HiperTM: High Performance, Fault-Tolerant Transactional Memory

Sachin Hirve, Roberto Palmieri, and Binoy Ravindran

Virginia Tech, Blacksburg VA 24060, USA,
{hsachin,robertop,binoy}@vt.edu

Abstract. We present HiperTM, a high performance active replication protocol for fault-tolerant distributed transactional memory. The active replication 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 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 1.2$\times$ better than atomic broadcast-based competitor (PaxosSTM).

1 Introduction

Software transactional memory (STM) [31] 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) 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 [25, 28] 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) must be ensured.

State-machine replication (or active replication) [29] 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 [7]) problem: i.e., how a group of processes can agree on a value in the presence of faults in partially synchronous systems. Paxos [18] 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* [17, 27], built on top of MultiPaxos [18], 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 [27]. *S-Paxos* [3] further improves JPaxos 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 [15], 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, the transaction is totally (or partially) executed and committed [19, 20]. However, guessing alternative serialization orders 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, in order to minimize the likelihood of mismatches between the optimistic order and the final delivery order. The protocol wraps 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 conducted experimental studies using benchmarks including TPC-C [5]. Our results reveal three important trends:

A) OS-Paxos provides a very limited number of out-of-order optimistic deliveries ($<1\%$ 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 $1.2\times$ better than state-of-the-art atomic broadcast-based competitor PaxosSTM [16], and up to $10\times$ better than Score [25].

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. The complete implementation of HiperTM, source codes, test-scripts, etc., is publicly available at https://bitbucket.org/hsachin/hipertm/.

2 System Model

We consider a classical distributed system model [12] consisting of a set of processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate via message passing. Process may fail according to the fail-stop (crash) model. A non-faulty process is called correct. We assume a partially synchronous system [18], where $2f + 1$ nodes are correct and at most f nodes are simultaneously faulty. We consider only non-byzantine faults, i.e., nodes cannot perform actions that are not compliant with the replication algorithm.

We consider a full replication model, where the application's entire shared dataset 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 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. 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.

3 HiperTM

3.1 Optimistic S-Paxos

Optimistic S-Paxos (or OS-Paxos) is an implementation of optimistic atomic broadcast [23] built on top of S-Paxos [3]. S-Paxos, as its predecessor JPaxos [17, 27], 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.

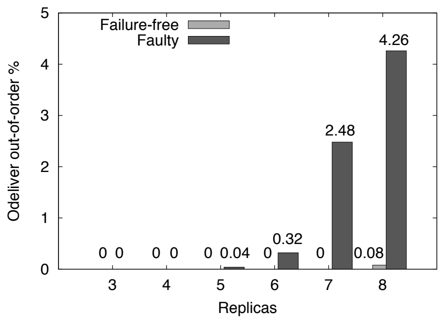
OS-Paxos provides an additional primitive, called $Odeliver(m)$, which is used for early delivering a previously broadcast message m before the $Adeliver$ for m is issued. OS-Paxos ensures that:

– 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 $Odeliver(m)$.

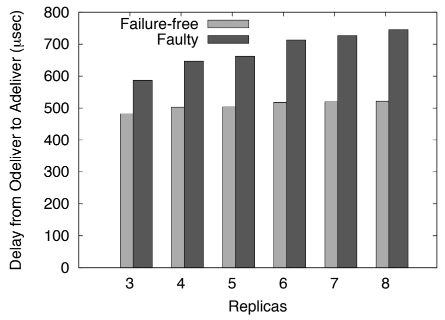
OS-Paxos's properties and primitives are compliant with the definition of optimistic atomic broadcast [23]. The sequence of $Odeliver$ notifications defines the so called *optimistic order* (or *opt-order*). The sequence of $Adeliver$ defines the so called *final order*. We now describe the architecture 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 [27] for increasing throughput. A replica creates a batch of client requests and distributes it to other replicas. The receiver replicas store the batch of requests and send an ack to all other replicas. When the replicas observe a majority of acks for a batch, it is considered as stable. 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 agreement 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.



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



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

Fig. 1. OS-Paxos performance.

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. If there is any out-of-order instance agreement, $Adeliver$ is delayed until all previous instances are $Adeliver$ed.

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. Reorderings (Figure 1(a)) are absent for failure-free experiments. 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 6 nodes. However, the impact on the overall performance is limited because the maximum number of reordering observed is lower than 5% with 8 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 $\approx$500 microseconds in case of failure-free runs and it increases up to $\approx$750 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; (b) or adopt a byte-code rewriting tool for transparently generating methods with the needed parameters from atomic blocks.

```
class Client{
void submitTransfer{
...
byte[] request;
request.put(TRANSFER);
request.putInt(sourceAccount);
request.putInt(destAccount);
request.putFloat(amount);
byte[] response;
response = client.invoke(request);
...
}
}
class Server{
...
transfer(ClientRequest, srcAcc, dstAcc,
     amount)
...
}
```

Fig. 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.

