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 misspeculations.
1 Introduction

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

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

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

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

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

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

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

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

Speculation is used in Sparkle to allow transactions to be processed “out of order”, i.e., to be tentatively executed in a serialization order that may potentially differ from the one established by the replica coordination phase. Thanks to speculative execution, not only can Sparkle take full advantage of modern multi-core micro-processors — by avoiding inherently non-scalable designs that rely on a single thread for executing [5] or scheduling transactions [47]. It also avoids any assumption on the a-priori knowledge of the transactions’ working sets — which increases the solution’s generality and widens the scope of its scalability.

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

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

- The deterministic concurrency control employed by Sparkle combines optimistic techniques with a timestamp-based locking scheme. The former aims to enhance parallelism. The latter increases the chances that the spontaneous serialization order of transactions matches the one established by the replica coordination phase, and allows for detecting possible divergences in a timely fashion — thus reducing the frequency and cost of misspeculation.

- Sparkle strives to remove the inter-partition confirmation phase of MPTs from the critical path of execution of other transactions via two complementary approaches: i) controlling, in a deterministic way, the final order of transactions, so as to schedule MPTs that access the same set of partitions consecutively; ii) taking advantage of this scheduling technique to establish the correctness of MPTs via a distributed coordination phase, which we call Speculative Confirmation (SC). SC is designed to minimize overhead, by exploiting solely information opportunistically piggybacked on remote read messages exchanged by MPTs, and maximize parallelism, by removing the MPT coordination phase from the critical path of transaction processing.

Through an extensive experimental study, encompassing two state of the art PRSM techniques [47] [5] and based on both synthetic and standard benchmarks, we show that Sparkle can achieve more than one order of magnitude throughput gains, while ensuring robust performance even when faced with challenging workloads characterized by high contention and frequent MPTs.

The reminder of the paper is organized as follows. Sec. 2 reports the related work of Sparkle. Sec. 3 describes our system model and state-machine replication. Sec. 5 details the Sparkle protocol. Sec. 6 presents the results of the experimental evaluation of Sparkle. Finally, Sec. 7 concludes the paper.

2 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) [25] or the state-machine replication (SMR) [30] approaches. In DUR-based systems, e.g., [7, 34, 28, 49], transactions are first locally executed at a single replica and then globally verified, via an agreement protocol based on consensus [25] and/or Two Phase Commit [19]. 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 [34, 41]) or at the consensus level (e.g., skipping communication steps in absence of conflicts among concurrently submitted transactions [40, 28, 49]).

Unlike DUR-systems, with SMR, e.g., [47, 14], 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, 48]. 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 [48].
Partially-replicated state machines. The PRSM approach \[36, 5, 47, 33\] 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 \[3, 26\] 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 \[31, 26\] have argued that this limitation can be circumvented by using a larger number of smaller partitions, delegating each partition to a different thread of the same 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 Sec. 6).

Other systems, e.g., \[47, 39\], 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. 6 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 \[39\] (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 \[47\], 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 mis-speculations.

Deterministic execution. The problem of designing efficient deterministic concurrency controls has also been studied for classical SMR systems adopting a full replication model \[21, 22, 38, 41, 27\]. Some of these works, e.g., \[22, 41, 27\], 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 \[27\] 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 \[2, 3, 37, 12, 13, 45\]. 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.

### 3 System and transaction model

#### System model

We consider the typical system model assumed by PRSM approaches, e.g., \[47, 5, 33\], in which application’s 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 [47]. 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 existing PRSM systems adopted during the ordering phase of transactions 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 [15]) to allow implementing consensus [16].

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. [47, 5, 33], 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., [47], 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 $\text{involved}(T)$. 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.

