HiperTM: High Performance, Fault-Tolerant Transactional Memory

Sachin Hirve, Roberto Palmieri[1], Binoy Ravindran

Virginia Tech, Blacksburg VA 24061, USA

Abstract

We present HiperTM, a high performance active replication protocol for fault-tolerant distributed transactional memory. The active replication processing paradigm allows transactions to execute locally, costing them only a single network communication step during transaction execution. Shared objects are replicated across all sites, avoiding remote object accesses. Replica consistency is ensured by a) OS-Paxos, an optimistic atomic broadcast layer that total-orders transactional requests, and b) SCC, a local speculative multi-version concurrency control protocol that enforces a commit order equivalent to transactions' delivery order. SCC executes write transactions serially without incurring any synchronization overhead, and runs read-only transactions in parallel (to write transactions) with non-blocking execution and abort-freedom. Our implementation reveals that HiperTM guarantees 0% of out-of-order optimistic deliveries and performance up to 3.5× better than atomic broadcast-based competitor (PaxosSTM) using the standard configuration of TPC-C benchmark.

Keywords: Distributed Transactional Memory, Fault-Tolerance, Speculative Processing, State-Machine Replication

Email addresses: hsachin@vt.edu (Sachin Hirve), robertop@vt.edu (Roberto Palmieri), binoy@vt.edu (Binoy Ravindran)

[1]Corresponding author. Address: 454 Durham Hall, Virginia Tech, Blacksburg VA 24061, USA. Tel: 540-231-0642. Email: robertop@vt.edu

Paper accepted at Journal of Theoretical Computer Science

1. Introduction

Software transactional memory (STM) [1] is a promising programming model for managing concurrency of transactional requests. STM libraries offer APIs to programmers for reading and writing shared objects, ensuring atomicity, isolation, and consistency in a completely transparent manner. STM 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 settings), where interactions with a stable storage often significantly degrade performance.

Besides performance, transactional applications usually require strong dependability properties that centralized, in-memory processing systems cannot guarantee. Fault-tolerant mechanisms often involve expensive synchronization with remote nodes. As a result, directly incorporating them into in-memory transactional applications (distributed software transactional memory or DTM [2, 3, 4, 5]) will reduce the performance advantage (of in-memory operations) due to network costs. For example, the *partial replication* paradigm allows transaction processing in the presence of node failures, but the overhead paid by transactions for looking-up latest object copies at encounter time limits performance. Current partial replication protocols [6, 7] 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 [8, 9]. *Full replication* is a way to annul network interactions while reading/writing objects. In this model, application's entire shared data-set is replicated across all nodes. However, to ensure replica consistency, a common serialization order (CSO) of transactions must be ensured.

State-machine replication (or active replication) [10] 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 [11]) problem: i.e., how a group of processes can agree on a value in the presence of faults in partially synchronous systems. Paxos [12] 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* [13, 14, 15], built on top of MultiPaxos [12], 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 [14]. *S-Paxos* [16] 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 [17], 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 [18, 19, 20, 21]. However, guessing alternative serialization orders [22, 23] – 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 paper, 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 process-

ing 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* [24], a high performance public cluster with 19 nodes[2] using benchmarks including TPC-C [25] 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.

C) HiperTM's transactional throughput is up to 3.5× better than PaxosSTM [26], 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.

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

The complete implementation of HiperTM is publicly available at `https://bitbucket.org/hsachin/hipertm/`.

The rest of the paper is organized as follow. Section 2 reports assumptions made by HiperTM. Section 3 represents the core of the paper and describes HiperTM with its two main components: OSPaxos and SCC. Correctness arguments and proofs are also presented in Section 3. The experimental evaluation of HiperTM is in Section 4; the related work is overviewed in Section 5; and the Section 6 concludes the paper.

2. System Model

We consider a classical distributed system model [27] consisting of a set of processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate via message passing links. Process may fail according to the fail-stop (crash) model. A non-faulty process is called correct. We assume a partially synchronous system [12], where $2f + 1$ nodes are required for tolerating at most f nodes that are simultaneously faulty. We consider only non-byzantine faults, i.e., nodes cannot perform actions that are not compliant with the replication algorithm.

