On Fault-tolerant and High Performance Replicated Transactional Systems

Sachin Hirve

Preliminary Examination Proposal submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

October 17, 2014
Blacksburg, Virginia

Keywords: Distributed Transaction Memory, Fault-tolerance, Active Replication,
Distributed Systems, On-line Transaction Processing

On Fault-tolerant and High Performance Replicated Transactional Systems

Sachin Hirve

(ABSTRACT)

With the recent technological developments in last few decades, there is a notable shift in the way business/consumer transactions are conducted. A majority of transaction now a days fall in Online Transaction Processing (OLTP) category, where transactions are triggered over the internet and transactional systems working in the background ensure that these transactions are processed. One of the most important requirements of these OLTP transaction systems is dependability.

Replication is a common technique that makes the services dependable and therefore helps in providing reliability, availability and fault-tolerance. Active replication based replicated transaction systems exploit full replication to avoid service interruption in case of node failures. Deferred Update Replication (DUR) and Deferred Execution Replication (DER) represent the two well known transaction execution models for replicated transactional systems. Under DUR, transactions are executed by clients before a global certification is invoked to resolve conflicts against other transactions running on remote nodes. On the other hand, DER postpones the transaction execution until the agreement on a common order of transaction requests is reached. Both DUR and DER require a distributed ordering layer, which ensures a total order of transactions even in case of faults.

In today's distributed transaction processing systems, performance is of paramount importance. Any loss in performance results e.g. latency resulting from slow processing of a client request or slow page load, results in loss of revenue for businesses. Although DUR model perform good when conflicts are rare, it is found to be worst impacted by high conflict workload profiles. In contrast, immunity from percentage of conflicts within transactions make DER an attractive choice of transaction execution model, but its serial execution results in limited performance and its total order layer for serializing all the transactions results in moderately high latencies. In addition to it, total order layer poses scalability challenges i.e. it does not scale with increase in system size.

In this dissertation, we propose multiple innovations and system optimizations to enhance the overall performance of replicated transactional systems. First, in HiperTM we exploit the time between the instance when a client broadcasts its request and the instance when its order is finalized, to speculatively execute the request and commit it when the final order arrives. To achieve this goal, we extend S-Paxos with optimistic delivery, resulting in OS-Paxos and build a transactional system based on DER model. HiperTM uses a novel, speculative concurrency control protocol called SCC, which processes write transactions serially, minimizing code instrumentation (i.e., locks or CAS operations). When a transaction is optimistically delivered by OS-Paxos, its execution speculatively starts, assuming the optimistic order as the processing order. For read-only transaction processing we use Multi-Version objects, which helps to execute write transactions in parallel, while eliminating the possible conflicts between read and write transactions.

Second, we designed and implemented Archie which improves over HiperTM in two aspects. As the first contribution, Archie includes a highly optimized total order layer that mixes optimistic-delivery and batching thus allowing the anticipation (thanks to the reliable optimistic notification) of a big amount of work (thanks to the batching) before the total order is finalized. Second contribution in Archie is a novel speculative parallel concurrency control that processes transactions speculatively with high parallelism, upon their optimistic notification, enforcing the same order as the sequence of optimistic notifications.

Both HiperTM and Archie perform well upto a certain number of nodes in the systems, beyond which the leader becomes the bottleneck. This motivates the design of Caesar, which is a transactional system based on a novel multi-leader partial order protocol. Caesar enforces a partial order on the execution of transactions according to their conflicts, by letting non-conflicting transactions to proceed in parallel and without enforcing any synchronization during the execution (e.g., no locks).

Finally, we stumbled upon the idea that not all read-only workloads require up-to-date data and application specific freshness and content-based constraints could be exploited to service read-only transactions to achieve high scalability. We designed Dexter, a replication framework which services the read-only requests according to the freshness guarantees specified by the application.

Our first major post-preliminary research goal is to optimize the DUR replication model. DUR systems pay high cost of local and remote aborts in case of high contention on shared objects, due to which their performance is adversely affected. Exploiting the knowledge of client's transaction locality, we propose to incorporate the benefits of state machine approach to scale-up the distributed performance of DUR systems. As our second contribution, we propose to incorporate HTM for transaction execution in replicated transaction systems, thereby eliminating the overheads associated with software solutions e.g., read-set, write-set, and multi-versioning etc. Since HTM does not have a notion of transaction order, ensuring the transaction execution compliant with total order is an interesting challenge that we would like to address. Lastly, in a replicated transactional system, transaction's latency also depends on the communication steps required by global ordering protocol to finalize its order. Fast decisions involving only two communication steps are optimal, but require the same set of dependencies observed by a quorum of nodes. Since this condition can not be guaranteed due to non-deterministic nature of message exchange among different nodes, achieving fast decisions is not always possible. As the last effort, we propose to design a multi-leader partial order protocol for ring based network which always ensures the fast decision, thereby improving the latency and performance of transactional systems.

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385.

Contents

List of Figures

List of Tables

Acknowledgement

We would like to thank the authors of S-Paxos [8] and PaxosSTM [117] for their willingness to share their projects' source codes.

Chapter 1

Introduction

With the technological developments in last few decades, we have seen a notable transformation in the way business transactions are conducted. For an example, no matter which country are you in, with the internet access and few computer clicks you can buy a merchandise from any part of the world within a few seconds. While customer only observes his/her purchase, there are a lot more activities that happen in background, such as updating the inventory of available items, charging the credit card of customer etc. This all is made possible by transactional processing systems which work in the background without the knowledge of regular customer.

Transactions are defined as a logical unit of work, which could be composed of multiple actions on some data, but appears to be indivisible and instantaneous for other transactions. When transactions successfully complete, they commit i.e. they change the data state and make it visible for other transactions. If they fail, they roll-back i.e. undo the changes made to data by different actions contained in transaction. From a long time, Database management systems (DBMS) have been seen as default model for processing transactions. Database systems have effectively exploited the hardware for concurrency for decades. Databases execute multiple queries simultaneously and possibly on several available processors to achieve good performance.

A database transaction supports *atomicity, consistency, isolation and durability* properties, also known as ACID properties. *Atomicity* property ensures that either the transaction completes successfully or none of its actions appear to execute. *Consistency* is application specific property which guarantees that transaction moves from one consistent state to another according to invariants on underlying data storage system. *Isolation* property requires that transactions do not observe changes made to shared data by other concurrent transactions, not yet completed. Lastly *durability* requires that once the transaction completes successfully, changes made by it are permanent (i.e. stored on stable storage).

Satisfying these properties has an associated cost to pay and even after meeting these requirements, services provided by these systems are prone to disruption from faults. Even though *durability* property promises that data changes are permanent, crash of the processing element (node) makes the system unavailable which is undesirable, specially in today's time when there are large scale transaction processing systems which have service level agreements (SLAs) to satisfy and any deviation from SLAs leads to revenue loss. Replication promises to address this problem by providing availability without any compromise on performance. Using replication, in case of a process failure, other processes can still deliver the request without service interruption and without the knowledge of clients.

1.1 Replication Model

Replication makes the data or services redundant and therefore helps in improving reliability, accessibility or fault-tolerance. Replication could be classified as: data replication, where identical data is stored at multiple storage locations; or computation replication, where same task is executed multiple times. A computational task is typically replicated in space, i.e. executed on separate devices, or it could be replicated in time, if it is executed repeatedly on a single device. Application and platform requirements play a part in selection of a particular approach for the distributed system under consideration. We focus on data replication as it also provides high availability. We treat objects as the quantum of data in the view of transactional systems.

Replication could be categorized depending on the degree of data replication provided in the system i.e. *no-replication, partial replication,* and *full replication.* Degree of replication denotes the number of object copies in the system and it impacts overall performance and data availability. *No-replication* model contains only one copy of data object in a system of multiple nodes, thereby has a replication degree 1. On the other hand, *full replication* model places each object at all nodes (replicas), resulting in a replication degree of N in a system consisting of n replicas. *Partial replication* model lies between these two extremes and could have degree of replication between 1 and N, which is selected by considering the system's requirements. As *no-replication* model does not provide better availability in presence of faults, we only focus on *partial replication* and *full replication* models in following discussion.

Partial replication paradigm allows transaction processing in the presence of node failures, but as data objects copies are hosted on a subset of replicas, the overhead paid by transactions for looking-up latest object copies at encounter time limits the achievable performance. Current partial replication protocols [87, 100] report performance in the range of hundreds to tens of thousands transactions committed per second, while centralized STM systems have throughput in the range of tens of millions [26, 27]. *Full replication* approach annuls the cost of object look-up and benefits from local execution since each data object is available at all replicas, but to ensure replica consistency, it requires a common serialization order (CSO) over transactions which itself involves network interactions with other replicas.

Both *partial-* and *full replication* mechanisms have their own pros and cons. Though *full replication* gives the benefit of local execution on replicated objects, while building the large scale systems, increased storage for hosting all data objects rules-out it as a viable option. In this case, *partial replication* not only becomes the viable option with better availability, but it also provides added computing power due to sheer number of nodes processing only a subset of transactions. In a typical large scale system, both *partial-* and *full replication* models could co-exist. For example, in geo-replicated system, different data centres work under *partial replication* approach for fault-tolerance, whereas nodes within individual data centre follow *full replication* model to ensure high availability and exploit the higher bandwidth of local area network for arriving at CSO to achieve high performance.

Orthogonal to the above classification, depending on how and when the updates are applied at different replicas, replication techniques could be classified as: *active* and *passive replication*. In *active replication* [102], client requests are ordered by an ordering protocol and each replica individually executes requests. Consistency is guaranteed since replicas deterministically process each request in the same order given the same initial state. In *passive replication*, also known as *primary-backup replication*, a replica (called primary) receives client requests and executes them. Subsequently primary updates the state of other (backup) replicas and sends back the response to client. If the primary replica fails (crashes), one of the backup replicas takes the role of primary and helps the system to make progress. *Primary-backup* approach with single primary suffers from limited performance since only one replica processes transactions. *Multi-primary replication* addresses this problem in a setting where data access could be partitioned and using this inherent data partitioning, different replicas can process some share of workload and backup the others. In case data accesses are not completely disjoint accesses, replicas slow down and give degraded performance due to need of increased coordination.

Both of these mechanisms i.e.*active- and passive replication*, have different trade-off. While *passive replication* could be a better choice for compute intensive workloads thereby saving computational resources, *active replication* has distinct advantage when the state updates are large enough to reach network bandwidth limit. In case of crash, *passive replication* could have detection and recovery delay, whereas *active replication* provides failure masking without noticeable performance degradation.

Active replication can be classified according to the time when transactions are ordered globally. On the one hand, transactions can be executed by clients before a global certification is invoked to resolve conflicts against other transactions running on remote nodes (approach also known as Deferred Update Replication or DUR) [117, 100, 8, 57]). Global certification phase requires exchange of network messages containing the object updates. On the other hand, clients can postpone the transaction execution after the agreement on a common order is reached. This way, they do not process transactions but they simply broadcast transaction requests to all nodes and wait until the fastest replica replies (we name it Deferred Execution Replication or DER) [70, 80, 48].

DUR approach gives high throughput (requests processed per second) when the conflicts among distributed transactions are rare and messages containing data updates are not big. In case the conflicts among distributed transactions are high, DUR severely suffers due to increase in remote aborts. Also if messages encapsulating data updates become very large, performance of DUR goes down due to limited network bandwidth. On the flip side, in case of DER, size of message containing the transaction request (and input parameters) is usually much smaller and is independent of object size, which helps to achieve a high throughput for defining the serialization order among requests. On top of that, harnessing the local execution, DER achieves moderately high throughput even for high conflict scenarios compared to DUR execution model.

Both DUR and DER approaches require a message ordering layer for defining the order of transactions so that each replica observes a consistent and unique order of transactional updates. While DUR approach requires the unique order of all object updates proposed by distributed concurrent transactions to ensure that each replica reaches same decision during certification phase, DER approach requires global order of all client requests so that each replica reaches an identical state after processing client requests following it.

Global order for transactions could be defined in two ways i.e., total order or partial order. Total order blindly defines the order of the transactions i.e., without taking into account the dependencies (or conflicts) among different transactional requests. Paxos [60] and S-Paxos [8] are widely known examples of total-order protocols. Partial order, on the other hand, only defines the global order for the conflicting transactions. Generalized Paxos [59] and Egalitarian-Paxos [78] are exmaples of such partial order protocols.

1.2 Motivation

In today's distributed transaction processing systems, performance is of paramount importance. Any loss in performance results e.g. latency resulting from slow processing of a client request or slow page load [39, 41], results in loss of revenue for businesses. As an example, google looses 0.44% of search sessions for each 400ms of increased page load time [39]. In another estimate, a 1-millisecond advantage in trading applications can add $100 million per year to a major brokerage firm [75]. In order to scale the request processing, usually transaction processing systems increase the system size i.e. add more processing elements. As majority of transaction request fall in the category of query transactions, therefore solutions which can ensure performance scaling for read-only workloads with the increase in system size are preferred. Another key insight is that not all read-only workloads need to access the latest data. With these goals in mind, we design transaction processing systems in this dissertation which can leverage local execution of read requests and different freshness guarantees, thereby giving high performance with low latencies for read workloads.

While high-performance for processing query transaction is preferred, ensuring the progress of write transactions with high throughput is equally important. This specially becomes of significance in case transactions experience high conflict scenario when they access same pool of shared data. Conflicts are resolved by letting one transaction proceed and aborting others which contend for same set of data objects, resulting in higher number of aborts for such transactions. Although DUR performs good when conflicts are rare, it is found to be worst impacted by high conflict workload profiles, since transactions execute in parallel before agreeing on an order and majority of them abort and restart. DER, on the other hand serializes all the update transactions before processing them serially, therefore it does not get affected by conflicting workloads.

Immunity from percentage of conflicts within transactions, make DER an attractive choice of transaction execution model, but its serial execution results in limited performance and moderately high latencies for write requests. This limitation can be addressed by designing parallel execution mechanisms for DER model, but honouring the order defined by request ordering layer along with parallel execution of request, makes it a challenging and interesting problem. We attempt to solve this problem in this dissertation.

DER requires a *total order* layer to serialize all the update transactions, which introduces a delay before a request could be processed resulting in longer latencies perceived by clients. This delay could be eliminated by anticipating the work and processing requests speculatively, but without an oracle, requests would be required to execute speculatively in multiple serialization orders. This could not only make it difficult to manage all those possible executions, but it could also result in wastage of computing resources as a lot of work is discarded if it does not match with output of *total order* i.e. *final order*. We exploit the early message delivery from *total order*, also known as *optimistic delivery* to guess the *final order* of request for speculative execution, so that when *final order* arrives, majority of work is already accomplished and if *optimistic delivery* order matches the *final order*, response to clients could be sent earlier.

Total order layer poses scalability challenges i.e. it does not scale with increase in system size. As the number of nodes in the system increase, single leader becomes the bottleneck of total order layer and its performance suffers. Also single leader total order protocols may fail to give high performance if the elected leader gets overloaded and starts to show degraded performance. Lastly, *total order* protocols, though provide a simpler implementation solution, they make request processing inefficient since transactions are forced to follow the order even if they are independent of each other i.e. non-conflicting.

Multi-leader *partial order* protocols [59, 78] address the challenges of single-leader total order protocols. On one hand existence of multiple leaders alleviates the problem of bottleneck created by single leader and degraded performance of overloaded leader as other nodes help the system to make progress. On the other hand, *partial order* protocol examines conflicts among different transaction prior to defining the order and as a result they enable non-conflicting transactions to execute concurrently whereas conflicting transactions are serialized. We attempt to design a multi-leader *partial order* messaging layer which could be exploited to build high performance transactional systems.

To summarize, in this research proposal, our main aim is to design high performance fault-tolerant distributed transactional systems. Observing the collective benefits of *active replication* e.g., full-failure masking, local computation, and high performance, we select it as our default replication model. Further, our focus is to design systems which can give high performance even under high conflict scenarios, therefore we select DER model of request execution. We use *optimistic delivery* to process requests in parallel, while their order is being finalized, to help reduce the request latency. Next we design a novel multi-leader partial order protocol and build a transactional system exploiting its benefits. Lastly, we exploit highly multi-core architectures to service transactional queries locally at each replica (as replica consistency is assured by *active replication*) using a multi-version concurrency control. This scheme reduces the contention among read and write accesses to objects, thereby scaling the performance of queries with the system size.

1.3 Summary of Current Research Contributions

Total-order can be formed by solving the *consensus* (or atomic broadcast [24]) problem. In this area, one of the most studied algorithm is Paxos [60]. Though Paxos's initial design was expensive (e.g., it required three communication steps), significant research efforts have focused on alternative designs for enhancing performance. A recent example is *JPaxos* [57, 98, 99], built on top of MultiPaxos [60], which extends Paxos to allow processes to agree on a sequence of values, instead of a single value. JPaxos incorporates optimizations such as batching and pipelining, which significantly boost message throughput [98]. *S-Paxos* [8] is another example that seeks to improve performance by balancing the load of the network protocol over all the nodes, instead of concentrating that on the leader.

We extend S-Paxos with *optimistic-delivery* and call it *OS-Paxos*. OS-Paxos optimizes the S-Paxos architecture for efficiently supporting optimistic deliveries, with the aim of minimizing the likelihood of mismatches between the optimistic order and the final delivery order. Based on *OS-Paxos*, we designed HiperTM, a high performance active replication protocol. This protocol wraps write transactions in transactional request messages and executes them on all the replicas in the same order.

HiperTM uses a novel, speculative concurrency control protocol called SCC, which processes write transactions serially, minimizing code instrumentation (i.e., locks or CAS operations). When a transaction is optimistically delivered by OS-Paxos, its execution speculatively starts, assuming the optimistic order as the processing order. Avoiding atomic operations allows transactions to reach maximum performance in the time available between the optimistic and the corresponding final delivery. Conflict detection and any other more complex mechanisms hamper the protocol's ability to completely execute a sequence of transactions within their final notifications – so those are avoided.

For read-only transaction processing we use Multi-Version objects, which helps to execute write transactions in parallel, while eliminating the possible conflicts between read and write transactions. Additionally read-only transactions are directly delivered to individual replicas and processed locally, without going through total-order layer since replica consistency is guaranteed by *total-order* layer.

An experimentation evaluation of HiperTM on a public cluster[1], revealed that serially processing optimistically delivered transactions guarantees a throughput (transactions per second) that is higher than atomic broadcast service's throughput (messages per second), confirming optimistic delivery's effectiveness for concurrency control in actively replicated transactional systems. Additionally, the reduced number of CAS operations allows greater concurrency, which is exploited by read-only transactions for executing faster.

From the experience with HiperTM, we learned about some possible improvements in our system. Firstly, assuming that *optimistic-order* matches *final-order*, any processing happening after arrival of *final-order* could be avoided, resulting in a lower latency and better performance. Second was to enhance the serial transaction execution to a high-performance concurrent one. Last one was to improve the *optimistic-delivery* mechanism to keep pace with the enhanced transaction processing.

Keeping these goals in sight, we designed Archie, an transactional framework based on state machine replication (SMR) that incorporates a set of protocol and system innovations that extensively use *speculation* for removing any non-trivial task after the delivery of the transaction's order. The main purpose of Archie is to avoid the time-consuming operations (e.g., the entire transaction execution or iterations over transaction's read and written objects) performed after this notification, such that a transaction can be immediately committed in case of no failure or node suspicion.

Archie incorporates *MiMoX*, a highly optimized total order layer, which proposes an architecture that mixes *optimistic-delivery* and *batching* [98] thus allowing the anticipation (thanks to the reliable optimistic notification) of a big amount of work (thanks to the batching) before the total order is finalized. Nodes take advantage of the time needed for assembling a batch to compute a significant amount of work before the delivery of the order is issued. This anticipation is mandatory in order to minimize (possibly remove) the transaction's serial phase.

At the core of Archie there is a novel speculative parallel concurrency control, named *ParSpec*, that processes transactions speculatively and concurrently, upon their optimistic notification, enforcing the same order as the sequence of optimistic notifications. *ParSpec* does it by executing transactions speculatively in parallel, but allowing them to speculative commit only in-order, thus reducing the cost of possible out-of-order executions. *ParSpec* makes modifications visible to the following speculative transactions and ready-to-commit snapshot of transaction's modifications are pre-installed into the shared data-set, but only

[1]We evaluate all our works on *PRObE* [32], a high performance public cluster.

publishes these modified object versions to non-speculative transactions post– *final-delivery* and validation step.

We implemented Archie in Java and our comprehensive experimental study with TPC-C [20], Bank and a distributed version of Vacation [13] benchmarks revealed that Archie outperforms all competitors including PaxosSTM [117], the classical state-machine replication [84] implementation and HiperTM [48], in most of the tested scenarios. on the other hand, when the contention is very low, PaxosSTM performs better than Archie.

Looking back at the experience with building HiperTM and Archie, we learned that when the systems size increases and thus the load of the system is high, total order based transactional system exhibits two well-known drawbacks which may limit its effectiveness: poor parallelism in the transaction execution and the existence of a single node (a.k.a. leader), which defines the order on behalf of all nodes. Processing transactions in accordance with a total order means effectively processing them serially, whereas *total order* can be seen as an unnecessary overestimation because the outcome of the commits of two non-conflicting transactions is independent from their commit order.

These two drawbacks are already addressed in literature. On one hand, more complex ordering techniques have been proposed, which allow the coexistence of multiple leaders at a time so that the system load is balanced among them, and the presence of a slow leader does not hamper the overall performance (as in the case of single leader) [69, 78]. On the other hand, the problem of ordering transactions according to their actual conflicts has been originally formalized by the Generalized Consensus [59] and Generic Broadcast [86] problems. The core idea consists of avoiding a "blind" total order of all submitted transactions whereas only those transactions that depend upon each other are totally ordered.

As current leaderless Generalized Paxos-based protocols suffer from serious performance penalties when deployed in systems where general purpose transactions (i.e., transactions that perform both read and write operations) are assumed, we designed a novel protocol that inherits the benefits from existing solutions while achieving high performance for transactional systems. We built Caesar, a replicated transactional system that takes advantage of an innovative multi-leader protocol implementing generalized consensus to enable high parallelism when processing transactions. The core idea is enforcing a partial order on the execution of transactions according to their conflicts, by leaving non-conflicting transactions to proceed in parallel and without enforcing any synchronization during the execution (e.g., no locks). The novelty of Caesar is in the way it is able to efficiently find a partial order among transactions without relying on any designated single point of decision, i.e., leaderless, and overcoming the limitations of recent contributions in the field of distributed consensus (e.g., [78]) when applied to transactional processing.

We implemented Caesar's prototype in Java and conducted an extensive evaluation involving three well-known transactional benchmarks like Bank, TPC-C [20], and distributed version of Vacation from the STAMP suite [13]. In order to cover a wide range of competitors, we compared Caesar against EPaxos [78], Generalized Paxos [59], and a transactional system using MultiPaxos [60] for ordering transactions before the execution. As a testbed, we used Amazon EC2 [2] and we scaled our deployment up to 43 nodes. To the best of our knowledge, this is the first consensus-based transactional system evaluated with such a large network scale. The results reveal that Caesar is able to outperform competitors in almost all cases, reaching the highest gain of more than one order of magnitude using Bank benchmark, and by as much as 8× running TPC-C.

While incorporating the system optimizations in HiperTM and Archie, we stumbled upon the idea that not all read-only workloads require up-to-date data and application specific freshness and content-based constraints could be exploited to service read-only transactions to achieve high scalability. As a result, we designed Dexter which is built upon HiperTM which is a *state machine* based transactional system. Since HiperTM is just a transaction processing abstraction, Dexter can very well be based on other alternative transactional systems [117].

Dexter's architecture divides nodes into one *synchronous* group and a set of *asynchronous* groups. Nodes in the synchronous group process write-transactions according to the classical *active replication* paradigm. Nodes in the asynchronous groups are lazily updated, and only serve read-only transactions with different constraints on data freshness or content. The asynchronous groups are logically organized as a set of levels and with each increasing level the expected freshness of objects decreases (staleness increases). The synchronous group stores the latest versions of the objects, and thereby serves those read-only requests that need access to the latest object versions. The main advantage of this architecture is that write transactions yield AR's traditional high performance, while at the same time, nodes can scale up for serving additional read-only workloads, exploiting the various levels of freshness that is available or expected.

For exploiting the aforementioned architecture, obviously, the application must specify the needed level of freshness guarantees. For this reason, Dexter provides a framework for inferring rules that are used for characterizing the freshness of a read-only transaction when it starts in the system. Using this framework, the programmer can describe the application's requirements. Rules can be defined based on the elapsed time since the last update on an object was triggered, as well as based on the content of the object (or the type of the object).

We implemented Dexter in Java and used HiperTM [48] for implementing the synchronous group. The rest of the infrastructure was built with classical *state machine replication* implementation. An extensive evaluation study aimed at testing the scalability of the system, revealed that Dexter outperform competitors by as much as 2×.

1.4 Summary of Proposed Post-Prelim Work

In the pre-preliminary work, we have focused on optimizing the transaction execution using optimistic delivery, exploiting transaction parallelism within *total ordered* transactions, harnessing the benefits of *partial order* etc. to build high performance fault-tolerant transactional systems. In post-preliminary work, we seek to further enhance replicated transactional systems by incorporating locally ordered transactions in DUR systems, exploiting Hardware Transaction Memory (HTM) for distributed transactions and lastly optimizing the multi-leader *partial order* protocol in a ring based network.

Current certification-based *deferred-update replication* systems pay high cost of local and remote aborts in case of high contention on shared objects, due to which their performance is adversely affected. Usually clients have a transaction locality i.e. with a high probability a client will access the same transaction processing node and same set of objects over and over. If object accesses are well partitioned in this manner, then a number of local aborts could be eliminated by ordering the conflicting transactions prior to the execution and certification phase. We plan to investigate in this direction and incorporate the benefits of local *state machine* approach to scale-up the distributed performance of DUR systems.

With our experience with HiperTM and Archie, we learnt that maintaining each component of software transactional memory e.g., read-set, write-set, and multi-versioning etc. has some unavoidable overheads. On the other hand, HTM is well known to yield high performance as it exploits the underlying hardware capabilities to manage contention while providing transactional (ACI) properties. Even more, recently HTM systems are gathering even more interest, after the introduction of support for transactional synchronization extensions in Intel® Core^{TM} Haswell processors, enabling ease of executing transactions on hardware. Considering these benefits, we plan to incorporate HTM support for in-order processing of distributed transactions to build high performance transactional systems without any major code instrumentation.

Apart from the execution time, transaction latency depends on the communication steps required by global ordering protocol to finalize the order of transaction. From our experience with Caesar, fast decisions involving only two communication steps are found to give optimal performance, but as a precondition it requires the same set of dependencies observed by a quorum of nodes. As message exchange among nodes is non-deterministic, this precondition is difficult to meet for each transaction. We find this as an interesting challenge and propose to design a multi-leader partial order protocol for ring based network which always ensures the fast decision, thereby improving the latency and performance of transactional systems.

1.5 Thesis Organization

This thesis proposal is organized as follows. In Chapter 2 we summarize the relevant and related previous work. In Chapter 5, we introduce and discuss HiperTM, a distributed transactional system built on top of OS-Paxos. Next, we present a highly concurrent transactional system Archie in 6, which incorporates optimizations to *total-order* layer and local concurrency control to achieve high performance. In chapter 7, we introduce Caesar, a high performance distributed transactional system based on multi-leader *partial order* protocol. In Chapter 8, we present Dexter, a transactional framework where we exploit application specific staleness and content-based constraints to scale the performance with the system size. Chapter 9 summarizes the thesis proposal and proposes post-preliminary work.

Chapter 2

Related Work

2.1 Transactional Replication Systems

Replication in transactional systems has been widely explored in the context of DBMS, including protocol specifications [51] and infrastructural solutions [104, 82, 83]. These proposals span from the usage of distributed locking to atomic commit protocols. [116] implements and evaluates various replication techniques, and those based on active replication are found to be the most promising.

Transactional replication based on atomic primitives had been widely studied in the last several years [52, 56, 117, 100]. Some of them focus on partial replication [100], while others target full replication [56, 117]. Partial replication protocols are affected by application locality: when transactions mostly access remote objects instead of local, the performance of the local concurrency control becomes negligible as compared to network delays. Full replication systems have been investigated in certification-based transaction processing [53, 14, 88]. In this model, transactions are first processed locally, and a total order service is invoked in the commit phase for globally certifying transaction execution (by broadcasting their read and write-sets).

Granola [21] is a replication protocol based on a single round of communication. Granola's concurrency control technique uses single-thread processing for avoiding synchronization overhead, and has a structure for scheduling jobs similar to speculative concurrency control.

Transactional systems can be categorized according to the scheme adopted for ordering transactions, i.e., total order or partial order. Total order finds its basis in Paxos [60], the original algorithm for establishing agreement among nodes in the presence of failures. A number of solutions can be listed here [70, 113, 50] but all of them suffer from a scalability bottleneck due to the presence of a single leader in the system. Recently, a redesign of the state-machine approach, called P-SMR, has been proposed in [70]. P-SMR is geared towards increasing the parallelism of transaction processing for total order-based distributed systems. P-SMR pursues this goal by creating a total order and then defining a set of worker threads per node, each processing a particular transaction conflict class. This way, transactions are pre-classified according to their conflicts and executed in parallel. Another approach, which extensively uses partitioned accesses and total ordered transactions is [71].

Calvin [113] assumes a partitioned repository of data (i.e., partial replication) and, on top of it, it builds a replication protocol. However, even though its partial replication model can be a way to enable scalability in case the mapping of data to nodes follows the distribution of the application's locality, it still requires the total order of all transactions. This is achieved via a logical entity, named sequencer, that is responsible for defining the total order by associating transactions with monotonically increasing sequencer numbers. To alleviate the pressure on a single point of processing, Calvin implements the logical sequencer as a set of distributed sequencers, and it defines a deterministic total order among messages from different sequencers via epoch numbers and sequencers' ids.

Eve [50] is another replicated transactional system that proposes the *execution-verify* approach. Roughly, Eve inherits the benefits from the DUR model, while falling back to the DER approach when the result of the optimistic execution is not compliant with other remote executions. This entails retrying the executions and committing them serially, after having established a total order for them. This approach reduces the load on the ordering layer because not all transactions necessarily undergo the ordering process. But in high contention scenarios, most speculative executions could be irreconcilable and Eve does not provide a specific solution to preserve high performance.

Mencius [69] is an ordering protocol that is able to establish a partial order on transactions on the basis of their dependencies. It is very close to the ordering scheme proposed by Caesar, even though it has a strong requirement that Caesar relaxes: it pre-assigns sending slots to nodes, and a sender can decide the order of a message at a certain slot s only after having heard from all nodes about the status of slots that precede s. This approach results in poor performance in case there is a slow or suspected node.

Alvin [115] is a recent transactional system that proposes an optimized version of the EPaxos's ordering protocol. Like Caesar, it only enforces an order among conflicting transactions, and it is able to avoid the expensive computation on the dependency graph enforced by EPaxos to find out the final execution order for a transaction. However, unlike Caesar and EPaxos, a transaction's leader in Alvin needs to re-collect dependencies from the other replicas in case it is not able to decide after the first round of communication. And this happens, in accordance with the scheme also adopted by EPaxos, in case the dependencies collected in the first round are discordant. This is not the case of Caesar, which forces an autonomously chosen decision only if it receives an explicit rejection in the first round of votes.

Rex [40] is a fault-tolerant replication system where all transactions are executed on a single node, thus retaining the advantages of pure local execution, while traces are collected reflecting the transaction's execution order. Then, Rex uses consensus to make traces stable across all replicas for fault-tolerance (without requiring a total order).

2.2 Optimistic Atomic Broadcast

The original Paxos algorithm for establishing agreement among nodes in the presence of failures was presented in [60], and later optimized in several works, e.g., [98, 57, 8]. These efforts do not provide any optimistic delivery. S-Paxos [8] introduced the idea of offloading the work for creating batches from the leader and distributing it across all nodes.

Optimistic delivery has been firstly presented in [52], and later investigated in [79, 80, 71]. [79] presents AGGRO, a speculative concurrency control protocol, which processes transactions in-order, in actively replicated transactional systems. In AGGRO, for each read operation, the transaction identifies the following transactions according to the opt-order, and for each one, it traverses the transactions' write-set to retrieve the correct version to read. The authors only present the protocol in [79]; no actual implementation is presented, and therefore overheads are not revealed. The work in [80] exploits optimistic delivery and proposes an adaptive approach to different networks models. The ordering protocol proposed in [71] is the first that ensures no-reordering between optimistic and final delivery in case of stable leader by relying on a network with a ring topology.

2.3 Scalable Read Processing

As read-only workloads form the majority of requests, considerable research efforts had been invested in improving the response time (latency) of query transactions. One of the ways to improve read-only workload performance is to service the query transactions using multi-version object storage. [68] presents a word-based Multi-version STM implementation in "C". Motivation for multi-version concurrency control is invisibility of read transactions to write transaction and long running read transactions may starve (abort and retry repetitively) in case there are a lot updates on the objects used by read transaction. [94], [93] and [10] are object-based multi-version STM implementations for centralized STMs and authors use multi-versions to improve the performance of read transactions.

Improvement over read-only workloads has been also studied in content-based caching techniques. In [66] a freshness-driven adaptive caching protocol is presented. They consider a model containing edge servers that cache data, web servers, application servers and backend DBMS. They propose changing the cache capacity for improving the cache hits and thereby reducing request response time. In [18] is presented the idea of updating cached objects (or validate them) when a predefined Time-To-Live (*TTL*) expires. It improves the performance as seen by the user, since getting a fresh data from main server is done offline and user only receives the fresh data from caching server. They also present policies to update/re-validate the objects after that their *TTL* is expired. The work in [12] considers search results caches used in Yahoo! search engine and introduces the concept of refreshing cache entries with expired *TTL*, leveraging idle cycles of engine. It prioritizes refreshing cache entries based on the access frequency and the duration of the cached entry. However, those techniques are not suited for transactional application with isolation and atomicity guarantees.

One of the observation over usual transaction workloads reveals that not all read-only workloads need to access the latest data. Some of the approaches proposed by database community [81, 31] tries to exploit this insight. Both [81, 31] use full replication with local databases ensuring local consistency and introduce a routing layer which enforces global consistency. Refresco [81] is based on single Master replication where the master node processes all the update (write) transactions and query (read-only) transactions are served by slave nodes. Slave nodes are updated asynchronously by master node through refresh transactions (single-primary lazy-backup). Leganet [31] uses full replication (lazy master replication)with local database ensuring local consistency and introduces a routing layer which enforces global consistency. Leganet runs all conflicting update (write) transactions in the same relative order at each node.

Pileus [111, 112] is a cloud storage based system which defines service level agreements (SLA) and gives the programmer the flexibility of selecting the consistency level for each read request. In Pileus, programmer could specify an SLA for read request and system translates the SLA for appropriate consistency guarantees which can be provided within desired latency bounds.

Chapter 3

Background

Replication has been studied extensively as a solution to provide improved availability for distributed systems. Replica consistency can be ensured if each replica observes same sequence of updates on replicated data. This is made feasible by *atomic broadcast* layer, a variant of *reliable broadcast*. *Atomic broadcast* is a broadcast messaging protocol that ensures that messages are received reliably and in the same order by all participant replicas [24]. Solution to *atomic broadcast* problem has been shown to reduce to consensus on messages [28]. Although many consensus algorithms [60, 15, 49, 63, 74, 73] have been studied till now, Paxos [60] is still one of the most widely studied consensus algorithm.