4 PRSM model

Sparkle is a deterministic distributed concurrency control designed to accelerate the execution phase of a generic PRSM system, e.g., [47, 33, 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:

1. all the (correct) replicas of the same partition deliver the same sequence, $B_1,...,B_n$, of transaction batches, where each batch contains the same totally ordered set of (single- or multi-partition) transactions;
2. if an MPT $T$ is delivered in the $i$-th batch by a partition, then $T$ is delivered in the $i-\text{th}$ 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 $S = \{P_1,...,P_n\}$, $T_1$ and $T_2$ are ordered in the same way by all the (correct) servers that replicate any partition in $S$, i.e., either $\forall P_i \in S \, T_1 \rightarrow T_2$ or $\forall P_i \in S \, T_2 \rightarrow T_1$;
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 ms, 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, 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 $T_1, T_2$, where $T_1$ precedes $T_2$ according to real-time order, $T_1$ is serialized before $T_2$ 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 [18, 42].

5 Sparkle

This section describes Sparkle’s deterministic concurrency control scheme. We start by discussing the processing of SPTs (5.1) and MPTs (5.2). Finally, we discuss how to optimize the treatment of read-only transactions (§5.4).

5.1 Single partition transactions

To maximize parallelism, Sparkle employs a multi-versioned, optimistic concurrency control that imposes no constraints on the processing order of transactions. Denoting with local $\bar{L}$s the logical timestamp that reflects the final order at a partition, threads select as the next transaction to start, the one with the smallest local $\bar{L}$s 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. 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. local $\bar{L}$s, 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 that 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 the pseudo-code and additional details on the management of SPTs.

**Start.** Upon activation, each transaction initializes three main data structures: its readset, write-set and abort_flag (Alg1, 1-3). The readset and write-set are private buffers that store the read and updated data items, 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. 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 local_ids of $T'$ is larger than that of $T$), $T$ aborts its execution. Conversely, if the locking transaction $T'$ precedes $T$, $T$ waits for $T'$ to finish execution. Once $T$ successfully obtains the lock on the data item, it applies the update to its write-set.

While executing a read operation, $T$ first attempts to read from its write-set 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. Similar to the above locking procedure, $T$ is suspended if the data item is currently being locked by any of its preceding transactions. 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 local_ids. 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, unless there are no uncommitted transactions preceding $T$, $T$ appends its local_ids to the read_dependencies of the data item. This allows $T$ to be aborted 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 local_ids, and releases the corresponding lock. Then, $T$ checks the read_dependencies tracked by this data item and aborts any (therein registered) transaction with a larger timestamp (as they missed $T$’s update on this item) by setting their abort_flag to true. 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. 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. Else, $T$ is considered to be speculatively-committed.

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.

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).

### 5.2 Multi Partition Transactions

During their execution (Sec. 5), the sub-transactions of an MPT disseminate the results of read operations on local data items to the other involved partitions. By letting MPTs execute speculatively, i.e., without waiting for the final commit of their preceding transactions, 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 $X$.

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, 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 $T^X_i$ (i denoting the final order of $T$ at $X$) is a conflict with some local transaction that *precedes* $T^X_i$ in the final order. It follows that when the last transaction, say $T^X_{i+1}$, that precedes a MPT, say $T^X_i$, at partition $X$ final commits, any instance of $T^X_{i+1}$ 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 $T^X_i$ the *final* LAN of transaction $T^X_{i+1}$.

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 $Y \in \mathcal{P}$, which stores 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 a twofold purpose: i) uniquely identifying a transaction instance, and ii) determining the global consistency of the snapshot it observed.

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, so called, *speculative confirmation* (SC) scheme. When $T^X_1$ speculatively commits, $T^X_1$ 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 $T^X_1$.

Partition $X$ can determine the final GAV for $T^X_i$ only if:

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

When these two conditions hold the final GAV of $T^X_i$ is computed as follows: for each involved partition $Y$, the $Y$-th entry of $T^X_i$’s final GAV is the largest LAN specified for $T^Y_i$ 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 $T^X_i$ can only be computed once $T^X_{i-1}$ has final committed. This implies, in its turn, that the final GAV of $T^X_{i-1}$ 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 $T^X_1$, which is guaranteed to never abort. As such, all the entries of $T^X_1$’s final GAV are necessarily equal to zero and $T^X_1$ can be used an initial “anchor” to bootstrap the SC scheme: as $T^X_1$ speculatively commits, it can be immediately final committed; when $X$ receives the SC messages from all the siblings of $T^X_1$, since these SC messages are tagged with $T^X_1$’s final GAV, $X$ can determine the final GAV of $T^X_2$, and so forth.