In any ordering communication step, a node contacts all the sites and waits for a *quorum* Q of replies. We assume $Q = f + 1$ such that a quorum can always be formed because $N - f \geq Q$. 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.

In order 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 [28] that is necessary to implement a leader election service [27].

A transaction is composed of a series of read and write operations, executed atomically (all or nothing). 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. 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) (Section 3.2

details the programming model). Threads submit the transaction request to a node, and wait until the node successfully commits that transaction.

OS-Paxos is the network service responsible for defining a total order among transactional requests. The requests are considered as network messages by OS-Paxos; it is not aware of the messages' content, it only provides ordering. After the message is delivered to a replica, the transactional request is extracted and processed as a transaction.

OS-Paxos delivers each message twice. The first is called *optimistic-delivery* (or opt-del) and the second is called *final-delivery* (or final-del). Opt-del notifies replicas that a new message is currently involved in the agreement process, and therefore opt-del's order cannot be considered reliable for committing transactions. On the other hand, final-del is responsible for delivering the message along with its order such that all replicas receive that message in the same order (i.e., total order). The final-del order corresponds to the transactions' commit order.

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). Each shared object is composed of: the last committed version, the last written version (not yet committed), and a list of previously committed versions. The last written version is the version generated by an opt-del transaction that is still waiting for commit. As a consequence, its timestamp is not specified. The timestamp is a monotonically increasing integer, which is incremented when a transaction commits. Our concurrency control ensures that only one writer can update the timestamp at a time. This is because, transactions are processed serially according to their opt-del order. Thus, there are no transactions validating and committing concurrently (Section 3.4 describes the concurrency control mechanism).

We assume that the transaction logic is *snapshot-deterministic* [20], i.e., the sequence of operations executed depends on the return value of previous read operations. Thus, any form of non-determinism is excluded.

3. HiperTM

3.1. Optimistic S-Paxos

Optimistic S-Paxos (or OS-Paxos) is an implementation of optimistic atomic broadcast [29] built on top of S-Paxos [16]. 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 [29]. 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 [14, 15] for increasing throughput. A replica

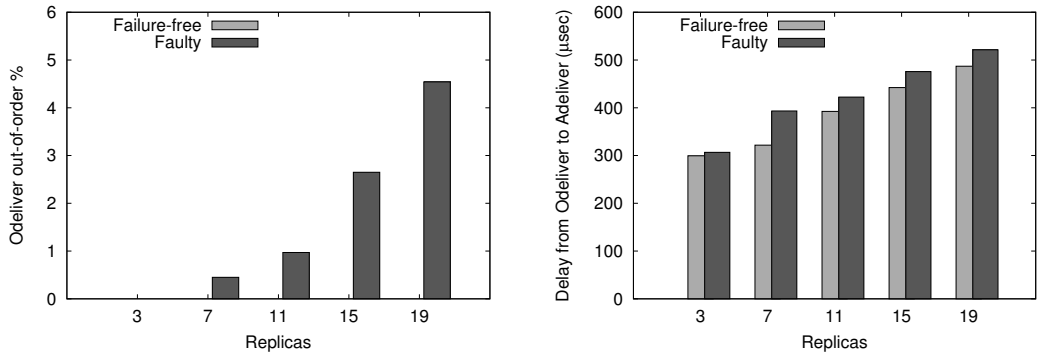
creates a batch of client requests and distributes it to other replicas. The receiver replicas forward this batch to all other replicas. When the replicas observe a majority of delivery for a batch, it is considered as stable batch. The leader then *proposes* an order (containing only batch IDs) for 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 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 1 shows the results. The experimental test-bed is the same used for the evaluation of HiperTM in Section 4 (briefly, we used 19 nodes interconnected via 40 Gbits network on PRObE [24] public cluster).

Reordering (Figure 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. How-

ever, 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 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*.