Usually in local area networks, under moderate loads, with high probability two messages *m* and *m'* would be received in the same order by all processes (replicas). *Optimistic atomic broadcast* was introduced by Pedone and Schiper [85], which exploits this knowledge to reduce the average delay for message delivery. *Optimistic atomic broadcast* helps to execute the request speculatively by earlier delivery of request i.e. *optimistic-delivery*, thereby reducing the cost of execution after the consensus.

While consensus and *optimistic atomic broadcast* forms the reliable messaging layer, transactional systems require a concurrency control mechanism at each processes which can provide concurrent request execution with high performance. These requests could access the same data and to ensure that data consistency is maintained, traditionally lock-based synchronization mechanisms have been used. Lock-based approaches are usually hard to maintain and have programmability, scalability and composability challenges. Transactional memory (TM) is an alternative synchronization solution which promises to address these challenges.

We use these different building blocks to design high-performance replicated transactional systems in this dissertation. In this chapter, we provide a brief discussion over these building blocks.

3.1 Paxos

In a distributed system, where multiple processes coordinate among themselves to achieve their individual goals using common shared resources, the need to incorporate consensus becomes implicit. In other words, any algorithm that helps to maintain a common order among multiple processes, in a model where some processes may fail, involves solving a consensus problem. Within a rich set of consensus algorithms [60, 15, 49, 63, 74, 73], Paxos [60] is one of the most widely studied consensus algorithm. Paxos was introduced by Leslie Lamport as a solution to finding an agreement among a group of participants even under failures.

After the initial design [60], there have been many variants of Paxos [65, 63, 62, 74] studied by research community. The Paxos family of protocols includes a spectrum of trade-offs between the number of processors, number of message delays before learning the agreed value, the activity level of individual participants, number of messages sent, and types of failures. Although no deterministic fault-tolerant consensus protocol can guarantee progress in an asynchronous network [29], Paxos guarantees safety (consistency), and the conditions that could prevent it from making progress are difficult to provoke.

3.1.1 The Paxos Algorithm

Paxos categorises processes by their roles in the protocol: *proposer*, *acceptor*, and *learner*. In a typical implementation a process may assume one or more roles. This does not affect the correctness of the protocol, but could improve the latency and throughput due to reduced number of messages by coalescing roles.

Client issues a request to the distributed system, and waits for a response. *Proposer* receives request and attempt to convince acceptors to agree on it by sending proposals, acting like a coordinator of client request, to move protocol forward. *Acceptors* act at the fault-tolerant memory of the protocol and responds to proposer's proposals to arrive on an agreement. Consensus over a proposal can only arrive if a quorum of acceptors agree to that proposal. *Learner* act as the replication factor for the protocol. Once consensus over a client request is formed, learner may process the request and respond to clients. Additional *learners* could be added to improve the system availability. Paxos requires a distinguished proposer, also known as *leader*, to make the protocol progress. Multiple proposers could believe that they are leaders, but it results in stalling of protocol due to continuous conflicting proposals.

In Paxos, consensus for each request constitutes a new ballot (Fig. 3.1) and processes could execute a series of ballots to agree on different requests. In each ballot one of the proposers, also called *leader* of the ballot, tries to convince other processes to agree on a value proposed by it. If ballot succeeds the value proposed is decided, otherwise other processes may start a new ballot. Each ballot is composed of two phases of communication:

Phase1: Proposer initiates a ballot by sending a *Prepare* message for a proposal to a quorum of acceptors. Each proposal is assigned a number, also called *view or Epoch*, which is higher than any previous proposal number used by this proposer.

Acceptors responds to the proposal and promises to ignore any later proposal lesser than current proposal's number N, if N is higher than any previous proposal number received by the Acceptor. Though responding to proposer is optional in case this condition is not met, as an optimization sending a `Nack` helps proposer to stop the attempt to create consensus on proposal number N.

Phase2: Proposer enters the second phase of ballot when it receives a quorum of responses from acceptors. If a response from some Acceptor contains a value, then proposer selects that value to its proposal. Otherwise if none of the Acceptors accepted a proposal up to this point, then the Proposer can choose any new value for its proposal. Proposer sends an *Accept* message to the quorum of Acceptors with the chosen value for the proposal.

Acceptors, on receiving *Accept* message with a proposal number N, accept the proposal iff they have not promised to participate in proposal higher than N. In this case, each acceptor sends *Accepted* message to proposers and learners informing them that it has accepted the proposal. Proposal is decided once learners receive a majority of *Accepted* messages for a proposal.

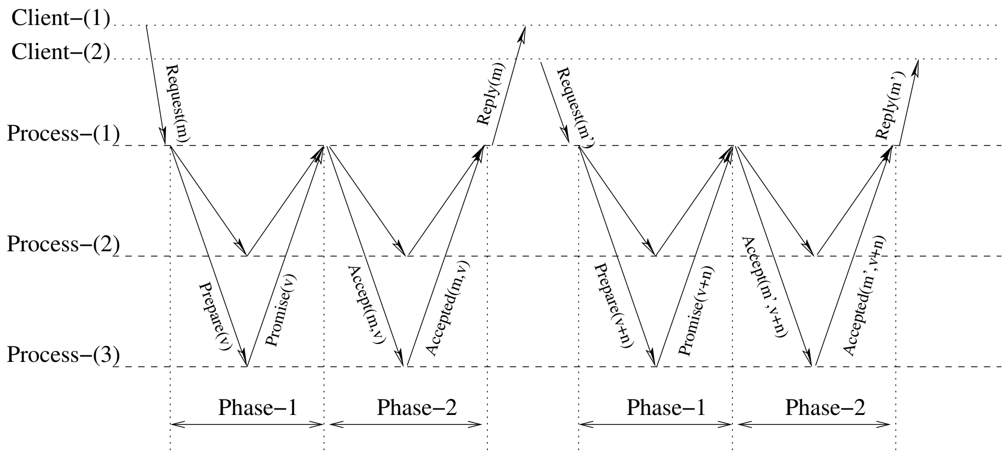


Figure 3.1: Consensus mechanism with classic Paxos ballot

3.1.2 Multi-Paxos

Paxos protocol requires phase-1 of ballot to select a leader (distinguished proposer), which then tries to get a consensus over a client request. Since it does not assume that leader could be stable, it suffers from significant amount of overhead when each request is agreed through a separate ballot. In case, leader process is relatively stable, phase-1 become redundant.

Multi-Paxos is an optimization over Paxos, where processes try to agree on a sequence

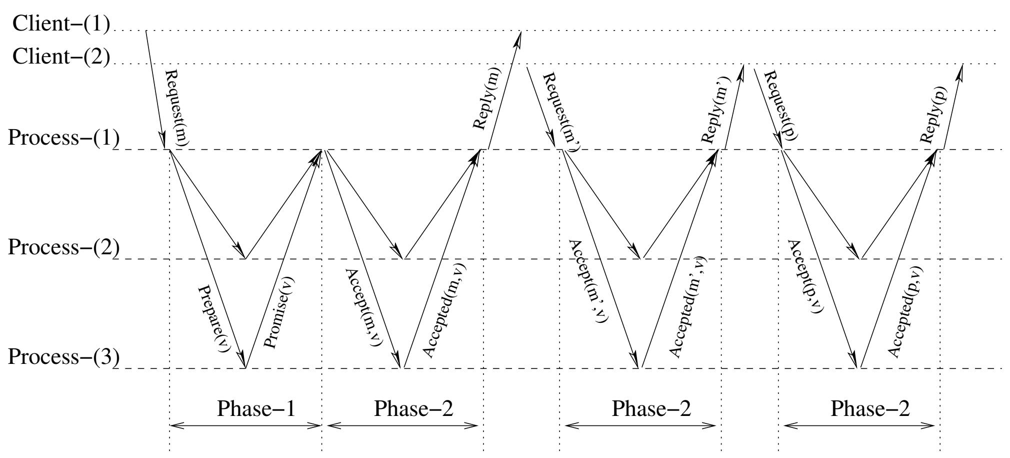


Figure 3.2: Consensus mechanism in Multi-Paxos

of client requests rather than single request with the same leader (Fig. 3.2). Multi-Paxos executes prepare phase (phase-1) for selecting a new leader for arbitrary number of future consensus instances. Afterwards, it only executes phase-2 for each instance, till this leader is suspected to have crashed by failure detector. This optimization reduces the number of communication steps for an instance, thereby improves the overall performance and latency.

3.1.3 Generalized Paxos

Paxos (as well as Multi-Paxos) "blindly" defines a total order for client requests in a single leader environment. Establishing a total order relying on one leader is widely accepted solution since it guarantees the delivery of a decision with the optimal number of communication steps [61], though existence of an overloaded or slow leader could become a bottleneck and limits its effectiveness. Fast Paxos [64] extends Classic Paxos by allowing fast rounds, in which a decision can be learned in two communication steps without relying on a leader but requires bigger quorums.

Additionally, defining a *total order* can be seen as an unnecessary overestimation of conflicts among client requests, because the outcome of two non-conflicting requests is independent from their commit order. As a consequence *total order* limits the parallelism with request execution, as all requests have to execute serially. The problem of ordering transactions according to their actual conflicts has been originally formalized by Generalized Paxos [59] which defines a *total order* on only those transactions that depend upon each other.

Generalized Paxos allows all processes (proposers) to send proposal for their client requests, thereby becoming the coordinator for their respective proposal. At the start, a request coordinator sends its proposal to other processes using *Propose* message. On receiving the proposal from an individual coordinator, acceptors evaluate proposal's conflict with the other concurrent proposals and broadcast their evaluated dependencies to others in the form of *Accepted* message. Each acceptor waits for fast quorum (Q_F) of *Accepted* messages for a given proposal. If the dependencies observed by fast quorum of *Accepted* messages are identical, proposed request is committed, thereby defining the order using the fast round similar to Fast Paxos. Otherwise if dependencies observed by different processes are non-identical, a dedicated leader resolves the conflict and defines the order for the proposed request. Leader then informs all other processes about the order by sending a *Stable* message. As a result, only those requests that depend upon each other are totally ordered, whereas each process is allowed to deliver an order that differs from the one delivered by another process, while all of them have in common the way conflicting requests are ordered.

3.2 Atomic Broadcast

Different reliable broadcast protocols support different properties. *FIFO-order broadcast* ensures that messages from the same process are delivered in the same order that the sender has broadcast them. But it does not guarantee any order for messages from different senders. On the other hand, *causal-order broadcast* ensures the global order for all causally dependent messages, but it does not guarantee order among unrelated messages. *Total-order broadcast*, also known as *atomic broadcast*, addresses the drawbacks of both of these broadcast protocols and orders all messages, even those that are from different senders and causally unrelated. Therefore, *atomic broadcast* provides much stronger ordering properties for all processes and without any knowledge about messages' causal dependency.

Total-order broadcast is called *atomic broadcast* because the message delivery occurs as if the broadcast were an indivisible "atomic" action: the message is delivered to all or to none of the processes and, if the message is delivered, every other message is ordered either before or after this message. *Atomic broadcast* defines two primitives: *ABcast(m)*, used by clients to broadcast a message *m* to all the processes; *ADeliver(m)*, event notified to each process for delivering message *m*. These primitives satisfy the following properties: *Validity*, if a correct process *ABcast* a message *m*, then it eventually *ADeliver m*; *Agreement*, if a process *ADelivers* message *m*, then all correct processes eventually *ADeliver m*; *Uniform integrity*, for any message *m*, every process *ADelivers m* at most once, and only if *m* was previously *ABcasted*; *Total order*, if some process *ADelivers m* before *m'*, then every process *ADelivers m* and *m'* in the same order.

3.3 Optimistic Atomic Broadcast

Optimistic atomic broadcast [85] was proposed as an optimization over *atomic broadcast*, which exploits the spontaneous total-order property: if processes *p* and *q* sends messages *m* and *m'* respectively to all processes, then both messages might be received in the same order by all processes. This property usually holds in local-area networks under moderate load conditions.

Apart from *ABcast(m)* and *ADeliver(m)*, *optimistic atomic broadcast* defines an additional primitive, called *ODeliver(m)*, which is used for early delivering a previously broadcast message m before the *Adeliver* for m is issued. Earlier delivery of message helps to reduce the latency as request (wrapped within message) is processed optimistically before the consensus decision arrives, but updates by the request are buffered. When message m is *Adelivered*, if the message order of *ODeliver(m)* matches *ADeliver(m)* then the buffered changes from request execution are applied to main memory (or the stable storage whatever the case may be).

In addition to the properties of *atomic broadcast*, *optimistic atomic broadcast* supports following properties: 1) If a process *Odeliver(m)*, then every correct process eventually *Odeliver(m)*; 2) If a correct process *Odeliver(m)*, then it eventually *Adeliver(m)*; 3) A process *Adeliver(m)* only after *Odeliver(m)*.

3.4 Transaction Memory

Taking the inspiration from database transactions, need for a similar abstraction in programming language semantic was felt for ensuring consistency of shared data among several processes. The usual way for managing concurrency in a system is using locks, which inherently suffers from programmability, scalability, and composability challenges [45]. Additionally, the implementation of complex algorithms based on manually implemented mutual exclusion supports becomes hard to debug, resulting in high software development time.

Herlihy and Moss [46] proposed hardware supported transactional memory which was followed by proposal on atomic multi-word operation known as *"Oklahoma update"* by Stone et al. [109]. These works became the starting point for research in hardware and software systems for implementing transactional memory. TM has been proposed in hardware(HTM) [89, 90, 17, 106], software (STM) [76, 26, 107, 10, 9, 108, 95] and hybrid approaches [23, 58, 22].

TM [42] provides the synchronization abstraction that promises to alleviate programmability, scalability and composability issues with lock-based approaches. In fact, leveraging the proven concept of atomic and isolated transactions, TM spares programmers from the pitfalls of conventional manual lock-based synchronization, significantly simplifying the development of parallel and concurrent applications. TM transactions are characterized by only in-memory operations, thus their performance is orders of magnitude better than that of non in-memory processing systems (e.g.. database systems), where interactions with stable storage often degrade performance.

In software, STM libraries offer APIs to programmers for reading and writing shared objects, ensuring atomicity, isolation, and consistency in a completely transparent manner. STM transactions optimistically execute, logging object changes in a private space. Two transactions conflict if they access the same object and one access is a write. When that happens, a contention manager resolves the conflict by aborting one and committing the other, yielding (the illusion of) atomicity. Aborted transactions are re-started, after rolling-back the changes.

In hardware, similar objective is achieved by *best-effort execution* enabled by underlying hardware. In cache extension based HTMs, conflict is detected when cache line of read-set/writeset of one transaction is written by another transaction, thereby invalidating the cache line. In addition to cache extension, HTM could be implemented either by extending the functionality of the memory ordering buffer (MOB) and re-order buffer (ROB) in modern x86 microprocessors, or modifications to the pipeline of an x86 microprocessors. In the pursuit of benefiting from HTM's performance, IBM and Intel [11, 92] have recently launched new processors with HTM support. Even though transaction execution is usually much faster in HTM as compared to STM, all transaction profiles are not suitable for HTM, resulting in keeping STM research alive. HTM transactions which are either very long or accesses many objects become prone to aborts due to timer interrupts and cache capacity miss respectively.

The challenges of lock-based concurrency control are exacerbated in distributed systems, due to the additional complexity of multi-computer concurrency (e.g., debugging efforts, distributed deadlocks, and composability challenges). Distributed TM (or DTM) [87, 80, 19] has been similarly motivated as an alternative to distributed locks. In addition to multi-computer synchronization primitive, DTM also adds extra computing power due to increased number of nodes. DTM can be classified based on the mobility of objects and transactions, with concomitant tradeoffs, including a) control flow [4], where objects are immobile and transactions invoke object operations through RMIs/RPCs and b) dataflow [114], where transactions are immobile, and objects are migrated to invoking transactions.

In this thesis, our focus is on software enabled DTM solutions i.e. we employ STM solutions to process transactions locally at a processing node (replica) in a distributed transaction processing system. Therefore for completeness we overview STM and its basic building blocks.

3.5 Software Transaction Memory

Conventional trend for handling shared objects during concurrent accesses is employing lock-based solutions [3], where shared object accesses are protected by locks. Lock-based approaches suffer from many drawbacks including deadlocks, livelocks, lock-convoying, priority inversion, and non-composability etc. Software transaction memory (STM) [105], a TM's software variant, has been seen as an alternative software based solution for accessing shared memory objects, without exposing locks in the programming interface. STM provides the synchronization abstraction that promises to addresses programmability, scalability and composability challenges associated with lock-based approaches. In the following sections, we provide an overview of different building blocks of a TM system.

3.5.1 Concurrency Control Mechanisms

In a TM system concurrent accesses to shared objects generates conflicts. A conflict could occur when two transactions perform conflicting operations on same data object i.e. either both write or one transaction reads and another writes concurrently. TM system detects the conflict and resolves it by delaying or aborting one of the two conflicting transactions. Based on how the conflicts are detected or resolved, TM concurrency control could be broadly divided into two approaches: *pessimistic concurrency control*, where the conflict occurrence, detection and resolution all happen simultaneously; and *optimistic concurrency control*, where the conflict detection and resolution happens later than conflict occurrence.

Pessimistic concurrency control allows a transaction to exclusively claim a data prior to modifying the data, preventing other transactions from accessing it. If conflicts are frequent, then *pessimistic concurrency control* pays off i.e. once a transaction has locks over its objects, it could run to completion without any hurdle. However in case conflicts are rare, then pessimistic concurrency control results in low throughput.

On the other hand, *optimistic concurrency control* allows multiple transactions to access data concurrently and let them continue to run even if they conflict, till these conflicts are caught and resolved by TM. In case conflicts are rare, *optimistic concurrency control* gives better throughput, as it allows higher concurrency among transactions as compared to *Pessimistic concurrency control*.

3.5.2 Version Management Systems

Depending upon the concurrency control being used, TM systems require mechanisms to manage the tentative writes of concurrent transactions i.e. version management. The first approach is *eager version management*, also known as *direct update* where transactions directly update the data in memory. Transactions maintain an *undo log* which holds all the values transaction has overwritten. If the transaction aborts, it roll-backs all the changes it has performed in memory by writing back the *undo log*. *Eager version management* requires *Pessimistic concurrency control* because the transaction requires exclusive access to the objects if it is going to overwrite them directly.

Second approach is *lazy version management*, also known as *deferred update* because the updates are delayed till the transaction doesn't commit. Transaction maintains a private *redo log* to write the tentative writes. Transaction's updates are buffered in this log and transaction reads from this log if a modified object is read again within the same transaction context. If a transaction commits, it updates the objects in main memory with these private logs. On abort, transaction's *redo log* is simply discarded.

3.5.3 Conflict Detection

With *pessimistic concurrency control*, conflict detection is trivial since lock over an object could only be acquired if it is not already held by another thread in conflicting mode. For *optimistic concurrency control*, transaction uses validation operation to check whether or not it has experienced a conflict. Successful validation implies that transaction could be serialized in history of transactions executed so far.

Conflicts could be detected at different times during transaction execution. Firstly, a transaction could be detected when it declares its intent to access a new object. This approach is also known as *eager conflict detection* [77]. Next, conflicts could be detected by executing validation operation any time, or even multiple times, during transaction's execution to examine the collection of locations it previously read or updated. Lastly, a conflict could be detected when a transaction attempts to commit by validating the complete set of read/-modified locations one final time. This last approach is also called *lazy conflict detection* [77].

Chapter 4

Common System Model

4.1 Assumptions

We consider a classical distributed system model [37] where a set of nodes/processes $\Pi = \{N_1, N_2, \ldots, N_{|\Pi|}\}$ communicate using message passing links. A node can be correct (or non-faulty), i.e., working properly; or faulty, i.e., crashed; or suspected, i.e., some node experienced interrupted interaction with it but it is still not marked as crashed.

Processes running distributed algorithms are subject to failures of different components. It could range from a process crash or message link failure to malicious malfunctioning of a process. Accordingly faults could be broadly classified as: 1) Crash faults, where process stops executing; 2) Omission faults, where a process doesn't send or receive a message; or 3) Arbitrary faults also known as *Byzantine* faults, where process deviates from the assigned algorithm leading to unpredictable behaviour.

For our systems, we assume that nodes may fail according to the fail-stop (crash) model [7]. We assume a partially synchronous system [60], where every message may experience an arbitrarily large, although finite, delay. We assume a maximum number of faulty nodes to be equal to f, and the number of nodes $|\Pi|$ equal to $2f + 1$. We consider only non-byzantine faults, i.e., nodes cannot perform actions that are not compliant with the replication algorithm.

Decisions about the final order of a transaction are made by collecting information from other nodes in the system. Depending upon the communication steps involved and consensus protocol, we leverage different *quorums*. When the leader, under single-leader consensus protocol such as multi-paxos, decides the final order of a transaction, it waits for a quorum $Q_C = f + 1$ of replies. Similarly for leaderless protocols (e.g., Caesar), when a leader decides the final order of a transaction after two communication delays, it waits for a quorum

$Q_F = f + \left\lfloor \frac{f+1}{2} \right\rfloor$ of replies[1]. On the other hand, if a decision cannot be accomplished after two communication delays, then a quorum $Q_C = f + 1$ is used, as in [60]. This way any two quorums always intersect, thus ensuring that, even though f failures happen, there is always at least one site with the last updated information that we can use for recovering the system.

To eventually reach an agreement on the order of transactions when nodes are faulty, we assume that the system can be enhanced with the weakest type of unreliable failure detector [38] that is necessary to implement a leader election [37].

4.2 Transaction Model and Processing

For the sake of generality and following the trend of [56, 48, 70], we adopt the programming model of software transactional memory (STM) [105] and its natural extension to distributed systems (i.e., DTM). DTM allows the programmer to simply mark a set of operations with transactional requirements as an "atomic block". The DTM framework transparently ensures the atomic block's transactional properties (i.e., atomicity, isolation, consistency), while executing it concurrently with other atomic blocks.

A transaction is a sequence of operations, each of which is either a *read* or a *write* on a shared object, wrapped in a clearly marked procedure that starts with the *begin* operation and ends with the *commit* (or *abort*) operation. This procedure is available at the system side and may or may not be available at the client side. We name a transaction as *write transaction* in case it performs at least one write operation on some shared object, otherwise the transaction is called *read-only transaction*.

We consider a full replication model, where the application's entire shared data-set is replicated across all nodes. Transactions are not executed on application threads. Instead, application threads, referred to as *clients*, inject transactional requests into the replicated system and service threads process transactions. These two groups of threads do not necessarily run on the same physical machine. Our transaction processing model is similar to the multi-tiered architecture that is popular in relational databases and other modern storage systems, where dedicated threads (different from threads that invoke transactions) process transactions.

Each request is composed of a key, identifying the transaction to execute, and the values of all the parameters needed for running the transaction's logic (if any). Threads submit the transaction request to a node, and wait until the node successfully commits that transaction.

Specifically for Caesar, which orders transactions according to their conflicts, we broadcast the objects expected to be accessed together with the transaction key and input parame-

[1]The size of the quorum is the same adopted in [78] for fast quorums.

ters. This additional information, although mandatory for avoiding overestimation of actual conflicts, does not represent a limitation especially if the business logic of the transaction is snapshot-deterministic (i.e., the sequence of performed operations depend only on the returned value of previous read operations). In fact, in this case usually once the values of the input parameters of the stored-procedure are known, then the set of objects accessed, which could conflict with other, is likely known.

We use a multi-versioned memory model, wherein an object version has two fields: *timestamp*, which defines the logical time when the transaction that wrote the version committed; and *value*, which is the value of the object (either primitive value or set of fields).

Chapter 5

Hiper TM

State-machine replication (or active replication) [102] is a paradigm that exploits full replication to avoid service interruption in case of node failures. In this approach, whenever the application executes a transaction T, it is not directly processed in the same application thread. Instead, a group communication system (GCS), which is responsible for ensuring the CSO, creates a transaction request from T and issues it to all the nodes in the system. The CSO defines a total order among all transactional requests. Therefore, when a sequence of messages is delivered by the GCS to one node, it guarantees that other nodes also receive the same sequence, ensuring replica consistency.

A CSO can be determined using a solution to the *consensus* (or atomic broadcast [24]) problem: i.e., how a group of processes can agree on a value in the presence of faults in partially synchronous systems. Paxos [60] is one of the most widely studied consensus algorithms. Though Paxos's initial design was expensive (e.g., it required three communication steps), significant research efforts have focused on alternative designs for enhancing performance. A recent example is *JPaxos* [57, 98, 99], built on top of MultiPaxos [60], which extends Paxos to allow processes to agree on a sequence of values, instead of a single value. JPaxos incorporates optimizations such as batching and pipelining, which significantly boost message throughput [98]. *S-Paxos* [8] is another example that seeks to improve performance by balancing the load of the network protocol over all the nodes, instead of concentrating that on the leader.

A deterministic concurrency control protocol is needed for processing transactions according to the CSO. When transactions are delivered by the GCS, their commit order must coincide with the CSO; otherwise replicas will end up in different states. With deterministic concurrency control, each replica is aware of the existence of a new transaction to execute only after its delivery, significantly increasing transaction execution time. An optimistic solution to this problem has been proposed in [52], where an additional delivery, called *optimistic delivery*, is sent by the GCS to the replicas prior to the final CSO. This new delivery is used to start transaction execution speculatively, while guessing the final commit order. If the guessed order matches the CSO, then the transaction, which is already executed (totally or partially), is ready to commit [79, 80, 72]. However, guessing alternative serialization orders [96, 97] – i.e., activate multiple speculative instances of the same transactions starting from different memory snapshots – has non-trivial overheads, which, sometimes, do not pay off.

In this chapter, we present HiperTM, a high performance active replication protocol. HiperTM is based on an extension of S-Paxos, called *OS-Paxos* that we propose. OS-Paxos optimizes the S-Paxos architecture for efficiently supporting optimistic deliveries, with the aim of minimizing the likelihood of mismatches between the optimistic order and the final delivery order. The protocol wraps write transactions in transactional request messages and executes them on all the replicas in the same order. HiperTM uses a novel, speculative concurrency control protocol called SCC, which processes write transactions serially, minimizing code instrumentation (i.e., locks or CAS operations). When a transaction is optimistically delivered by OS-Paxos, its execution speculatively starts, assuming the optimistic order as the processing order. Avoiding atomic operations allows transactions to reach maximum performance in the time available between the optimistic and the corresponding final delivery. Conflict detection and any other more complex mechanisms hamper the protocol's ability to completely execute a sequence of transactions within their final notifications – so those are avoided.

For each shared object, the SCC protocol stores a list of committed versions, which is exploited by read-only transactions to execute in parallel to write transactions. As a consequence, write transactions are broadcast using OS-Paxos. Read-only transactions are directly delivered to one replica, without a CSO, because each replica has the same state, and are processed locally.

We implemented HiperTM and experimentally evaluated on *PRObE* [32], a high performance public cluster with 19 nodes[1] using benchmarks including TPC-C [20] and Bank. Our results reveal three important trends:

A) OS-Paxos provides a very limited number of out-of-order optimistic deliveries (0% when no failures happen and <5% in case of failures), allowing transactions processed – according to the optimistic order – to more likely commit.

B) Serially processing optimistically delivered transactions guarantees a throughput (transactions per second) that is higher than atomic broadcast service's throughput (messages per second), confirming optimistic delivery's effectiveness for concurrency control in actively replicated transactional systems. Additionally, the reduced number of CAS operations allows greater concurrency, which is exploited by read-only transactions for executing faster.

[1]We selected 19 because, according to Paxos's rules, this is the minimum number of nodes to tolerate 9 simultaneous faults.

C) HiperTM's transactional throughput is up to 3.5× better than PaxosSTM [117], a state-of-the-art atomic broadcast-based competitor, using the classical configuration of TPC-C.

With HiperTM, we highlight the importance of making the right design choices for fault-tolerant DTM systems. To the best of our knowledge, HiperTM is the first fully implemented transaction processing system based on speculative processing, built in the context of active replication.

5.1 Optimistic S-Paxos

Optimistic S-Paxos (or OS-Paxos) is an implementation of optimistic atomic broadcast [85] built on top of S-Paxos [8]. S-Paxos can be defined in terms of two primitives (compliant with the atomic broadcast specification):

- *ABcast*(m): used by clients to broadcast a message m to all the nodes
- *Adeliver*(m): event notified to each replica for delivering message m

These primitives satisfy the following properties:

- *Validity.* If a correct process *ABcast* a message m, then it eventually *Adeliver* m.
- *Uniform agreement.* If a process *Adeliver*s a message m, then all correct processes eventually *Adeliver* m.
- *Uniform integrity.* For any message m, every process *Adeliver*s m at most once, and only if m was previously *ABcast*ed.
- *Total order.* If some process *Adeliver*s m before m', then every process *Adeliver*s m and m' in the same order.

OS-Paxos provides an additional primitive, called *Odeliver*(m), which is used for delivering a previously broadcast message m before the *Adeliver* for m is issued. OS-Paxos ensures that:

- If a process *Odeliver*(m), then every correct process eventually *Odeliver*(m).
- If a correct process *Odeliver*(m), then it eventually *Adeliver*(m).
- A process *Adeliver*(m) only after *Odeliver*(m).

OS-Paxos's properties and primitives are compliant with the definition of optimistic atomic broadcast [85]. The sequence of *Odeliver* notifications defines the so called *optimistic order* (or *opt-order*). The sequence of *Adeliver* defines the so called *final order.* We now describe the architecture of S-Paxos to elaborate the design choices we made for implementing *Odeliver* and *Adeliver*.

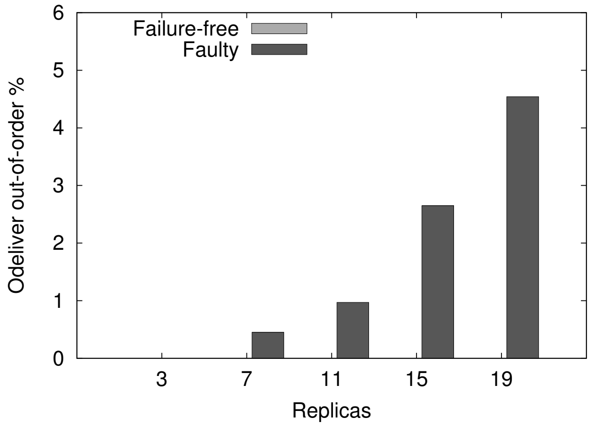
S-Paxos improves upon JPaxos with optimizations such as distributing the leader's load across all replicas. Unlike JPaxos, where clients only connect to the leader, in S-Paxos each replica accepts client requests and sends replies to connected clients after the execution of the requests. S-Paxos extensively uses the batching technique [98, 99] for increasing throughput. A replica creates a batch of client requests and distributes it to other replicas. The receiver replicas forward this batch to all other replicas. When the replicas observe a majority of delivery for a batch, it is considered as stable batch. The leader then *proposes* an order (containing only batch IDs) for non-proposed stable batches, for which, the other replicas reply with their agreement i.e., *accept* messages. When a majority of agreements for a proposed order is reached (i.e., a consensus instance), each replica considers it as *decided.*

S-Paxos is based on the MultiPaxos protocol where, if the leader remains stable (i.e., does not crash), its proposed order is likely to be accepted by the other replicas. Also, there exists a non-negligible delay between the time when an order is proposed and its consensus is reached. As the number of replicas taking part in the consensus agreement increases, the time required to reach consensus becomes substantial. Since the likelihood of a proposed order to get accepted is high with a stable leader, we exploit the time to reach consensus and execute client requests speculatively without commit. When the leader sends the proposed order for a batch, replicas use it for triggering *Odeliver*. On reaching consensus agreement, replicas fire the *Adeliver* event, which commits all speculatively executed transactions corresponding to the agreed consensus.

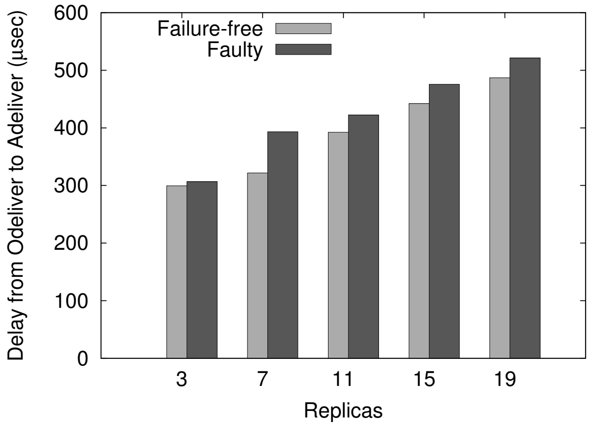
Network non-determinism presents some challenges for the implementation of *Odeliver* and *Adeliver* in S-Paxos. First, S-Paxos can be configured to run multiple consensus instances (i.e., pipelining) to increase throughput. This can cause out-of-order consensus agreement e.g., though an instance a precedes instance b, b may be agreed before a. Second, the client's request batch is distributed by the replicas before the leader could propose the order for them. However, a replica may receive a request batch after the delivery of a proposal that contains it (due to network non-determinism). Lastly, a proposal message may be delivered after the instance is decided.

We made the following design choices to overcome these challenges. We trigger an *Odeliver* event for a proposal only when the following conditions are met: 1) the replica receives a propose message; 2) all request batches of the propose message have been received; and 3) *Odeliver* for all previous instances have been triggered i.e., there is no "gap" for *Odeliver*ed instances. A proposal can be *Odeliver*ed either when a missing batch from another replica is received for a previously proposed instance, or when a proposal is received for the previously received batches. We delay the *Odeliver* until we receive the proposal for previously received batches to avoid out-of-order speculative execution and to minimize the cost of aborts and retries.

The triggering of the *Adeliver* event also depends on the arrival of request batches and the majority of accept messages from other replicas. An instance may be decided either after the receipt of all request batches or before the receipt of a delayed batch corresponding to the instance. It is also possible that the arrival of the propose message and reaching consensus is the same event (e.g., for a system of 2 replicas). In such cases, *Adeliver* events immediately follow *Odeliver*. Due to these possibilities, we fire the *Adeliver* event when: *1)* consensus is reached for a proposed message; and *2)* a missing request batch for a decided instance is received; and *3)* the corresponding instance has been *Odeliver*ed. If there is any out-of-order instance agreement, *Adeliver* is delayed until all previous instances are *Adeliver*ed.



(a) % of out-of-order *Odeliver* w.r.t. *Adeliver*



(b) Time between *Odeliver* and *Adeliver*

Figure 5.1: OS-Paxos performance.

In order to assess the effectiveness of our design choices, we conducted experiments measuring the percentage of reordering between OS-Paxos's optimistic and final deliveries, and the average time between an *Odeliver* and its subsequent *Adeliver*. We balanced the clients injecting requests on all the nodes and we reproduced executions without failures (Failure-free) and manually crashing the actual leader (Faulty). Figure 5.1 shows the results. The experimental test-bed is the same used for the evaluation of HiperTM in Section 5.5 (briefly, we used 19 nodes interconnected via 40 Gbits network on PRObE [32] public cluster).

Reordering (Figure 5.1(a)) is absent for failure-free experiments (Therefore the bar is not visible in the plot). This is because, if the leader does not fail, then the proposing order is always confirmed by the final order in OS-Paxos. Inducing leader to crash, some reorder appears starting from 7 nodes. However, the impact on the overall performance is limited because the maximum number of reordering observed is lower than 5% with 19 replicas. This confirms that the optimistic delivery order is an effective candidate for the final execution order. Figure 5.1(b) shows the average delay between *Odeliver* and *Adeliver*. It is in the rage of ≈300 microseconds to ≈500 microseconds in case of failure-free runs and it increases up to ≈550 microseconds when leader crashes. The reason is related to the possibility that the process of sending the proposal message is interrupted by a fault, forcing the next elected leader to start a new agreement on previous messages.

