and 15%, respectively. OrderStatus and StockLevel transactions 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 (i.e., their working set depends on the database state and cannot be predicted statically).

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. Except in §VI-C, we populate 12 warehouses per data partition in all TPC-C experiments.

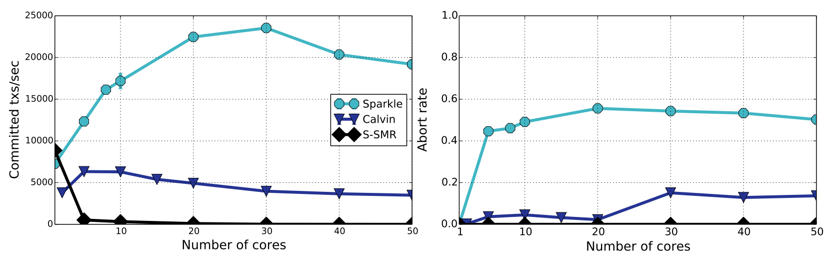
C. Single node deployment

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.

When testing Sparkle and Calvin we deploy a single data partition, and increase the total number of worker threads up to



(a) No conflict.



(b) High conflict.

Fig. 3: Single node deployment, TPC-C 90% update workload.

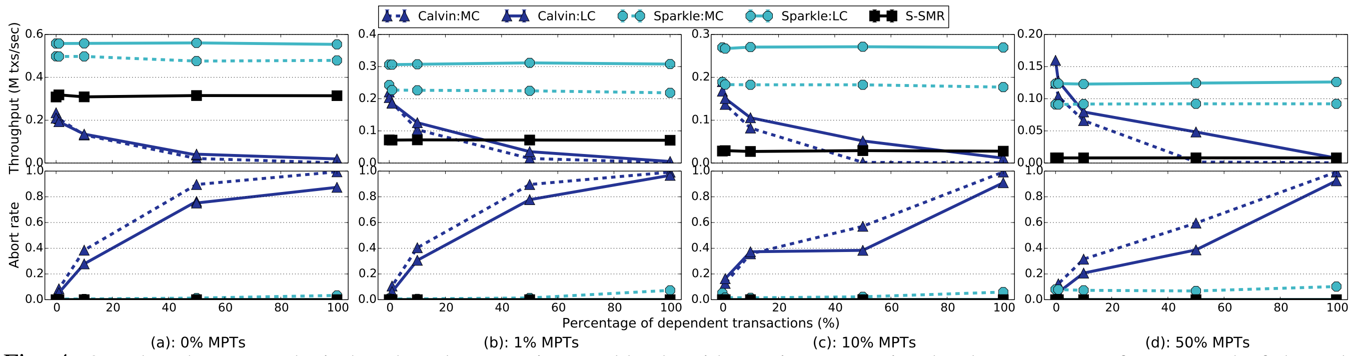


Fig. 4: 8 nodes cluster, synthetic benchmark generating workloads with varying contention level, percentage of MPTs and of dependent transaction. *MC* stands for medium contention and *LC* for low contention.

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 to generate 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. 3a 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 with S-SMR may be stored on a different partition (unlike Sparkle and Calvin, which do not need to use multiple partitions per node to enable parallelism). 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 data exchanges between sibling sub-transactions impose a synchronization phase between the worker threads of different partitions and leads to frequent stalls in the processing.

In the high contention workload (Fig. 3b), 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 transactions. The most dramatic performance drop, though, is experienced by S-SMR. In this case, when using more than a single thread, the data (which 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 force the worker threads of different partitions to synchronize hundreds of times to process a single transaction.

D. 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. 4, the results for the synthetic

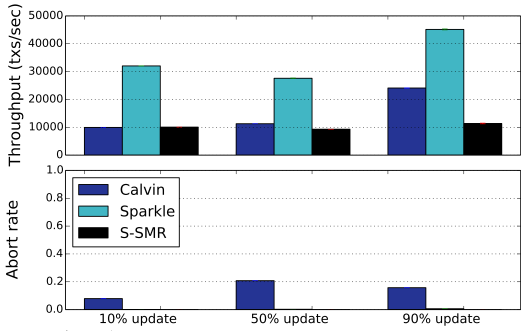


Fig. 5: 8 nodes cluster, TPC-C workloads.

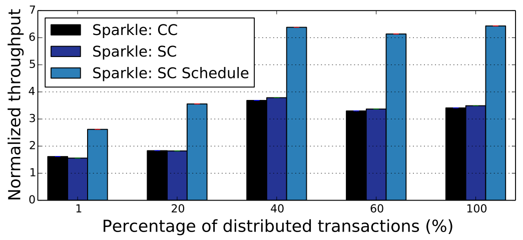


Fig. 6: Throughput of $Sparkle : CC$, $Sparkle : SC$ and $Sparkle : SC + schedule$, normalized to that of $Sparkle : Cons$, while varying the percentage of MPTs.

benchmark considering four scenarios, which differ by the percentage of MPTs they generate. In each of the 4 plots in Fig. 4 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. 4a 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 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 than 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 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 transactions, Calvin thrashes, 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 turns out to be wrong during execution, these transactions have to be aborted and re-executed. Note that the high frequency of abort of dependent transactions imposes overhead not only to worker threads, but also to Calvin's scheduler thread – upon each abort and restart of a (dependent) transaction, the scheduler thread has to release and acquire its locks, incurring non-negligible overhead.