Figure 2 exemplifies a scenario in which $T^X_1$ aborts the first instance of $T^X_2$ due to a local conflict on data item $k$ and notifies partition $Y$ via an SC message (the SC message is sent by $T^Y_1$ upon final commit since, being the first transaction of the batch, it cannot abort and omits the speculative commit phase). Upon reception of $T^X_1$’s SC message, $Y$ establishes the final GAV for $T^Y_2$, i.e., $[1,0]$, and $T^Y_2$ restarts with that GAV. When this instance of $T^Y_2$ speculatively commits, it emits an SC message that is used at partition $X$ to establish the final GAV for $T^X_3$. After speculatively committing, the instance of $T^X_3$ with GAV $[1,0]$ can be final committed, since its GAV coincides with $T^X_2$’s final GAV.

**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 discussion write operations, as these are managed as in SPTs.

### Data structures. Each partition maintains two data-structures 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 $X$ by 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 $T^*$ (l. 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 instace 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 (l. 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 (l. 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 (l. 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 
receipt of a SC message from partition $Y$ (l. 23): i) the 
SCMSG_GAV associated with partition $Y$ is updated 
with the newly received GAV 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 $T^*$ to be finally committed, its GAV must coincide 
with the final GAV for $T$. This is determined (l. 23) after 
waiting for transaction $T_{i-1}$ (i.e., the transaction imme-
diately 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, l. 23). After 
this moment, in fact, no instance of $T_i$ can any longer be 
aborted at any partition. Thus, if a speculatively commit-
ted transaction instance $T^*$ returns from waitFinalGAV() 
without being aborted, it means that none of the instances 
of $T_i$’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 finally 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.

5.3 Correctness arguments

In this section, we discuss the correctness of Sparkle. 
Specifically, we intend to show that Sparkle’s concurrency 
control ensures serializability and that it enforces a 
serialization order, which we denote with $<_S$, which can 
be obtained by applying the deterministic transactions 
grouping and scheduling rules discussed in ?? to the final 
order determined by the dispatching phase.
We start by discussing the case of batches consisting solely of SPTs. Next, we consider the case of batches composed by a single group of homogeneous MPTs. Finally, we analyze the case of generic batches composed by SPTs and heterogeneous MPTs.

### Single-partition transactions

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

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

### Multi-partition transactions

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

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

Next, we show that if all the MPTs preceding \( T_x \)
(T_{1P1}^{x} \ldots T_{xP1}^{n}) observe a globally consistent snapshot, then also T_{xP1}^{z} does. By contradiction, let us assume that T_{xP1}^{z} (T_{x}^{z}’s sub-transaction at partition P_{1}) final commits after observing an inconsistent snapshot. Using the same arguments used for the case of SPTs, it is easy to show that T_{xP1}^{z} cannot final commit if it observes a locally inconsistent snapshot (condition C1). We focus, therefore, on proving that T_{xP1}^{z} cannot final commit after having received data T_{xP1}^{pk}, active at partition P_{k}, which has observed a non-globally consistent (condition C2) snapshot. There are two cases to consider:

1. T_{xP1}^{P1} received remote data from an instance of T_{xP1}^{pk}, having lan l, which observed a non-locally consistent snapshot at P_{k}, i.e., that instance of T_{xP1}^{pk} missed the write of some local MPT, T_{wP1}^{k}, where w < x. In this case, though, T_{xP1}^{p1}, before final committing, must have cause the abort of the instance of T_{xP1}^{pk} with lan = l. Also, upon its speculative commit, T_{xP1}^{p1} must have sent a SC message w_{x} to P_{i}, advertising a LAN that must be necessarily larger than l. Also, T_{xP1}^{P1} can only final commit at P_{i} after having received the SC of T_{xP1}^{pk}. Upon its reception, though, T_{xP1}^{P1} would detect that it received data from an instance of T_{xP1}^{pk} that has observed an inconsistent local snapshot and abort.