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 series 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 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.

We use the speculative processing technique for executing optimistically (but not yet finally) delivered transactions. This approach has been proposed in [15] in the context of traditional DBMS. In addition to [15], 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. 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 transaction's completion.

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. Although complete buffers can be managed without synchronization because only one writing transaction is active at a time, installing a new version as committed requires synchronization. Therefore, each object maintains also a list of previously committed versions. This is exploited by read-only transactions to execute independently from the write transactions.

Read-only transactions are marked by programmers and 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 1(b), 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. SCC is able to process ≈95K write transactions per second, in-order, while ≈250K read-only transactions are executing in parallel on different cores (Bank benchmark on experimental test-bed). 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).

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.

```
upon Read(Transaction $T_i$, Object $X$) do
  if ($T_i$.readOnly == FALSE)
    if ($\exists$ version $\in X$.completeBuffer
      $T_i$.ReadSet.add($X$.completeBuffer);
      return $X$.completeBuffer.value;
    else
      $T_i$.ReadSet.add($X$.lastCommittedVersion);
      return $X$.lastCommittedVersion.value;
  else
    if (r-timestamp == 0)
      r-timestamp = $X$.lastCommittedVersion.timestamp;
      return $X$.lastCommittedVersion.value;
    $P$= {all versions $V \in X$.committedVersions s.t.
    $V$.timestamp $\leq$ r-timestamp}
    if ($\exists$ version $V_{cx} \in P$)
       $V_{cx}$ = maximum-timestamp($P$)
      return $V_{cx}$.value;
    else
      return $X$.lastCommittedVersion.value;

upon Write(Transaction $T_i$, Object $X$, Value $v$) do
  Version $V_x$ = createNewVersion($X$,$v$);
  $X$.completeBuffer = $V_x$;
  $T_i$.WriteSet.add($V_x$);
upon Commit(Transaction $T_i$) do
  if (Validation($T_i$) == FALSE)
    return $T_i$.abort&restart();
  c-timesamp = replica-timestamp;
  c-timesamp++;
  $\forall V_x \in T_i$.WriteSet do
    $V_x$.timestamp = c-timestamp;
    $X$.lastCommittedVersion = $V_x$;
  replica-timestamp = c-timesamp;
boolean Validation(Transaction $T_i$)
  $\forall V_x \in T_i$.ReadSet do
    if ($V_x \neq X$.lastCommittedVersion)
    return FALSE;
  return TRUE;
```

Fig. 3. SCC's pseudo code.

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.

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 Figure 3. We show the core steps of the concurrency control protocol such as reading/writing a shared object and validating/committing a write transaction.

3.5 Properties

HiperTM ensures 1-copy serializability, opacity, lock-freedom and abort-freedom for read-only transactions. We avoid formal proofs due to space constraints, but sketch the basic arguments as follows:

Opacity [11]. A protocol ensures opacity if it guarantees three properties: (Op.1) committed 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; and (Op.3) all live and aborted transactions observe a consistent system state.

HiperTM ensures opacity for each replica. It 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. 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. (Op.3) is guaranteed by serial execution, which prevents concurrent accesses to same objects. 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 of objects is always consistent.

1-Copy serializability [2]. 1-copy serializability is guaranteed because each replica commits the same set of write transactions in the same order notified by the optimistic atomic broadcast layer. Read-only transactions activated on different nodes cannot observe any two write transactions that are serialized differently on those nodes.

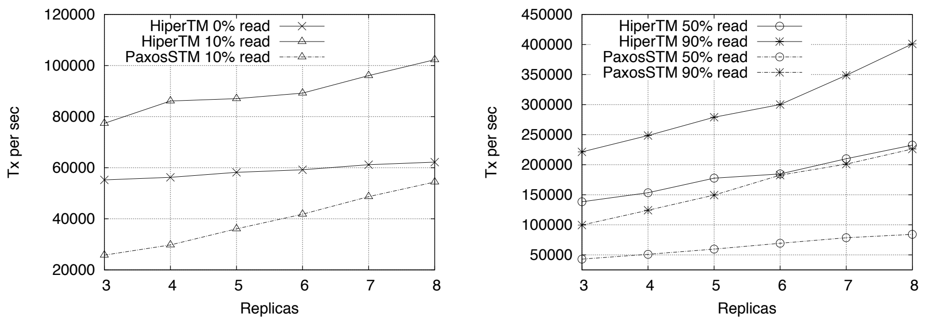
Lock-freedom [10]. Lock-freedom guarantees that there always exists at least one thread that makes progress, which rules out deadlocks and livelocks. In HiperTM, this is a direct consequence of the fact that transactions aborted due to unsuccessful validation and already $Adeliver$ed can restart their execution and cannot be aborted anymore due to serial execution and its highest priority for subsequent commit.