The results highlight the trade-off between a more reliable optimistic delivery order and the time available for speculation. On one hand, anticipating the optimistic delivery results in additional time available for speculative processing transactions, at the cost of having an optimistic delivery less reliable. On the other hand, postponing the optimistic delivery brings an optimistic order that likely matches the final order, restricting the time for processing. In HiperTM we preferred this last configuration and we designed a lightweight protocol for maximizing the exploitation of the time between *Odeliver* and *Adeliver*.

5.2 The Protocol

Application threads (clients), after invoking a transaction using the *invoke* API, wait until the transaction is successfully processed by the replicated system and its outcome becomes available. Each client has a reference replica for issuing requests. When that replica becomes unreachable or a timeout expires after the request's submission, the reference replica is changed and the request is submitted to another replica. Duplication of requests is handled by tagging messages with unique keys composed of client ID and local sequence number.

Replicas know about the existence of a new transaction to process only after the transaction's *Odeliver*. The opt-order represents a possible, non definitive, serialization order for transactions. Only the sequence of *Adeliver*s determines the final commit order. HiperTM overlaps the execution of optimistically delivered transactions with their coordination phase (i.e., defining the total order among all replicas) to avoid processing those transactions from scratch after their *Adeliver*. Clearly, the effectiveness of this approach depends on the likelihood that the opt-order is consistent with the final order. In the positive case, transactions are probably executed and there is no need for further execution. Conversely, if the final order contradicts the optimistic one, then the executed transactions can be in one of the following two scenarios: *i)* their serialization order is "equivalent" to the serialization order defined by the final order, or *ii)* the two serialization orders are not "equivalent". The notion of equivalence here is related to transactional conflicts: when two transactions are non-conflicting, their processing order is equivalent.

Consider four transactions. Suppose $\{T_1,T_2,T_3,T_4\}$ is their opt-order and $\{T_1,T_4,T_3,T_2\}$ is their final order. Assume that the transactions are completely executed when the respective *Adeliver*s are issued. When $Adeliver(T_4)$ is triggered, T_4's optimistic order is different from its final order. However, if T_4 does not conflict with T_3 and T_2, then its serialization order, realized during execution, is equivalent to the final order, and the transaction can be committed without re-execution (case *i)*). On the contrary, if T_4 conflicts with T_3 and/or

T_2, then T_4 must be aborted and restarted in order to ensure replica consistency (case *ii)*). If conflicting transactions are not committed in the same order on all replicas, then replicas could end up with different states of the shared data-set, violating correctness (i.e., the return value of a read operation can be different if it is executed on different replicas).

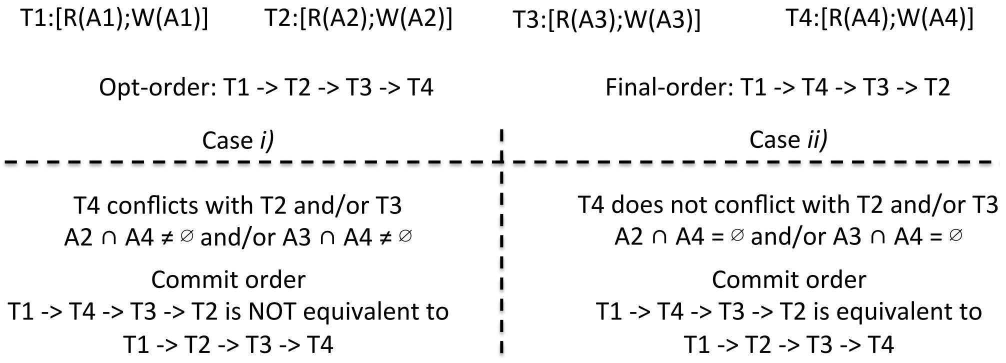

T1:[R(A1);W(A1)]
T2:[R(A2);W(A2)]
T3:[R(A3);W(A3)]
T4:[R(A4);W(A4)]
Opt-order: T1 -> T2 -> T3 -> T4
Final-order: T1 -> T4 -> T3 -> T2
Case i)
Case ii)
T4 conflicts with T2 and/or T3
A2 ∩ A4 ≠ ∅ and/or A3 ∩ A4 ≠ ∅
Commit order
T1 -> T4 -> T3 -> T2 is NOT equivalent to
T1 -> T2 -> T3 -> T4
T4 does not conflict with T2 and/or T3
A2 ∩ A4 = ∅ and/or A3 ∩ A4 = ∅
Commit order
T1 -> T4 -> T3 -> T2 is equivalent to
T1 -> T2 -> T3 -> T4

Figure 5.2: Example of committing transactions $\{T_1,T_2,T_3,T_4\}$ varying the conflict of accessed objects, in case the final order contradicts the optimistic order.

Figure 5.2 pictures the previous two cases. For the sake of clarity, we assume each transaction performing one read operation and one write operation on the same object. We distinguish between case *i)* and case *ii)* by, respectively, assigning different values to accessed objects (left column in the figure) or same values (right column in the figure). However, in both the cases, transaction T_4 reads and writes the same object managed by T_1, thus $A1$ is equals to $A4$ (due to the compact representation of the example, each object's name is different but it can refer to same object). In case *i)*, where object $A2$ (or $A3$) is the same as $A4$, the validation of T_4 after T_1 cannot complete successfully because the value of $A2$ (or $A3$) read by T_4 does not correspond to the actual committed value in memory, namely the one written by T_1. On the contrary, the right column shows the case *ii)* where object $A2$ (or $A3$) is different from $A4$. This way, T_4 can successfully validate and commit even if its speculative execution order was different. This is because the actual dependencies with other transactions of T_4 are the same as those in the final order (i.e., T_1 has to commit before T_4). As a result, T_1 is still committed before T_4, allowing T_4 to commit too.

5.2.1 Write Transaction Processing

We use the speculative processing technique for executing optimistically (but not yet finally) delivered write transactions. (We recall that only write transactions are totally ordered through OS-Paxos). This approach has been proposed in [52] in the context of traditional DBMS. In addition to [52], we do not limit the number of speculative transactions executed in parallel with their coordination phase, and we do not assume a-priori knowledge on transactions' access patterns. Write transactions are processed serially, without parallel activation (see Section 5.3 for complete discussion). Even though this approach appears inconsistent with the nature of speculative processing, it has several benefits for in-order processing, which increase the likelihood that a transaction will reach its final stage before its *Adeliver* is issued.

In order to allow next conflicting transaction to process speculatively, we define a *complete buffer* for each shared object. In addition to the last committed version, shared objects also maintain a single memory slot (i.e., the complete buffer), which stores the version of the object written by the last completely executed optimistic transaction. The complete buffer could be empty if no transactions wrote a new version of that object after the previous version became committed. We do not store multiple completed versions because, executing transactions serially needs only one uncommitted version per object. When an *Odeliver*ed transaction performs a read operation, it checks the complete buffer for the presence of a version. If the buffer is empty, the last committed version is considered; otherwise, the version in the complete buffer is accessed. When a write operation is executed, the complete buffer is immediately overwritten with the new version. This early publication of written data in memory is safe because of serial execution. In fact, there are no other write transactions that can access this version before the current transaction completes.

After executing all its operations, an optimistically delivered transaction waits until *Adeliver* is received. In the meanwhile, the next *Odeliver*ed transaction starts to execute. When an *Adeliver* is notified by OS-Paxos, a handler is executed by the same thread that is responsible for speculatively processing transactions. This approach avoids interleaving with transaction execution (which causes additional synchronization overhead). When a transaction is *Adeliver*ed, if it is completely executed, then it is validated for detecting the equivalence between its actual serialization order and the final order. The validation consists of comparing the versions read during the execution. If they correspond with the actual committed version of the objects accessed, then the transaction is valid, certifying that the serialization order is equivalent to the final order. If the versions do not match, the transaction is aborted and restarted. A transaction *Adeliver*ed and aborted during its validation can re-execute and commit without validation due to the advantage of having only one thread executing write transactions.

The commit of write transactions involves moving the written objects from transaction local buffer to the objects' last committed version. In addition, each object maintains also a list of previously committed versions, which is exploited by read-only transactions to execute independently from the write transactions. In terms of synchronization required, the complete buffer can be managed without it because only one write transaction is active at a time. On the other hand, installing a new version as committed requires synchronization because of the presence of multiple readers (i.e., read-only transactions) while the write transaction could (possibly) update the list.

5.2.2 Read-Only Transaction Processing

Read-only transactions are marked by programmers and they are not broadcast using OS-Paxos, because they do not need to be totally ordered. When a client invokes a read-only transaction, it is locally delivered and executed in parallel to write transactions by a separate pool of threads. In order to support this parallel processing, we define a timestamp for each replica, called *replica-timestamp*, which represents a monotonically increasing integer, incremented each time a write transaction commits. When a write transaction enters its commit phase, it assigns the replica-timestamp to a local variable, called *c-timestamp*, representing the committing timestamp, increases the c-timestamp, and tags the newly committed versions with this number. Finally, it updates the replica-timestamp with the c-timestamp.

When a read-only transaction performs its first operation, the replica-timestamp becomes the transaction's timestamp (or *r-timestamp*). Subsequent operations are processed according to the r-timestamp: when an object is accessed, its list of committed versions is traversed in order to find the most recent version with a timestamp lower or equal to the r-timestamp. After completing execution, a read-only transaction is committed without validation. The rationale for doing so is as follows. Suppose T_R is the committing read-only transaction and T_W is the parallel write transaction. T_R's r-timestamp allows T_R to be serialized *a)* after all the write transactions with a c-timestamp lower or equal to T_R's r-timestamp; and *b)* before T_W's c-timestamp and all the write transactions committed after T_W. T_R's operations access versions consistent with T_R's r-timestamp. This subset of versions cannot change during T_R's execution, and therefore T_R can commit safely without validation.

Whenever a transaction commits, the thread managing the commit wakes-up the client that previously submitted the request and provides the appropriate response.

5.3 Speculative Concurrency Control

In HiperTM, each replica is equipped with a local speculative concurrency control, called SCC, for executing and committing transactions enforcing the order notified by OS-Paxos. In order to overlap the transaction coordination phase with transaction execution, write transactions are processed speculatively as soon as they are optimistically delivered. The main purpose of the SCC is to completely execute a transaction, according to the opt-order, before its *Adeliver* is issued. As shown in Figure 5.2, the time available for this execution is limited.

Motivated by this observation, we designed SCC. SCC exploits multi-versioned memory for activating read-only transactions in parallel to write transactions that are, on the contrary, executed on a single thread. The reason for single-thread processing is to avoid the overhead for detecting and resolving conflicts according to the opt-order while transactions are executing. During experiments on the standalone version of SCC, we found it to be capable of processing ≈95K write transactions per second, while ≈250K read-only transactions in parallel on different cores (we collected these results using Bank benchmark on experimental test-bed's machine). This throughput is higher than HiperTM's total number of optimistically delivered transactions speculatively processed per second, illustrating the effectiveness of single-thread processing.

Single-thread processing ensures that when a transaction completes its execution, all the previous transactions are executed in a known order. Additionally, no atomic operations are needed for managing locks or critical sections. As a result, write transactions are processed faster and read-only transactions (executed in parallel) do not suffer from otherwise overloaded hardware bus (due to CAS operations and cache invalidations caused by spinning on locks) and they are also never stopped.

Transactions log the return values of their read operations and written versions in private read- and write-set, respectively. The write-set is used when a transaction is *Adeliver*ed for committing its written versions in memory. However, for each object, there is only one uncommitted version available in memory at a time, and it corresponds to the version written by the last optimistically delivered and executed transaction. If more than one speculative transaction wrote to the same object, both are logged in their write-sets, but only the last one is stored in memory in the object's complete buffer. We do not need to record a list of speculative versions, because transactions are processed serially and only the last can be accessed by the current executing transaction.

Algorithm 1 Read Operation of Transaction T_i on Object X.

```
if $T_i$.readOnly = FALSE then
    if $\exists$ version $\in X$.completeBuffer then
        $T_i$.ReadSet.add($X$.completeBuffer)
        return $X$.completeBuffer.value
    else
        $T_i$.ReadSet.add($X$.lastCommittedVersion)
        return $X$.lastCommittedVersion.value
    end if
else
    if r-timestamp = 0 then
        r-timestamp $\leftarrow X$.lastCommittedVersion.timestamp
        return $X$.lastCommittedVersion.value
    end if
    $P \leftarrow$ {set of versions $V \in X$.committedVersions s.t. $V$.timestamp $\leq$ r-timestamp
    if $P \neq \emptyset$ then
        $V_{cx} \leftarrow \exists$ version $V_k \in P$ s.t. $\forall V_q \in P \Rightarrow V_k$.timestamp $\geq V_q$.timestamp  $\triangleright$ $V_{cx}$ has the maximum timestamp in $P$
        return $V_{cx}$.value
    else
        return $X$.lastCommittedVersion.value
    end if
end if
```

Algorithm 2 Write Operation of Transaction T_i on Object X writing the Value v.

```
Version $V_x \leftarrow$ createNewVersion($X$,$v$)
$X$.completeBuffer $\leftarrow V_x$
$T_i$.WriteSet.add($V_x$)
```

The read-set is used for validation. Validation is performed by simply verifying that all the objects accessed correspond to the last committed versions in memory. When the optimistic order matches the final order, validation is redundant, because serially executing write transactions ensures that all the objects accessed are the last committed versions in memory. Conversely, if an out-of-order occurs, validation detects the wrong speculative serialization order.

Consider three transactions, and let $\{T_1,T_2, T_3\}$ be their opt-order and $\{T_2,T_1, T_3\}$ be their final order. Let T_1 and T_2 write a new version of object X and let T_3 reads X. When T_3 is speculatively executed, it accesses the version generated by T_2. But this version does not correspond to the last committed version of X when T_3 is *Adelivered*. Even though T_3's optimistic and final orders are the same, it must be validated to detect the wrong read version. When a transaction T_A is aborted, we do not abort transactions that read from T_A (cascading abort), because doing so will entail tracking transaction dependencies, which has a non-trivial overhead. Moreover, a restarted transaction is still executed on the same processing thread. That is equivalent to SCC's behavior, which aborts and restarts a transaction when its commit validation fails.

Algorithm 3 Validation Operation of Transaction T_i.

```
1: for all $V_x \in T_i$.ReadSet do
2:     if $V_x \neq X$.lastCommittedVersion then
3:         return FALSE
4:     end if
5: end for
6: return TRUE
```

Algorithm 4 Commit Operation of Transaction T_i.

```
1: if Validation($T_i$) = FALSE then
2:     return $T_i$.abort&restart
3: end if
4: c-timesamp ← replica-timestamp
5: c-timesamp *gets* c-timesamp + 1
6: for all $V_x \in T_i$.WriteSet do
7:     $V_x$.timestamp ← c-timestamp
8:     $X$.lastCommittedVersion ← $V_x$
9: end for
10: replica-timestamp = c-timesamp
```

A task queue is responsible for scheduling jobs executed by the main thread (processing write transactions). Whenever an event such as *Odeliver* or *Adeliver* occurs, a new task is appended to the queue and is executed by the thread after the completion of the previous tasks. This allows the events' handlers to execute in parallel without slowing down the executor thread, which is the SCC's performance-critical path.

As mentioned, read-only transactions are processed in parallel to write transactions, exploiting the list of committed versions available for each object to build a consistent serialization order. The growing core count of current and emerging multicore architectures allows such transactions to execute on different cores, without interfering with the write transactions.

One synchronization point is present between write and read transactions, i.e., the list of committed versions is updated when a transaction commits. In order to minimize its impact on performance, we use a concurrent sorted Skip-List for storing the committed versions.

The pseudo code of SCC is shown in Algorithms 1-4. We show the core steps of the concurrency control protocol such as reading a shared object (Algorithm 1), writing a shared object (Algorithm 2), validating a write transaction (Algorithm 3) and committing a write transaction (Algorithm 4).

5.4 Properties

HipertTM satisfies a set of properties that can be classified as local to each replica and global to the replicated system as a whole. For what concern the former, each replica has a concurrency control that operates isolated, without interactions with other nodes. For this reason, we can infer properties that hold for non distributed interactions. On the other side, a client of HiperTM system does not see specific properties local to each replica because the system is hidden by the semantic of API exposed (i.e., `invoke`).

We name a property as global if it holds for the distributed system as a whole. Specifically, a property is global if there is no execution involving distributed events such that the property is not ensured. In other words, the property should work for transactions executing within the bounds of single node, as well as involving transactions (concurrent or not) executing or executed on other nodes.

5.4.1 Formalism

We now introduce the formalism that will be used for proving HiperTM's correctness properties.

According to the definition in [1], an history H is a partial order on the sequence of operations Op executed by the transactions, where Op's values are in the set $\{begin, read, write, commit, abort\}$. When a transaction T_i performs the above operations, we name them as b_i, c_i, a_i respectively. In addition, a write operation of T_i on a the version k of the shared object x is denoted as $w_i(x_k)$; and we refer a read operation the corresponding read operation as $r_i(x_k)$. In addition H implicitly induces a total order $\ll$ on committed object versions [1].

We now use a direct graph as a representation of an history H where committed transaction in H are the graph's vertexes and there exists a directed edge between two vertexes if the respective transactions are conflicting. We name this graph as Direct Serialization Graph (or $DSG(H)$). More formally, a vertex in DSG is denoted as V_{T_i} and represents the committed transaction T_i in H. Two vertexes V_{T_i} and V_{T_j} are connected with an edge if T_i and T_j are conflicting transactions, namely there are two operations Op_i and Op_j in H, performed by

T_i and T_j respectively, on a common shared object, such that at least one of them is a write operation.

We distinguish three types of edges depending on the type of conflicts between T_i and T_j:

- *Direct read-dependency* edge if there exists an object x such that both $w_i(x_i)$ and $r_j(x_i)$ are in H. We say that T_j directly read-depends on T_i and we use the notation $V_{T_i} \xrightarrow{wr} V_{T_j}$.
- *Direct write-dependency* edge if there exists an object x such that both $w_i(x_i)$ and $w_j(x_j)$ are in H and x_j immediately follows x_i in the total order defined by $\ll$. We say that T_j *directly write-depends* on T_i and we use the notation $V_{T_i} \xrightarrow{ww} V_{T_j}$.
- *Direct anti-dependency* edge if there exists an object x and a committed transaction T_k in H, with $k \neq i$ and $k \neq j$, such that both $r_i(x_k)$ and $w_j(x_j)$ are in H and x_j immediately follows x_k in the total order defined by $\ll$. We say that T_j *directly anti-depends* on T_i and we use the notation $V_{T_i} \overset{rw}{\dashrightarrow} V_{T_j}$.

Finally, it is worth to recall two important aspects of HiperTM that will be used in the proof.

- **(SeqEx)**. HiperTM processes write transactions serially, without interleaving their executions. This means that for any pair of operations Op_i^1 and Op_i^2 performed by a transaction T_i such that Op_i^1 is executed before Op_i^2, there is no operation Op_j, invoked by a write transaction T_j, that can be executed in between Op_i^1 and Op_i^2 by the HiperTM's local concurrency control.
- **(ParRO)**. The second aspect is related to the read-only transactions. When such a transaction starts, it cannot observe objects written by write transactions committed after the starting time of the read-only transaction. Intuitively, the read-only transaction, thanks to the multi-versioning, could read in the past. This mechanism allows read-only transactions to fix the set of available versions to read at the beginning of their execution, without taking into account concurrent commits.

5.4.2 Global Properties

For the purpose of the following proofs, we scope out the speculative execution when the transactions are optimistically delivered. In fact, this execution is only an anticipation of the execution that happens when a transaction is final delivered. For the sake of clarity, we assume that a transaction T is activated as soon as the final delivery for T is received. This assumption does not limit the generality of the proofs because any transaction speculative executed is validated when the relative final delivery is received (Algorithm 4, Line 1). If the speculative order does not match the final order, then the transaction is re-executed

(Algorithm 4, Line 2). Thus the speculative execution can be seen only for improving performance, but in terms of correctness, only the execution after the final delivery matters. In fact, speculative transactions are not committed. The validation (Algorithm 3) performs a comparison between the read versions of the speculative execution with actual committed versions in memory. Due to (SeqEx), there are no concurrent transactions validating at the same time, thus if, the validation succeeds, then the transaction does not need the re-execution, otherwise it is re-executed from the very beginning.

Theorem 1. *HiperTM guarantees 1-copy serializability (1CS) [6], namely for each run of the protocol the set of committed transactions appear as they are executed sequentially, i.e. whichever pair of committed transactions T_i, T_j, serialized in this order, every operation of T_i precedes in time all the operations of T_j as executed on a single copy of the shared state.*

Proof. We conduct this proof relying on the DSG. In particular, as also stated in [6], a history H with a version order $\ll$ is 1-copy serializable if the $DSG(H)$ on H does not contain any oriented cycle.

To show the acyclicity of the $DSG(H)$ graph, we first prove that for each history H, every transaction committed by the protocol appears as instantaneously executed in a unique point in time t (*Part1*); subsequently we rely on those t values to show a mathematical absurd confirming that $DSG(H)$ cannot contain any cycle (*Part2*).

In order to prove *Part1* of the proof, we assign to each transaction T committed in H a commit timestamp, called *CommitOrd(T,H)*. $CommitOrd(T, H)$ defines the time where the transaction T appears committed in H. We distinguish two cases, namely when T is a write transaction or a read-only transaction.

- If T is a write transaction, $CommitOrd(T, H)$ is the commit timestamp of T in H (Algorithm 4 Line 4) , which matches also the final order that OS-Paxos assigned to T. This is because: *i)* OS-Paxos defines a total order among all write transactions and, *ii)*, (SeqEx) does not allow interleaving of operations' executions. This way, given a history H, $CommitOrd(T, H)$ is the time when T commits its execution in H and no other write transaction executes concurrently.

- If T is a read-only transaction, $CommitOrd(T, H)$ is the node's timestamp (Algorithm 1 Line 11) when T starts in H (read-only transactions are delivered and executed locally to one node, without remote interactions). In fact, (ParRO) prevents the read-only transaction to interfere with executing write transaction, implicitly serializing the transaction before those write transactions. In this case, $CommitOrd(T, H)$ is the timestamp that precedes any other commit made by write transactions after T started.

According to the definition of $DSG(H)$, there is an edge between two vertexes V_{T_i} and V_{T_j} when T_i and T_j are conflicting transaction in H. We now show that, if such an edge exists, then $CommitOrd(T_i, H) \leq CommitOrd(T_j, H)$. We do this considering the scenarios where H is only composed of write transactions (WOnly), then we extend it integrating read-only transaction (RW).

(WOnly).

- If there is a direct read-dependency between T_i and T_j (i.e., $V_{T_i} \xrightarrow{wr} V_{T_j}$), then it means that there exists an object version x_i that has been written by T_i and read by T_j via $r_j(x_i)$. Since (SeqEx), T_i and T_j cannot interleave their executions thus all the object versions accessed by T_j have been already committed by T_i before T_j starts its execution. If T_j starts after T_i means also that $CommitOrd(T_i, H) < CommitOrd(T_j, H)$.
- Similar argument can be made if T_j directly write-depends on T_i ($V_{T_i} \xrightarrow{ww} V_{T_j}$). Here both T_i and T_j write a version of object x, following the order T_i, T_j (i.e., T_j overwrites the version written by T_i). As before, through (SeqEx) we can infer that $CommitOrd(T_i, H) < CommitOrd(T_j, H)$.
- If T_j directly anti-depends on T_i ($V_{T_i} \overset{rw}{\dashrightarrow} V_{T_j}$), it means that there exists an object x such that T_i reads some object version x_i and T_j writes a new version of x, namely x_j, after T_i. By the definition of directly anti-dependency and given that the transaction execution is serial (SeqEx), it follows that, if T_j creates a new version of x after T_i read x, then T_i committed its execution before activating T_j, thus $CommitOrd(T_i, H)$ $< CommitOrd(T_j, H)$.

(RW).
If we enrich a history H with read-only transactions, the resulting $DSG(H)$ contains at least a vertex V_{T_r}, corresponding to the read-only transaction T_r, such that, due to (ParRO), the only type of outgoing edge that is allowed to connect V_{T_r} to any other vertex, namely an edge where V_{T_r} is the source vertex, is a directly anti-dependency edge. In fact, no other transaction can have any direct read-dependency or direct write-dependency with T_r, because T_r does not create new object versions. In this case, say T_r the transaction reading the object version x_r and T_w the transaction writing a new version of x, called x_w. Due to (ParRO), any concurrent write transaction (such as T_w), that commits after T_r's begin, acquires a timestamp that is greater than T_r's timestamp, thus also the new versions committed by T_w(such as x_w) are tagged with a higher timestamp. This prevents read-only transactions to access those new versions. In other words, T_r cannot see the modifications made by T_w after its commit. This serializes T_w's commit operation after T_r's execution, thus $CommitOrd(T_r, H) < CommitOrd(T_w, H)$.

Nevertheless, V_{T_r} is clearly connected with edges from other vertexes corresponding to write transactions (T_w) previously committed. In this case, due to (ParRO), $CommitOrd(T_r, H)$ is not strictly greater than $CommitOrd(T_w, H)$ but $CommitOrd(T_w, H) \leq CommitOrd(T_r, H)$ because otherwise T_r is always forced to read in the past even having fresher object versions committed before T_r's starting. However this does not represent a limitation because, if a cycle on DSG involves a vertex that represents a read-only transaction (V_{T_r}), then all its outgoing edges will connect to vertexes with a $CommitOrd$ strictly greater than the $CommitOrd(T_r, H)$.

We have proved that for each $DSG(H)$ on H and for each $V_{T_i} \rightarrow V_{T_j}$ edge in $DSG(H)$, $CommitOrd(T_i, H) \leq CommitOrd(T_j, H)$ holds. In order to prove *Part2*, we now show that $DSG(H)$ cannot contain any oriented cycle. To do that, we observe that, if $DSG(H)$ is composed of only write transactions, then $CommitOrd(T_i, H) < CommitOrd(T_j, H)$. In addition, if there is a path in $DSG(H)$ that is: $T_{W0}, T_{W1} \ldots T_{Wi}, T_R, T_{Wi+1}, \ldots T_{Wn}$ where T_{Wi} is the i-th write transaction and T_R is the read-only transaction, then $CommitOrd(T_R, H)$ $< CommitOrd(T_{Wi+1}, H)$. Having said that, we can now show why $DSG(H)$ cannot have cycles involving only write transactions or read-only transactions. This is because, if such a cycle existed it would lead to the following absurd: for each V_{T_i} in the cycle we have $CommitOrd(T_i, H) < CommitOrd(T_i, H)$. □

Theorem 2. *HiperTM guarantees wait freedom of read-only transactions [43], namely that any process can complete a read-only transaction in a finite number of steps, regardless of the execution speeds of the other processes.*

Proof. Due to (ParRO) and the (SeqEx), the proof is straightforward. In fact, with (SeqEx) there are no locks on shared objects [35] (one transaction processes and commits at a time). The only synchronization point between a read-only transaction and a write transaction is the access to the version list of objects. However, those lists are implemented as wait-free [44], thus concurrent operations on the shared list always complete. This prevents the thread executing write transactions to possibly stop (or slow-down) the execution of a read-only transaction.

In addition, read-only transactions cannot abort. Before issuing the first operation, a read-only transaction saves the replica-timestamp in its local r-timestamp and use it for selecting the proper committed versions to read. The acquisition of the replica-timestamp always completes despite any behavior of other threads because the increment of the replica-timestamp does not involve any lock acquisition, rather we use atomic-increment operations. The subset of all the versions that the read-only transaction can access during its execution is fixed when the transaction defines its r-timestamp. Only one write transaction, T_W, is executing when a read-only transaction, T_{RO} acquires the r-timestamp. Due to the atomicity of replica-timestamp's update, the acquisition of the r-timestamp can only happen before or after the atomic increment.

i) If T_W updates the replica-timestamp before T_{RO} acquires the r-timestamp, T_{RO} is serialized after T_W, but before the next write transaction that will commit.

ii) On the contrary, if the replica-timestamp's update happens after, T_{RO} is serialized before T_W and cannot access the new versions that T_W just committed.

In both cases, the subset of versions that T_{RO} can access is defined and cannot change due to future commits. For this reason, when a read-only transaction completes its execution, it returns the values to its client without validation. □

5.4.3 Local Properties

HiperTM guarantees a variant of opacity [34] locally to each replica. It cannot ensure Opacity as defined in [34] because of the speculative execution.

In fact, even if such execution is serial, data written by a committed transaction are made available to the next speculative execution. This usually happens before the actual commit of the transaction, which occurs only after its final order is notified. Opacity can be summarized as follow: a protocol ensures opacity if it guarantees three properties: (Op.1) committed and aborted transactions appear as if they are executed serially, in an order equivalent to their real-time order; (Op.2) no transaction accesses a snapshot generated by a live (i.e., still executing) or aborted transaction.

As an example, say H an history of transactions $\{T_1, T_2, T_3\}$ reading and writing the same shared objects (i.e., T_1=T_2=T_3=[read(x); write(x)]). The optimistic order defines the following execution: T_1, T_2, T_3. We now assume that the final order for those transactions is not yet defined.

T_1 starts as soon as it is optimistic delivered. It completes its two operations and, according to the serial speculative execution, T_2 starts. Clearly T_2 accesses to the version of object x written by T_1 before T_1 actually commits (that will happen when the final order of T_1 will be delivered), breaking (Op.2).

However, we can still say that HiperTM guarantees a variant of opacity if we assume one of these two scenarios.

a) The speculative execution is just an anticipation of the real execution that happens when a transaction is final delivered. The validation procedure is responsible for decoupling speculative and non-speculative execution. This way, we can scope out the speculative execution and analyze only the execution after the final delivery of transactions.

b) We can enrich the type of operations admitted by opacity with the *speculative commit*. Given that, when a transaction completes its speculative execution, it does the speculative commit, exposing new versions to only other speculative transactions.

We show this by addressing all the above clauses of opacity and, considering that this is a local property (i.e., valid within the bound of a replica), we will refer to HiperTM as SCC.

SCC satisfies (Op.1) because each write transaction is validated before commit, in order to certify that its serialization order is equivalent to the optimistic atomic broadcast order, which reflects the order of the client's requests. When a transaction is aborted, it is only because its serialization order is not equivalent to the final delivery order (due to network reordering). However that serialization order has been realized by a serial execution. Therefore, the transaction's observed state is always consistent. Read-only transactions perform their operations according to the r-timestamp recorded from the replica-timestamp before their first read. They access only the committed versions written by transactions with the highest c-timestamp lower or equal to the r-timestamp. Read-only transactions with the same r-timestamp have the same serialization order with respect to write transactions. Conversely, if they have different r-timestamps, then they access only objects committed by transactions serialized before.

(Op.2) is guaranteed for write transactions because they are executed serially in the same thread. Therefore, a transaction cannot start if the previous one has not completed, preventing it from accessing modifications made by non-completed transactions. Under SCC, optimistically delivered transactions can access objects written by previous optimistically (and not yet finally) delivered transactions. However, due to serial execution, transactions cannot access objects written by non-completed transactions. (Op.2) is also ensured for read-only transactions because they only access committed versions.

5.5 Implementation and Evaluation

HiperTM's architecture consists of two layers: network layer (OS-Paxos) and replica speculative concurrency control (SCC). We implemented both in Java: OS-Paxos as an extension of S-Paxos, and SCC from scratch. To evaluate performance, we used two benchmarks: Bank and TPC-C [20]. Bank emulates a monetary application and is typically used in TM works for benchmarking performance [117, 87, 21]. TPC-C [20] is a well known benchmark that is representative of on-line transaction processing workloads.

We used PaxosSTM [117, 55] as a competitor. PaxosSTM implements the deferred update replication scheme and relies on a non-blocking transaction certification protocol, which is based on atomic broadcast (provided by JPaxos).

We used the *PRObE* testbed [32], a public cluster that is available for evaluating systems research. Our experiments were conducted using 19 nodes in the cluster. Each node is a physical machine equipped with a quad socket, where each socket hosts an AMD Opteron 6272, 64-bit, 16-core, 2.1 GHz CPU (total 64-cores). The memory available is 128GB, and the network connection is a 40 Gigabits Ethernet.

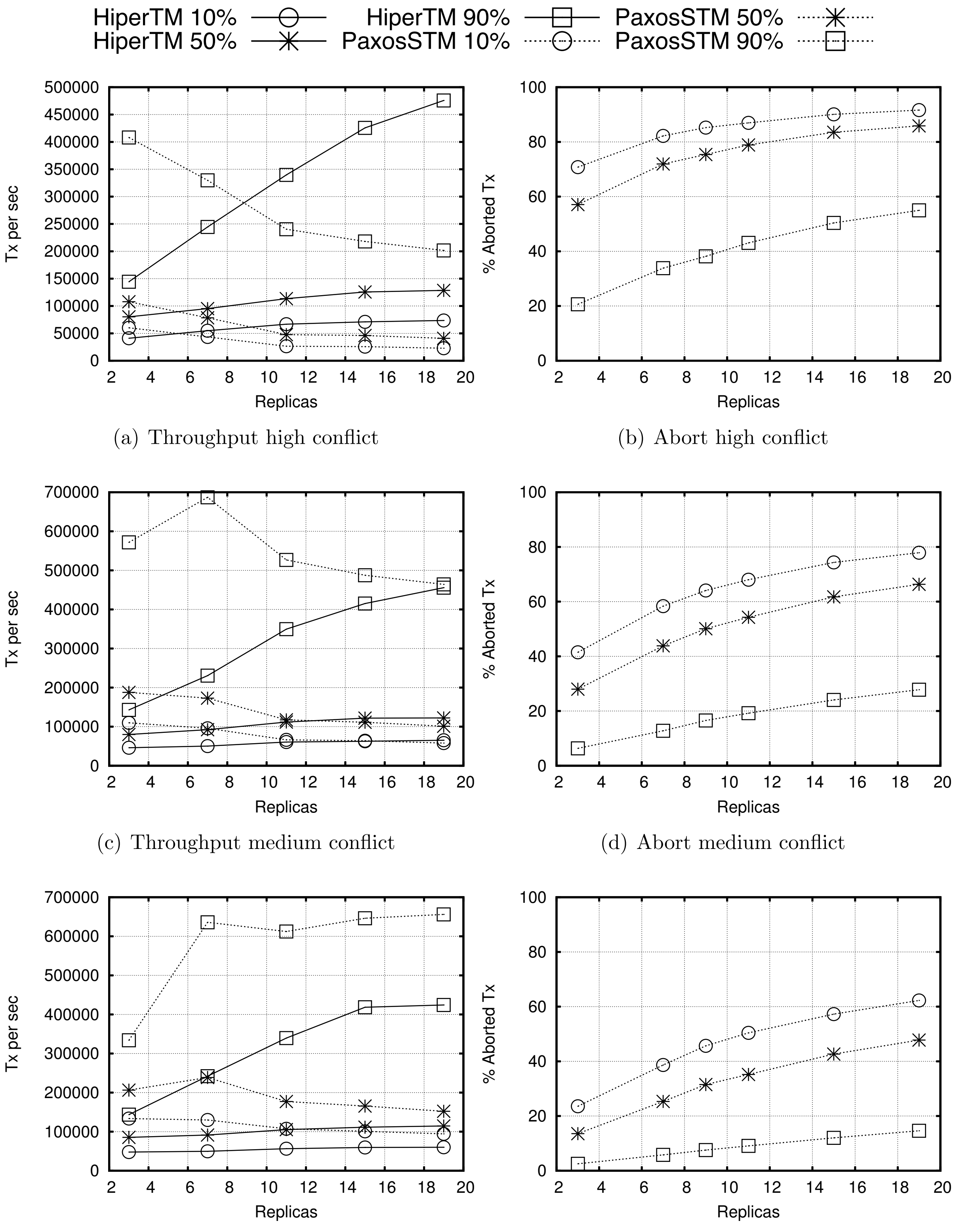
HiperTM is configured with a pool of 20 threads serving read-only transactions while a single thread is reserved for processing write transactions delivered by OS-Paxos. Clients are balanced on all the replicas. They inject transactions for the benchmark and wait for the reply. We configured PaxosSTM for working with the same configuration used in [55]: 160 parallel threads per nodes are responsible to execute transactions while JPaxos (i.e., the total order layer) leads their global certification. Data points plotted are the average of 6 repeated experiments.