2. T_{xP1}^{P1} observed a locally consistent data at P_{k}, but received remote data from some sibling T_{xP1}^{Pj} that observed either a locally or a globally inconsistent snapshot. If T_{xP1}^{Pj} observed a globally inconsistent snapshot, it means that there must exist some sub-transaction, say T_{xP1}^{pm}, which must have externalized a locally inconsistent snapshot to T_{xP1}^{Pj} either directly or, indirectly, i.e., T_{xP1}^{pm} must have initiate the propagation of an inconsistent snapshot that affected a chain of sibling sub-transactions that eventually led to the reception of stale data by T_{xP1}^{Pj}. Let us assume that the inconsistent local snapshot was externalized by an instance of T_{xP1}^{pm} with lan = l and a ran = r, and that the final lan and ran of T_{xP1}^{pm} are l' and r', respectively. Note that r must be the same ran with which T_{xP1}^{Pj} committed, as otherwise, this version would not be accepted by T_{xP1}^{Pj}. But for T_{xP1}^{P1} to commit, by the inductive assumption, all the transactions preceding Tx at P_{i} must have already committed on a globally consistent snapshot and must have updated the lan of T_{xP1}^{pm} at partition P_{i} to l'. So, for T_{xP1}^{P1} to commit it must have received remote data from T_{xP1}^{pm} with lan = l' and ran = r'. In this case, however, T_{xP1}^{P1} could not have accepted any read from T_{xP1}^{pk} with ran r — a contradiction.

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

We note that the transition from processing the MPTs in the first group, G_{1}, to processing the following SPTs does not raise any correctness issue: in Sparkle, in fact, a SPT is final committed only if all its preceding transactions (i.e., the MPTs in G_{1} in this case) are final committed and processing the first group of MPT in a batch is equivalent to processing a batch of homogeneous MPTs — whose correctness we have discussed above.

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

Let us denote the first transaction of G_{i} (i > 1) as T and the set of partitions accessed by T as P = \{P_{1},...,P_{n}\}. For any of the sub-transactions T_{P1}^{i} to be final committed at partition P_{j} ∈ P, P_{j} must have received a CC from every partition in P, which informs P_{j} of the final lan of every sub TRANSACTION of T_{P1}^{i}. We denote the final lan of T_{P1}^{i} notified by the CC sent by partition P as lan_{P1}^{i}. For T_{P1}^{j} to be final committed, not only the lan of T_{P1}^{i} must coincide with lan_{P1}^{i}. For each P_{k} ≠ P_{j} ∈ P, T_{P1}^{i} must have received remote data from an instance of T_{P1}^{i} having lan = lan_{P1}^{i} — which guarantees that T_{P1}^{i} has observed a locally consistent snapshot. Additionally, the remote data received by T_{P1}^{i} from every sibling must be tagged with the same ran — which, as discussed above, ensures that T_{P1}^{i} must have also observed a globally consistent snapshot. This ensures the correctness of T.
5.4 Read-only Transactions

Since read-only transactions do not alter the state of the data store, they can be executed at a single partition's replica and serialized in an arbitrary order with respect to the other transactions, provided that they observe a consistent snapshot of the data store. As such, in order to minimize overheads, in Sparkle read-only transactions are executed concurrently with the remaining update transactions, but in a non-speculative fashion, i.e., by assigning them a timestamp (local_id) associated with a final committed transaction. This allows sparing read-only transactions from the overheads associated with registering themselves among the read dependencies of the keys they read — which becomes unnecessary since, being serialized after a final committed update transaction, read-only transactions are guaranteed to observe a stable snapshot.

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

6 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 [47], which, as discussed in Sec. 2, adopt radically different techniques to implement a deterministic concurrency control scheme.