Figs. 4b, 4c and 4d 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 §VI-C, 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 §VI-C) 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!

The throughput of Calvin and Sparkle throughput reduces more gradually as the MPT increases. This is because both of them allow activating the processing of different transactions, whenever an MPT is blocked waiting for remote data. Similar to what already observed in Figure 4a, also in this case, the throughput of Calvin drops dramatically in presence of even a small fraction 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, 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 5 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.

1) Benefits of SC and scheduling: Next, we conduct an experiment aimed to quantify 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 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 variants:

• *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:CC*: in which, as in Sparkle, MPTs disseminate to their siblings the data they read locally in a speculative fashion. Unlike Sparkle, though, this variant uses a conservative confirmation (CC) scheme, which sends confirmation messages only when transactions final commit, and not when they speculatively commit. The CC scheme is significantly simpler than SC, as, with CC, a transaction generates exactly one confirmation message, and not an a priori unknown number, as it is the case for SC. However, with CC, a partition can send the confirmation for its $i+1$-th transaction, only upon final committing its 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. 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.

• *Sparkle:SC*, which uses SCs but not scheduling;

• *Sparkle:SC+Schedule*, which uses SC and scheduling.

We use the low conflict micro benchmark configuration and generate varying ratios of MPTs. For better readability, in Fig. 6 we report the normalized throughput of the three protocols allowing speculative reads across partitions against *Sparkle:Cons*. The plot allows us to draw three main conclusions. First, all variants achieve significant (up to approx. $3\times$) w.r.t. *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:CC* 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 $2\times$ throughput increase when compared to *Sparkle:CC*.

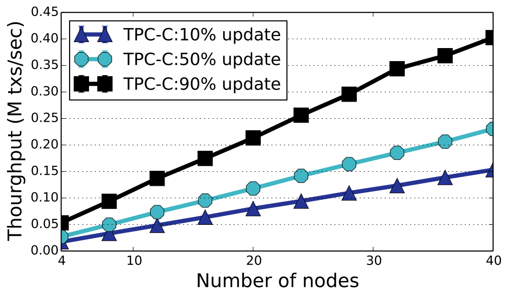


Fig. 7: The performance of Sparkle for three workloads varying the scale of the system.

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 5, and, hence, omitted).

Figure 7 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.

VII. 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.

References

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

[2] C. Basile, Z. Kalbarczyk, and R. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings of the 33th International Conference on Dependable Systems and Networks*, pages 149–158, June 2003.

[3] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: How well does it actually pound nails. In *Proceedings of the 2nd Workshop on Determinism and Correctness in Parallel Programming*, 2011.

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

[5] C. E. Bezerra, F. Pedone, and R. V. Renesse. Scalable state-machine replication. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 331–342, June 2014.

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

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 261–264, Hollywood, CA, 2012. USENIX Association.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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 *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 343–354. IEEE, 2014.

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

[14] 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.

[15] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 233–242, Aug 1997.

[16] S. Frølund and R. Guerraoui. e-transactions: End-to-end reliability for three-tier architectures. *IEEE Trans. Softw. Eng.*, 28(4):378–395, Apr. 2002.

[17] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM International Conference on Management of Data*, pages 173–182, New York, NY, USA, 1996. ACM.

[18] Grid'5000. https://www.grid5000.fr/, 2018.

[19] 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.

[20] 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.

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

[22] Intel. Threading building blocks. https://www.threadingbuildingblocks.org/, 2018.

[23] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 477–484, July 2002.

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

[25] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 237–250, Hollywood, CA, 2012. USENIX.

[26] 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.

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

[28] 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.

[29] Z. Li. Sparkle codebase. https://github.com/marsleezm/spec_calvin.

[30] 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.

[31] Z. Li, P. Van Roy, and P. Romano. Transparent speculation in geo-replicated transactional data stores. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 255–266. ACM, 2018.

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

[33] R. Palmieri, F. Quaglia, and P. Romano. Aggro: Boosting stm replication via aggressively optimistic transaction processing. In *Proceedings of the Ninth IEEE International Symposium on Network Computing and Applications*, pages 20–27, July 2010.

[34] 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.

[35] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

[36] S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues. Specula: Speculative replication of software transactional memory. In *Proceedings of the 31st IEEE International Symposium on Reliable Distributed Systems*, pages 91–100, 2012.

[37] F. Quaglia and P. Romano. Ensuring e-transaction with asynchronous and uncoordinated application server replicas. *IEEE Transactions on Parallel and Distributed Systems*, 18(3):364–378, March 2007.

[38] P. Romano and M. Leonetti. Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning. In *Computing, Networking and Communications (ICNC)*, 2012.

[39] M. M. Saad, M. J. Kishi, S. Jing, S. Hans, and R. Palmieri. Processing transactions in a predefined order. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 120–132, New York, NY, USA, 2019. ACM.

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

[41] A. Thomson. Calvin codebase. https://github.com/yaledb/calvin.

[42] 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.

[43] 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.

[44] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 263–278. ACM, 2015.