5.5.1 Bank Benchmark

Bank benchmark is characterized by short transactions with few objects accessed (i.e., in the range of 2-4 objects), resulting in small transactions' read-set and write-set. A sanity check is implemented to test the correctness of the execution. The nature of this benchmark causes very high performance.

In order to conduct an exhaustive evaluation, we changed the application workload such that strengths and weaknesses of HiperTM are highlighted. Specifically, we varied the percentage of read-only transactions in the range of 10%, 50%, 90% and the contention level in the system by decreasing the total number of shared objects (i.e., accounts in Bank benchmark) available. This way we defined three contention level: *low*, with 5000 objects, *medium*, with 2000 objects, and *high*, with 500 objects. During the experiments we collected transactional throughput (Figure 5.3) and latency (Figure 5.4). In addition, for what concerns PaxosSTM, we gathered also the percentage of remote aborts. This information is available only for PaxosSTM because HiperTM does not certify transactions globally thus it cannot end up in aborting transactions. Only if the optimistic order does not match the final order and the transaction's read-set is not valid, then a transaction can be aborted in HiperTM. However, each transaction is aborted only once (at most) because it immediately restarts and commits without any possible further invalidation.

Figure 5.3 shows the throughput of Bank benchmark. For each workload configuration (i.e., low,medium,high conflict) we reported the observed abort percentage of PaxosSTM. The trend is clear from the analysis of the plots, PaxosSTM has a great performance compared with HiperTM because it is able to exploit the massive multi threading (i.e., 160 threads) for the transaction processing when the system is characterized by few conflicts. When contention becomes greater, namely when number of nodes increases or the amount of shared objects decreases, the certification phase of PaxosSTM hampers its scalability. On the contrary, HiperTM suffers from the single thread processing when the system has low contention, but outperforms PaxosSTM when the contention starts to increase. As a result, HiperTM scales better than PaxosSTM when the number of nodes increases.



(a) Throughput high conflict

(b) Abort high conflict

(c) Throughput medium conflict

(d) Abort medium conflict

(e) Throughput low conflict

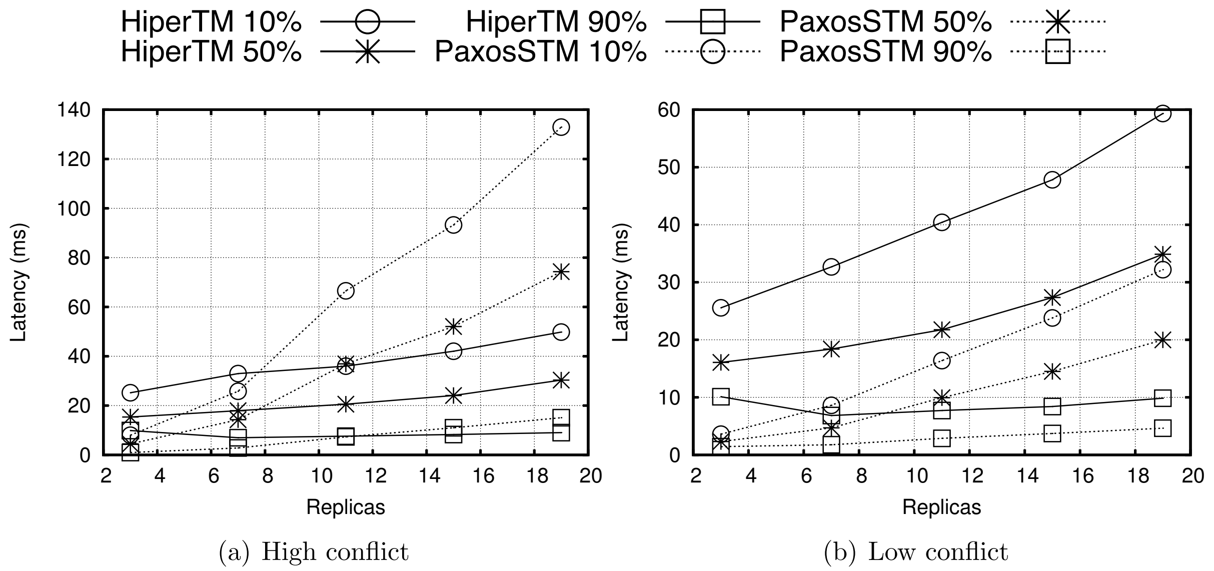
(f) Abort low conflict

Figure 5.3: Throughput and abort percentage of HiperTM and PaxosSTM for Bank benchmark.

In all plots, even where the absolute performance is better than HiperTM, the PaxosSTM's trend highlights its lack of scalability. This is mainly because, when a huge number of threads flood the system with transactional requests (where each request is the transaction's read-set and write-set), the certification phase is not able to commit transactions as fast as clients would inject requests. In addition, with higher contention, remote aborts play an important role as scalability bottleneck. As an example, with 11 nodes and high conflict scenario, PaxosSTM aborts 80% of transactions when configured with 50% of read-only workload.

Increasing the percentage of read-only workload increases performance of both competitors due to local multi-versioning concurrency control. However, HiperTM always scales when the size of the system increases. This is because HiperTM does not saturate the total order layer since messages are very small (i.e., the id of the transaction to invoke and parameters) and it does not require any certification phase. After an initial ordering phase, transactions are always committed suffering from at most one abort which, anyway, is not propagated through the network but it is handled locally by the concurrency control.

It is worth to notice the trend of PaxosSTM for low node count in the plot in Figure 5.3(a). Here, even though the number of shared objects is low, with such few replicas, the overall contention is not high, thus PaxosSTM behaves as in medium contention scenario (see Figure 5.3(c) when the percentage of abort is around 20% and with 90% of read-only transactions). However, after 9 nodes, HiperTM starts outperforming PaxosSTM and keeps scaling, reaching its peak performance improvement, that is 2.35× at 19 nodes.



(a) High conflict (b) Low conflict

Figure 5.4: Latency of HiperTM and PaxosSTM for Bank benchmark.

Figure 5.4 shows the latency measured in the same experiments reported in Figure 5.3. As expected, it follows the inverse trend of the throughput and for this reason we decided not to show the case of medium contention but the two extreme cases with high and low contention.

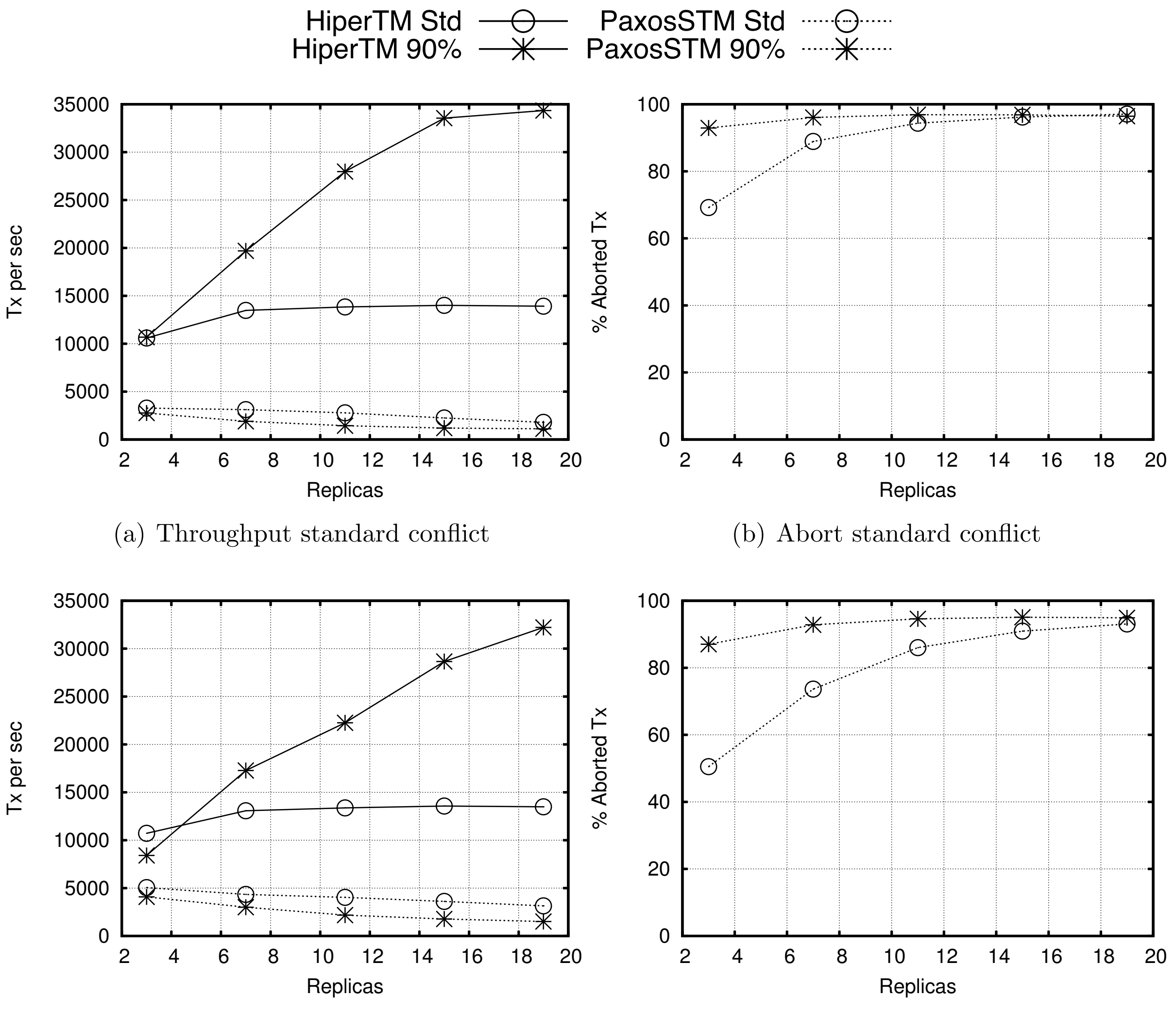
Both PaxosSTM and HiperTM rely on batching as a way to improve performance of the total order layer. Waiting for the creation of a batch consumes the most part of the reported latency. In addition for PaxosSTM, when a transaction aborts, client has to reprocess the transaction and issue a new certification phase through the total order layer. For this reason, PaxosSTM's latency starts increasing for high conflict scenarios.

5.5.2 TPC-C Benchmark

TPC-C [20] is a real application benchmark composed of five transaction profiles, each either read-only (i.e., `Order Status`, `Stock Level`) or read-write (i.e., `Delivery`, `Payment`, `New Order`) . Transactions are longer than Bank benchmark, with high computation and several objects accessed. The specification of the benchmark suggests a mix of those transaction profiles (i.e., 4% `Order Status`, 4% `Stock Level`, 4% `Delivery`, 43% `Payment`, 45% `New Order`), resulting in a write intensive scenario. In order to wide the space of tested configurations, we measured the performance with a read intensive workload (i.e., 90% read-only) by changing the above mix (i.e., 45% `Order Status`, 45% `Stock Level`, 3.3% `Delivery`, 3.3% `Payment`, 3.3% `New Order`).

In terms of application contention, TPC-C defines a hierarchy of dependencies among defined objects, however the base object that controls the overall contention is the *warehouse*. Increasing the number of shared warehouses results in lower contention. The suggested configuration of TPC-C is to use as warehouses as the total number of nodes in the system. Therefore for the purpose of this test, we ran the benchmark with 19 warehouses and also with 50 warehouses in order to generate a low conflict scenario. We collected the same information as in Bank benchmark.

Figure 5.5 reports the throughput of HiperTM and PaxosSTM, together with the abort rate observed for PaxosSTM. PaxosSTM's abort rate results confirm that the contention in the system is much higher than in Bank benchmark. In addition, the certification phase of PaxosSTM now represents the protocol's bottleneck because read-set and write-set of transactions are large, thus each batch of network messages does not record many transactions and this limits the throughput of the certification phase. Both these factors hamper PaxosSTM's scalability and high performance. On the other hand, HiperTM orders transactions before their execution and it leverages OS-Paxos just for broadcasting transactional requests, thus it is independent from the application and from the contention in the system. This allows HiperTM to scale while increasing nodes and resulting in performance by as much as 3.5× better in case of standard configuration of TPC-C, and by more than one order of magnitude for the 10% read-only scenario. HiperTM's performance in Figures 5.5(a) and 5.5(c) are almost the same, this confirms how HiperTM, and the active replication paradigm, is independent from application's contention. Unfortunately, with long transactions as in TPC-C, HiperTM cannot match the performance of Bank benchmark because of the single thread processing.



(a) Throughput standard conflict

(b) Abort standard conflict

(c) Throughput low conflict

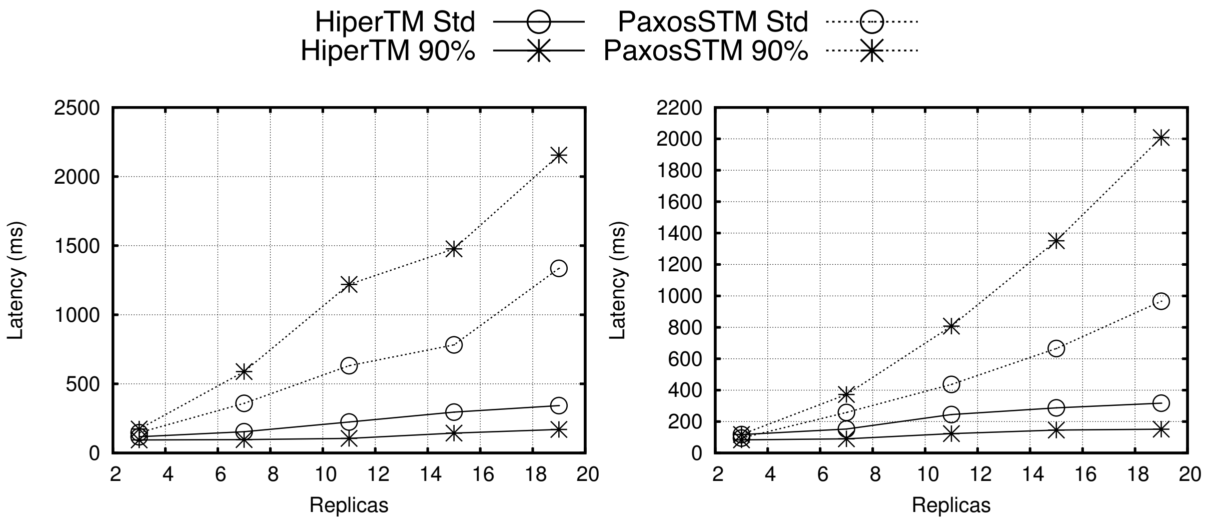
(d) Abort low conflict

Figure 5.5: Throughput and abort percentage of HiperTM and PaxosSTM for TPC-C benchmark.

The Figure 5.6 shows the latency measured in the above experiments. Clearly, lower throughput and longer transactions caused higher latency.

5.6 Summary

At its core, our work shows that optimism pays off: speculative transaction execution, started as soon as transactions are optimistically delivered, allows hiding the total ordering latency, and yields performance gain. Single-communication step is mandatory for fine-grain transactions. Complex concurrency control algorithms are sometimes not feasible when the available



(a) Standard conflict (b) Low conflict

Figure 5.6: Latency of HiperTM and PaxosSTM for TPC-C benchmark.

processing time is limited.

Implementation matters. Avoiding atomic operations, batching messages, and optimizations to counter network non-determinism are important for high performance.

Chapter 6

Archie

From the experience with HiperTM, we identified few possible improvements in our system. Firstly, assuming that *optimistic-order* matches *final-order*, any transactional processing happening after arrival of *final-order* could be avoided, which could result in a lower latency and better performance. Second one was overcoming the limitation of serial execution of transactions by a high-performance concurrent one. Last improvement was to enhance the *optimistic-delivery* mechanism to keep pace with the improved transaction processing.

We present Archie, a State Machine Approach (SMA) based transactional scheme that incorporates these protocol and system innovations that extensively use *speculation* for removing any non-trivial task after the delivery of the transaction's order. The main goal of Archie is to avoid the time-consuming operations (e.g., the entire transaction's execution or iterations over transaction's read and written objects) performed after this notification, such that a transaction can be immediately committed.

In order to accomplish the above goal, we designed *MiMoX*, an optimized sequencer-based total order layer which inherits the advantages of two well-known mechanisms: the *optimistic notification* [52, 71], issued to nodes prior to the establishment of the total order; and *batching* [98, 30], as a means for improving the throughput of the global ordering process. MiMoX proposes an architecture that mixes these two mechanisms, thus allowing the anticipation (thanks to the optimistic notification) of a big amount of work (thanks to batching) before the total order is finalized. Nodes take advantage of the time needed for assembling a batch to compute a significant amount of work before the delivery of the order is issued. This anticipation is mandatory in order to minimize (and possibly eliminate the need for) the transaction's serial phase. As originally proposed in [71], MiMoX guarantees that if the sequencer node (*leader*) is not replaced during the ordering process (i.e., either suspected or crashed), the sequence of optimistic notifications matches the sequence of final notifications. As a distinguishing point, the solution in [71] relies on a ring topology as a means for delivering transactions optimistically, whereas MiMoX does not assume any specific topology.

At the core of Archie there is a novel speculative parallel concurrency control, named *ParSpec*, that processes/certifies transactions upon their optimistic notification and enforces the same order as the sequence of optimistic notifications. The key enabling point for guaranteeing the effectiveness of ParSpec is that the majority of transactions speculatively commit before the total order is delivered. This goal is reached by minimizing the overhead caused by the enforcement of a predefined order on the speculative executions. ParSpec achieves this goal by the following steps:

- Executing speculative transactions in parallel, but allowing them to speculative commit only in-order, thus reducing the cost of detecting possible out-of-order executions;
- Dividing the speculative transaction execution into two stages: the first, where the transaction is entirely speculatively executed and its modifications are made visible to the following speculative transactions; the second, where a ready-to-commit snapshot of the transaction's modifications is pre-installed into the shared data-set, but not yet made available to non-speculative transactions.

A transaction starts its speculative commit phase only when its previous transaction, according to the optimistic order, becomes speculatively-committed and its modifications are visible to other successive speculative transactions. The purpose of the second stage concerns only the non-speculative commit, thus it can be removed from the speculative transaction's critical path and executed in parallel. This approach increases the probability of speculatively committing a transaction before the total order is notified. The final commit of an already speculatively-committed transaction consists of making the pre-installed snapshot available to all. In case the MiMoX's leader is stable during the execution, ParSpec realizes this task without iterating over all transaction's written objects but, rather, it just increases one local timestamp. Clients are informed about their transactions' outcome while other speculative transactions execute. As a result, transaction latency is minimized and ParSpec's high throughput allows more clients to submit requests.

The principles at the base of Archie can be applied in both DUR- and DER-based systems. For the purpose of this work, we optimized Archie to cope with the DER model. This is because DER has three main benefits over DUR. First, it makes application behavior independent of failures. When a node in the system crashes or stops serving incoming requests, other nodes are able to transparently service the same request, process the transaction, and respond back to the application. Second, it does not suffer from aborts due to contention on shared remote objects because a common serialization order is defined prior to starting transaction (local) execution, thus yielding high performance and better scalability in medium/high contention scenarios [56]. Third, with DER, the size of network messages exchanged for establishing the common order does not depend on the transaction's logic (i.e., the number of objects accessed). Rather, it is limited to the name of the transaction and, possibly, its input parameters, which reduces network usage and increases the ordering protocol's performance.

As commonly adopted in several SMA-based transactional systems [56, 48, 80] and thanks to the full replication model, Archie does not broadcast read-only workloads through MiMoX; read-only requests are handled locally, in parallel with the speculative execution. Processing write transactions (both conflicting and not conflicting) in the same order on all nodes allows Archie to guarantee 1-copy-serializability [7].

We implemented Archie in Java and we conducted a comprehensive experimental study using benchmarks including TPC-C [20], Bank and a distributed version of Vacation [13]. As competitors, we selected one DUR-based: PaxosSTM [117] – a high-performance open source transactional system; and two DER-based: one non-speculative (SM-DER [101]) and one speculative (HiperTM [48]) transactional system.

Our experiments on *PRObE* [32], a state-of-the-art public cluster, reveal Archie's high-performance and scalability. On up to 19 nodes, Archie outperforms all competitors in most of the tested scenarios. As expected, when the contention is very low, PaxosSTM behaves better than Archie.

The work makes the following contributions:

- Archie is the first fully-implemented DER-based transactional system that eliminates costly operations during the serial phase by anticipating the work through speculative parallel execution.
- MiMoX is the first total order layer that guarantees a reliable optimistic delivery order (i.e., the optimistic order matches the total order) without any assumption on the network topology, and maximizes the overlapping time (i.e., the time between the optimistic and relative total order notifications) when the sequencer node is not replaced (e.g,. due to a crash).
- ParSpec is the first parallel speculative concurrency control that removes from the transaction's critical path the task to install written objects and implements a lightweight commit procedure to make them visible.

6.1 MiMoX

MiMoX is a network system that ensures total order of messages across remote nodes. It relies on Multi-Paxos [60], an algorithm of the Paxos family, which guarantees agreement on a sequence of values in the presence of faults (i.e., total order). MiMoX is sequencer-based – i.e., one elected node in the system, called the *leader*, is responsible for defining the order of the messages.

MiMoX provides the APIs of Optimistic Atomic Broadcast [52]: `broadcast(m)`, which is used by clients to broadcast a message m to all nodes; `final-delivery(m)`, which is used for notifying each replica on the delivery of a message m (or a batch of them); and `opt-delivery(m)`, which is used for early-delivering a previously broadcast message m (or a batch of them) before the `final-delivery(m)` is issued.

Each MiMoX message that is delivered is a container of either a single transaction request or a batch of transaction requests (when batching is used). The sequence of `final-delivery(m)` events, called *final order*, defines the transaction serialization order, which is the same for all the nodes in the system. The sequence of `opt-delivery(m)` events, called *optimistic order*, defines the optimistic transaction serialization order. Since only the final order is the result of a distributed agreement, the optimistic order may differ from the final order and may also differ among nodes (i.e., each node may have its own optimistic order). As we will show later, MiMox guarantees the match between the optimistic and final order when the leader is not replaced (i.e., stable) during the ordering phase.

6.1.1 Ordering Process

MiMoX defines two types of batches: *opt-batch*, which groups messages from the clients, and *final-batch*, which stores the identification of multiple opt-batches. Each final-batch is identified by an unique `instance_ID`. Each opt-batch is identified by a pair <`instance_ID`, `#Seq`>.

When a client broadcasts a request using MiMoX, this request is delivered to the leader which aggregates it into a batch (the opt-batch). In order to preserve the order of these steps, and for avoiding synchronization points that may degrade performance, we rely on single-thread processing for the following tasks. For each opt-batch, MiMoX creates the pair <`instance_ID`, `#Seq`>, where `instance_ID` is the identifier of the current final-batch that will wrap the opt-batch, and `#Seq` is the position of the opt-batch in the final-batch. When the pair is defined, it is appended to the final-batch. At this stage, instead of waiting for the completion of the final-batch and before creating the next opt-batch, MiMoX sends the current opt-batch to all the nodes, waiting for the relative acknowledgments. Using this mechanism, the leader informs nodes about the existence of a new batch while the final-batch is still accumulating requests. This way, MiMoX maximizes the overlap between the time needed for creating the final-batch with the local processing of opt-batches; and enables nodes to effectively process messages, thanks to the reliable optimistic order.

Each node, upon receiving the opt-batch, immediately triggers the optimistic delivery for it. As in [71], we believe that within a data-center the scenarios where the leader crashes or becomes suspected are rare. If the leader is stable for at least the duration of the final-batch's agreement, then even if the opt-batch is received out-of-order with respect to other opt-batches sent by the leader, this possible reordering is still nullified by the ordering information (i.e., #Seq) stored within each opt-batch.

After sending the opt-batch, MiMoX loops again serving the next opt-batch, until the completion of the final-batch. When ready, MiMoX uses the Multi-Paxos algorithm for establishing an agreement among nodes on the final-batch. The leader *proposes* an order for the final-batches, to which the other replicas reply with their agreement – i.e., *accept* messages. When a majority of agreement for a proposed order is reached, each replica considers it as *decided*.

The message size of the final-batch is very limited because it contains only the identifiers of opt-batches that have already been delivered to nodes. This makes the agreement process fast and includes a high number of client messages.

6.1.2 Handling Faults and Re-transmissions

MiMoX ensures that, on each node, an *accept* is triggered for a *proposed* message (or batch) m only if all the opt-batches belonging to m have been received. Enforcing this property prevents loss of messages belonging to already *decided* messages (or batches).

As an example, consider three nodes $\{N_1,N_2,N_3\}$, where N_1 is the leader. The final-batch (FB) is composed of three opt-batches: OB_1, OB_2, OB_3. N_1 sends OB_1 to N_2 and N_3. Then it does the same for OB_2 and OB_3. But N_2 and N_3 do not receive both messages. After sending OB_3, the FB is complete, and N_1 sends the *propose* message for FB. Nodes N_2 and N_3 send the *accept* message to the other nodes, recognizing that there are unknown opt-batches (i.e., OB_2 and OB_3). The only node having all the batches is N_1. Therefore, N_2 and N_3 request N_1 for the re-transmission of the missing batches. In the meanwhile, each node receives the majority of *accept* messages from other nodes and triggers the *decide* for FB. At this stage, if N_1 crashes, even though FB has been agreed, OB_2 and OB_3 are lost, and both N_2 and N_3 cannot retrieve their content anymore.

We solve this problem using a dedicated service at each node, which is responsible for re-transmitting lost messages (or batches). Each node, before sending the *accept* for an FB, must receive all the opt-batches. The FB is composed of the identification of all the expected opt-batches. Thus, each node is easily able to recognize the missing batches. Assuming that the majority of nodes are non-faulty, the re-transmission request for one or multiple opt-batches is broadcast to all the nodes such that, eventually the entire sequence of opt-batches belonging to FB is rebuilt and the *accept* message is sent.

Nodes can detect a missing batch before the *propose* message for the FB is issued. Exploiting the sequence number and the FB's ID used for identifying opt-batches, each node can easily find a gap in the sequence of the opt-batches received, that belong to the same FB (e.g., if OB_1 and OB_3 are received, then, clearly, OB_2 is missing). Thus, the re-transmission can be executed in parallel with the ordering, without additional delay. The worst case happens when the missing opt-batch is the last in the sequence. In this case, the *propose* message of FB is needed to detect the gap.

6.1.3 Evaluation

We evaluated MiMoX's performance by an experimental study. We focused on MiMoX's scalability in terms of the system size, the average time between optimistic and final delivery, the number of requests in opt-batch and final-batch, and the size of client requests. We used the *PRObE* testbed [32], a public cluster that is available for evaluating systems research. Our experiments were conducted using 19 nodes (tolerating up to 9 failures) in the cluster. Each node is equipped with a quad socket, where each socket hosts an AMD Opteron 6272, 64-bit, 16-core, 2.1 GHz CPU (total 64-cores). The memory available is 128GB, and the network connection is a high performance 40 Gigabit Ethernet.

For the purpose of the study, we decided to finalize an opt-batch when it reaches the maximum size of 12K bytes and a final-batch when it reaches 5 opt-batches, or when the time needed for building them exceeds 10 msec, whichever occurs first. All data points reported are the average of six repeated measurements.

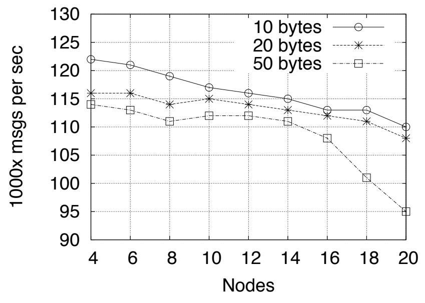


Figure 6.1: MiMoX's message throughput.

Figure 6.1 shows MiMoX's throughput in requests ordered per second. For this experiment, we varied the number of nodes participating in the agreement and the size of each request. Clearly, the maximum throughput (122K requests ordered per second) is reached when the node count is low (3 nodes). However, the percentage of degradation in performance is limited when the system size is increased: with 19 nodes and request size of 10 bytes, the performance decreases by only 11%.

Figure 6.1 shows also the results for request sizes of 20 and 50 bytes. Recall that Archie's transaction execution process leverages the ordering layer only for broadcasting the transaction ID (e.g., method or store-procedure name), along with its parameters (if any), and not the entire transaction business logic. Other solutions, such as the DUR scheme, use the total order layer for broadcasting the transaction read- and write-set after a transaction's completion, resulting in larger request size than Archie's. In fact, our evaluations with Bank and TPC-C benchmarks revealed that almost all the transaction requests can be compacted between 8 and 14 bytes. MiMoX's performance for a request size of 20 bytes is quite close to that for 10 byte request size. We observe a slightly larger gap with 19 nodes and 50 byte request size, where the throughput obtained is 104K. This is a performance degradation lesser than 15% with respect to the maximum throughput. This is because, with smaller requests (10 or 20 bytes), opt-batches do not get filled to the maximum size allowed, resulting in smaller network messages. On the other hand, larger requests (50 bytes) tend to fill batches sooner, but these bigger network messages take more time to traverse.

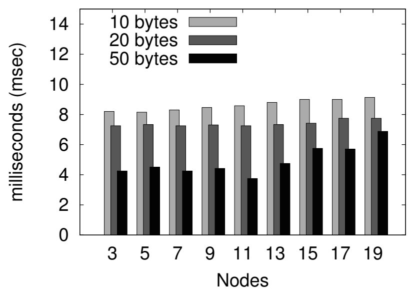


Figure 6.2: Time between optimistic/final delivery.

Figure 6.2 shows MiMoX's delay between the optimistic and the relative final delivery, named overlapping time. This experiment is the same as that reported in Figure 6.1. MiMoX achieves a stable overlapping time, especially for a request size of 10 bytes, of ≈8 *msec*. This delay is non-negligible if we consider that Archie processes transactions locally. Using bigger requests, the final-batch becomes ready sooner because less requests fit in one final-batch. As a result, the time between optimistic and final delivery decreases. This is particularly evident with a request size of 50 bytes, where we observe an overlapping time that is, on average, 4.6 *msec*.

The last results motivate our design choice to adopt DER as a replication scheme instead of DUR.

Request size (bytes)	Final-batch size	Opt-batch size	% Re NF	% Re F
10	4.91	230.59	0%	1.7%
20	4.98	166.45	0%	3.4%
50	5.12	90.11	0%	4.8%

Table 6.1: Size of requests, batches, and % reorders.

Table 6.1 shows other information collected from the previous experiments. It is interesting to observe the number of opt-batches that makes up a final-batch (5 on average) and the number of client requests in each opt-batch (varies from 90 to 230 depending on the request size). This last information confirms the reason for the slight performance drop using requests of 50 bytes. In fact, in this case each opt-batch transfers ≈ 4500 bytes in payload, as compared to ≈ 2300 bytes for request size of 10 bytes.

In these experiments, we used TCP connections for sending opt-batches. Since MiMoX uses a single thread for sending opt-batches and for managing the final-batch, reorders between optimistic and final deliveries cannot happen except when the leader crashes or is suspected. Table 6.1 supports this. It reports the maximum reordering percentages observed when leader is stable (column *Re NF*) and when the leader is intentionally terminated after a period of stable execution (column *Re F*), using 19 nodes.

6.2 PARSPEC

ParSpec is the concurrency control protocol that runs locally at each node. MiMoX delivers each message or a batch of messages twice: once optimistically and once finally. These two events are the triggers for activating ParSpec's activities. Without loss of generality, hereafter, we will refer to a message of MiMoX as a batch of messages.

Transactions are classified as *speculative*: i.e., those that are only optimistically delivered, but their final order has not been defined yet; and *non-speculative*: i.e., those whose final order has been established. Among speculative transactions, we can distinguish between *speculatively-committed* (or *x-committed* hereafter): i.e., those that have completely executed all their operations and cannot be aborted anymore by other speculative transactions; and *active*: i.e., those that are still executing operations or that are not allowed to speculatively commit yet. Moreover, each transaction T records its optimistic order in a field called $T.OO$. T's optimistic order is the position of T within its opt-batch, along with the position of the opt-batch in the (possible) final-batch.

ParSpec's main goal is to activate in parallel a set of speculative transactions, as soon as they are optimistically delivered, and to entirely complete their execution before their final order is notified.

As a support for the speculative execution, the following meta-data are used: `abort-array`, which is a bit-array that signals when a transaction must abort; `LastX-committedTx`, which stores the ID of the last x-committed transaction; and `SCTS`, the speculative commit timestamp, which is a monotonically increasing integer that is incremented each time a transaction x-commits. Also, each node is equipped with an additional timestamp, called `CTS`, which is an integer incremented each time a non-speculative transaction commits.

For each shared object, a set of additional information is also maintained for supporting ParSpec's operations: (1) the list of committed and x-committed versions; (2) the version written by the last x-committed transaction, called `spec-version`; (3) the boolean flag called `wait-flag`, which indicates that a speculative active transaction wrote a new version of the object, and `wait-flag`.OO, the optimistic order of that transaction; and (4) a bit-array called `readers-array`, which tracks active transactions that already read the object during their execution. Committed (or x-committed) versions contain `VCTS`, which is the `CTS` (or the `SCTS`) of the transaction that committed (or x-committed) that version.

The size of the `abort-array` and `readers-array` is bounded by `MaxSpec`, which is an integer defining the maximum number of speculative transactions that can run concurrently. `MaxSpec` is fixed and set a priori at system start-up. It can be tuned according to the underlying hardware.

When an opt-batch is optimistically delivered, ParSpec extracts the transactions from the opt-batch and processes them, activating `MaxSpec` transactions at a time. Once all these speculative transactions finish their execution, the next set of `MaxSpec` transactions is activated. As it will be clear later, this approach allows a quick identification of those transactions whose history is not compliant anymore with the optimistic order, thus they must be aborted and restarted.

In the `abort-array` and `readers-array`, each transaction has its information stored in a specific location such that, if two transactions T_a and T_b are optimistically ordered, say in the order $T_a > T_b$, then they will be stored in these arrays respecting the invariant $T_a > T_b$.

Since the optimistic order is a monotonically increasing integer, for a transaction T, the position $i = T.OO$ `mod MaxSpec` stores T's information. When `abort-array`$[i]$=1, T must abort because its execution order is not compliant anymore with the optimistic order. Similarly, when an object *obj* has `readers-array`$[i]$=1, it means that the transaction T performed a read operation on *obj* during its execution.

Speculative active transactions make available new versions of written objects only when they x-commit. This way, other speculative transactions cannot access intermediate snapshots of active transactions. However, when `MaxSpec` transactions are activated in parallel, multiple concurrent writes on the same object could happen. When those transactions reach their x-commit phase, different speculative versions of the same object could be available for readers. As an example, consider four transactions $\{T_1,T_2,T_3,T_4\}$ that are optimistically delivered in this order. T_1 and T_3 write to the same object O_a, and T_2 and T_4 read from O_a. When T_1 and T_3 reach the speculative commit phase, they make two speculative versions of O_a available: $O_a^{T_1}$ and $O_a^{T_3}$. According to the optimistic order, T_2's read should return $O_a^{T_1}$ and T_4's read should return $O_a^{T_3}$. Even though this approach maximizes concurrency, its implementation requires traversing the shared lists of transactional meta-data, resulting in high transaction execution time and low performance [5].