S-SMR, analogously to other recent systems [42] essentially circumvents the problem by processing transactions sequentially (using a single thread) at each partition: this avoids any overhead due to concurrency control, but can only exploit the parallelism potential of multi-core architectures by deploying multiple logical partitions (one per available core) at each server. As we will see, this leads to an increase of the frequency of MPTs, which impose a much larger inter-thread synchronization overhead than SPTs.

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

Our study aims to answer the following main questions:

- How effective is Sparkle in exploiting the parallelism potential of large multi-core architectures? (Sec. 6.3)
- What throughput gains does Sparkle achieve, with respect to the state of the art techniques, in a typical deployment scenario, like the one illustrated in Fig. 1, where data is fully replicated across data centers, and, within a data center, data is shards across the nodes of a medium size cluster (composed by 8 servers) (Sec. 6.4)
- How relevant are Sparkle's SC and scheduling techniques to regulate the execution of MPTs? And do the performance gains brought about by processing MPTs in a speculative fashion justify the additional complexity that speculation introduce? (Sec. 6.4.1)
- How does Sparkle's performance scale when deployed over large scale clusters (composed by up to 40 servers) (Sec. 6.4)

6.1 Implementation and setup

We implemented Sparkle and S-SMR, based on Calvin's publicly available code base [1] which was implemented in C++. 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

---

[1] https://github.com/yaledb/calvin
counts (i.e., above 20 threads). Therefore, in our implementation, we replaced it with \texttt{concurrent_hash_map} from Intel’s Threading Building Blocks library \cite{threaddb}, a highly-scalable library designed for multicore settings. The repository containing the implementation of all protocols and benchmarks are publicly available at \url{github.com/marsleezm/spec_calvin}.

Except for the experiments reported in Sec. 6.3 and Sec. 6.4.2, all our tests were conducted on the Grid’5000 cluster \cite{grid5000} using 8 \textit{genepi} machines. Each machine has two 4-cores Intel Xeon E5420 QC CPUs, and is equipped with one gigabit ethernet card. Unless otherwise noted, all protocols use three cores for auxiliary tasks needed for the evaluation (e.g., network communication and workload generation); other than that, Calvin dedicates one core for serializing lock requests and four other cores to execute transactions, Sparkle uses five cores to execute transactions, and S-SMR only uses one core to execute transactions. The large scale deployment tests (Sec. 6.3) were conducted using Grid’5000’s \textit{paravance} machines, each equipped with two 6-cores Intel Xeon E5-2630 v3 CPUs. To quantify the scalability of Sparkle on large multi-core architectures (Sec. 6.3), we use a machine equipped with two Intel Xeon E5-2648L v4 CPUs, consisting in total of 28 cores (56 hardware threads).

The presented results are the average of three runs. We also report the results’ standard deviation, but since the performance of different runs are overall consistent (usually less then 5\% of difference), many experiments’ standard deviation are not visible.

As in prior work \cite{prior_work}, 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 \cite{batching} \cite{batching2} 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.

6.2 Benchmarks

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

\textbf{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%). Specifically, we control contention by varying the number of index keys of each partition (using the remaining keys as normal keys): in the low contention scenario, each partition uses 50000 index keys; 1000 index keys per partition are used, instead, for the medium contention scenario.

TPC-C. The TPC-C benchmark has been long studied in the literature and we adopt a typical sharding policy [10], whereby data is sharded by warehouse, except for the Item table which, being read-only, is fully replicated across all partitions. This benchmark specifies five transaction profiles: NewOrder, Payment, OrderStatus, StockLevel and Delivery.

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

We consider three different transaction mixes, containing 10%, 50% and 90% update transactions. All transaction mixes always contain only 4% of Delivery transactions, while the other two update and read-only transactions evenly share the rest of the proportion. For example, the 90% update workload consists of 42% of NewOrder and Payment transactions each, 4% of Delivery transactions and 5% of OrderStatus and StockLevel transactions each. Except in Sec. 6.3 we populate 12 warehouses per data partition in all TPC-C experiments.