3.2. Programming model

Classical transactional applications based on STM/DTM delimit portions of source code containing operations which must be executed transactionally, according to the application's logic (all or nothing). Those blocks are managed by the STM/DTM library and executed according to the concurrency control rules at hand. Some programming languages use *annotations* for marking transactional code (e.g., annotation), while others explicitly invoke APIs offered by the STM/DTM library in order to open and commit a transactional context (e.g., store-procedure).

HiperTM's programming model follows the latter approach. Since transactions must be ordered through the total order layer (i.e., OS-Paxos), the concurrency control mechanism cannot process transactions in the same application thread (this is the only difference with the traditional, API-based transactional programming model). In HiperTM, programmers can either:

(a) wrap transactions in a method with the necessary parameters and call a library API (i.e., *invoke(type par1, type par2, ...)*) to invoke that transaction; or

(b) adopt a byte-code rewriting tool for transparently generating methods with the needed parameters from atomic blocks.

```
1  class Client{
2          void submitTransfer{
3                  ...
4                  byte[] request;
5                  request.put(TRANSFER);
6                  request.putInt(sourceAccount);
7                  request.putInt(destAccount);
8                  request.putFloat(amount);
9                  byte[] response;
10                 response = client.invoke(request);
11                 ...
12         }
13 }
14 class Server{
15         ...
16         transfer(ClientRequest, srcAcc, dstAcc, amount);
17         ...
18 }
```

Figure 2: Transfer transaction profile of Bank benchmark on HiperTM.

Figure 2 shows how the transaction profile of the *transfer* operation of Bank benchmark is managed by HiperTM. Transfer requires three parameters: the source account, the destination account, and the amount to be transferred. These parameters are stored in the request and sent for execution using *invoke*.

3.3. 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.

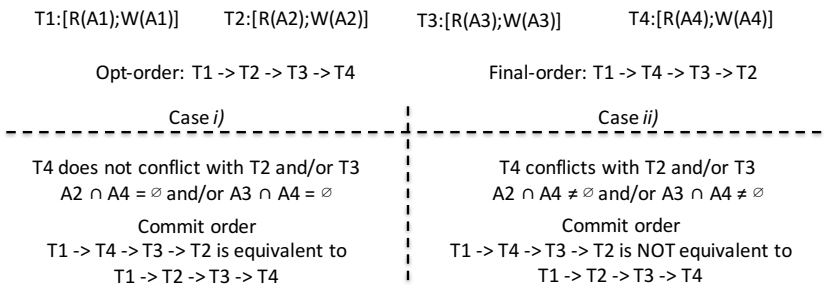


Figure 3: Example of committing transactions $\{T_1,T_2,T_3,T_4\}$ varying the conflict of accessed objects, in case the final order contradicts the optimistic order.

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).

Figure 3 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.

3.3.1. 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 [17] in the context of traditional DBMS. In addition to [17], 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 3.4 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.

3.3.2. 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.

3.4. 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 3.3, 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.

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

Algorithm 1 Read Operation of Transaction T_i on Object X.

```
1: if T_i.readOnly = FALSE then
2:     if ∃ version ∈ X.completeBuffer then
3:         T_i.ReadSet.add(X.completeBuffer)
4:         return X.completeBuffer.value
5:     else
6:         T_i.ReadSet.add(X.lastCommittedVersion)
7:         return X.lastCommittedVersion.value
8:     end if
9: else
10:     if r-timestamp = 0 then
11:         r-timestamp ← X.lastCommittedVersion.timestamp
12:         return X.lastCommittedVersion.value
13:     end if
14:     P ← {set of versions V ∈ X.committedVersions s.t. V.timestamp ≤ r-timestamp
15:     if P ≠ ∅ then
16:         V_cx ← ∃ version V_k ∈ P s.t. ∀ V_q ∈ P ⇒ V_k.timestamp ≥ V_q.timestamp        ▷ V_cx has the
    maximum timestamp in P
17:         return V_cx.value
18:     else
19:         return X.lastCommittedVersion.value
20:     end if
21: end if
```

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

```
1: Version V_x ← createNewVersion(X,v)
2: X.completeBuffer ← V_x
3: T_i.WriteSet.add(V_x)
```

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.