ParSpec finds an effective trade-off between performance and overhead for managing meta-data. In order to avoid maintaining a list of speculative versions, ParSpec allows an active transaction to x-commit only when the speculative transaction optimistically ordered just before it is already x-committed. Formally, given two speculative transactions T_x and T_y such that $T_y.OO = \{T_x.OO\} + 1$, T_y is allowed to x-commit only when T_x is x-committed. Otherwise, T_y keeps spinning even when it has executed all of its operations. T_y easily recognizes T_x's status change by reading the shared field `LastX-committedTx`. We refer to this property as *rule-comp*. By *rule-comp*, read and write operations become efficient. In fact, when a transaction T reads an object, only one speculative version of the object is available. Therefore, T's execution time is not significantly affected by the overhead of selecting the appropriate version according to T's history. In addition, due to *rule-comp*, even though two transactions may write to the same object, they can x-commit and make available their new versions only in-order, one after another. This policy prevents any x-committed transaction to abort due to other speculative transactions.

In the following, ParSpec's operations are detailed.

6.2.1 Transactional Read Operation

When a write transaction T_i performs a read operation on an object X, it checks whether another active transaction T_j is writing a new version of X and T_j's optimistic order is prior to T_i's. In this case, it is useless for T_i to access the `spec-version` of X because, eventually, T_j will x-commit, and T_i will be aborted and restarted in order to access T_j's version of X. Aborting T_i ensures that its serialization order is compliant with the optimistic order. T_i is made aware about the existence of another transaction that is currently writing X through X.wait-flag, and about its order through X.wait-flag.OO. If X.wait-flag=1 and X.wait-flag.OO $<$ T_i.OO, then T_i waits until the previous condition is no longer satisfied. For the other cases, namely when X.wait-flag=0 or X.wait-flag.OO $>$ T_i.OO, T_i proceeds with the read operation without waiting, accessing the `spec-version`. Specifically, if X.wait-flag.OO $>$ T_i.OO, then it means that another active transaction T_k is writing to X. But, according to the optimistic order, T_k is serialized after T_i. Thus, T_i can simply ignore T_k's concurrent write.

After successfully retrieving X's value, T_i stores it in its read-set, signals that a read operation on X has been completed, and sets the flag corresponding to its entry in X.readers-array. This notification is used by writing transactions to abort inconsistent read operations that are performed before a previous write takes place.

6.2.2 Transactional Write Operation

The *rule-comp* prevents two or more speculative transactions from x-committing in parallel and in any order. Rather, they progressively x-commit, according to the optimistic order. In ParSpec, transactional write operations are buffered locally in a transaction's write-set. Therefore, they are not available for concurrent reads before the writing transaction x-commits. The write procedure has the main goal of aborting those speculative active transactions that are serialized after (in the optimistic order) and *a)* wrote the same object, and/or *b)* previously read the same object (but clearly a different version).

When a transaction T_i performs a write operation on an object X and finds that X.wait-flag $= 1$, ParSpec checks the optimistic order of the speculative transaction T_j that wrote X. If X.wait-flag.OO $>$ T_i.OO, then it means that T_j is serialized after T_i. So, an abort for T_j is triggered because T_j is a concurrent writer on X and only one X.spec-version is allowed for X. On the contrary, if X.wait-flag.OO $<$ T_i.OO (i.e., T_j is serialized before T_i according to the optimistic order) then T_i, before proceeding, loops until T_j x-commits.

Since a new version of X written by T_i will eventually become available, all speculative active transactions optimistically delivered after T_i that read X must be aborted and restarted so that they can obtain X's new version. Identifying those speculative transactions that must be aborted is a lightweight operation in ParSpec. When a speculative transaction x-commits, its history is fixed and cannot change because all the speculative transactions serialized before it have already x-committed. Thus, only active transactions can be aborted. Object X keeps track of readers using the `readers-array` and ParSpec uses it for triggering an abort: all active transactions that appear in the `readers-array` after T_i's index and having an entry of 1 are aborted. Finally, before including the new object version in T_i's write-set, ParSpec sets X.wait-flag = 1 and X.wait-flag.OO = T_i.OO.

Finally, if a write operation is executed on an object already written by the transaction, its value is simply updated.

6.2.3 X-Commit

A speculative active transaction that finishes all of its operations enters the speculative commit (x-commit) phase. This phase has three purposes: the first *(A)* is to allow next speculative active transactions to access the new speculative versions of the written objects; the second *(B)* is to allow subsequent speculative transactions to x-commit; the third *(C)* is to prepare "future" committed versions (not yet visible) of the written objects such that, when the transaction is eventually final delivered, those versions will be already available and its commit will be straightforward. However, in order to accomplish *(B)*, only *(A)* must be completed while *(C)* can be executed later. This way, ParSpec anticipates the event that triggers the x-commit of the next speculative active transactions, while executing *(C)* in parallel with that.

Step (A). All the versions written by transaction T_i are moved from T_i's write-set to the `spec-version` field of the respective objects and the respective `wait-flags` are cleared. This way, the new speculative versions can be accessed from other speculative active transactions. At this time, a transaction T_j that accessed any object conflicting with T_i's write-set objects and is waiting on `wait-flags` can proceed.

In addition, due to *rule-comp*, an x-committed transaction cannot be aborted by any speculative active transaction. Therefore, all the meta-data assigned to T_i must be cleaned up for allowing the next `MaxSpec` speculative transactions to execute from a clean state.

Step (B). This step is straightforward because it only consists of increasing *SCTS*, the speculative commit timestamp, which is incremented each time a transaction x-commits, as well as increasing `LastX-committedTx`.

Step (C). This step is critical for avoiding the iteration on the transaction's write-set to install the new committed versions during the serial phase. However, this step does not need to be in the critical path of subsequent active transactions. For this reason *(C)* is executed in parallel to subsequent active transactions after updating `LastX-committedTx`, such that the chain of speculative transactions waiting for x-commit can evolve.

For each written object, a new committed, but not yet visible, version is added to the object's version list. The visibility of this version is implemented leveraging *SCTS*. Specifically, *SCTS* is assigned to the *VCTS* of the version. *SCTS* is always greater than *CTS* because the speculative execution always precedes the final commit. This way (as we will show in Section 6.2.6) no non-speculative transaction can access that version until *CTS* is equal to *SCTS*. If the MiMox's leader is stable in the system (i.e., the optimistic order is reliable), then when *CTS* reaches the value of *SCTS*, then the speculative transaction has already been executed, validated and all of its versions are already available to non-speculative transactions.

6.2.4 Commit

The commit event is invoked when a final-batch is delivered. At this stage, two scenarios can happen: *(A)* the final-batch contains the same set of opt-batches already received in the same order, or *(B)* the optimistic order is contradicted by the content of the final-batch.

Scenario (A) is the case when the MiMoX's leader is not replaced while the ordering process is running. This represents the normal case within a data-center, and the best case for Archie because the speculative execution can be actually leveraged for committing transactions without performing additional validation or re-execution. In fact, ParSpec's *rule-comp* guarantees that the speculative order always matches the optimistic order, thus if the latter is also confirmed by the total order, it means that the speculative execution does not need to be validated anymore.

In this scenario, the only duty of the commit phase is to increase *CTS*. Given that, when $CTS{=}Y$, it means that the x-committed transaction with $SCTS{=}Y$ has been finally committed. Non-speculative transactions that start after this increment of *CTS* will be able to observe the new versions written during the step *(C)* of the x-commit of the transaction with $SCTS{=}Y$.

Using this approach, ParSpec eliminates any complex operation during the commit phase and, if most of the transactions x-commit before their notification of the total order, then they are committed right away, paying only the delay of the total order. If the transaction does not contain massive non-transactional computation, then the iteration on the write-set for installing the new committed versions, and the iteration on the read-set for validating the transaction, have almost the same cost as running the transaction from scratch after the final delivery. This is because, once the total order is defined, transactions can execute without any overhead, such as logging in the read-set or write-set.

In scenarios like (B), transactions cannot be committed without being validated because the optimistic order is not reliable anymore. For this reason, the commit is executed using a single thread. Transaction validation consists of checking if all the versions of the read objects during the speculative execution correspond to the last committed versions of the respective objects. If the validation succeeds, then the commit phase is equivalent to the one in scenario (A). When the validation fails, the transaction is aborted and restarted for at most once. The re-execution happens on the same committing thread and accesses all the last committed versions of the read objects.

In both the above scenarios, clients must be informed about transaction's outcome. ParSpec accomplishes this task asynchronously and in parallel, rather than burdening the commit phase with expensive remote communications.

6.2.5 Abort

Only speculative active transactions and x-committed transactions whose total order has already been notified can be aborted. In the first case, ParSpec uses the abort mechanism for restarting speculative transactions with an execution history that is non-compliant with the optimistic order. Forcing a transaction T to abort means simply to set the T's index of the `abort-array`. However, the real work for annulling the transaction context and restarting from the beginning is executed by T itself by checking the `abort-array`. This check is made after executing any read or write operation and when T_i is waiting to enter the x-commit phase. The abort of a speculative active transaction consists of clearing all of its meta-data before restarting.

In the second case, the abort is needed because the speculative transaction x-committed with a serialization order different from the total order. In this case, before restarting the transaction as non-speculative, all the versions written by the x-committed transaction must be deleted from the objects' version lists. In fact, due to the snapshot-deterministic execution, the new set of written versions can differ from the x-committed set, thus some version could become incorrectly available after the increment of *CTS*.

6.2.6 Read-Only Transactions

When a read-only transaction is delivered to a node, it is immediately processed, accessing only the committed versions of the read objects. This way, read-only workloads do not interfere with the write workloads, thus limiting the synchronization points between them. A pool of threads is reserved for executing read-only transactions. Before a read-only transaction T_i performs its first read operation on an object, it retrieves the *CTS* of the local node and assigns this value to its own timestamp (T_i.TS). After that, the set of versions available to T_i is fixed and composed of all versions with $VCTS \leq T_i$.TS – i.e., T_i cannot access new versions committed by any T_j ordered after T_i. Some object could have, inside its version list, versions with a $VCTS > T_i$.TS. These versions are added from x-committed transactions, but not yet finally committed, thus their access is prohibited to any non-speculative transaction.

6.3 Consistency Guarantees

Archie ensures 1-Copy Serializability [7] as a global property, and it ensures also that any speculative transaction (active, x-committed and aborted) always observes a serializable history, as a local property.

1-Copy Serializability. Archie ensures 1-Copy Serializability. The main argument that supports this claim is that transactions are validated and committed serially. We can distinguish two cases according to the reliability of the optimistic delivery order with respect to the final delivery order: *i)* when the two orders match, and the final commit procedure does not accomplish any validation procedure; *ii)* when the two orders do not match, thus the validation and a possible re-execution are performed.

The case *ii)* is straightforward to prove because, even though transactions are activated and executed speculatively, they are validated before being committed. The validation, as well as the commit, process is sequential. This rule holds even for non-conflicting transactions. Combining serial validation with the total order of transactions guarantees that all nodes eventually validate and commit the same sequence of write transactions. The ordering layer ensures the same sequence of delivery even in the presence of failures, therefore, all nodes eventually reach the same state.

The case *i)* is more complicated because transactions are not validated after the notification of the final order; rather, they directly commit after increasing the commit timestamp. For this case we rely on MiMoX, which ensures that all final delivered transactions are always optimistically delivered before. Given that, we can consider the speculative commit as the final commit because, after that, the transaction is ensured to not abort anymore and eventually commit. The execution of a speculative transaction is necessarily serializable because all of its read operations are done according to a predefined order. In case a read operation accesses a version such that its execution becomes not compliant with the optimistic order anymore, the reader transaction is aborted and restart. In addition, transactions cannot speculatively commit in any order or concurrently. They are allowed to do so only serially, thus reproducing the same behavior as the commit phase in case *ii)*.

Read-only transactions are processed locally without a global order. They access only committed versions of objects, and their serialization point is defined when they start. At this stage, if we consider the history composed of all the committed transactions, when a read-only transaction starts, it defines a prefix of that history such that it cannot change over time. Versions committed by transactions serialized after this prefix are not visible by the read-only transaction. Consider a history of committed write transactions, $\mathcal{H}$=$\{T_1, \ldots, T_i, \ldots, T_n\}$. Without loss of generality, assume that T_1 committed with timestamp 1; T_i committed with timestamp i; and T_n committed with timestamp n. All nodes in the system eventually commit $\mathcal{H}$. Different commit orders for these transactions are not allowed due to the total order enforced by MiMoX. Suppose that two read-only transactions T_a and T_b, executing on node N_a and N_b, respectively, access the same shared objects. Let T_a perform its first read operation on X accessing the last version of X committed at timestamp k, and T_b at timestamp j. Let P_a and P_b be the prefixes of $\mathcal{H}$ defined by T_a and T_b, respectively. $P_a(\mathcal{H})$=$\{T_1, \ldots, T_k\}$ such that $k \leq i$ and $P_b(\mathcal{H})$= $\{T_1, \ldots, T_j\}$ such that $j \leq i$. P_a and P_b can be either coincident, or one is a prefix of the other because both are prefixes of $\mathcal{H}$: i.e., if $k < j$, then P_a is a prefix of P_b; if $k > j$, then P_b is a prefix of P_a; if $k = j$, then P_a and P_b coincide.

Let P_a be a prefix of P_b. Now, $\forall$ T_u, $T_v \in P_a$, T_a and T_b will observe T_u and T_v in the same order (and for the same reason, it is true also for the other cases). In other words, due to the total order of write transactions, there are no two read-only transactions, running on the same node or different nodes, that can observe the same two write transactions serialized differently.

Serializable history. In ParSpec, all speculative transactions (including those that will abort) always observe a history that is *serializable*. This is because new speculative versions are exposed only at the end of the transaction, when it cannot abort anymore; and because each speculative transaction checks its abort bit after any operation. Assume three transactions T_1, T_2 and T_3, optimistically ordered in this way. T_1 x-commits a new version of object A, called A_1 and T_2 overwrites A producing A_2. It also writes object B, creating B_2. Both T_2 and T_3 run in parallel while T_1 already x-committed. Now T_3 reads A from T_1 (i.e., A_1) and subsequently T_2 starts to x-commit. T_2 publishes the A_2's speculative version and flags T_3 to abort because its execution is not compliant with the optimistic order anymore. Then T_2 continues its x-commit phase exposing B_2's speculative version. In the meanwhile, T_3 starts a read operation on B before being flagged by T_2, and it finds B_2. Even though T_3 is marked as aborted, it already started the read operation on B before checking the abort-bit. For this reason, this check is done after the read operation. In the example, when T_3 finishes the read operation on B, but before returning B_2 to the executing thread, it checks the abort-bit and it aborts due to the previous read on A. As a result, the history of a speculative transaction is always (and at any point in time) compliant with the optimistic order, thus preventing the violation of serializability.

6.4 Implementation and Evaluation

We implemented Archie in Java: MiMoX's implementation inherited JPaxos's [98, 57] software architecture, while ParSpec has been built from scratch. As a testbed, we used PRObE [32] as presented in Section 6.1. ParSpec does not commit versions on any stable storage. The transaction processing is entirely executed in-memory while fault-tolerance is ensured through replication.

We selected three competitors to compare against Archie. Two are state-of-the-art, open-source, transactional systems based on state-machine replication. One, PaxosSTM [117] implements the DUR model, while the other, HiperTM [48], complies with the DER model. As the third competitor, we implemented the classical DER scheme (called SM-DER) [101], where transactions are ordered through JPaxos [57] and processed in a single thread after the total order is established.

PaxosSTM [117] processes transactions locally, and relies on JPaxos [57] as a total order layer for their global certification across all nodes. On the other hand, HiperTM [48], as Archie, exploits the optimistic delivery for anticipating the work before the notification of the final order, but it processes transactions on single thread. In addition, HiperTM's ordering layer is not optimized for maximizing the time between optimistic and final delivery.

Each competitor provides its best performance under different workloads, thus they represent a comprehensive selection to evaluate Archie. Summarizing, PaxosSTM ensures high-performance in workloads with very low contention, such that remote aborts do not kick-in. HiperTM, as well as SM-DER, are independent from the contention because they process transactions using a single thread but their performance is significantly affected by the length of transactions (any operation is on the transaction's critical path). This way, workloads composed of short transactions represent their sweet spot. In addition, SM-DER excels for workloads where contention is very high. Here the intuition is that, if only few objects are shared, then executing transactions serially without any overhead is the best solution.

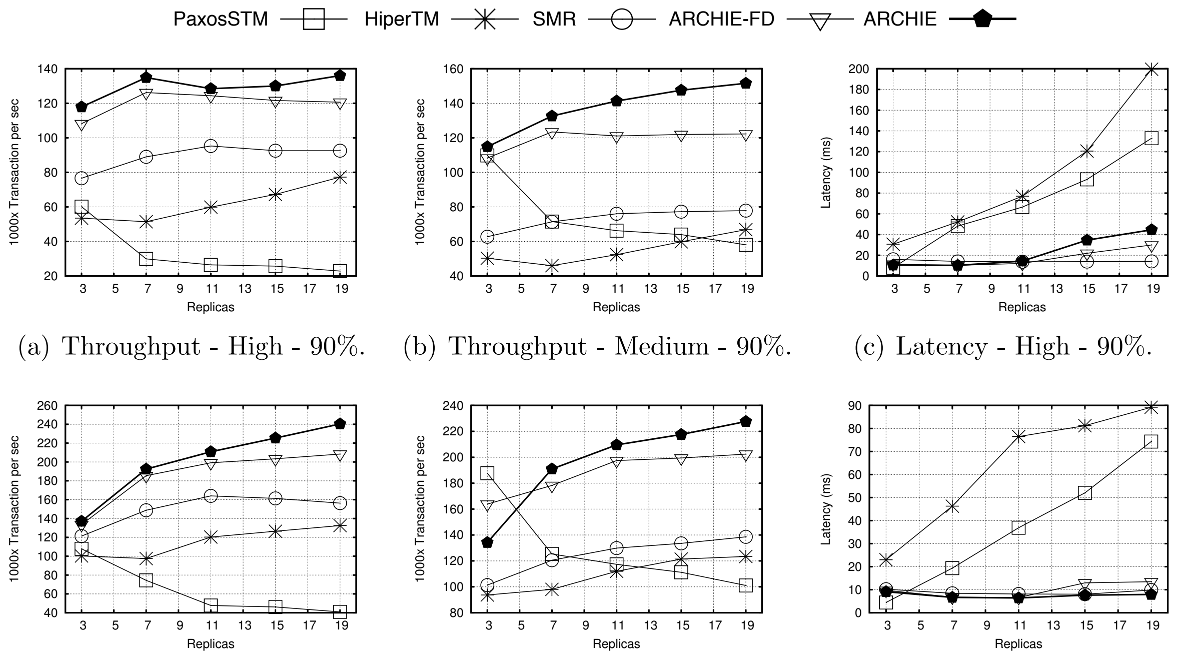
We provided two versions of Archie: one that exploits the optimistic delivery and one that postpones the parallel execution until the transactions are final delivered. This way, we can show the impact of the anticipation of the work, with respect to the parallel execution. The version of Archie that does not use the optimistic delivery, called Archie-FD, replaces the x-commit with the normal commit. In contrast with Archie, Archie-FD installs the written objects during the commit. For the purpose of the study, we configured `MaxSpec` and the size of the thread pool that serves read-only transactions as 12. This configuration resulted in an effective trade-off between performance and scalability on our testbed. However, these parameters can be tuned for exploring different trade-offs for the hardware and application workload at hand.

The benchmarks adopted in this evaluation include Bank, a common benchmark that emulates bank operations, TPC-C [20], a popular on-line transaction processing benchmark, and Vacation, a distributed version of the famous application included in the STAMP suite [13]. We scaled the size of the system in the range of 3-19 nodes and we also changed the system's contention level by varying the total number of shared objects available. All the competitors benefit from the execution of local read-only transactions. For this reason we scope out read-only intensive workloads. Each node has a number of clients running on it. When we increase the nodes in the system, we also slightly increase the number of clients accordingly.

This also means that the concurrency and (possibly) the contention in the system moderately increase. This is also why the throughput tends to increase for those competitors that scale along with the size of the system. In practice, we used on average the following total number of application threads balanced on all nodes: 1000 for TPC-C, 3000 for Bank, and 550 for Vacation.

6.4.1 Bank Benchmark

We configured the Bank benchmark for executing 10% and 50% of read-only transactions, and we identified the high, medium and low contention scenarios by setting 500, 2000, and 5000 total bank accounts, respectively. We report only the results for high and medium contention (Figure 6.3) because the trend in low contention scenario is very similar to the medium contention though with higher throughput.



(a) Throughput - High - 90%. (b) Throughput - Medium - 90%. (c) Latency - High - 90%.

(d) Throughput - High - 50%. (e) Throughput - Medium - 50%. (f) Latency - High - 50%.

Figure 6.3: Performance of Bank benchmark varying nodes, contention and percentage of write transactions.

Figure 6.3(a) plots the results of the high contention scenario. PaxosSTM suffers from a massive amount of remote aborts (≈85%), thus its performance is worse than others and it is not able to scale along with the size of the system. Interestingly, SM-DER behaves better than HiperTM because HiperTM's transaction execution time is higher than SM-DER's due to the overhead of operations' instrumentation. This is particularly evident in Bank, where transactions are short and SM-DER's execution without any overhead provides better performance. In fact, even if HiperTM anticipates the execution leveraging the optimistic delivery, its validation and commit after the total order nullify any previous gain. We observed also the time between the optimistic and final delivery in HiperTM to be less than 1 *msec*, which limits the effectiveness of its optimistic execution.

The two versions of Archie perform better than others but still Archie-FD, without the speculative execution, pays a penalty in performance around 14% against Archie. This is due to the effective exploitation of the optimistic delivery. Consistently with the results reported in Section 6.1, we observed an average time between optimistic and final delivery of 8.6 *msec*, almost 9× longer than HiperTM. However, as showed in Figure 6.3(c), Archie's average transaction latency is still much lower than others. The peak throughput improvement over the best competitor (i.e., SM-DER) is 54% for Archie and 41% for Archie-FD.

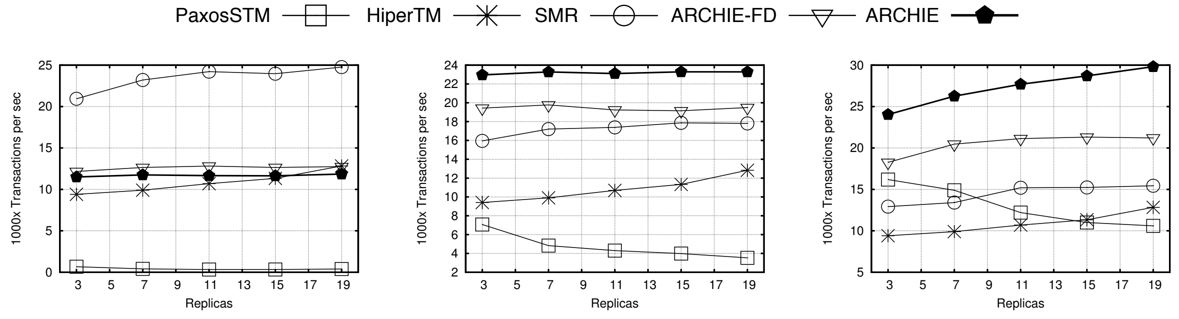
Figure 6.3(b) shows the results with an increased number of shared objects in the system. In these experiments the contention is lower than before, thus PaxosSTM performs better. With 3 nodes, its performance is comparable with Archie but, by increasing the nodes and thus the contention, it degrades. Here Archie's parallel execution has a significant benefit, reaching a speed-up by as much as 95% over SM-DER. Due to the lower contention, also the gap between Archie and Archie-FD increased up to 25%.

Figures 6.3(d), 6.3(e), 6.3(f) show the results with higher percentage of read-only transactions (50%). Recall that all protocols exploit the advantage of local processing of read-only transactions but absolute numbers are higher than before, as well as latency is reduced, but the trends are still similar.

6.4.2 TPC-C Benchmark

TPC-C is characterized by transactions accessing several objects and the workload has a contention level usually higher than other benchmarks (e.g., Bank). The mix of TPC-C profiles is the same as the default configuration, thus generating 92% of write transactions. We evaluated three scenarios, varying the total number of shared warehouses (the most contented object in TPC-C) in the range of {1,19,100}. With only one warehouse, all transactions conflict each other (Figure 6.4(a)) thus SM-DER behaves better than other competitors. In this case, the parallel execution of Archie is not exploited because transactions are always waiting for the previous speculative transaction to x-commit and then start almost the entire speculative execution from scratch. Increasing the number of nodes, HiperTM behaves better than Archie because of minimal synchronization required due to the single thread processing. However, when the contention decreases (Figure 6.4(b)), Archie becomes better than SM-DER by as much as 44%. Further improvements can be seen in Figure 6.4(c) where contention is much lower (96% of gain).

Archie is able to outperform SM-DER when 19 warehouses are deployed, because it bounds the maximum number of speculative transactions that can conflict each other (i.e., `MaxSpec`).



(a) #1 warehouse. (b) #19 warehouses. (c) #100 warehouses.

Figure 6.4: Performance of TPC-C benchmark varying nodes and number of warehouses.

We used 12 as `MaxSpec`, thus the number of possible transactions that can conflict with each other is less than the total number of shared objects, thus reducing the abort percentage from 98% (1 warehouse) to 36% (19 warehouses) (see also Figure 6.6). Performance of SM-DER worsens from Figure 6.4(a) to Figure 6.4(b). Although it seems counterintuitive, it is because, with more objects, the cost of looking up a single object is less than with 19 objects.

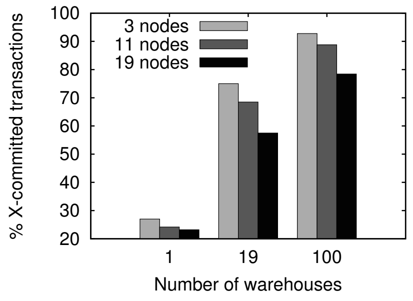


Figure 6.5: % of x-committed transactions before the notification of the total order.

Figure 6.5 shows an interesting parameter that helps to understand the source of Archie's gain: the percentage of speculative transactions x-committed before their total order is notified. It is clear from the plot that, due to the high contention with only one warehouse, Archie cannot exploit its parallelism thus almost all transactions x-commit after their final delivery is issued. The trend changes by increasing the number of warehouses. In the configuration with 100 warehouses, the percentage of x-committed transactions before their final delivery is in the range of 75%-95%. The performance related to this data-point is shown in Figure 6.4(c) where Archie is indeed the best, and the gap with respect to Archie-FD increased up to 41%.

Figure 6.6 reports the percentage of aborted transactions of the only two competitors that can abort: PaxosSTM and Archie. PaxosSTM invokes an abort when a transaction does not pass the certification phase, while Archie aborts a transaction during the speculative

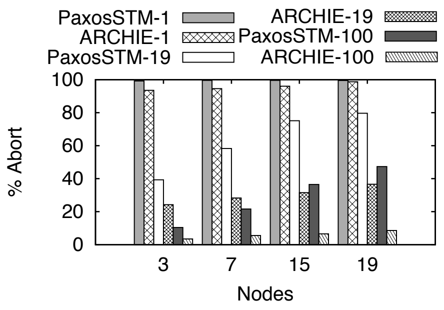


Figure 6.6: Abort % of PaxosSTM and Archie.

execution. Recall that, in PaxosSTM, the abort notification is delivered to the client, which has to re-execute the transaction and start again a new global certification phase. On the other hand, Archie's abort is locally managed and the re-execution of the speculative transaction does not involve any client operation, thus saving time and network load. In this plot, we vary the number of nodes in the system and, for each node, we show the observed abort percentage changing with the number of warehouses as before. The write intensive workload generates a massive amount of aborted transactions in PaxosSTM while in Archie, thanks to the speculative processing of `MaxSpec` transactions at a time, the contention does not increase significantly. The only case where Archie reaches 98% is with only one shared warehouse.

6.4.3 Vacation Benchmark

The Vacation Benchmark is an application originally proposed in the STAMP suite [13] for testing centralized synchronization schemes and often adopted in distributed settings (e.g., [117]). It reproduces the behavior of clients that submit booking requests for vacation related items.

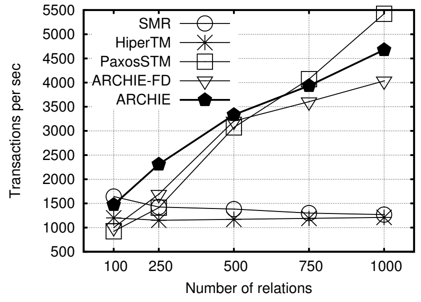


Figure 6.7: Throughput of Vacation benchmark.

Vacation generates longer transactions than the other considered benchmarks, thus also its total throughput is lower. Figure 6.7 shows the results. In this experiment we varied the total number of *relations* (object used for defining the contention in the system) and we fixed the number of nodes to 11. Vacation's clients do not perform any read-only transaction, however those transactions can still occur as a result of unsuccessful booking requests. However, the actual number of read-only transactions counted is less than 3%, thus their impact on performance is very limited.

With only 100 relations, SM-DER performs slightly better than the others, while increasing objects, and thus decreasing contention, Archie is the best until 750 relations. After that, the contention is so low that the certification-based approach of PaxosSTM prevails. From the results it is clear how competitors based on single thread processing (SM-DER and HiperTM) suffer in low contention scenarios because they cannot take advantage of any parallelism.

6.5 Summary

Most replicated transactional systems based on total order primitives suffer from the problem of single thread processing but they still represents one of the best solutions in scenarios in which majority of transactions access few shared objects. Archie's main goal is to alleviate the transaction's critical path by eliminating non-trivial operations performed after the notification of the final order. In fact, if the sequencer node is stable in the system, Archie's commit procedure consists of just a timestamp increment. Results confirmed that Archie outperformed competitors in write intensive workloads with medium/high contention.

Chapter 7

Caesar

The current technological trend in the area of network infrastructure (e.g., Amazon EC2 [2], Google Cloud [33], RedHat OpenShift [91]) proposes a change in the way computational resources are provisioned. Networks now are elastic, namely they can easily expand and shrink according to the application needs and the expected workload. For this reason building scalable services is considered a mandatory requirement for forthcoming solutions. This aim becomes more challenging when the application involves transactional requirements because: on one hand, logical contention among accessed objects likely hampers the system's scalability (i.e., adding resources to the system does not entail providing the same request's response time); and on the other hand, distributed transactional systems hardly give up stringent fault-tolerance requirements, by enforcing data replication in order to cope failures of nodes.

The widely used technique for building fault-tolerant transactional systems is providing a global order among transactions spawned on all nodes so that their execution changes the replicated shared state in the same way on all nodes (i.e., the so called State Machine Replication Approach [84]). Two major deployments of the State Machine Replication Approach have been proposed in literature: one executes transactions optimistically on the originating node and then a global "certification" is invoked to validate the optimistic execution against other concurrent transactions in the system (e.g., [115, 103, 117, 84]); the other one orders transactions before their execution and, once the global order is defined, transactions are processed accordingly (e.g., [47, 70, 78]).

In this work we assume the latter approach because its execution model prevents transactions from aborting (i.e., the order is defined before the execution) and the size of messages exchanged among nodes is very small (i.e., the transaction's footprint consists of only the transaction's name and its parameters), which is highly desirable for avoiding the saturation of network bandwidth for high node count.

Up to a few years ago, a reasonable network size for such a system was in the range of 8 to 20 nodes [70, 47, 78], however nowadays many dozens of nodes is the expected deployment. In this scenario – i.e., when the system's size increases and thus the load of the system is high – the above replication model has two well-known drawbacks which may limit its effectiveness: poor parallelism in the transaction execution and the existence of a single node (a.k.a. *leader*), which defines the order on behalf of all nodes. Regarding the former, processing transactions in accordance with a total order typically means processing them serially (see Chapter 2 for alternative solutions which assume partitioned accesses). In addition, ordering transactions in the same way on all nodes can be seen as an unnecessary overestimation of the problem of transactional replication, because the outcome of the commits of two non-conflicting transactions is independent from their commit order. Regarding the latter, establishing a total order relying on one leader represents the de-facto solution since it guarantees the delivery of a decision with the optimal number of communication steps [61].

The two mentioned problems are already addressed in literature. On one hand, more complex ordering techniques have been proposed, which allow the coexistence of multiple leaders at a time so that the work in the system is balanced among them, and the presence of a slow leader cannot hamper the overall performance (as in the case of single leader) [69, 78]. On the other hand, the problem of ordering transactions according to their actual conflicts has been originally formalized by the Generalized Consensus [59] and Generic Broadcast [86] problems. The core idea consists of avoiding a "blind" total order of all submitted transactions whereas only those transactions that depend upon each other are totally ordered. This way each node is allowed to deliver an order that differs from the one delivered by another node, while all of them have in common the way conflicting transactions are ordered. The effort of classifying conflicting transactions is then exploited during the processing phase because those non-conflicting transactions can be activated in parallel without worrying about any form of synchronization and therefore it enables the real parallelism (i.e., performance increases along with the number of threads).

Combining the above design choices into a unique protocol that inherits the benefits from all of them is the challenge we address in this work. Specifically, we show how existing solutions, which provide leaderless Generalized Paxos-based protocols, suffer from serious performance penalties when deployed in systems where general purpose transactions (i.e., transactions that perform both read and write operations) are assumed. Among the related proposals (overviewed in Chapter 2), EPaxos [78] stands up due to its generality, low latency and capability of reducing the number of nodes to contact during the ordering process[1].

EPaxos [78] is a consensus algorithm that orders commands[2] taking into account their content (as Generalized Paxos) but allows multiple leaders to exist together, each with the assignment of defining the order for a subset of sent commands. The major innovation of EPaxos is that it is able to reply to the client as soon as the message is stable in the system , but before its order is actually finalized. The actual delivery of the command to the processing layer is

[1]EPaxos decreases the quorum size by one as compared to Fast Paxos [64].

[2]A command is a general container of information and does not have any transactional semantic.

delayed until some local computation is performed (e.g., analysis of a dependency graph for computing the set of strongly connected components).

Thanks to this innovation, EPaxos is able to provide very high performance (e.g., throughput in the range of a few, up to five, hundred-thousand commands delivered per second, and low latency using 7 nodes) but it does not find its best case when deployed in a transactional processing system where clients need to understand the result of a computation (e.g., the result of a read transaction or the outcome of a transaction that invokes an explicit abort[3]) before issuing the next transaction. In this scenario, all those operations needed for the actual execution become part of the transactions critical path, significantly slowing down the performance (as we will show in our evaluation study, Section 7.4.1).

The weakness of EPaxos derives from the way it avoids the usage of a single leader. It is an egalitarian protocol where the "democracy" is the only form of government admitted to regulate decisions. Therefore, the transaction execution order entirely depends on the ballots, and the relevant cost paid to reach a decision is moved from the voting phase to the counting of votes.

We present Caesar, a replicated transactional system that takes advantage of an innovative multi-leader protocol implementing generalized consensus to enable high parallelism when processing transactions. The core idea is enforcing a partial order on the execution of transactions according to their conflicts, by leaving non-conflicting transactions to proceed in parallel and without enforcing any synchronization during the execution (e.g., no locks). The novelty of Caesar is in the way it is able to efficiently find a partial order among transactions without relying on any designated single point of decision, i.e., leaderless, and overcoming the limitations of recent democratic contributions in the field of distributed consensus (e.g., [78]) when applied to transactional processing.