Abort-freedom of read-only transactions. Before issuing the first operation, read-only transactions save the replica-timestamp in their local *r*-timestamp and use it for selecting the proper committed versions to read. The subset of all the versions that read-only transactions can access during their execution is fixed when the transactions define their *r*-timestamp. Only one write transaction is executing when a read-only transaction acquires the *r*-timestamp. If this write transaction updates the replica-timestamp before the other acquires the *r*-timestamp, the read-only transaction is serialized after the write transaction, but before the next write transaction eventually commits. On the contrary, if the replica-timestamp's update happens after, the read-only transaction is serialized before the write transaction and cannot access the write transaction's just committed objects. In both cases, the subset of versions that the read-only transaction 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.

4 Implementation and Evaluation

HiperTM's architecture consists of two layers: network layer (OS-Paxos) and replica 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. Bank emulates a bank application and is typically used in TM works for benchmarking performance [16, 25, 6]. TPC-C [5] is a well known benchmark that is representative of on-line transaction processing workloads.

We used PaxosSTM [16] and Score [25] as competitors. 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). Score is a



(a) Write-intensive workload (b) Read workload

Fig. 4. Performance of HiperTM and PaxosSTM for the Bank benchmark.

partial replication-based DTM protocol (in contrast to HiperTMs state machine-based replication, which yields full failure-masking). Score is designed to scale up to hundred nodes with a replication degree of two. Both represent state-of-the-art DTM.

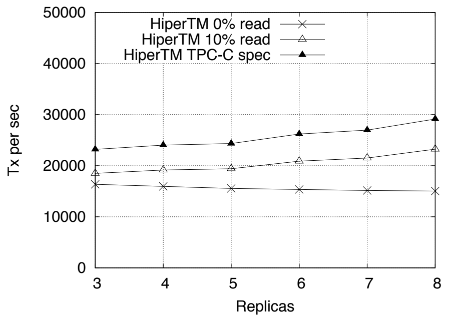
Our test-bed consists of 8 nodes interconnected using a 1Gb/s switched network. Four of the nodes are 64-core AMD Opteron machines (128GB RAM, 2.3 GHz), while the other four are 48-core AMD Opteron (32GB RAM, 1.7 GHz). For each benchmark, we varied the percentage of read-only transactions to cover all configuration settings. Clients are balanced on all the replicas. They inject transactions for the benchmark and wait for the reply. A pool of 20 threads are deployed for serving read-only transactions. Data points plotted are the average of 10 repeated experiments.

Figure 4 shows the throughput of the Bank benchmark[1]. Figure 4(a) shows results for 0% read and 10% read, and Figure 4(b) shows results for read-intensive workloads (50% and 90% read). HiperTM OS-Paxos sustains its throughput in all configurations, achieving almost constant throughput for write-only transactions with increasing replica count. Interestingly, with 3 replicas, HiperTM's write-only transaction throughput outperforms PaxosSTM's throughput for 10% of read-only transactions. This speed-up is directly due to HiperTM's speculative processing of write transactions, which allows SCC to commit most of the transactions when they are $Adeliver$ed.

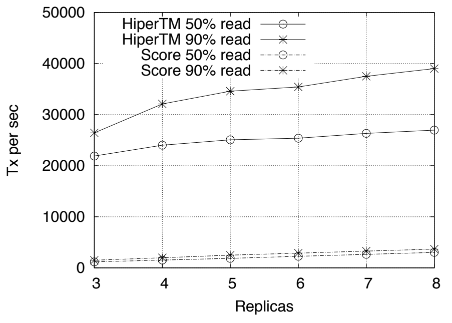
Performance as well as system scalability significantly increases when read-only transactions are varied from 10% to 90%. This is mainly because, read-only transactions can execute locally without involving OS-Paxos and other nodes for computation. The maximum throughput observed is $\approx$400K transactions processed per second in the entire system, with a maximum speed-up of $\approx 1.2\times$ over PaxosSTM.