The abort handler is responsible for removing the complete buffer of each object written by the aborted transaction. As a consequence of that, any newly activated speculative transaction will not observe any speculative version while reading, thus it will be forced to read the committed version.

A task queue is responsible for scheduling jobs executed by the main thread (processing write transactions). Whenever an event such as *Odeliver*

Algorithm 3 Validation Operation of Transaction T_i.

```
1: for all V_x ∈ T_i.ReadSet do
2:     if V_x ≠ 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 ∈ T_i.WriteSet do
7:     V_x.timestamp ← c-timestamp
8:     X.lastCommittedVersion ← V_x
9: end for
10: replica-timestamp = c-timesamp
```

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).

3.5. 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.

3.5.1. Formalism

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

According to the definition in [30], 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 [30].

We now use a directed graph as a representation of an history H where committed transaction in H are the graph's vertices and there exists a directed edge between two vertices if the respective transactions are conflicting. We name this graph as Directed 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 vertices 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:

- *Directed 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 $V_{T_i} \xrightarrow{wr} V_{T_j}$.

- *Directed 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 $V_{T_i} \xrightarrow{ww} V_{T_j}$.

- *Directed 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.

3.5.2. 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) [31], 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 [31], 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 vertices 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 directed 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 directed read-dependency or directed 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 vertices 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 in spite of 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 vertices 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 [32], 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 [33] (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 [34], 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. □

3.5.3. Local Properties

HiperTM guarantees a variant of opacity [35] locally to each replica. It cannot ensure Opacity as defined in [35] 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.

4. 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 [25]. Bank emulates a monetary application and is typically used in TM works for benchmarking performance [26, 6, 36]. TPC-C [25] is a well known benchmark that is representative of on-line transaction processing workloads.

We used PaxosSTM [26, 5] 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 [24], 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 [5]: 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.

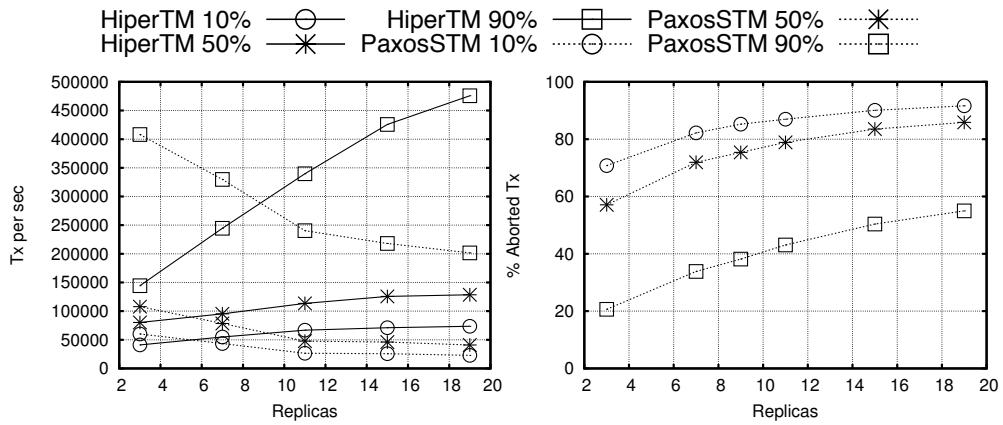
4.1. Bank Benchmark

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 4) and latency (Figure 5). 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 4 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