Inspired by Mencius [69], each node in Caesar has a preassigned set of available positions that can be associated with transactions activated on that node, and any two sets of two different nodes have an empty intersection. In Caesar, a transaction T activated on node N is executed on all nodes at position P_T (chosen among the ones available for N) after the execution of any other transaction conflicting with T and chosen at a position less than P_T. To do that, unlike Mencius, N does not need to gather information from all nodes in the system to understand the status of all positions less than P_T. Rather, Caesar borrows the idea of exchanging quorums of dependencies from protocols like EPaxos [78], thus guaranteeing high performance also in presence of failures or non-homogeneous nodes (e.g., slow or temporarily unreachable nodes).

However, Caesar provides a set of innovations that make the proposed ordering protocol well suited for On-Line Transactional Processing (OLTP) workloads, where the transaction's execution flow depends on the outcome of read operations. This kind of workload is unquestionably the most diffused among common transactional applications.

[3]The business logic of the transaction calls an abort due to the results of some read operation.

Besides the capability of being democratic like EPaxos, because the order of a transaction can be decided upon the voting of a quorum of nodes, Caesar is also able to confine the democracy in case the voting for a transaction does not provide a positive expected outcome. In that case, in fact, the leader of that transaction autonomously decides the final order that will be used to execute the transaction. This way, unlike EPaxos, deciding the final execution order is not a process entailing the linearization of a conflict graph built using transactions dependencies.

We implemented Caesar's prototype in Java and conducted an extensive evaluation involving three well-known transactional benchmarks like TPC-C [20] and the distributed version of Vacation from the STAMP suite [13], and a Bank benchmark. In order to cover a wide range of competitors, we contrasted Caesar against EPaxos [78], Generalized Paxos [59], and a transactional system using MultiPaxos [60] for ordering transactions before the execution. As a testbed, we used Amazon EC2 [2] and we scaled our deployment up to 43 nodes. To the best of our knowledge, this is the first consensus-based transactional system evaluated with such a large network scale. The results reveal that Caesar is able to outperform competitors in almost all cases, reaching the highest gain of more than one order of magnitude using Bank benchmark, and by as much as 8× running TPC-C.

7.1 Leaderless Transactions' Ordering

7.1.1 Overview

The idea of enforcing an order only among conflicting commands has already been originally formalized by the Generalized Consensus [59] and Generic Broadcast [86] problems and followed by a set of implementations and optimizations, e.g., Generalized Paxos [59], EPaxos [78], and Alvin [115].

In contrast to them, Caesar offers an integrated solution for disseminating conflicting transactions in a way that is prone to scalability. It avoids the usage of the designated leader, as adopted by the classical Paxonian algorithms (e.g., [86, 59, 16, 110]), in order to either define the outcome of the consensus for the delivery position of a transaction, or act as a contention solver in case of multiple nodes do not agree on the delivery sequence of (dependent) transactions. In addition, it differs from two recent implementations of generalized consensus, i.e., EPaxos [78] and Alvin [115]. EPaxos relaxes the need of a designated leader by moving the complexity of the protocol from the agreement phase (where nodes exchange information with the aim of defining the delivery position of a transaction with respect to the other dependent transactions) to the delivery phase. In fact, its delivery phase requires the linearization of a conflict graph built by taking into account the transaction dependencies collected during the agreement phase in order to produce the final partial order. Caesar does not need any computation after the transaction's delivery. Alvin does not require

EPaxos's graph processing but involves expensive rounds of communication for exchanging the transaction dependencies missed by the leader during the ordering phases, thus keeping the message footprint large. Caesar collects dependencies only once during the ordering phase, thus it reduces the network utilization.

The core idea behind Caesar is to define a deterministic scheme for the assignment of *delivery positions* (i.e., positions in the final order that are associated with positive integers) to submitted transactions, by following the general design of *communication history*-based total order broadcast protocols [25, 69] where message delivery order is determined by the senders.

In Caesar, for each transaction T we define a unique transaction leader L_T that is responsible for establishing the final delivery position of T. L_T is either the node that broadcast T, or any other elected node if the latter is crashed or has been suspected by the failure detector. Assuming N_i as L_T, the final delivery position of T can only be an unused position P_T such that $P_T \bmod |\Pi| = i$.

Caesar delivers a transaction T in position P_T if and only if $T \in deps_{T'}$ for each directly dependent transaction T' delivered in position $P_{T'} > P_T$, where $deps_{T'}$ is the set of transactions which T' directly depends on. We say that two transactions are *directly dependent*, or also *conflicting*, if they both access a common object and at least one of them writes on that object. Thanks to this property, T' can be executed without any additional synchronization mechanisms (e.g., locks) as soon as T commits because, this way, once T' starts its execution, its accessed objects cannot be requested by any other concurrent transaction.

Therefore, when node L_T has to define a delivery position for T, it selects the next available position among the unused ones as defined above, and it tries to "convince" the other nodes to accept T at that position issuing a *proposal* phase. A generic node N_j can accept T at a certain position in case no other transaction T', which conflicts with T, will be delivered on N_j at a later position without having observed T, i.e., T' does not have T in its dependency set $deps_{T'}$. In fact, if that was the case, a transaction T', which is ready to be executed on node N_i, would have no clue about a possible later delivery of T, which should actually be executed before T'. The replies collected by L_T in this phase are also used for building the final dependency set $deps_T$ of T.

In case L_T receives at least one negative acknowledgement about the acceptance of T during the *proposal* phase, it executes a *retry* phase under no democracy in order to force a new position. Subsequently, L_T broadcasts the final agreed position P_T to all, whether the position is the result of the *proposal* phase or the *retry* phase, together with the final dependency set $deps_T$. Finally, once $deps_T$ is delivered to a node, T can be processed on that node as soon as all transactions in its $deps_T$ have committed on that node.

It is worth to note that, in both EPaxos and Alvin, when a leader collects discordant dependencies in the *proposal* phase, it has to go through an additional phase (similar to Caesar's *retry* phase) in order to finalize the order of a transaction. This is not the case for Caesar, in which a leader collecting discordant dependencies decides without undergoing the *retry* phase, as far as the position proposed in the *proposal* phase is accepted by other nodes.

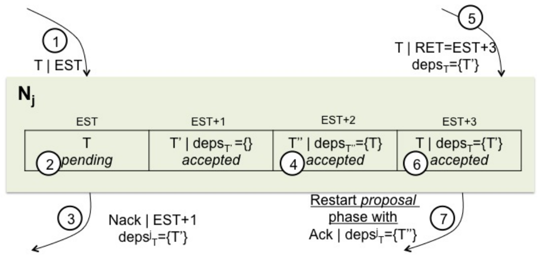
Besides the general intuition of Caesar illustrated above, there are a number of key aspects the protocol takes into account in order to provide the system's liveness, high-performance and scalability guarantee. First, since a leader needs to retry a proposal phase for a transaction in case a proposed position has been refused by some node, one might think that the ordering process is prone to execute a potentially infinite number of *proposal* phases for a certain transaction. However, as we will see in the section discussing the correctness of the protocol (Section 7.3), Caesar's design guarantees that the retry of a proposal is executed at most once per transaction. This is fundamental because it ensures that, given a transaction T and T's non-faulty leader L_T, the decision process for T will always converge in a finite and known sequence of steps.

The second critical aspect regards the number of replies a node should wait for before moving forward in its ordering process. In Caesar, a node does not need to sense all the other nodes before taking any decision (unlike Mencius [69]), in fact the proposal for a transaction can be accepted by the transaction's leader if it is acknowledged by at least a quorum of nodes. As it will be clear later, waiting for the replies from a quorum of nodes is enough to consider a proposed position as *accepted*. The actual size of the quorums used by Caesar depends on the possibility of having the so called *fast decision*, namely the capability of deciding a transaction in two communication delays when an initial proposal receives a confirmation from a quorum of nodes.

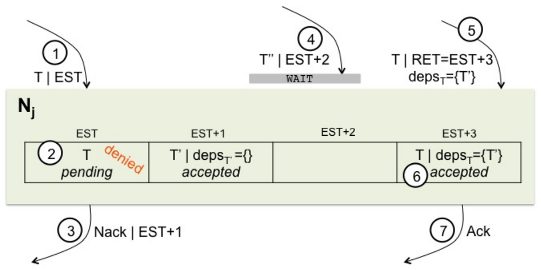
7.1.2 Unfolding the Idea

Before moving into all the details of the protocol (Section 7.2), it is important to explain how the core idea of Caesar, shown in the previous subsection, has been unfolded in order to obtain the final working protocol. Here we illustrate how the rules executed by the protocol have been inferred, along with an intuition on the correctness of those rules.

The main flow of the decision process about the position of execution of a transaction T goes through the following four steps: (1) the leader L_T selects an initial proposal for T (i.e., EST); *(2)* L_T requests to a quorum Q_F of nodes the confirmation of EST; *(3)* in case the replying quorum does not unanimously confirm EST, L_T issues a retry for T with a new position RET, and waits for the acceptance of the new proposal by a quorum Q_C of nodes; *(4)* L_T broadcasts the final position P_T of execution of T as either EST or RET, depending on whether the initial proposal has been confirmed or not. In the process of confirming a proposal for T (step *(2)*), L_T and the other nodes also agree on the set of transactions that should precede T in the partial order and that are conflicting with T (i.e., $deps_T$). $deps_T$ is the only information used locally by a node to determine when T can be delivered for the execution, and it is therefore broadcast with the final position P_T. Given the above steps, we now show why a node may not confirm a proposal, and how the protocol behaves in this case.



(a) Wrong scenario where T'' does not wait for the retry of T. T has to recompute dependencies by restarting a *proposal* phase.



(b) Correct scenario where T'' waits for the retry of T.

Figure 7.1: Acceptance rule Acc2 on *denied* positions.

Acceptance rule 1: looking at the future

Intuitively, since a transaction T' can be delivered for the execution only when all transactions in $deps_{T'}$ have already been executed, the position accepted for any transaction in $deps_{T'}$ should be less than the position accepted for T'. Therefore, let us consider a node N_j receiving an initial proposal EST for a transaction T. Caesar defines the following acceptance rule for the tuple $[N_j, EST, T]$.

- Acc1$[N_j,\ EST,\ T]$. N_j can confirm the position EST only if there is no transaction T' conflicting with T and accepted with a position greater than EST, such that T does not belong to $deps_{T'}$.

In case Acc1$[N_j,\ EST,\ T]$ is true, N_j can send a confirmation to L_T with the set of transactions conflicting with T and that are known by N_j to be possibly accepted in a position less than EST, i.e., $deps^j_T$. Note that this set will be used by L_T to compute the final $deps_T$.

On the contrary, in case Acc1$[N_j,\ EST,\ T]$ is false, N_j cannot confirm T at position EST, and therefore it sends a negative acknowledgement Nack to L_T, by notifying the maximum position that denied EST for T. This information is used by L_T to choose a new position, i.e., RET, such that Acc1$[N_j,\ RET,\ T]$ must be true. In addition, N_j stores the result of the failed proposal for T at position EST as *denied* until L_T actually sends the new proposal for T.

Acceptance rule 2: looking at the past

Remembering the *denied* proposals is the key aspect that allows Caesar to avoid recomputing and re-exchanging dependencies while L_T retries a new position for T. In particular, let us consider the scenario depicted in Figure 7.1(a), where node N_j sends a negative acknowledgement Nack for the proposal EST of T (in steps 1-3 because Acc1$[N_j,\ EST,\ T]$ is false). We assume that this is because N_j has already accepted a transaction T' at position $EST+1$, such that T' conflicts with T, and T is not contained in $deps_{T'}$. In case the NACK from N_j is in the quorum of replies received by L_T, L_T is forced to select a new proposal for T that has to be greater than $EST+1$ (i.e., $RET = EST+3$ in the example). In the meanwhile, N_j could accept another transaction T'' at position $EST+2$ and such that T'' and T are conflicting, and T is contained in $deps_{T''}$ (step 4 in Figure 7.1(a)). Afterwards, T is received by N_j with the retry at the new position $RET = EST+3$, and it finds T'' as conflicting and accepted at position $EST+2$ (which is clearly less than RET). As a consequence, N_j should include T'' in the dependencies of T and notify L_T, which on its side has to build again $deps_T$ as part of a new restarted proposal phase for T.

To solve this problem Caesar uses information about *denied* proposals in order to guarantee that a retry always succeeds without the need of re-exchanging dependencies. Intuitively, if a node N_j receives a proposal EST for a transaction T, it blocks the agreement process for T in case there is a transaction T' such that T and T' are conflicting and N_j previously sent a NACK for a position EST' of T' that is less than EST. This way we derive the following acceptance rule for the tuple $[N_j, EST, T]$.

- ACC2$[N_j, EST, T]$. N_j stops processing T until there is no transaction T' such that T and T' are conflicting and N_j previously sent a NACK for a position EST' of T' less than EST, i.e., T' was *denied* at position EST'.

Therefore, by using the rule in the scenario depicted in Figure 7.1(a), we force transaction T'' to wait for the arrival of T at a new position. We depict the example of the new behavior in Figure 7.1(b). When T'' is received by N_j (step 4), rule ACC2$[N_j, EST+2, T'']$ blocks the processing of T'' because there is a conflicting transaction T at a *denied* position EST, where EST is clearly less than $EST+2$. After the arrival of T at the new position RET, T'' can be resumed.

7.2 Protocol Details

Given the description of the main idea behind the design of Caesar and the explanation on how the idea is supported by the acceptance rules ACC1 and ACC2, in this section we provide all details of the protocol. An execution of a transaction T in Caesar goes throughout three phases, named *Proposal* phase, *Retry* phase and *Processing* phase.

7.2.1 Proposal Phase

The *Proposal* phase starts when T is submitted by a client to a node N_i in the system. N_i becomes T's transaction leader L_T and it becomes responsible for finding a serialization order of T's execution against other concurrent transactions that are conflicting with T.

To do that, N_i selects its estimated execution order of T (i.e., EST), and it broadcasts a CONVINCE message containing T, EST and $deps_T^i$ to all nodes[4]. EST is a position greater than any other position observed by N_i so far and such that $EST \bmod |\Pi| = i$; $deps_T^i$ is the set of all transactions conflicting with T and that have been observed by N_i with a position less than EST.

By doing that, N_i tries to convince at least a quorum of nodes to serialize the processing of T at position EST. After that, it simply waits for Q_F replies in order to decide whether EST is the final position for T or not. If all the replies in the quorum of size Q_F are ACK messages, N_i can confirm the processing order of T as EST, otherwise if the quorum contains at least a NACK message, T needs to undergo the additional *Retry* phase.

As we explained in the overview of the protocol, the final delivery for processing a transaction T relies on the set of conflicting transactions which T depends on. Therefore, in the case EST is confirmed, N_i needs to also determine the final set of conflicting transactions that have to be executed at positions less than EST. In practice, an ACK message from a node N_j also contains the set of transactions $deps_T^j$ known by N_j and that are going to be possibly delivered at a position less than EST. In this case, N_i receives Q_F ACK messages for T at position EST and therefore it can broadcast an EXEC message with parameters P_T and $deps_T$ to confirm that T has to be executed at position P_T after transactions in $deps_T$. In this case, P_T is clearly equal to EST and $deps_T$ is computed as the union of the sets $deps_T^j$, where $deps_T^j$ is contained in the ACK message from N_j.

Before analyzing the case where N_i receives at least a NACK message in the quorum of replies, let us understand how a node N_j, receiving a CONVINCE message from N_i, computes its reply. As a precondition, N_j first needs to execute the acceptance rule ACC2[N_j, EST, T]. Therefore, N_j stops executing steps for T in case N_j stores a transaction T' such that T and T' are conflicting and N_j previously sent a NACK message for a position EST' of T' that is less than EST. In fact, in this case T' is a transaction denied at a smaller position EST' (see example of Figure 7.1).

If that is not the case, N_j can send an ACK message if ACC1[N_j, EST, T] is true, namely there is no transaction T' conflicting with T and *accepted* with a position greater than EST, such that T is not in $deps_{T'}$. As we will also clarify later on, a transaction T' is *accepted* on N_j if N_j received either an EXEC message or a RETRY message for T'; otherwise we say that T' is *pending* on N_j.

Therefore, N_j executes the following steps for T: *(i)* it waits for the set *watchSet* of all transactions conflicting with T and having a position greater than EST to be accepted; *(ii)* if ACC1[N_j, EST, T] is true, N_j sends an ACK message to N_i, where $deps_T^j$ is the set with all transactions conflicting with T and that are either *pending* or *accepted* on N_j with a position less than EST; *(iii)* if ACC1[N_j, EST, T] is false, then N_j sends a NACK message

[4]We always assume that the destinations of a broadcast are all nodes in the system, including the node broadcasting the message.

to N_i by specifying the greatest position in *watchSet* (i.e., $maxWatch$), and again the $deps_T^j$ computed as the set of all transactions conflicting with T and either *pending* or *accepted* on N_j with a position less than or equal to $maxWatch$.

The intuition here is the following. If N_j confirms the position EST, it informs N_i about the transactions that it knows should be executed before T. Otherwise, if N_j cannot confirm the position EST because there exist other transactions (i.e., *watchSet*) that will be executed at positions greater than EST and that are not aware about the existence of T at position EST, then N_i has to retry with a position greater than the ones of transactions in *watchSet*.

7.2.2 Retry Phase

In case N_i receives at least a NACK message in the quorum of replies, it has to retry with a new position for T. It selects as new available position RET, which is the position that is greater than any other position both contained in the NACK messages and ever observed by N_i so far. Clearly RET has to be admissible for N_i and therefore such that $RET \bmod |\Pi| = i$. In addition, N_i also computes $deps_T$ at this stage, as it would have done at the end of a successful *proposal* phase, i.e., by considering the union of all sets $deps_T^j$ contained in the ACK/NACK messages of the received quorum. Afterwards, N_i broadcasts a RETRY message with the pair RET and $deps_T$ in order to inform all the other nodes about its decision on T.

This decision, i.e., RET and $deps_T$, is the final of N_i for T because of the twofold motivation: *(i)* as it will be clarified in Section 7.3, at this stage there cannot exist a node N_j receiving RET from N_i such that ACC1$[N_j,\ RET,\ T]$ is false, therefore forcing to change the position; *(ii)* as previously explained in the example of Figure 7.1, there cannot exist a node N_j proposing to further update $deps_T$ upon the reception of RET on a retry.

However, the decision process for T cannot stop here because we have to be sure that what N_i decided survives even in case N_i becomes faulty later on. For this reason, in this phase N_i waits for a quorum of Q_C replies for acknowledging the reception of the message RETRY. After that, its decision can be sent to all nodes by broadcasting the EXEC message, along with the final position $P_T = RET$ and $deps_T$.

A node N_j receiving a RETRY message for T at position RET and dependencies $deps_T$ updates the status of T as *accepted* and sends an acknowledgement back to N_i. In addition, N_j deletes any information (if any) about an old proposal for T so that any transaction T', which experienced a false ACC2$[N_j,\ EST',\ T']$ due to T, will be unblocked.

7.2.3 Execution Phase

A node N_j receiving an EXEC message for T at position P_T and dependencies $deps_T$ updates the status of T as *accepted* (if it was still *pending*) and it marks T as *ready* to be executed.

The execution phase of Caesar is really lightweight because any transaction T marked as *ready* can be executed as soon as all transactions belonging to its dependency set, i.e., $deps_T$, commit. Furthermore, the actual execution of a transaction can proceed without any instrumentation because it does not require any synchronization while accessing shared objects. In fact, any other transaction that could have developed a direct or transitive dependency with T has been either executed before T or it will execute after the completion of T.

In order to support such an execution, *ready* transactions should not create circular dependencies. This problem can be trivially solved by removing every transaction T' from $deps_T$ if T' is ready at a position greater than the one of T. It is worth noticing that this operation does not entail missing the dependency between T and T', because the protocol guarantees that in this scenario T is always contained in $deps_{T'}$, since T is ordered before T'.

Finally, the client requesting the execution of T receives a reply with the transaction's outcome when the execution of T completes on T's current leader.

7.2.4 Size of the Quorums and Handling of Failures

The way we select the values of Q_F and Q_C, i.e., fast and classic quorums, affects the behavior of Caesar in how it guarantees correctness even in the presence of failures. As a general scheme, if a node N_i suspects a node N_j, N_i tries to become the leader of any *pending* transaction T whose leader is currently N_j, so that T can be finalized. This way we have to guarantee that, if there exists at least one node that already decided to execute T at a certain position, N_i cannot choose to execute T at a different position. As a result, the size of the quorums should be such that when N_i caches up and decides to finalize the execution of T, it is able to gather enough information from the other correct nodes in order to avoid mistakes.

To accomplish this recovery, N_i gets information about T and T's dependencies from a quorum of Q_C nodes and it explores a set of cases to find out the final position of T. The trivial cases are the ones where there exists at least one node in the quorum that already accepted T, i.e., it received either an EXEC message or a RETRY message for T. In those cases, in fact, N_i needs only to force that decision by re-executing a *retry* phase followed by an *execution* phase.

In the other cases, T was at most *pending* at all nodes in the quorum that replied to N_i. Thus these Q_C nodes only observed at most the CONVINCE message for T that was sent in the *proposal* phase from T's old leader N_j. Therefore we choose the size of Q_F in the *proposal* phase such that if N_j decided after having received Q_F ACK messages in the *proposal* phase, then the Q_C replies gathered by N_i would contain a majority of ACK messages for T. Equivalently, we say that the number of gathered NACK messages, which is at most equal to $|\Pi| - Q_F$, has to be less then $\frac{Q_C}{2}$:

$$|\Pi| - Q_F < \frac{Q_C}{2} \tag{7.1}$$

Actually we can do better than that, because in case T is conflicting with another transaction T', and N_i received information from the leader of T' during T's recovery, N_i can just understand the relative position of T by looking at the decision for T', i.e., if T is not in $deps_{T'}$ then T' has to be in $deps_T$; and, vice-versa, if T is in $deps_{T'}$ then T' should not be inserted in $deps_T$. We can then adjust Eq. 7.1 by saying that the number of gathered NACK messages has to be less then $\frac{Q_C}{2}$, only in case we do not gather information from a leader of a transaction conflicting with T:

$$|\Pi| - Q_F - 1 < \frac{Q_C}{2} \tag{7.2}$$

N_i's recovery proceeds by choosing the position confirmed by a possible majority of ACK messages in the quorum of Q_C replies that are gathered before. If that majority does not exist, N_i can decide the position of T by looking at the decisions made by the leaders of all transactions reported by the Q_C replies as conflicting with T.

Since the number of nodes $|\Pi|$ is known, we need an additional equation to obtain the values of Q_F and Q_C. In particular we have to avoid that two new leaders of two conflicting and concurrent transactions T and T', here called *opponents*, both believe that the old leaders of T and T' respectively had both decided in the *proposal* phase (hence after having collected Q_F ACK messages) and such that $T \notin deps_{T'}$ and $T' \notin deps_T$ (which is clearly impossible). After f failures and ignoring the reply from the other opponent, each opponent cannot collect a sufficient number of replies ($\frac{|\Pi|-f}{2}$) in the recovery phase which totaled up to f is greater than or equal to Q_F:

$$\frac{|\Pi| - f}{2} + f - 1 < Q_F \tag{7.3}$$

If we minimize the ratio $\frac{|\Pi|}{f}$ by considering $|\Pi| = 2f + 1$, we obtain the following sizes for the classic and fast quorums, respectively:

$$Q_C = f + 1 \tag{7.4}$$

$$Q_F = f + \left\lfloor \frac{f+1}{2} \right\rfloor \tag{7.5}$$

Note that these are the same quorum sizes adopted by EPaxos [78] and the recovery phase we are applying follows the one in EPaxos.

7.3 Correctness

Due to space constraints we do not provide the formal proof on the correctness of Caesar even thought we gave some hints on why the protocol works in Sections 7.1.2 and 7.2. However in this section we provide informal arguments on how Caesar is able to guarantee *One-Copy Strict Serializability* (1CSS) on the history of the committed transactions.

A transaction T is executed and committed on a node N_i only after all transactions conflicting with T and contained in $deps_T$ have been executed and committed on N_i. So, by iterating the reasoning for the transactions in $deps_T$, we can then conclude that any two transactions T and T' ready to execute on a node N_i are processed and committed sequentially (i.e., one after the other) on N_i if there is an either direct or transitive dependency between the two. This is enough to guarantee that there always exists a serial history of committed transactions $\mathcal{S}$ (where all transactions are executed sequentially) that is equivalent to the history $\mathcal{H}$ of the transactions committed on N_i. In particular $\mathcal{S}$ is the serial history containing all and only transactions committed in $\mathcal{H}$ and it is obtained from $\mathcal{H}$ as follows:

- The order of execution of any pair of transactions that are executed sequentially in $\mathcal{H}$ is preserved in $\mathcal{S}$.
- The order of execution of any pair of transactions that are executed concurrently in $\mathcal{H}$ is any of the two possible ones.

For the formal definition of histories, equivalence of two histories and Strict Serializability we refer to [7, 1]. However, intuitively since $\mathcal{S}$ is serial, and the two histories have the same return values for the executed read operations and produce the same transactional state, $\mathcal{H}$ is Serializable. In addition $\mathcal{H}$ is also Strict because the equivalent serial history $\mathcal{S}$ does not revert the order of execution of non-concurrent dependent transactions.

To show that any history $\mathcal{H}$ committed by Caesar (here we are not referring to a particular node anymore) is 1CSS, we need to show that it is equivalent to a serial history $\mathcal{S}$ as it was executed over a single logical non-replicated copy of the transactional state. This is trivially verified in Caesar because the partial order of transactions is the same on all nodes and $\mathcal{S}$ can be built as shown above by starting from the history $\mathcal{H}$ committed on any of the nodes (all nodes execute and commit exactly the same history).

We have shown that Caesar guarantees One-Copy Strict Serializability thanks to two properties of the ordering protocol: *i)* the partial order of transactions is the same on all nodes and *ii)* if two transactions T and T' are conflicting and they are decided at positions P_T and $P_{T'}$ respectively, where $P_T < P_{T'}$, then T is contained in $deps_{T'}$.

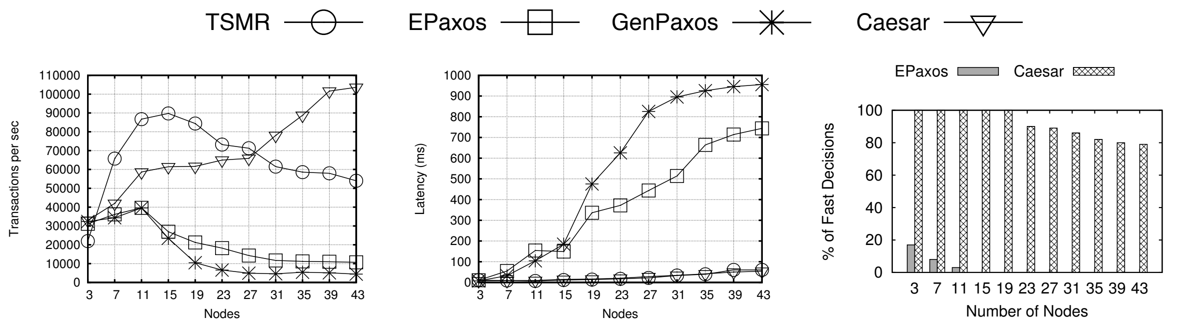
In a failure-free execution the former property is verified since the decision on the order of a transaction T, i.e., position P_T and dependencies $deps_T$, is taken by a unique node, namely T's leader L_T, which broadcasts the decision to all the other nodes. Further, in a scenario with failures, even though T's leader fails before all nodes have learnt its decision about T, but after the decision has been learnt by at least one node in the system, Caesar guarantees that T's new leader does not enforce any decision different from the one already taken by T's old leader (see Section 7.2.4).

Concerning the latter property, we can show that if we suppose that the property is not guaranteed by contradiction, we can obtain an absurd. In particular we can suppose that there are two conflicting transactions T and T' that are decided at positions P_T and $P_{T'}$ respectively, where $P_T < P_{T'}$, and such that T is not contained in $deps_{T'}$. We have to distinguish two different scenarios depending on whether T' executes a *retry* phase or not.
T' executes a *retry*. We suppose that T' first selects a position $EST' < P_T$ (because T is not in $deps_{T'}$) and then it retries with position $P_{T'}$ due to a NACK received in the *proposal* phase. Thanks to the acceptance rule ACC1 (Section 7.1.2), if there is a node sending a NACK for T', then there are at least Q_C nodes that do the same (because a NACK is sent due to an *accepted* transaction). Therefore, there is at least a node N_j where the *proposal* phase for T waits till the reception of T' at new position $P_{T'}$ due to the acceptance rule ACC2 (Section 7.1.2). Furthermore, since T' is accepted without T in $deps_{T'}$, then N_j forces T to retry at a new position $P_T > P_{T'}$, due to the acceptance ACC1, by contradicting the hypothesis $P_T < P_{T'}$.
T' does not execute a *retry*. We suppose that the first proposal EST' by T' is the final one $P_{T'}$. Since T' has been accepted without T in its $deps_{T'}$, there exists at least one node N_j that forces a *retry* phase for T at position $P_T > P_{T'}$. And clearly that contradicts the hypothesis $P_T < P_{T'}$.

As a last note we show that the *retry* phase for a transaction T cannot be executed multiple times in Caesar. In fact, there cannot exist a node N_j replying in the *retry* phase of T and such that ACC1$[N_j, P_T, T]$ due to a conflicting transaction T'. This is because the only two admissible cases are the following: *(i)* P_T is greater than the position $P_{T'}$ of T' because T observed T' before the *retry*; *(ii)* T is in the dependency set of T', namely $deps_{T'}$, because T' observed T during its *proposal* phase.

7.4 Implementation and Evaluation

We implemented a prototype of Caesar in Java and we contrasted it with the three most related competitors in literature: EPaxos, Generalized Paxos (called GenPaxos), and a transactional system based on the State Machine Replication Approach [84] where transactions are ordered before executions using the MultiPaxos protocol [60] (called TSMR). This selection allows us to identify strengths and weaknesses of Caesar because each of these competitors performs well in a particular scenario and suffers in others. The main goal of this evaluation is showing how Caesar scales up along with the size of the system, while all competitors (especially those based on single leader) slow down after a certain amount of nodes. We did not include Alvin in this evaluation because Alvin adopts a different replication scheme where transactions are executed locally at the originating node before being globally ordered. This approach has advantages and disadvantages when compared with Caesar's model, thus



(a) Throughput. (b) Latency. (c) % Fast Decisions.

Figure 7.2: Performance of TPC-C benchmark.

a system comparison would be misleading.

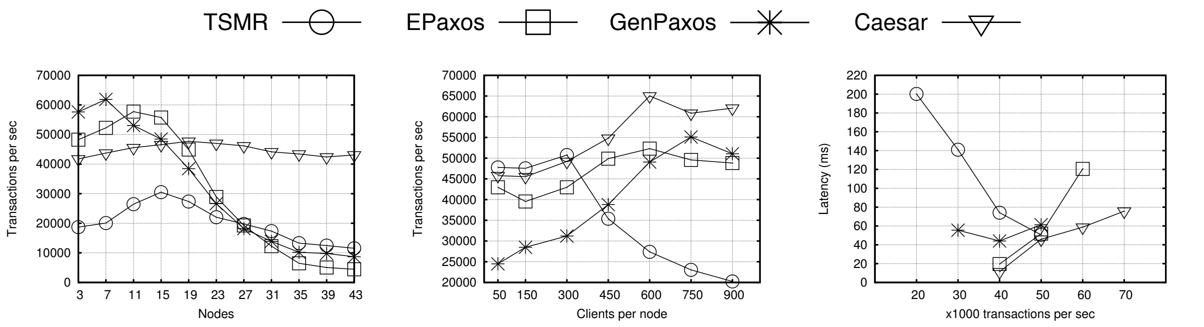
In order to conduct a fair comparison, all competitors have been re-implemented using the same programming language and the same runtime framework developed for Caesar. This way, all take advantage of the same low level mechanisms applied to Caesar, e.g., *batching* network messages in a single but bigger message.

In order to pursue this goal, we used the Amazon EC2 infrastructure reserving 43 nodes within the category *c3.8xlarge*. To the best of our knowledge, this is the first evaluation showing a Paxos-based transactional system with such a large network. Each machine has 32 CPU cores and 60GB of memory. All nodes are located on the same Amazon datacenter and interconnected through a 10Gbps network. All results are the average of 5 samples.

The evaluation study goes through three well-known benchmarks: two of them are real applications (i.e., TPC-C [20] and Vacation [13]), and one is the lightweight Bank benchmark. For each benchmark we identified certain contention levels by changing the total number of shared objects. This way we analyze scenarios where few objects are shared, thus total order (as TSMR) should pay off more than others, or scenarios where the shared dataset is large, thus the probability to access common objects is lower and conflicting-aware ordering protocols (such as GenPaxos, EPaxos and Caesar) should gain more. All configurations do not include the classical optimization of running read-only transactions locally using a multi-versioning concurrency control because this way we can clearly compare the rules used by the different competitors for ordering (and processing) transactions. In all the experiments we set a fixed amount of clients per node. This way, increasing the nodes also means increase the load of the system.

7.4.1 TPC-C and Vacation Benchmarks

TPC-C and Vacation perform a non trivial amount of work per transaction. TPC-C is the popular on-line transaction processing benchmark, and Vacation is the distributed version



(a) Throughput.

(b) Throughput varying clients using 19 nodes.

(c) Throughput Vs Latency using 19 nodes.

Figure 7.3: Performance of Vacation benchmark.

of the famous application included in the STAMP suite [13] that reproduces the behavior of clients that submit booking requests for vacation related items.

Figure 7.2 shows the performance collected running TPC-C. The benchmark has been configured using the classical configuration, which suggests to deploy as many warehouses (the most contented object) as the total number of nodes in the system, thus we deployed 43 of them. Most TPC-C transactions perform an operation to a warehouse before to execute other operations. Given this access pattern, TPC-C's workload is notoriously considered very conflicting. This is clear also from Figure 7.2(a) where the best competitor is TSMR due to the presence of a single leader which orders all transactions indiscriminately, without looking into their conflicts. However, after 27 nodes, TSMR leader's resources saturate, thus slowing down its ordering process.

On the other side, Caesar is slower than TSMR for low node count (due to also the lower number of clients in the system), but it starts being very effective where TSMR shows scalability issues because of the exploitation of multiple leaders and its fast decision rule, which allows the delivery of an ordered transaction even if the dependency sets collected for that transaction are discordant. Interestingly, EPaxos is not able to exploit the fast decision properly because, with few shared objects, the probability to collect different dependency sets during the *proposal* phase is high. This causes a second phase, which involves further communication steps, and thus it explains EPaxos's low throughput. Figure 7.2(c) confirms this intuition, showing that Caesar is able to deliver an ordered and ready-to-process transaction with the fast decision in at least 80% of the cases. We do not include GenPaxos in this plot because it uses the same conditions as EPaxos for fast delivering an ordered transaction so their performance would be very similar (we also increased the readability of the plot). As overall improvement, Caesar gains up to 8× better than EPaxos at high node count, and almost one order of magnitude against TSMR.

Figure 7.2(b) reports the latency measured during the previous experiment. If follows the same (inverse) trend of the throughput, thus EPaxos shows the worst client perceived latency.