Figure 5 compares HiperTM's performance with Score [25][2]. In these experiments, we ran TPC-C with the same configuration as Bank, and also with the configuration suggested by the TPC-C specification (Score's results are not available for all configurations). TPC-C's average transaction execution time is much higher than Bank's, due to the nature of its transactions, impacting overall throughput. However, HiperTM is

[1] Results of PaxosSTM that we used are also available in [16].
[2] Score's results are available in [25].



(a) Write-intensive workload



(b) Read workload

Fig. 5. Performance of HiperTM and Score for the TPC-C benchmark.

still able to overlap transaction execution with coordination for committing a transaction when its $Adeliver$ arrives. HiperTM outperforms Score with 8 replicas by up to $10\times$. Its effectiveness is particularly evident here, as Score pays the cost for looking-up remotely accessed objects, unlike HiperTM, which executes transactions locally.

We also measured the impact of the leader's failure on transactional throughput with the TPC-C benchmark. Due to space constraints, we skip plots, but summarize the key trend: we observed a maximum degradation of $\approx$30% after the crash. This is because, clients directly connected to the leader's replica need time to detect the failure and reroute their connections (for write and read transactions) to other replicas. After this time window, the performance returns to a stable state.

5 Related Work

Replication in transactional systems has been widely explored in the context of DBMS, including protocol specifications [14] and infrastructural solutions [30, 26, 21, 22]. These proposals span from the usage of distributed locking mechanisms to atomic commit protocols. [32] implements and evaluates various replication techniques, and those based on active replication are found to be the most promising.

In [1, 13], 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 [15] 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 [16, 4, 24]. 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). [4] is based on OAB and [24] is based on AB. 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.

Granola [6] 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

The authors would like to thank the authors of S-Paxos [3], PaxosSTM [16], and Score [25] for their willingness to share source codes.

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

References

1. D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In *Euro-Par 97*.
2. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
3. M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-Paxos: Offloading the Leader for High Throughput State Machine Replication. In *SRDS '12*.
4. N. Carvalho, P. Romano, and L. Rodrigues. Scert: Speculative certification in replicated software transactional memories. In *SYSTOR '11*.

5. T. Council. TPC-C benchmark. 2010.
6. J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference '12*.
7. X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4), 2004.
8. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06*.
9. A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. PLDI '09.
10. K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge Univ. Computer Laboratory, 2003.
11. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP '08*.
12. R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. 2006.
13. R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *SRDS 2000*.
14. B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *VLDB 2000*.
15. B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE TKDE*, 15(4), 2003.
16. T. Kobus, M. Kokociński, and P. Wojciechowski. Practical considerations of distributed STM systems development (abstract). In *WDTM '12*.
17. J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper. JPaxos: State machine replication based on the Paxos protocol. Technical Report EPFL-REPORT-167765, Faculté Informatique et Communications, EPFL, July 2011. 38pp.
18. L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, pages 133–169, 1998.
19. R. Palmieri, F. Quaglia, and P. Romano. AGGRO: Boosting STM replication via aggressively optimistic transaction processing. In *NCA '10*.
20. R. Palmieri, F. Quaglia, and P. Romano. OSARE: Opportunistic speculation in actively replicated transactional systems. In *SRDS '11*.
21. M. Patino-Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4), 2005.
22. F. Pedone and S. Frølund. Pronto: High availability for standard off-the-shelf databases. *J. Parallel Distrib. Comput.*, 68(2), 2008.
23. F. Pedone and A. Schiper. Optimistic atomic broadcast. In *DISC*, 1998.
24. S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues. SPECULA: Speculative replication of software transactional memory. In *SRDS '12*.
25. S. Peluso, P. Romano, and F. Quaglia. SCORe: A scalable one-copy serializable partial replication protocol. In *Middleware*, 2012.
26. F. Perez-Sorrosal, M. Pati no-Martinez, R. Jimenez-Peris, and B. Kemme. Consistent and scalable cache replication for multi-tier j2ee applications. In *Middleware '07*.
27. N. Santos and A. Schiper. Tuning paxos for high-throughput with batching and pipelining. In *ICDCN '12*.
28. N. Schiper, P. Sutra, and F. Pedone. P-store:genuine partial replication in WAN. In *SRDS 10*.
29. F. B. Schneider. *Replication management using the state-machine approach*. ACM Press/Addison-Wesley Publishing Co., 1993.
30. P. Sebastiano, R. Palmieri, F. Quaglia, and B. Ravindran. On the viability of speculative transactional replication in database systems: a case study with PostgreSQL. In *NCA '13*.
31. N. Shavit and D. Touitou. Software transactional memory. PODC '95.
32. M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE TKDE*, 17(4), 2005.