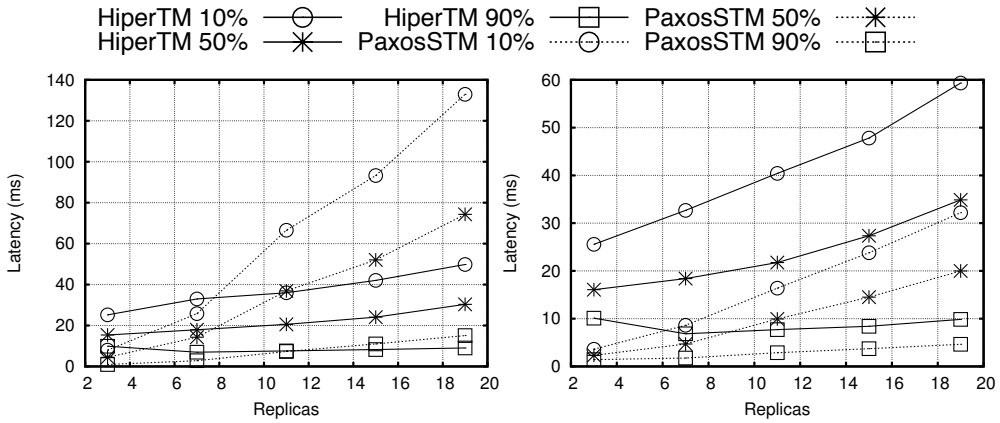
Figure 4: Throughput and abort percentage of HiperTM and PaxosSTM for Bank benchmark.

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 4(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 4(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.

Figure 5 shows the latency measured in the same experiments reported in Figure 4. 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.



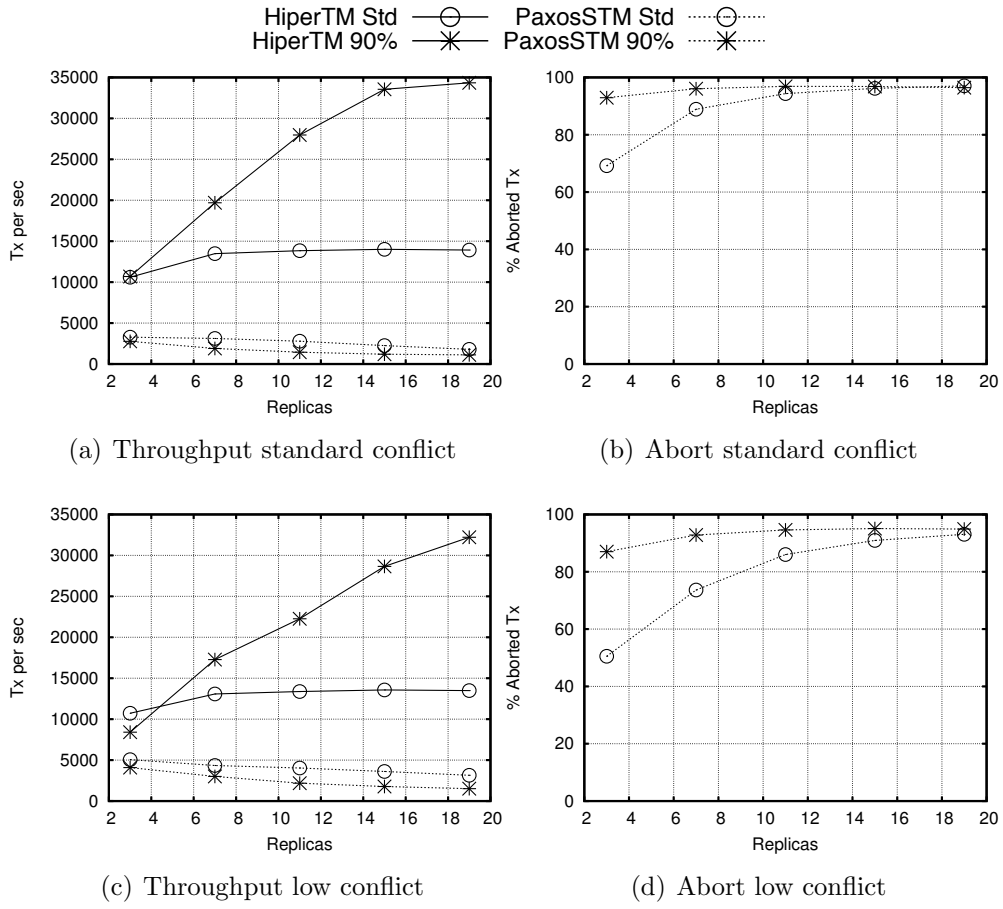
(a) High conflict (b) Low conflict

Figure 5: Latency of HiperTM and PaxosSTM for Bank benchmark.