6.3 Single node deployment

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

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

Fig. 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 on S-SMR is stored on a different partition (which needs to resort to smaller data partitions to enable parallelism), and which can be instead be stored in the same (and only) partition with Sparkle and Calvin. Despite in this test, communi-
cation between the sub-transactions of a MPT take place via efficient Unix Domain sockets, MPTs impose a large overhead as the need to exchange data between sibling sub-transactions imposes not only additional load on the memory sub-system but also an inherent synchronization phase between the worker threads of different partitions (which leads to frequent stalls in the processing).

Next, we examine the high conflict workload (Fig. 3). Given the high contention probability, the absolute peak throughput achieved by Sparkle is clearly lower than in the previous scenario. Yet, we observe up to approx. 6× speed-up versus the best baseline, i.e., Calvin, which scales only up to 5 threads, as in the previous workload, before being bottlenecked by the sequencing thread. This striking performance gain is achieved despite, as expectable, Sparkle incurs a high contention rate, given its speculative nature and the high probability of conflicts between of transactions. The most dramatic performance drop, though, is experienced by S-SMR. In this case, in fact, when using more than a single thread, the data (which we recall is populated with a single warehouse) has to be sharded over multiple partitions (in this case we partition by district up to 10 threads, and then using random hashing), forcing most transactions to access more than a single partition. This is particularly onerous for long read-only transactions, such as OrderStatus, which access hundreds of keys and, as such, force the worker threads of different partitions to synchronize hundreds of times to process a single transaction.

6.4 Distributed deployment

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

We start by presenting, in Fig. 4, the results for the synthetic benchmark considering four scenarios, which differ by the percentage of MPTs they generate. In each of the 4 plots in Fig. 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, as expected, its performance is slightly reduced in the MC workload, but is not affected by the rate of dependent transactions. In this scenario S-SMR also achieves approx. 60% lower throughput 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 inherently spares it from the overhead of concurrency control but also intrinsically limits its scalability.

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

Figs. 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 the throughput of S-SMR’s drops significantly as the rate of MPTs grows. As already mentioned in Sec. 6.3, MPTs incur a large overhead with S-SMR, due to the synchronization they impose between the worker threads of different partitions. Since S-SMR uses a single worker thread per partition, whenever a MPT is forced to block waiting for remote data from a sibling partition, no other transaction can be processed at that partition — unlike in Calvin or Sparkle. In distributed settings, as the communication latency between partitions is strongly amplified (with respect to the single machine scenario considered in Sec. 6.3, the performance toll imposed by MPT also grows radically and S-SMR’s throughput is severely throttled by network latency: with 50% MPTs, S-SMR’s throughput drops by about 40× compared with the case of no MPTs!

On the other hand, the throughput of Calvin and Sparkle throughput reduces more gradually as the MPT increases. This is due to their multi-threaded execution model, which allows them to activate the processing of different transactions, whenever an MPT is blocked waiting for remote data. Similar to what already observed in Figure 4, 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, thanks to its SC and scheduling mechanisms (which we will evaluate more in detail in Sec. ??), 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 6 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 workloads 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.

6.4.1 Benefits of SC and scheduling

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

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

We use the low conflict micro benchmark configuration and generate varying ratios of MPTs. For better readability, in Fig. 6 we report the normalized throughput of the three protocols with multi-partition speculative read against Sparkle:Cons. The plot allows us to draw three main conclusions. First, all variants achieve significant (up to approx. 3×) with respect to Sparkle:Cons, confirming the relevance of using speculative processing techniques to cope with MPTs. Second, unless coupled with scheduling, SC provides no perceivable benefit with respect to a simpler CC approach: without scheduling, most MPTs need to resort to using a CC scheme, hence the throughputs of Sparkle:EC: and Sparkle:SC is almost identical. Finally, it allows us to quantify the gains reaped through the joint use of scheduling and SC: up to 2× throughput increase when compared to Sparkle:EC.

6.4.2 Large scale deployment

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

7 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