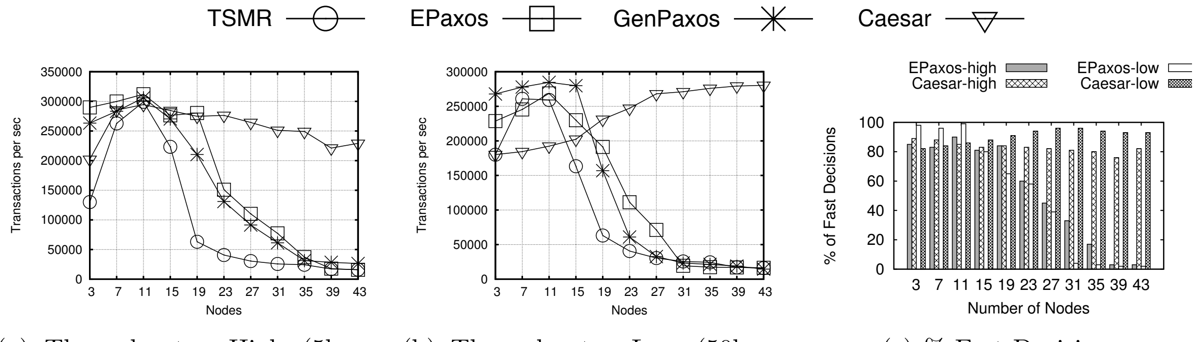
This can be explained considering the number of operations that EPaxos needs to perform before replying to the client, which includes the analysis of the dependency graph and the breakage of strongly connected components. Those operations are not taken into account if transactions do not read any object but they only write. In this case EPaxos can actually reply to the client before the accomplishment of the above operations. However such a workload is not representative of usual transactional applications such as TPC-C.

GenPaxos is consistently slower than EPaxos, especially after 11 nodes. Interestingly, from the analysis of the plot in Figure 7.2(b), GenPaxos provides latency up to 40% lower than EPaxos in the range of 3-15 nodes. This is because within a high performance network infrastructure and without an overloaded leader, solving conflicting proposals using a designated leader, as GenPaxos, helps to reduce latency.

Table 7.1 reports some further performance statistics related to EPaxos and the impact of local computation before starting the execution of a transaction. We report for three network deployments (11,19 and 31 nodes): the average size of the dependency graph, found every time the analyzing procedure starts; the percentage of strongly connected components (SCC) having more than one node involved against all the SCC found (including the trivial ones with only one nodes); the average number of nodes within a SCC. Clearly, increasing the size of the system, and thus the contention, the number of SCC grows, along with its size.

Nodes	Average size of graph	% SCC >1	% Average size of SCC
11	2.91	4%	1.27
19	26.36	17%	4.25
31	44.91	47%	9.65

Table 7.1: Costs of EPaxos's graph analysis using TPC-C.



(a) Throughput - High (5k accounts). (b) Throughput - Low (50k accounts). (c) % Fast Decisions.

Figure 7.4: Performance of Bank benchmark.

The performance of Vacation's benchmark is reported in Figure 7.3. We configured it to share 2000 relations (the key object type of Vacation). The trend of all competitors is similar to the one showed before but Vacation produces in general a less conflicting workload than TPC-C, thus EPaxos is exposing better performance than others until 19 nodes, then Caesar takes over and sustains its performance while increasing the system's size. Interestingly, TSMR is constantly slower than other multi-leader protocols even for low node count. The reason is related to the network message size, which is high in Vacation thus reducing the benefit of leader's batching. We recall that batching in case of TSMR is the key component for providing high performance.

The performance of GenPaxos clearly shows its capability to exploit the fast decision even in case the leader is close to its saturation point, thus outperforming TSMR but still being slower than EPaxos. The inverse situation happens when EPaxos degrades and GenPaxos places its performance again in between TSMR and EPaxos (but this time performing better than EPaxos).

Figure 7.3(b) reports the throughput varying the number of clients in the range of 50 to 900 and fixing the number of nodes as 19. This plot shows how the competitors react when the load of the system is progressively increased. All our experiments assume a constant number of clients per node, thus there could be cases where the overall system is not properly loaded (especially in the left part of the plots). Figure 7.3(b) covers this case. Despite TSMR, which does not increase its performance along with the number of clients, both EPaxos and Caesar scale up until 600 clients per node, and then sustain.

In Figure 7.3(c) we integrated the results showed in Figure 7.3(b) with the client perceived latency. Specifically, for each competitor we report the latency measured when the system was providing a certain throughput (using 19 nodes). Given that not all competitors provide all the plotted throughput, some data points are not shown. The plot shows an interesting aspect of TSMR, high latency with low throughput. This is mainly because the system is not properly loaded, thus the batching mechanism has to wait a certain amount of time before sending out the batch. This timeout is never reached in other cases where the batch is sent due to space depletion.

7.4.2 Bank Benchmark

The Bank benchmark is an application that mimics the operations of a monetary system. Each transaction accesses two objects (i.e., accounts) and move a certain amount of money from one account to another. This benchmark is widely used because it is configurable and the transaction execution time is very limited, thus pointing out other predominant costs in the system (e.g., the way transactions are ordered). Given that, Bank represents a good scenario for TSMR because the serial execution after the delivery does not have a big impact. However, after a certain amount of nodes we expect TSMR's performance go down due to saturation of leader's resources.

Figure 7.4 shows the transactional throughput varying the number of nodes and the contention level. We managed to change it by increasing the number of shared *accounts* in the system (i.e., more account means less contention). We identified 5000 and 50000 as high and low contention level, respectively. Each transaction accesses two accounts uniformly distributed across all available. TSMR is not affected by the different contention levels because it processes totally ordered transactions serially, thus the only impact of having more shared objects is in the management of the data repository, which is slightly more expensive. However, in both the tested scenarios (Figures 7.4(a) and 7.4(b)) after 15 nodes, the performance of the leader goes down significantly.

EPaxos performs better than Caesar until 19 nodes, then the contention starts increasing and this entails reducing the probability of delivering transactions using a fast decision (Figure 7.4(c)) and a bigger graph to analyze for EPaxos. Both these overheads push EPaxos's performance down. Caesar is not affected by such a overhead, thus it sustains its performance even in presence of high contention (Figure 7.4(a)), or increases it in the lower contention scenario (Figure 7.4(b)). Figure 7.4(c) highlights the impact of delivering with a fast decision also in presence of discordant dependencies (but with a confirmed order). EPaxos's fast decision probability suddenly drops increasing the contention whereas Caesar still exploits this important feature.

GenPaxos's performance is in between TSMR and EPaxos in almost in all data points. In the scenario with 5k accounts, GenPaxos is able to move the saturation point of TSMR a bit further, but it still saturates before EPaxos, which does not rely on a centralized leader for executing the second phase after an unsuccessful proposal phase. The same happens at lower contention level, where the fast decision and a still non-overloaded leader allow GenPaxos to behave better than others until 15 nodes. From the analysis of the plot in Figures 7.4(a) and 7.4(b) we can deduct that EPaxos outperforms Caesar by as much as 28% using 11 nodes, and TSMR performs better than Caesar by as much as 29% using 7 nodes. However, when the system's size increases, Caesar finds its sweet spot and outperforms both EPaxos and TSMR by more than one order of magnitude.

7.5 Conclusion

Caesar shows that having egalitarian multi-leaders in a high node count replicated transactional system does not pay off in presence of interactive workload, where transaction's outcome can be delivered to the client only after the actual transaction's processing. However, single-leader replication protocols do not scale along with the size of the system, thus the presence of multiple leaders is still desirable. Caesar proposes a solution for accomplishing this goal: enabling high performance and scalability of strongly consistent transactional systems. Results confirmed the claim by outperforming competitors on almost all tested cases.

Chapter 8

Dexter

Past work has shown that AR-based solutions outperform competitor approaches [48, 80, 54], especially on high-contention workloads (i.e., where transactions are mostly conflicting) and those with a higher percentage of read-only requests. However, AR-based solutions suffer from poor scalability.

AR exploits the ordering layer for reproducing the same transactional state on multiple nodes such that read-only workloads can be executed quickly without remote interactions. This technique scales: with increasing number of nodes, more read-only requests can be locally served on those nodes, increasing the overall performance. However, the ordering layer is expensive in terms of the number of messages exchanged for reaching agreement among the nodes (number of messages exchanged typically increase quadratically with the number of participant nodes). Thus, when the system size increases, the overall performance increases up to a certain point, after which the ordering layer becomes the performance bottleneck, significantly hampering further scalability.

In this work, we overcome AR's non-scalability. We present Dexter, an an AR-based transactional system that scales beyond the ordering layer's performance bottleneck. Our key idea is to exploit application specifics to achieve high scalability.

AR solutions usually exploit local concurrency control protocols that rely on multi-versioned objects. In particular, when a read-only transaction is delivered on a node, the transaction uses the local timestamp to determine the set of shared objects that can be seen during the transaction's execution, preserving serializability. As an example, when a write transaction T_w commits its writing on an object O_w, it appends a new version of O_w, called O_{w2}. This way, if a read-only transaction T_r that started before T_w's commit wants to read O_w, then T_r will not access O_{w2}; otherwise, T_r's transactional history could become non-serializable. Instead, T_r will access a previous version of O_w that is compliant with its previously acquired timestamp. As a result, read-only transactions are executed in parallel, without conflicting with on-going write transactions, and their execution cannot be aborted. Abort-freedom of read-only transactions is a highly desirable property, because it allows read-only workload – the common case – to be processed quickly even though the objects read are not always the latest copy available in the system.

Dexter leverages the abort-freedom property for serving those class of transactions that do not necessarily require access to the latest object versions. As an example, many online vendors (e.g., Amazon, Walmart, Ebay) keep inventory of various items at different locations. A customer who is planning to buy an item needs to know about the available inventory (in addition to price and other information) to make a timely decision. Sometimes, it is not necessary (from the vendor's standpoint) to respond with the most up-to-date information if the item has a large inventory and is not sold that often. This is because, from the customer's standpoint, it is irrelevant whether the inventory that is displayed about the item's availability is up-to-date or stale if the inventory is significantly larger than the maximum amount that a single customer would usually buy.

Dexter's architecture divides nodes into one *synchronous* group and a set of *asynchronous* groups. Nodes in the synchronous group process write-transactions according to the classical AR paradigm. Nodes in the asynchronous groups are lazily updated, and only serve read-only transactions with different requirements on data freshness. The asynchronous groups are logically organized as a set of levels, with each group maintaining objects with a certain degree of freshness. The synchronous group stores the latest versions of the objects, and thereby serves those read-only requests that need access to the latest object versions. The asynchronous group at the first level manages read-only requests on objects that are not the latest, but at the same time, are not too old as well. At the second level, as well as at further levels, the expected freshness of objects decreases accordingly. The main advantage of this architecture is that write transactions yield AR's traditional high performance, while at the same time, nodes can scale up for serving additional read-only workloads, exploiting the various level of freshness that is available or expected.

For exploiting the aforementioned architecture, obviously, the application must specify the needed level of freshness guarantees. For this reason, Dexter provides a framework for inferring rules that are used for characterizing the freshness of a read-only transaction when it starts in the system. Using this framework, the programmer can describe the application's requirements. Rules can be defined based on the elapsed time since the last update on an object was triggered, as well as based on the content of the object (as well as the type of the object). Using this framework, the programmer defines rules in content-based style, e.g., "when the field F_1 of an object O is in the range of α and β, then a read-only request can be handled at the level L," or in time-based style, e.g., "the version of object O that is ϵ old is stored at the level M."

Dexter also provides a more explicit API to the application such that the programmer can directly force the desired level for read-only requests. If the application has a specific class of queries that must access the latest versions of the objects, then the programmer can mark those requests with level 0, which corresponds to the synchronous group, and the request will be directly delivered and managed by the synchronous group.

Dexter processes transactions to match application rules. It distinguishes between the execution flows of read and write transactions. Write transactions submitted by application threads are delivered and processed in the synchronous group using a high-performance active replication protocol, without interfering with the asynchronous groups. In contrast, read-only transactions that are not tagged by the application to be handled at a specific level are not delivered to the synchronous group, but rather to the asynchronous group at the last level (i.e., with the greatest index according to the previous definition). Each asynchronous group accepts a transaction request and selects one node for its processing. If the node, during the transaction execution, recognizes that at least one rule is not satisfied, then the read-only request is propagated to one level up. This way, the asynchronous group that is able to satisfy all the rules will reply to the application, without consuming resources in the synchronous group, thereby allowing it to process write workloads without any slow-down.

The downside of this architecture is that, read-only transactions may not be processed completely locally, potentially increasing their latency. However, the number of expected levels in the system is limited, thus limiting the maximum number of hops that a read-only transaction has to perform. In addition, not all the read-only requests traverse all the levels (otherwise the application is not actually designed for exploiting Dexter's architecture). Thus, the architecture is particularly effective when the majority of the requests can be served by asynchronous groups with the lower levels of data freshness.

As we argued before, in many e-commerce-like applications, most of the queries do not need to access the latest object versions; older versions are likely sufficient. In particular, the perception of time between the system and the user is different. When the number of items available in the stock for a needed purchase is very high, a refresh time of 10 seconds is negligible from the user's perception. From the system's perception, making asynchronous updates every 10 seconds could help offload a number of read-only transactions from the synchronous group to the asynchronous groups, which reduces interference to read-write transactions, improving write transactions' throughput and, in general, overall performance.

We implemented Dexter in Java. We used HiperTM [48], an AR-based transactional protocol and system for implementing the synchronous group, directly leveraging its implementation. The rest of the infrastructure was built from scratch. We conducted an extensive evaluation study aimed at testing the scalability of the system. Our experiments on PRObE [32], an high-performance public cluster, reveal that Dexter improves throughput by as much as 2× against HiperTM using Bank, and 2.1× using TPC-C.

8.1 System Model and Assumptions

As the system setup demands of Dexter are different than the previous works, we need to refine the system model presented in Chapter 4. Since Dexter requires bounds on network communication delays a synchronous network [67], rather than a partially synchronous one.

Dexter's architecture divides nodes into one *synchronous* group and a set of *asynchronous* groups. Nodes in the synchronous group process write-transactions according to the classical AR paradigm, thus they require a consensus algorithm for ordering incoming transactions. In this group, the usual assumption is having $2f + 1$ correct nodes, where at most f nodes are faulty.

Nodes in the asynchronous groups are lazily updated, and only serve read-only transactions with different requirements on data freshness. In the asynchronous groups, we do not globally order transactions, because updates from the synchronous group are already ordered. In these groups we assume a number of correct nodes and a $\diamond$S failure detector [15], sufficient for supporting the desired leader-election algorithm (Dexter's solution is independent from the specific leader-election algorithm actually used).

Nodes share a synchronized clocks running a clock synchronization protocol, such as Network Time Protocol (NTP) [76]. In our dedicated test-bed we measured a maximum skew of 1 *msec*. We also assume that all the delays configured in Dexter are longer than this time.

8.2 Architecture Overview

Dexter groups nodes according to different levels of freshness guarantees. We define the freshness of a shared object based on two factors: time and number of updates received. Focusing only on time-based freshness can be misleading. For example, if the last update on an object was received Δ time units ago, we cannot say that the object is Δ time units old, because it is possible that no other updates could have happened during Δ and therefore the object is still fresh. In contrast, if several updates occur on the object during Δ, then it should be considered old, having a lower level of freshness.

As previously mentioned, Dexter defines two group types: the synchronous group and the asynchronous group. The synchronous group is defined as the set of nodes that serve both read-only and write transactions, and there is only one such group in the system. Overall system progress cannot be guaranteed without this group. Read-only transactions running on the synchronous group always operate on the most recent versions of the accessed objects.

The protocol for executing transactions in the synchronous group is based on active replication [102, 80, 48, 79]. This design choice is made not only for obtaining high-performance, but also because, in active replication, each node is aware of all the write transactions submitted by any client. This means that, each correct node can reproduce the same state of the whole shared data-set independently from the others. This property is fundamental to Dexter, because one of the tasks of the synchronous group is to propagate recent updates to the asynchronous groups. In addition, by keeping the size of the synchronous group small, we also inherit AR's key benefits including faster execution of write transactions and

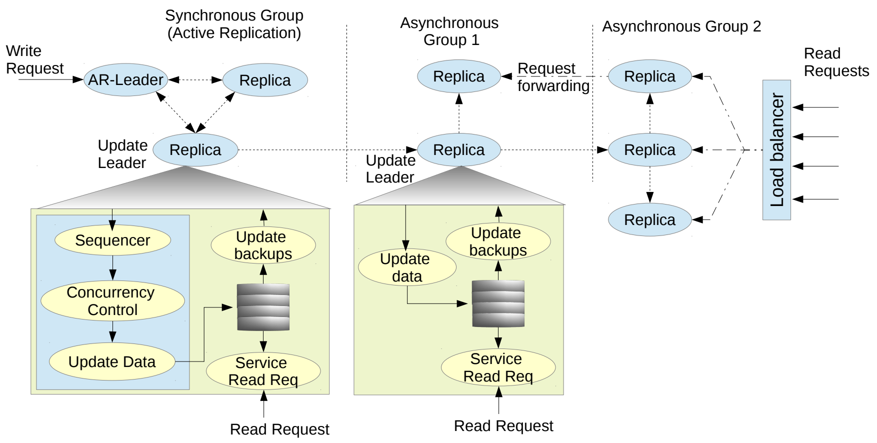


Figure 8.1: Dexter's architecture.

high-throughput of the ordering layer [8].

Recall that, in active replication, write transactions are executed in order. Therefore, the impact of transactions' conflicts is also minimal. In addition, each node is equipped with a multi-versioned concurrency control protocol for serving read-only transactions without incurring abort costs due to logical contention with concurrently executing write transactions [48, 54].

Dexter defines multiple asynchronous groups. Each of them is responsible for serving read-only transactions issued by clients. Write transactions are never delivered to asynchronous groups; instead, they get deferred updates from the synchronous group. The process of disseminating updates across the asynchronous groups is optimized for the size of the updates. If the synchronous group is able to process ≈100K write transactions per second, then each node produces a large amount of data to be propagated per second [1]. [2].

The synchronous group contains two leaders: one for ordering the write transaction requests received from the clients, called the *AR-leader*, and another for transferring the updates to the asynchronous groups, called the *update-leader*. The AR-leader [8, 60] is elected (in the presence of failures) using a classical leader election algorithm [60]; the update-leader is similarly elected. Moreover, we transition the role of the update-leader among the different nodes during the evolution of the system, because the process of pushing updates to asynchronous nodes requires computational resources that are also needed for executing the active replication protocol. If the same node ends up as both the update-leader and the AR-leader, then that node's performance will likely decrease. If the AR-leader slows down, it degrades the ordering layer's performance, and as a consequence, degrades the performance of write

[1]Consider an integer object of 4 bytes. Let transactions access two such objects. If 50% of the transactions access different objects, then a node generates 400K bytes of new data each second.

[2]As an example, if the transfer unit per transaction is 4 bytes then each second a node generates 400K bytes of new data to transfer.

transactions. Any other correct node within the synchronous group is a good candidate for being an update-leader, because every node has the entire shared data-set consistent and updated (due to the active replication protocol). For this reason, we exclude the AR-leader from the set of possible new update-leaders.

The performance of the update-leader could be affected during the processing of the deferred updates. However, ordering layers usually base their decision on the majority of the replies collected. During the time needed for deferring all the updates, the update-leader will unlikely end up in this majority. As a result, this likely will not affect the performance of the ordering protocol.

Each asynchronous group also elects an update-leader using the same algorithm as the synchronous group. This node is responsible for managing incoming updates.

Groups are logically organized as a chain. Let SG denote the synchronous group and AG_n denote the asynchronous group located at position n of the chain. The whole system can be logically viewed as a chain of updates e.g., $SG \rightarrow AG_1 \rightarrow AG_2 \rightarrow \ldots \rightarrow AG_n$. The synchronous group, through the update-leader, forwards the updates to the first asynchronous group (i.e., AG_1). AG_1's update-leader receives those updates and broadcasts them to other nodes in the same group. AG_1's update-leader also propagates updates to AG_2. Generalizing, the asynchronous group AG_i elects an update-leader that receives updates from group AG_{i-1} and forwards new updates to AG_{i+1}. We name each position in the chain as *level*: level 0 corresponds to SG; level i represents AG_i.

We define three parameters (also called system parameters in the rest of the chapter) that can trigger the event of forwarding updates from level 0 to level 1:

- δ, which represents the time between two updates. When the update-leader finishes pushing updates to the next level, it triggers the next update forwarding process after time δ;
- λ, which is the number of object updates seen by a replica since the last forwarding. It corresponds to the sum of the size of the write-sets of committed transactions (i.e., $\sum |Tx.writeset|$);
- Λ, which is the maximum number of updates observed by a replica per object.

If one of these parameters exceed a predefined threshold, the update-leader starts forwarding updates to level 1 of the chain (i.e., AG_1). The δ parameter is useful when the workload is uniform, which means that objects are almost updated with the same probability. However, considering only this parameter for triggering updates could be misleading if the workload becomes more write-intensive. In fact, committing a significantly higher number of write transactions results in several object versions in the SG, which also makes the objects stored in AG_1 much older (and less fresher) than expected. In this scenario, the λ parameter is fundamental. When λ becomes greater than a threshold, updates are computed and pushed to level 1 such that the desired freshness of objects stored at that level is still maintained. The Λ parameter is similar to λ but it is defined over each object. Λ is needed to cope with “hot spot” objects (i.e., objects with frequent updates). If the number of updates is less than λ, but the great majority of those affect few objects, a new forwarding process towards AG_1 is triggered (for the same reason as that with λ). All of these thresholds are application-specific and could be changed by the programmer according to application needs.

The update-leader of level 0 sends all the updates to the update-leader of level 1. It does not broadcast to all the nodes in level 1; otherwise, given the size of the data transfer, the increased network utilization could potentially degrade overall performance. The level 1's update-leader disseminates the collected updates from level 0 among its group's nodes and also propagates older updates to level 2.

This architecture is optimized for deployment scenarios where the network infrastructure between levels are configured such that sending an intra-level message does not interfere with other inter-level messages. However, the performance of our solution are equivalent to broadcasting updates from the update-leader to all the nodes in the next group with a completely shared network infrastructure.

When the level 0 pushes updates to level 1, it tags the message with the time elapsed, ϵ, since the previous transfer. This time is used by the update-leader of level 1 as a timeout for propagating those updates to level 2. This approach allows to reproduce the same schedule of updates triggered by SG. If the application workload is stable, then an update U_1, sent from SG to AG_1 at time T_1, will be propagated from AG_1 to AG_2 at time T_2=T_1 + δ. It will arrive at the third level at time T_3=T_2 + δ. In this scenario, $\epsilon = \delta$. However, if after sending U_1, SG's update-leader receives a number of write transaction requests such that either the λ or Λ threshold is exceeded, then SG propagates a new update U_2 to the first level of the chain. In this case, the elapsed time ϵ between U_1 and U_2 is smaller than δ. In order to avoid significantly stale object versions at subsequent levels, the update-leader of level 1 will only wait for ϵ, rather than δ.

Due to the active replication paradigm, the synchronized clock among nodes, and the assumption that any of the update intervals are larger than the nodes' clock synchronization skew, each node can calculate all the aforementioned parameters and end up with the same decision as the others (see Section 8.4.2). This is particularly important if the update-leader crashes; a new node is then easily elected as the next update-leader.

In order to exploit the proposed architecture, read-only transactions are delivered from application threads to the last level of the chain, and, according to the fulfillment of rules, they are either served from the outermost level, or sent to the previous level. Eventually, a read-only transaction is executed either at the level where all its applicable rules have been satisfied, or in the synchronous group. Within each level, read-only transactions are balanced on all the nodes of the group.

The proposed architecture allows fast execution of write transactions, minimizing their interference with read-only workloads, which are mostly processed externally at the other levels. Groups AG_1, ..., AG_n can be seen as an extension of SG, and they allow the system to scale up performance with size increases, exploiting application characteristics.

Clearly, this framework has limitations. For example, if the transactional application is mission-critical, then having only the synchronous group (and no asynchronous groups) is likely a better design choice. This is because, if the application is not able to exploit the asynchronous groups, then the overhead for checking the various rules and computing the additional parameters does not payoff.

8.3 Rule-based framework

Dexter provides a framework for defining application-specific rules such that the chain for executing read-only transactions, as described in Section 8.2, can be exploited. For easy programmability, Dexter minimizes programmer interventions and also provides specific APIs that enable specification of more complex requirements.

We define δ, λ, and Δ (described in Section 8.2) as system parameters.

A transaction can specify the desired execution level of the chain by using a version of the invoke API (described in Section 8.1 that explicitly provides the level (i.e., `invoke(type par1, type par2, ..., levelN)`). In addition, to be consistent with the classical active replication programming model [48, 80], the `invoke(type par1, type par2, ...)`) API of a read-only transaction delivers the transaction request to the synchronous group. Therefore, if the programmer does not specify any system parameters, Dexter will default to classical active replication, processing all transactions (read and write) in the synchronous group. Otherwise, there are two execution configurations that the programmer can use for exploiting different levels of object freshness: exclusively time-based (TB) or time/content-based (CB).

With the TB configuration, the time between pushing updates between levels (i.e., δ) is the only mandatory system parameter. At the application side, the API for invoking a read-only transaction is extended with a new parameter, called δ_{App}, that describes the maximum time since the objects accessed by the transaction has been updated. This way, the previous invoke API becomes `invoke(type par1, type par2, ...,` δ_{App}`)`. When a read-only transaction traverses the chain, δ_{App} is used for selecting the appropriate level (see Section 8.4.3 for complete discussion).

Dexter offers another execution configuration for inferring more complex rules based on both the content and the freshness of objects accessed. The framework allows to define groups of rules. Each group is identified by an ID such that a transaction can specify which group of rules should be applied while it is processed. To exploit this feature, the programmer should use another version of the invoke API that is overloaded with the corresponding group's ID.

As an example, let RG_1 and RG_2 be two groups of rules. Let Tp_1 and Tp_2 be two application transaction profiles. When the programmer needs to invoke Tp_1, she[3] can decide whether RG_1 or RG_2 or none of them can be applied. The same happens with Tp_2. In this case, the invoking command will change as Tp_2`(type par1, type par2, ..., `RG_1`)` or Tp_1`(type par1, type par2, ..., `RG_2`)` accordingly.

Each rule, named $\Re$, is classified as:

- time-based, called $\Re_T$; or
- content-based, called $\Re_C$.

$\Re$ can be applied to an entity Ψ that can either be an object field (e.g., `warehouse.zip_code`) or an object type (e.g., `warehouse`). $\Re$ is logically defined as a triple [Ψ, *expression*, *level*] where:

- Ψ is the entity (i.e., object field or type) accessed by the read-only transaction;
- *expression* is either an expression that can be evaluated using logical (*and*, *or*, *not*) and/or mathematical ($<, \leq, >, \geq, =$) operators, if $\Re_C$ is the rule's type; or it represents the maximum elapsed time since Ψ was refreshed, if $\Re_T$ is the rule's type; and
- *lev* is the level in the chain that can serve the read-only transaction.

Let T_R be the read-only transaction currently processing at node N_R. The semantics of a rule $\Re$ depends on the rule's type:

- Rule $\Re_C$. If the entity Ψ has a value in N_R that satisfies *expression*, then T_R is executed at the level *lev* in the chain.
- Rule $\Re_T$. If the last time that the entity Ψ has been refreshed is less than *expression*, then T_R is executed at the level *level* in the chain.

In other words, a rule represents a programmer's hint. By exploiting rules, a programmer provides a hint to Dexter on how to recognize if a read-only transaction can be executed at a certain level in the chain, according to application needs.

Rules belonging to the same group are evaluated as *or*. When a read-only transaction is processed, if at least one rule is not satisfied, the transaction is not executed at the current level and is forwarded to the previous level.

For the sake of simplicity and for faster processing of rules, we scope out equations involving multiple entities.

Rules are known to all the nodes in the system, but they do not apply to nodes of the synchronous group. Here, read-only transactions access the freshest data available in the system, thus there is no fall back plan.

[3]We assume the programmer is a woman.

8.4 Processing transactions in Dexter

We now describe the three main steps involved in transaction processing in Dexter. We first describe how transactions (raed-only and write) are processed in the synchronous group. We then discuss the mechanisms for propagating updates from the synchronous to asynchronous levels. Finally, the execution flow of read-only transactions in the asynchronous groups, as well as the mechanism for checking rules, are detailed.

8.4.1 Handling Transactions In The Synchronous Level

Write transactions are executed in the synchronous group according to the active replication paradigm. This paradigm requires two main building blocks for handling a write transaction. The first is a total order protocol (e.g., Atomic Broadcast [52]) that defines a global order among write transactions issued by clients. The second is a local concurrency control protocol responsible for processing those transactions according to the already defined order. This determinism on transaction processing is mandatory. Otherwise, nodes will end up in different states, violating the nature of state-machine replication and preventing the local execution of read-only transactions without global synchronization. Another advantage of deterministic, in-order commit is that, it ensures one-copy-serializability [7].

Application threads (i.e., clients) use the appropriate *invoke* API for invoking a write transaction (i.e., tagged with a flag signaling that the transaction is write). After that, a thread waits until the transaction is successfully processed by the replicated system and its outcome becomes available. Each client has a reference node for issuing requests. When that node becomes unreachable, or a timeout expires after the request's submission, the reference node is changed and the request is re-submitted to another node. Duplication of requests is handled by tagging messages with unique keys composed of the client ID and the local sequence number.

Dexter relies on the HiperTM [48] active replication system for processing transactions in the synchronous group. Within the AR paradigm, HiperTM speculatively processes transactions for maximizing the overlap between the global coordination of transactions and the local transaction execution. HiperTM uses Optimistic S-Paxos (or OS-Paxos), an implementation of optimistic atomic broadcast (OAB) [85, 52], built on top of S-Paxos [8], for defining the global order of write transactions. OAB enriches classical Atomic Broadcast with an additional delivery, called *optimistic delivery*, which is sent to nodes prior to the establishment of message's global order. This new delivery is used by local concurrency control to start transaction execution speculatively, while guessing the final commit order. If the guessed order matches the final order, the transaction is already totally (or partially) executed and can be committed. OS-Paxos shows good scalability (for up to 20 nodes) and high-performance [48]. The HiperTM implementation is open-source.

IIn HiperTM, each replica is equipped with a local speculative concurrency control protocol for executing and committing write transactions, enforcing the order notified by OS-Paxos. In order to overlap the transaction coordination phase with transaction execution, write transactions are processed speculatively, as soon as they are optimistically delivered, on a single thread. The reason for single-thread processing is to avoid the overhead for detecting and resolving conflicts according to the optimistic delivery order while transactions are executing. Additionally, no atomic operations are needed for managing locks on critical sections.

The speculative concurrency control uses a multi-versioned model, wherein an object version has two fields: *obj-timestamp*, which defines the time when the transaction that wrote the version committed; and *value*, which is the value of the object. Each shared object includes the last committed version and a list of previously committed versions.

Read-only transactions are not broadcast using the ordering layer, because they do not need to be totally ordered. When a client invokes a read-only transaction, it is locally delivered and executed in parallel to write transactions by a separate pool of threads. In order to support this parallel processing, we define a logical timestamp for each node, called *node-timestamp*, which represents a monotonically increasing integer, incremented each time a write transaction commits. When a write transaction commits, it increases the node-timestamp and tags the newly committed versions with this the increased node-timestamp.

When a read-only transaction performs its first operation, the node-timestamp becomes the transaction timestamp, called *transaction-timestamp*). Subsequent operations are processed according to this value: when an object is accessed, its list of committed versions is traversed in order to find the most recent version with a obj-timestamp lower or equal to the transaction-timestamp.

One synchronization point is present between write and read-only transactions, i.e., the list of committed versions is updated when a transaction commits. On one hand, when the read-only workload is intensive, write transactions get delayed because of multiple threads traversing the same concurrent data-structure. On the other hand, with a write intensive workload where the contention level is not minimal, read-only transactions suffer from write load.

8.4.2 Propagating updates

The mechanism for propagating updates (forwarding hereafter) from level 0 to the first level of the chain is computed asynchronously by the update-leader. There are two modes of forwarding. In the first, the update-leader creates a batch of all the latest versions of written (shared) objects since the last forwarding. In the second, the update-leader prepares a batch of all the transactional requests, rather than objects, received since the last forwarding. The decision about adopting the first or the second solution is a trade-off.

On the one hand, with the first mode, there is no duplication of objects that were written multiple times since the last forwarding, but usually, the size of shared objects is large. On the other hand, batching requests means that, transferring a smaller batch, at the cost of re-executing transactions, including those that have write-write conflicts, results in overwritten objects. This trade-off can be explored depending on the application workload characteristics. If the write transactions are mostly conflicting on a restricted data-set, then forwarding objects is the best solution. On the contrary, workloads with very large data-sets and few conflicts are good candidates for forwarding transactions.

As already discussed in Section 8.2, the update-leader decides when updates are pushed according to the system's execution configuration (i.e., TB or CB) and the setting of system parameters (i.e., δ, λ, Δ).

When the system works in exclusively time-based mode (i.e., TB), the only system parameter that determines when to begin pushing the updates from the synchronous group to level 1 is δ. When a time δ since the last forwarding is elapsed, the update-leader collects all the issued requests (or all the latest versions of written objects) and creates the forwarding batch. This batch is sent to the update-leader of level 1[4]. Each batch is tagged with ϵ, which represents the time elapsed since the last forwarding. In the TB mode, ϵ always equals δ.

In the time/content-based mode, the forwarding mechanism is more complex. Here, two other system parameters are also taken into account. In this mode, δ represents the maximum time between two forwardings. If higher than λ number of objects has been written since the last forwarding, the update-leader will not wait δ, but it will immediately trigger a new forwarding. The same happens if the update-leader's concurrency control protocol observes higher than Δ number of writes on the same type of object. As a consequence, when the system is in the CB mode, ϵ associated with each batch can differ from δ.

Whichever mode the system is configured, the local concurrency control protocol needs to be adapted for tracking object modifications. However, this overhead can seriously slow-down the execution of the committing thread of write transactions. This thread should avoid any overhead because it represents the concurrency control's critical path. For this reason, we rely on an additional thread, the *forwarding thread*, for monitoring updated objects, which is not synchronized with the committing thread.

Three data structures are needed for implementing the monitoring system. One is a hash map, called *objTrack*, where key is the ID of the written object and value is the reference to the last written version of that object since the last forwarding. The second one is also a hash map, called *tyTrack*, where key is the type of the written object and value is a counter that tracks the number of writes occurred on that type of object since the last forwarding. The third is a counter that tracks the number of objects written since the last forwarding.

The concurrency control protocol reserves a thread for committing transactions in order.

[4]We avoid low-level mechanisms for recognizing and caching next level's update-leader. Briefly, if the next level update-leader has changed since the last forwarding, the old update-leader notifies the change.

The transactions are maintained in a queue with the committing thread as the queue's server. When a transaction is committed, it is inserted into another queue that is managed by the forwarding thread. As an example, let T be a committed transaction accessing two objects (x,y) of the same type (T) (i.e., T=$\{O_x^T$, $O_y^T\}$). The forwarding thread *i)* updates two buckets of objTrack (keys=x,y) with references to O_x and O_y; *ii)* increases the counter associated with the key T of tyTrack; and *iii)* increases the counter for tracking new writes by 2. After that, the forwarding thread simply checks whether any of the system parameters' threshold has been exceeded, and if so, creates the batch to forward.

The aforementioned process can be done only in a single thread because, otherwise, the sequence of updates is lost and the references to the last committed version of the objectS can point to a wrong object.

The forwarding thread runs on all the nodes of the synchronous group, and not just on the update-leader. This is because, if the update-leader crashes, another node can take over (i.e., the new update-leader) and correctly complete the forwarding process. (Recall that node clocks are synchronized, and that we assume that δ is much higher than the maximum clock skew).