4.2. TPC-C Benchmark

TPC-C [25] 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.



(a) Throughput standard conflict

(b) Abort standard conflict

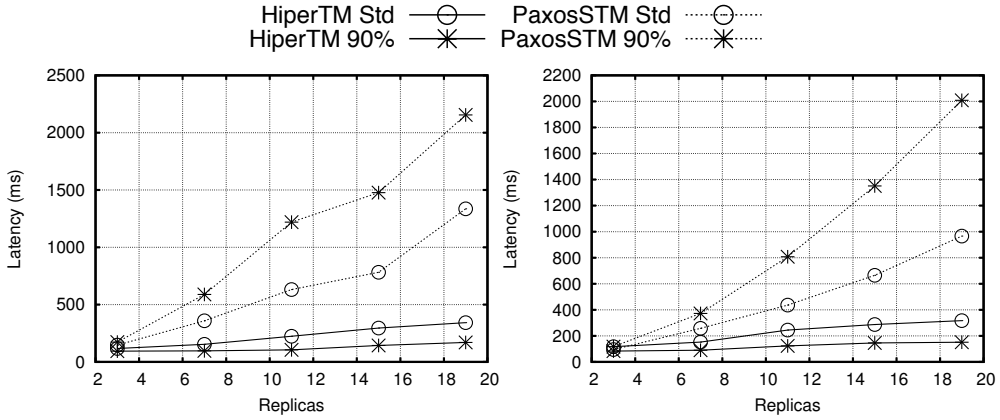
(c) Throughput low conflict

(d) Abort low conflict

Figure 6: Throughput and abort percentage of HiperTM and PaxosSTM for TPC-C benchmark.

Figure 6 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 6(a) and 6(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) Standard conflict (b) Low conflict

Figure 7: Latency of HiperTM and PaxosSTM for TPC-C benchmark.

The Figure 7 shows the latency measured in the above experiments. Clearly, lower throughput and longer transactions caused higher latency.

5. Related Work

Replication in transactional systems has been widely explored in the context of DBMS, including protocol specifications [37] and infrastructural solutions [38, 39, 40, 41]. These proposals span from the usage of distributed locking mechanisms to atomic commit protocols. [42] implements and evaluates various replication techniques, and those based on active replication are found to be the most promising.

In [43, 44], two active replication techniques are presented. Both rely on atomic broadcast for ordering transaction requests, and execute them only when the final order is notified. In contrast HiperTM, based on optimistic

atomic broadcast, begins to process transactions before their final delivery, i.e., when they are optimistically delivered.

Speculative processing of transactions has been originally presented in [17] and further investigated in [19, 20]. [19] 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 [19]; no actual implementation is presented, and therefore overheads are not revealed.

In HiperTM, all the design choices are motivated by real performance issues. Our results show how single-thread processing and multi-versioned memory for parallel activation and abort-freedom of read-only transactions are the best trade-off in terms of performance and overhead for conflict detection, in systems based on total order services similar to OS-Paxos. In contrast to [20], HiperTM does not execute the same transaction in multiple serialization orders, because OS-Paxos, especially in case of failure-free execution, guarantees no-reorders.

Full replication based on total order has also been investigated in certification-based transaction processing [26, 45, 46]. 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). [45] is based on OAB while [46] is based on classical atomic broadcast layer. Both suffer from (O)AB's scalability bottleneck when message size increases. In HiperTM, the length of messages does not depend on transaction operations; it is only limited by the signature of invoked transactions along with their parameters.

Object access pattern impacts both state-machine and certification-based replication differently. Though certification-based transactional systems exhibit high performance for low contention scenarios, they suffer from high abort rates in case of high contention. On the other hand, state-machine replication is less impacted by system's contention level, but even for low contention workloads its performance is limited. An evidence of this phenomenon can be seen in Section 4 through our comparison between HiperTM and PaxosSTM. Recently, a hybrid replication approach [5] has been proposed. It tries to bring the best of both worlds by starting the transactions in certification-based mode and switching to state-machine mode when conflicts are detected. This approach limits the number of aborts for conflicting

transaction while still providing high throughput for non-conflicting transactions. When compared with HiperTM, the hybrid approach presented in [5] does not exploit the optimism while ordering transactions and its performance is significantly impacted from the prediction of the oracle component that is responsible for switching between modes. In addition, when transactions in certification-based mode and state-machine mode coexist, then they influence each other on the system's critical path because the certification phase, as well as the execution of state-machine transactions are performed by the same thread.

Granola [36] 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 SCC.

6. Conclusions

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 processing time is limited.

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

Acknowledgement

This work is supported in part by the AFOSR under grant FA9550-15-1-0098.

References

[1] N. Shavit, D. Touitou, Software transactional memory, in: PODC, 1995, pp. 204–213.

[2] M. Couceiro, P. Romano, N. Carvalho, L. Rodrigues, D2stm: Dependable distributed software transactional memory, in: PRDC, 2009, pp. 307–313.

[3] D. Hendler, A. Naiman, S. Peluso, F. Quaglia, P. Romano, A. Suissa, Exploiting locality in lease-based replicated transactional memory via task migration, in: DISC, 2013, pp. 121–133.

[4] A. Turcu, B. Ravindran, R. Palmieri, Hyflow2: a high performance distributed transactional memory framework in scala, in: PPPJ, 2013, pp. 79–88.

[5] T. Kobus, M. Kokocinski, P. T. Wojciechowski, Hybrid replication: State-machine-based and deferred-update replication schemes combined, in: ICDCS, 2013, pp. 286–296.

[6] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, L. E. T. Rodrigues, When scalability meets consistency: Genuine multiversion update-serializable partial data replication, in: ICDCS, 2012, pp. 455–465.

[7] N. Schiper, P. Sutra, F. Pedone, P-store: Genuine partial replication in wide area networks, in: SRDS, 2010, pp. 214–224.

[8] D. Dice, O. Shalev, N. Shavit, Transactional locking II, in: DISC, 2006, pp. 194–208.

[9] A. Dragojevic, R. Guerraoui, M. Kapalka, Stretching transactional memory, in: PLDI, 2009, pp. 155–165.

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

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

[12] L. Lamport, The part-time parliament, ACM Trans. Comput. Syst. (1998) 133–169.

[13] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, A. Schiper, JPaxos: State machine replication based on the Paxos protocol, Tech. Rep. EPFL-REPORT-167765, Faculté Informatique et Communications, EPFL, 38pp (Jul. 2011).

[14] N. Santos, A. Schiper, Tuning paxos for high-throughput with batching and pipelining, in: ICDCN, 2012, pp. 153–167.

[15] N. Santos, A. Schiper, Optimizing paxos with batching and pipelining, Theor. Comput. Sci. 496 (2013) 170–183.

[16] M. Biely, Z. Milosevic, N. Santos, A. Schiper, S-paxos: Offloading the leader for high throughput state machine replication, in: SRDS, 2012, pp. 111–120.

[17] B. Kemme, F. Pedone, G. Alonso, A. Schiper, M. Wiesmann, Using optimistic atomic broadcast in transaction processing systems, IEEE TKDE 15 (4).

[18] S. Hirve, R. Palmieri, B. Ravindran, Archie: a speculative replicated transactional system, in: Middleware, 2014, pp. 265–276.

[19] R. Palmieri, F. Quaglia, P. Romano, AGGRO: boosting STM replication via aggressively optimistic transaction processing, in: NCA, 2010, pp. 20–27.