When the update-leader of level 2 receives the updates (either objects or transactions), it broadcasts them to other nodes at the same level. The update-leader also schedules a forwarding of updates to level 3 after the time ϵ specified in the batch just received from level 1 elapses. This way, batches are propagated to other asynchronous levels.

When a node receives a batch, it writes the new objects or executes the new requests. In the former case, all the objects are first made permanent on the shared-data. Only after all the objects are installed, the node timestamp is updated with the maximum timestamp associated with the new objects. In the latter case, transactions are executed in the order defined by the batch (that corresponds to the execution order in the synchronous group), and the node timestamp is updated only when the last transaction commits. As a result, only the read-only transactions that start after updating the node timestamp are allowed to access those versions.

8.4.3 Handling read-only transactions

Read-only transactions delivered directly to the synchronous group are processed as previously described in Section 8.4.1. They do not need to check rules while processing because the synchronous group already maintains the freshest data available. The asynchronous groups also execute read-only transactions in the same way, but, in addition, they pay the cost for checking rules.

Dexter also allows a read-only transaction to execute directly at a level specified by the programmer. In this case, the transaction's behavior is similar to the previous case. The concurrency control protocol does not check any rules while processing the transaction; the transaction simply accesses the most recent versions of the objects stored at that level.

In contrast, when a read-only transaction is not bound to a specific level, it is delivered to the last level in the chain of asynchronous groups. When a transaction arrives at a level, it is delivered to a node through a router, which implements the classical round-robin-based load-balancing approach. The core idea is to run the read-only transaction in the last level of the chain. If the transaction satisfies all the rules, then the node at that level replies to the client; otherwise, the transaction is forwarded back to the previous level. The process ends when the transaction detects that it cannot run at level 1, and is finally forwarded to the synchronous group where it will be successfully processed.

It is unlikely that a read-only transaction will traverse the entire chain. If it does, then it means that either the application is not designed for exploiting Dexter's architecture, or the system parameters have been misconfigured for the workload at hand (e.g., δ is much smaller than what is necessary).

If the system is configured in exclusively time-based mode, then a read-only transaction starts at the latest level and traverses the chain, until it reaches the level whose time of last forwarding is lower than δ_{App}. That level has been last updated, which is prior to the maximum time required by the transaction (i.e., δ_{App}) and therefore can serve the transaction. Similar to the previous scenarios, this transaction also does not need to check for the satisfaction of all the rules, as it only needs to check for the last time when the node, and thus the level, has been updated.

When the specification and enforcement of rules is enabled in the system (i.e., time/content-based mode), then read-only transactions will incur some overhead for rule checking. Rules are stored on all the nodes as a hash map, called *ruleMap*, where key is the type of the object and value is the list of rules associated with that object type. As an example, if a programmer wants to infer a rule on the type `warehouse` in the TPC-C [20] benchmark, then the rule is stored at key=`warehouse` in the hash map. Similarly, if the programmer wants to infer a rule on the field `zip_code` of a `warehouse` (i.e., `warehouse.zip_code`), then the rule will be appended to the same entry of the hash map as before, because both affect the type of the object `warehouse`.

A read-only transaction that runs when the rules are enabled must check whether its rules are satisfied for each accessed object. When a read operation completes, the concurrency control protocol looks-up in ruleMap to check whether rules have been defined for the entry associated with the type of the accessed object. If rules exist, the protocol checks whether there exists at least one rule that is not satisfied. If a rule is not satisfied, the read-only transaction is aborted and forwarded to the previous level. The checking of rules is done at object encounter-time instead of at commit time, because, this way, part of the useless computation is saved – e.g., if one rule of the first accessed object is not satisfied, it is useless to continue to process the transaction.

The overhead of checking rules on the transaction's critical path is not minimal when the number of rules per object type significantly increases. This overhead can be mitigated by deploying more nodes at each level. However, the rules must be defined (by the programmer) with the goal of improving performance, and not degrading it.

8.5 Correctness

Dexter guarantees 1-copy serializability [7] and opacity [34]. This is simply because, all modifications made by Dexter do not violate these same properties of HyperTM, which is the underlying active replication protocol used by Dexter for processing read and write transactions. We demonstrate our claim by analyzing the behavior of read-only and write transactions executed at the synchronous level, as well as, at the asynchronous levels. (We skip a formal proof due to space constraints.)

In the synchronous group, read-only and write transactions follow the same protocol rules as HiperTM. 1-copy serializability is easily ensured because each node commits the same set of write transactions in the same order notified by the optimistic atomic broadcast layer. In addition, those transactions are also processed and committed in a single thread, without concurrency. Read-only transactions activated on different nodes cannot observe any two write transactions that are serialized differently on those nodes, because the ordering layer provides a total order among conflicting and non-conflicting transactions, without any discrimination.

HiperTM ensures opacity [34] because, in addition to single thread processing and using a predefined total order of write transactions, read-only transactions perform their operations according to the snapshot of the node-timestamp taken when the transactions begin. During their execution, they access only the committed versions written by transactions with the highest object-timestamp that is lower or equal to the transaction-timestamp.

In any asynchronous group, updates are applied while read-only transactions are being processed, but the node-timestamp is updated with the maximum value among all the new object-timestamps only at the end of the updates. This means that, those objects are invisible for ongoing and new read-only transactions started before the modification of the node-timestamp. Only when the updates are finally set, the newly activated read-only transactions will be able to observe these new objects.

The only difference between the processing of read-only transactions in the synchronous group and an asynchronous group is abort-freedom. When executed at level 0, read-only transactions cannot abort because the subset of all the accessible object versions is fixed when the transactions define their transaction-timestamp, and it cannot change. In contrast, in the asynchronous groups, they can be aborted due to rule checking. Such aborts are not due to contention, but can be seen as due to external factors. Thus, in the asynchronous levels, the abort-freedom property cannot be ensured.

However, we can provide an upper bound on the number of retries of read-only transactions. Since a read-only transaction aborts not because of contention but because of violation of rules, it can restart only as many times as the number of asynchronous groups. After that, the transaction will be successfully served in the synchronous group.

8.6 Evaluation

We implemented Dexter in java, extending the implementation of HiperTM [48], publicly available as open-source project. We used the *PRObE* testbed [32], a public cluster that is available for evaluating systems research. Our experiments were conducted using 48 nodes in the cluster. Each node is a physical machine equipped with 8 cores. The network connection is a dedicated InfiniBand 40Gbps. As benchmarks we use two typical applications for assessing performance of transactional systems such as Bank benchmark and TPC-C [20]. The former mimics the operations of a bank monetary application; the latter is a well known benchmark that is representative of on-line transaction processing workloads.

Our architecture is composed of the synchronous group with 20 nodes running HiperTM, and 3 asynchronous groups, each with 8 nodes, for serving read-only transactions with different levels of objects' freshness. For the purpose of these experiments, we configured Dexter for forwarding transactions instead of objects.

All the results reported are the average of 5 samples.

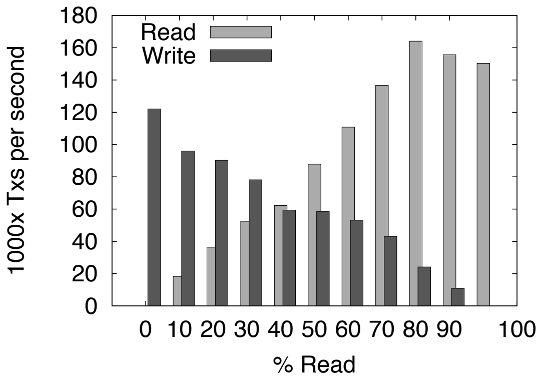


Figure 8.2: Throughput with Bank and different % of read-only transactions at synchronous node.

The first part of our experimental study regards the impact of read-only workload on write workload in a node deployed at the synchronous level. In Figures 8.2 and 8.3 we report the performance of read-only and write transactions running Bank and TPC-C respectively. In these plots we vary the percentage of read-only transactions (stock-level and order-status) while write transactions are continuously injected in the system. In Bank, the performance of write transactions drop from ≈120K to ≈60K when the read-only are 40%. This is because of contention on the shared lists for storing objects versions[5]. Similar trend is for TPC-C but this benchmark is much more processing-intensive than Bank, thus the general throughput of read-only transactions is lower than Bank. This is why the performance improvement of read-only transactions between having 50% of write transactions and 0% is around 35% while in Bank it is 77%. These results confirm the need for offloading the nodes of the synchronous group from the burden of processing all the read-only transactions in the system. The cost to pay is reducing the performance of write transactions.

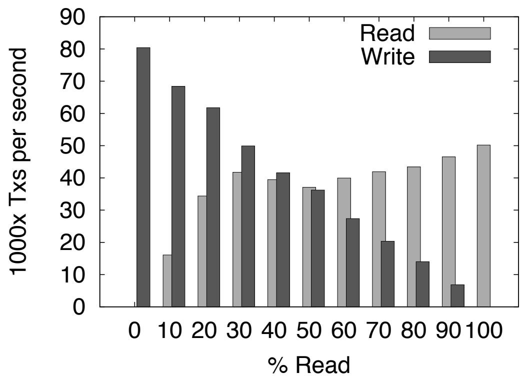


Figure 8.3: Throughput with TPC-C and different % of read-only transactions at synchronous node.

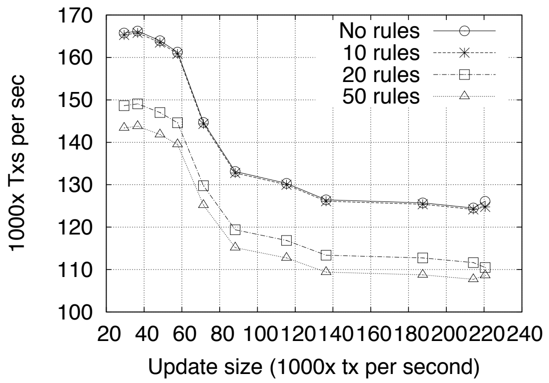


Figure 8.4: Throughput of read-only transactions with Bank, varying the update size at asynchronous node.

We now test the performance of an asynchronous node deployed in a generic level of the chain, when updates are pushed from the previous level. The amount of work to perform depends on the write workload of the system. For this reason, in these experiments we measured the throughput of read-only transactions (the only type of transaction that such

[5]HiperTM uses concurrent Skip-List for this purpose.

node is allowed to process) as a function of the size of update messages. This size is reported in terms of transactions per second. In case the update time since the last forwarding is δ, then the total number of transactions to apply will be δ times the throughput reported in the x-axes.

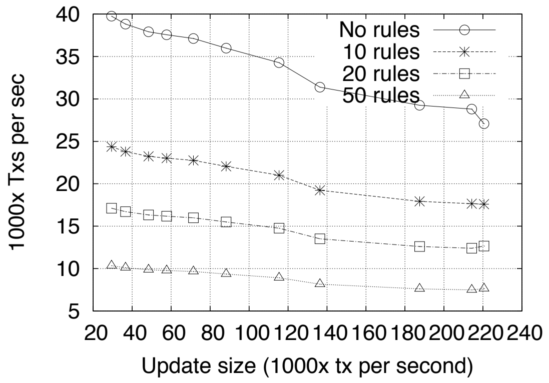


Figure 8.5: Throughput of read-only transactions with TPC-C, varying the update size at asynchronous node.

Figures 8.4 and 8.5 reports the results for Bank and TPC-C. Each plot shows 4 lines corresponding to the number of rules per object that the node checks while processing the read-only transaction. We tested with {0,10,20,50} where 0 represents the execution in purely time-based and the others emulate different configurations. Clearly checking 50 rules per object degrades significantly the performance, especially in TPC-C where transactions are long. Interesting, using Bank, the performance of 0 rules and 10 rules are very similar. The reason is related to the amount of type of objects in the system. Bank has very few different types of object while TPC-C is more complex.

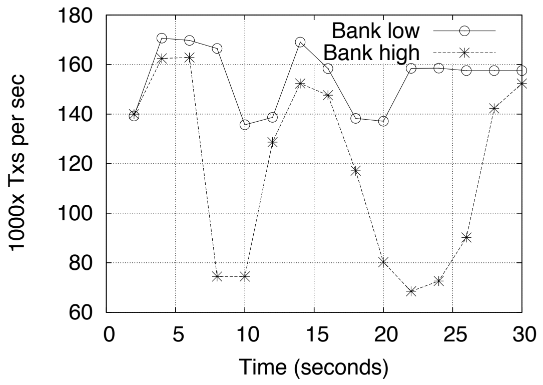


Figure 8.6: Throughout of read transactions at asynchronous node, pushing updates every 12 seconds.

In Figure 8.6, we report the throughput (read-only transactions) of an asynchronous node as function of time. We report only Bank benchmark because we found the same behavior in TPC-C. Updates are pushed every 12 seconds. For each benchmark we have two configurations: low and high. Low means that updates contain around 80K transaction per second, while for high is 200K. Each object type has 10 rules associated. The results clearly reveal how the performance of asynchronous nodes is affected by installing updates from previous levels. However, performance is still reasonably high even when updates arrive.

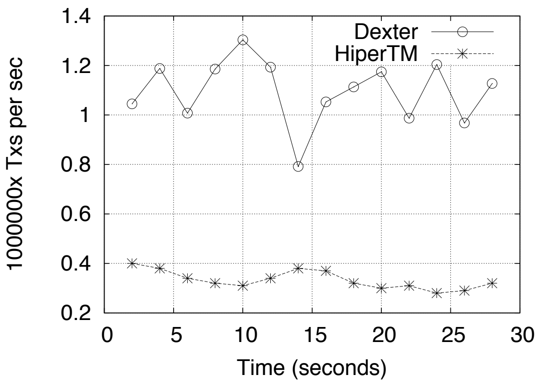


Figure 8.7: Throughput of Dexter with 48 nodes and 3 levels in the chain using Bank.

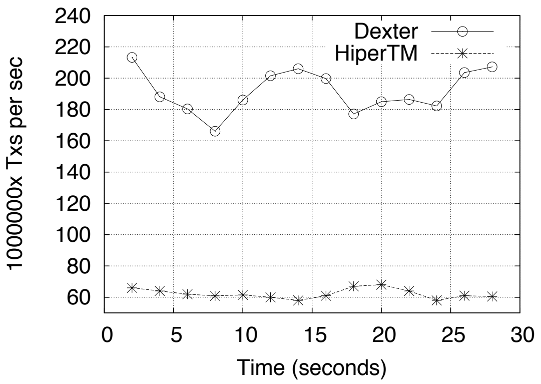


Figure 8.8: Throughput of Dexter with 48 nodes and 3 levels in the chain using TPC-C.

Finally we measure the performance of the entire system, and compare it with the performance of HiperTM. As already said, we ran HiperTM on our test-bed and its performance does not scale after 20 nodes. After that, the load of the ordering layer is so high that write transactions execution time is delayed significantly. In this scenario, read-only transactions could access very old versions of data because the progresso of the whole system is hampered. Dexter, instead, exploiting application rules, can be configured for running with much higher number of nodes. In this experiments we deployed Dexter with a chain of 3 levels, each with 8 nodes, in addition to the synchronous level that is composed of 20 nodes. Read-only transactions are injected continuously in the system. The parameter δ is set as 5 seconds and the system configuration model is time/content-based. For each object type we define 10 rules to check while processing the read-only transaction. These rules are logical expressions based on the values of the object fields. We designed the rules such that transactions can be balanced among levels of the chain. Figure 8.7 shows the results. As expected, the speed-up is reasonable high, up to 1.3 Million transactions per second served, with improvement up to 2× with respect to HiperTM. Clearly, the major contribution of this really high throughput is made by read-only transactions, while write transactions sustain their performance around 100K.

Figure 8.8 shows TPC-C performance under the same configuration as Bank. Here absolute numbers are lower than before but the speed-up of Dexter against HiperTM is still up to 2.1×.

8.7 Summary

Active replication is a powerful technique for obtaining high transactional performance with full-failure masking. However, it suffers from poor scalability, as it relies on a consensus-based algorithm for ensuring global consistency of the replicated state.

Our work shows that, it is possible to overcome this scalability bottleneck by exploiting application characteristics. Our key insight is that, not all read-only transactions need to access the latest data; for many, “sufficiently fresh” data suffice. By enabling the application to specify the level of data freshness that it needs, the shared data-set can be maintained at different levels of freshness, avoiding costly global synchronization. Read-only transactions access progressively fresh data, scaling up performance. The cost of enforcing the application’s data freshness rules can be mitigated through careful design and implementation choices.

In some sense, our result can be viewed as a generalization of multi-version concurrency control (CC), where read-only transactions commit in “the past” by reading older versions. Multi-version CC does not specify data freshness (we do), but mostly guarantees abort-freedom for read-only workloads (we do not, though retries are bounded). Taken together, our work illustrates how scalability can be achieved by exploiting multi-version CC’s insight (of reading old data), with full-failure masking.

Chapter 9

Conclusions

In this dissertation proposal, we have made several contributions aimed at improving the performance of replicated transactional systems. We analysed the different aspects of such systems e.g., ordering layer, transaction execution model, and application requirement characteristics etc., and tried to find the opportunities to optimize them in different ways. With HiperTM, we exploited the time between the client request is known to replicas and the time its order is finalized to process it speculatively. In Archie, we further optimized this parameter and improved the concurrent execution of requests to get high performance. With Caesar, we designed a new transaction ordering protocol to benefit from relaxed lock-free execution for non-conflicting transactions. Lastly, in Dexter we exploit the application characteristics to scale the read-only loads, which usually comprise the majority of transactional load.

At its core, HiperTM shows that optimism pays off: speculative transaction execution, started as soon as transactions are optimistically delivered, allows hiding the total ordering latency, and yields performance gain. Experimental evaluation and comparison with PaxosSTM [54] revealed the performance benefits achieved from *optimistic delivery* and serial processing over DUR model, especially in high contention workloads. While DUR model of PaxosSTM suffers from remote aborts due to conflicting transactions, HiperTM achieves zero aborts when the leader is not faulty or not suspected.

We presented Archie, which further optimizes the *optimistic delivery*, exploits parallelism for concurrent transactions and alleviates the transaction's critical path by eliminating non-trivial operations performed after the notification of the final order. Exhaustive evaluation against multiple competitors [54, 48] on well known benchmarks [20, 13] showed that Archie outperforms its competitors specially in medium to low conflict scenarios. Archie limits the number of visible conflicts by spawning a predefined number of worker threads and achieves higher throughput and lower abort rates for all benchmarks.

In Caesar, we tackle the problems associated with single sequencer based total order and propose a novel message ordering protocol which enables high performance and scalability of strongly consistent transactional systems. Caesar allows lock-free execution of non-conflicting transactions. Experimental study revealed that transactional systems based on competitors [60, 59, 78] do not scale as the number of replicas increase beyond 19.

Active replication faces a scalability limit as it relies on a consensus-based algorithm for ensuring global consistency of the replicated state. In Dexter, we overcome this scalability bottleneck by exploiting application characteristics. Our key insight is that, not all read-only transactions need to access the latest data; for many, "sufficiently fresh" data suffice. We provide the application an ability to specify the level of data freshness that it needs, while shared data-set is maintained at different levels of freshness. Read-only transactions access progressively fresh data, scaling up performance. By evaluation study we show that exploiting staleness of shared data, system throughput could scale further together with increasing system size.

9.1 Proposed Post-Prelim Work

In the pre-prelim work we have focused mainly on solving the problems in building high performance fault-tolerant DER systems. Moving forward we plan to apply similar optimizations in DUR systems. In addition to it, considering the efficient transaction execution model of HTM, we plan to incorporate it in our future work for building replicated transactional systems. We elaborate our future proposals in the following sections.

9.1.1 Ordering Transactions in DUR

The *deferred update replication* (DUR) [84] is a well-known scheme where transactions execute locally and their commit phase (including the transaction validation procedure) is deferred until a total order [60] among all nodes is established. This total order is required because it imposes a common serialization order among all transactions in the system, which is used to verify the global correctness of transactions' execution. In fact before commit, each transaction has to undergo a *certification phase* where the transaction validates the consistency of its read operations, performed during the execution, against write operations done by other concurrent transactions in the system. To accomplish this task, a total order is leveraged so that all nodes know a unique order to follow while performing the certification. If the snapshot observed is still consistent, then the transaction can safely commit by updating the shared state with its written objects. The sequence of commit necessarily matches the global total order.

DUR-based protocols find their best scenario in terms of performance when transactions running on different nodes (remote transactions) rarely conflict with each other (e.g., well-partitioned accesses across nodes). This way an executed transaction, which is waiting for its global certification, is likely to commit because all its read operations cannot be invalidated by remote transactions due to the well-partitioned accesses. In such an execution environment, the DUR scheme allows the (massive) parallelization of application threads running locally at each node, therefore ensuring high performance. However, even if the application exposes well-partitioned accesses across different nodes, the local parallelism is effectively exploited only in case local concurrent transactions hardly request same objects.

As an example, consider TPC-C [20], the classical transactional benchmark widely used for evaluating distributed synchronization protocols. Most TPC-C transactions access a `warehouse` before performing other operations. The usual deployment of TPC-C is to pin one (or a set of) `warehouse` to each node and let transactions generated on that node to likely request that warehouse. This configuration, which is representative of several applications with well-partitioned accesses, matches DUR's needs in terms of few remote aborts, but it also reduces the parallelism of local application threads due to conflicts. As a result, even if application threads do not suffer remote aborts, they are still highly prone to aborts due to contention within local application threads.

We plan to propose solution for reducing the local conflicts in DUR system by introducing the notion of an order for local transactions and then by propagating the state changes made by one transaction execution to another along the chain of subsequent conflicting transactions, according to the defined order. It should be noted that this order is not necessarily known (or pre-determined) before starting the transaction execution, rather it could be determined while transactions are executing taking into account their actual conflicts. Later transactions from one node would be submitted to the global certification layer in the same order as their optimistic execution order. This way, the local transaction ordering would be always compliant with the final commit order, thereby resulting in fewer aborts.

9.1.2 Executing Distributed Transactions on HTM

With our previous works i.e., HiperTM and Archie, we used software transactional memory for local transactional execution. Though this approach resulted in high performance, maintaining each component of STM e.g., read-set, write-set, and multi-versioning etc. has non-negligible overheads. On the other hand, Hardware Transaction Memory (HTM) is well known to yield high performance as it exploits the underlying hardware capabilities to manage contention while providing transactional (ACI) properties similar to STM. There is also a renewed interest in HTM in the light of recent transactional synchronization extensions of Intel®Haswell chips.

HTM transactions are best effort transaction i.e. there is no guarantee that HTM transactions will eventually commit. For example, a transaction on Intel's *Haswell* chip could fail due to other reasons such as capacity failure, page faults, system calls etc., apart from conflicts with other concurrent HTM transactions. As a result, usually HTM transactions have a fall-back software path, which helps to commit these transactions in software. Even with this added complexity and overhead, HTM transactions are found to be at least at par with their software counterparts. On the other hand, in favourable conditions, HTM yields much higher performance than STM.

Considering the performance benefits, we propose to incorporate HTM transactions in *total order* based DER systems. Since HTM transactions do not have any notion of order among transactions, we plan to introduce an order aware transaction execution engine above HTM to ensure that the HTM transactions follow the *total order*. This work will also include a fall-back software path for the transaction which could not commit in HTM due to repetitive failures due to variety of reasons. We expect that HTM transactions together with order aware execution engine will further boost the performance of replicated transactional systems.

9.1.3 Multi-leader Partial Order Ring-based Transactional System

Reaching a fast decision in a consensus protocol is highly desirable, as it reduces the latency perceived by clients. The same thought was the motivation for designing Fast Paxos [64] which ensures two communication steps to order a client request if discordant states of the execution are not received by a quorum of nodes. Even though Fast Paxos gives an optimal number of communication steps for such cases, it is dependent on an elected leader to resolve the discordant states (if they happen). On the other hand, Generelized Paxos [59] orders transactions according to their actual conflicts but it also relies on an elected leader for resolving discordant states. With Caesar, we designed a multi-leader partial order solution which orders transactions according to their actual conflicts and solves the problem of bottleneck created by a single leader. But even Caesar is prone to additional communication steps when at least a node in the quorum observes discordant state.

Reducing the number of communication steps even under possibility of discordant states observed by different nodes in a quorum is a challenging problem. As ring network topology is found to provide optimal performance [36], it becomes our natural choice for designing a high performance ordering protocol. In summary, as our last contribution to this thesis, we propose to design a multi-leader partial order protocol which will ensure fast decision for every transaction will follow a fast decision for finalizing its order, thereby improving the latency and performance of transactional systems.

Bibliography

[1] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* PhD thesis, Massachusetts Institute of Technology, 1999. AAI0800775.

[2] Amazon Inc. Elastic Compute Cloud, November 2008. URL: http://aws.amazon.com/ec2.

[3] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, January 1990.

[4] Ken Arnold, Robert Scheifler, et al. *Jini Specification.* Addison-Wesley, 1999.

[5] João Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe, and Rachid Guerraoui. Unifying thread-level speculation and transactional memory. In *Middleware*, 2012.

[6] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[8] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. S-Paxos: Offloading the Leader for High Throughput State Machine Replication. In *SRDS*, 2012.

[9] Nathan G. Bronson, Hassan Chafi, and Kunle Olukotun. Ccstm: A library-based stm for scala. Scala Days, April 2010.

[10] Joao Cachopo and Antonio Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, December 2006.

[11] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 225–236, New York, NY, USA, 2013. ACM.

[12] Berkant Barla Cambazoglu, Flavio P. Junqueira, Vassilis Plachouras, Scott Banachowski, Baoqiu Cui, Swee Lim, and Bill Bridge. A refreshing perspective of search engine caching. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 181–190, New York, NY, USA, 2010. ACM.

[13] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08*, 2008.

[14] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. Scert: Speculative certification in replicated software transactional memories. In *SYSTOR '11*, 2011.

[15] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[16] Bernadette Charron-Bost and Andre Schiper. Improving fast paxos: Being optimistic with no overhead. In *PRDC*, pages 287–295, 2006.

[17] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 29(2):6–16, March 2009.

[18] E. Cohen and H. Kaplan. Refreshment policies for web content caches. In *IEEE INFOCOM '01*, 2001.

[19] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luis Rodrigues. D2STM: Dependable distributed software transactional memory. PRDC, 2009.

[20] TPC Council. TPC-C benchmark. 2010.

[21] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference '12*, 2012.

[22] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[23] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[24] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4), 2004.

[25] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36, 2004.

[26] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In Shlomi Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*. Springer, 2006.

[27] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. PLDI '09.

[28] Richard Ekwall and Andr Schiper. Solving atomic broadcast with indirect consensus. In *In IEEE International Conference on Dependable Systems and Networks (DSN*, 2006.

[29] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[30] Roy Friedman and Robbert van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *HPDC*, pages 233–242, 1997.

[31] Stéphane Gançarski, Hubert Naacke, Esther Pacitti, and Patrick Valduriez. The leganet system: Freshness-aware transaction routing in a database cluster. *Inf. Syst.*, 32(2):320–343, April 2007.

[32] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research. volume 38, June 2013.

[33] Google. Google Cloud Platform, 2014. URL: https://cloud.google.com.

[34] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPOPP '08*, 2008.

[35] Rachid Guerraoui and Michal Kapalka. The semantics of progress in lock-based transactional memory. In *POPL*, pages 404–415, 2009.

[36] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst.*, 28(2):5:1–5:32, July 2010.

[37] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., 2006.

[38] Rachid Guerraoui and André Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1-2):297–316, 2001.

[39] Nicolas Guillaume. For google, 400ms of increased page load time, results in 0.44% lost search sessions, February 2013. Available at `http://www.cedexis.com/blog/for-google-400ms-of-increased-page-load-time-results-in-044-lost-search-sessions/`.

[40] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: replication at the speed of multi-core. In *EuroSys*, pages 11:1–11:14. ACM, 2014.

[41] James Hamilton. The cost of latency, October 2009. Available at `http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx`.

[42] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[43] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[44] Maurice Herlihy. Technical perspective - highly concurrent data structures. *Commun. ACM*, 52(5):99, 2009.

[45] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06*. ACM, 2006.

[46] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[47] Sachin Hirve, Roberto Palmieri, and Binoy Ravindran. Archie: A speculative replicated transactional system. In *Middleware*, 2014.

[48] Sachin Hirve, Roberto Palmieri, and Binoy Ravindran. HiperTM: High Performance, Fault-Tolerant Transactional Memory. In *ICDCN*, 2014.

[49] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 0:245–256, 2011.

[50] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: execute-verify replication for multi-core servers. In *OSDI '12*, 2012.

[51] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *VLDB 2000*, 2000.

[52] Bettina Kemme, Fernando Pedone, Gustavo Alonso, Andre Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE TKDE*, 15(4), 2003.

[53] T. Kobus, T. Kokocinski, and P.T. Wojciechowski. Practical considerations of distributed STM systems development (abstract). In *WDTM '12*, 2012.

[54] Tadeusz Kobus, Maciej Kokocinski, and Pawel T. Wojciechowski. Paxos STM. TR-ITSOA-OB2-1-PR-10-4. Technical report, Instytut Informatyki, Politechnika Poznanska., 2010.

[55] Tadeusz Kobus, Maciej Kokocinski, and Pawel T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *ICDCS*, pages 286–296. IEEE, 2013.

[56] Tadeusz Kobus, Maciej Kokocinski, and Pawe T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *ICDCS*, 2013.

[57] Jan Kończak, Nuno Santos, Tomasz urkowski, Pawe T. Wojciechowski, and Andr Schiper. JPaxos: State machine replication based on the Paxos protocol. Technical report, EPFL, 2011.

[58] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[59] Leslie Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[60] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, pages 133–169, 1998.

[61] Leslie Lamport. Future directions in distributed computing. chapter Lower Bounds for Asynchronous Consensus, pages 22–23. 2003.

[62] Leslie Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[63] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[64] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[65] Leslie Lamport and Mike Massa. Cheap paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 307–, Washington, DC, USA, 2004. IEEE Computer Society.

[66] Wen-Syan Li, Oliver Po, Wang-Pin Hsiung, K. Selçuk Candan, and Divyakant Agrawal. Freshness-driven adaptive caching for dynamic content web sites. *Data Knowl. Eng.*, pages 269–296, November 2003.

[67] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. In *PODC '91*, pages 1–9, 1991.

[68] Li Lu and Michael L. Scott. Generic multiversion stm. In Yehuda Afek, editor, *DISC*, volume 8205 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 2013.

[69] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In *OSDI*, pages 369–384, 2008.

[70] P. J. Marandi, Benevides Bezerra, and Fernando Pedone. Rethinking state-machine replication for parallelism. In *ICDCS*, 2014.

[71] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. High performance state-machine replication. In *DSN*, pages 454–465, 2011.

[72] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. High performance state-machine replication. In *DSN*, pages 454–465, 2011.

[73] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In *DSN*, 2012.

[74] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *DSN*, 2010.

[75] Richard Martin. Wall street's quest to process data at the speed of light, April 2007. Available at `http://www.informationweek.com/wall-streets-quest-to-process-data-at-the-speed-of-light/d/d-id/1054287?`

[76] D.L. Mills. Internet time synchronization: the network time protocol. *Communications, IEEE Transactions on*, 39(10):1482–1493, 1991.

[77] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *in HPCA*, pages 254–265, 2006.

[78] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *SOSP*, 2013.

[79] Roberto Palmieri, Francesco Quaglia, and Paolo Romano. AGGRO: Boosting STM replication via aggressively optimistic transaction processing. In *NCA '10*, 2010.

[80] Roberto Palmieri, Francesco Quaglia, and Paolo Romano. OSARE: Opportunistic speculation in actively replicated transactional systems. In *SRDS*, 2011.

[81] Ccile Le Pape, Stphane Ganarski, and Patrick Valduriez. Refresco: Improving query performance through freshness control in a database cluster. In Jacques Le Maitre, editor, *BDA*, pages 153–173, 2004.

[82] Marta Patino-Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4), 2005.

[83] Fernando Pedone and Svend Frølund. Pronto: High availability for standard off-the-shelf databases. *J. Parallel Distrib. Comput.*, 68(2), 2008.

[84] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, July 2003.

[85] Fernando Pedone and André Schiper. Optimistic atomic broadcast. In *DISC*, 1998.

[86] Fernando Pedone and André Schiper. Generic broadcast. In *DISC*, 1999.

[87] S. Peluso, P. Romano, and F. Quaglia. SCORe: A scalable one-copy serializable partial replication protocol. In *Middleware*, 2012.

[88] Sebastiano Peluso, Joao Fernandes, Paolo Romano, Francesco Quaglia, and Luís Rodrigues. SPECULA: Speculative replication of software transactional memory. In *SRDS '12*, 2012.

[89] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[90] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 5–17, New York, NY, USA, 2002. ACM.

[91] Red Hat. Open Shift, 2013. URL: https://www.openshift.com.

[92] J. Reinders. Transactional synchronization in Haswell. 2013.

[93] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *DISC '06*, 2006.

[94] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *In Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT '06)*, June 2006.

[95] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 221–228, New York, NY, USA, 2007. ACM.

[96] Paolo Romano, Roberto Palmieri, Francesco Quaglia, Nuno Carvalho, and Luís Rodrigues. Brief announcement: on speculative replication of transactional systems. In *SPAA*, pages 69–71, 2010.

[97] Paolo Romano, Roberto Palmieri, Francesco Quaglia, Nuno Carvalho, and Luís Rodrigues. An optimal speculative transactional replication protocol. In *ISPA*, pages 449–457, 2010.

[98] Nuno Santos and André Schiper. Tuning paxos for high-throughput with batching and pipelining. In *ICDCN*, 2012.

[99] Nuno Santos and André Schiper. Optimizing paxos with batching and pipelining. *Theor. Comput. Sci.*, 496:170–183, 2013.

[100] N. Schiper, P. Sutra, and F. Pedone. P-store:genuine partial replication in WAN. In *SRDS*, 2010.

[101] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[102] Fred B. Schneider. *Replication management using the state-machine approach.* ACM Press/Addison-Wesley Publishing Co., 1993.

[103] Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable deferred update replication. In *DSN*, pages 1–12, 2012.

[104] Peluso Sebastiano, Roberto Palmieri, Francesco Quaglia, and Binoy Ravindran. On the viability of speculative transactional replication in database systems: a case study with PostgreSQL. In *NCA '13*, 2013.

[105] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, 1995.

[106] J.E. Smith and G.S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, Dec 1995.

[107] Michael F. Spear, Maged M. Michael, and Christoph von Praun. Ringstm: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM.

[108] Jaswanth Sreeram, Romain Cledat, Tushar Kumar, and Santosh Pande. Rstm: A relaxed consistency software transactional memory for multicores. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 428–, Washington, DC, USA, 2007. IEEE Computer Society.

[109] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the oklahoma update. *IEEE Parallel Distrib. Technol.*, 1(4):58–71, November 1993.

[110] Pierre Sutra and Marc Shapiro. Fast genuine generalized consensus. In *SRDS*, pages 255–264, 2011.

[111] Douglas Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, and Marcos K. Aguilera. Transactions with consistency choices on geo-replicated cloud storage. Technical Report MSR-TR-2013-82, September 2013.

[112] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13.

[113] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[114] E Tilevich and Y Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *ECOOP*, 2002.

[115] Alexandru Turcu, Sebastiano Peluso, Roberto Palmieri, and Binoy Ravindran. Be general and don't give up consistency in geo-replicated transactional systems. In *OPODIS*, 2014.

[116] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE TKDE*, 17(4), 2005.

[117] Pawel T. Wojciechowski, Tadeusz Kobus, and Maciej Kokocinski. Model-driven comparison of state-machine-based and deferred-update replication schemes. In *SRDS*, 2012.