[20] R. Palmieri, F. Quaglia, P. Romano, OSARE: opportunistic speculation in actively replicated transactional systems, in: SRDS, 2011, pp. 59–64.

[21] P. J. Marandi, M. Primi, F. Pedone, High performance state-machine replication, in: DSN, 2011, pp. 454–465.

[22] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, L. Rodrigues, Brief announcement: on speculative replication of transactional systems, in: SPAA, 2010, pp. 69–71.

[23] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, L. Rodrigues, An optimal speculative transactional replication protocol, in: ISPA, 2010, pp. 449–457.

[24] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar, An integrated experimental environment for distributed systems and networks, in: OSDI, 2002.

[25] T. Council, TPC-C benchmark, 2010.

[26] P. T. Wojciechowski, T. Kobus, M. Kokocinski, Model-driven comparison of state-machine-based and deferred-update replication schemes, in: SRDS, 2012, pp. 101–110.

[27] R. Guerraoui, L. Rodrigues, Introduction to Reliable Distributed Programming, 2006.

[28] R. Guerraoui, A. Schiper, Genuine atomic multicast in asynchronous distributed systems, Theor. Comput. Sci. 254 (1-2) (2001) 297–316.

[29] F. Pedone, A. Schiper, Optimistic atomic broadcast, in: DISC, 1998, pp. 318–332.

[30] A. Adya, Weak consistency: A generalized theory and optimistic implementations for distributed transactions, Ph.D. thesis, aAI0800775 (1999).

[31] P. A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency control and recovery in database systems, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[32] M. Herlihy, Wait-free synchronization, ACM Trans. Program. Lang. Syst. 13 (1) (1991) 124–149.

[33] R. Guerraoui, M. Kapalka, The semantics of progress in lock-based transactional memory, in: POPL, 2009, pp. 404–415.

[34] M. Herlihy, Technical perspective - highly concurrent data structures, Commun. ACM 52 (5) (2009) 99.

[35] R. Guerraoui, M. Kapalka, On the correctness of transactional memory, in: PPOPP, 2008, pp. 175–184.

[36] J. A. Cowling, B. Liskov, Granola: Low-overhead distributed transaction coordination, in: USENIX Annual Technical Conference, 2012, 2012, pp. 223–235.

[37] B. Kemme, G. Alonso, Don't be lazy, be consistent: Postgres-r, A new way to implement database replication, in: VLDB, 2000, pp. 134–143.

[38] S. Peluso, R. Palmieri, F. Quaglia, B. Ravindran, On the viability of speculative transactional replication in database systems: A case study with postgresql, in: NCA, 2013, pp. 143–148.

[39] F. Perez-Sorrosal, M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, Consistent and scalable cache replication for multi-tier J2EE applications, in: Middleware, 2007, pp. 328–347.

[40] M. Patino-Martinez, R. Jiménez-Peris, B. Kemme, G. Alonso, MIDDLE-R: Consistent database replication at the middleware level, ACM Trans. Comput. Syst. 23 (4).

[41] F. Pedone, S. Frølund, Pronto: High availability for standard off-the-shelf databases, J. Parallel Distrib. Comput. 68 (2).

[42] M. Wiesmann, A. Schiper, Comparison of database replication techniques based on total order broadcast, IEEE TKDE 17 (4).

[43] D. Agrawal, G. Alonso, A. El Abbadi, I. Stanoi, Exploiting atomic broadcast in replicated databases (extended abstract), in: Euro-Par, 1997, pp. 496–503.

[44] R. Jiménez-Peris, M. Patiño-Martínez, S. Arévalo, Deterministic scheduling for transactional multithreaded replicas, in: SRDS, 2000, pp. 164–173.

[45] N. Carvalho, P. Romano, L. E. T. Rodrigues, Scert: Speculative certification in replicated software transactional memories, in: SYSTOR, 2011, p. 10.

[46] S. Peluso, J. Fernandes, P. Romano, F. Quaglia, L. E. T. Rodrigues, SPECULA: speculative replication of software transactional memory, in: SRDS, 2012, pp. 91–100.