Archie: A Speculative Replicated Transactional System

Sachin Hirve
Virginia Tech
hsachin@vt.edu

Roberto Palmieri
Virginia Tech
robertop@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

ABSTRACT

We present Archie, a high performance fault-tolerant transactional system. Archie complies with the State Machine Approach, where the transactional state is fully replicated and total ordered transactions are executed on the replicas. Archie avoids the serial execution after transactions get ordered, which is the typical bottleneck of those protocols, by anticipating the work and using speculation to process transactions in parallel, enforcing a predefined order. The key feature of Archie is to avoid any non-trivial operation to perform post total order's notification, in case the sequencer node remains stable (only a single timestamp increment is needed for committing a transaction). This approach significantly shortens the transaction's critical path. The contention of speculative execution is always kept limited by activating a fixed number of transactions at a time. A comprehensive evaluation, using three competitors and three well known benchmarks, shows that Archie outperforms competitors in all medium/high contention scenarios.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Transaction processing; D.4.5 [**Software**]: Fault-tolerance

General Terms

Algorithms, Performance

Keywords

Fault-Tolerance, Replication, Speculation

1. INTRODUCTION

The State Machine Approach (SMA) [23] is a well-known coordination technique for building fault-tolerant services where all distributed nodes receive and process the same sequence of requests. Many replicated transactional schemes provide SMA with a transaction semantics (we name them SMA-based transactional systems, hereafter) so that transactions can be processed in a way such that the overall system is always *available* even in presence of faults. In such systems, each node replicates the entire shared state (*full replication*). As high-level characterization, SMA-based transactional systems can be classified according to the time when transactions are ordered globally. On the one hand, transactions can be executed by clients before a global certification is invoked to resolve conflicts against other transactions running on remote nodes. This approach is known as *Deferred Update Replication* (DUR) [26, 22, 3, 14]. On the other hand, clients can postpone the transaction execution until the agreement on a common order is reached. This way, they do not process transactions but simply broadcast transaction requests to all nodes and wait until the fastest replica replies. This method is known as *Deferred Execution Replication* (DER) [16, 10].

In both of these cases, an additional layer for ordering transactions is needed. This layer implements a *total order* protocol, such as Multi-Paxos [15], for agreeing on a set of previously proposed values in the presence of faults. Enforcing this order while transactions are certified/executed is usually done serially [26, 13, 10] (*serial phase* hereafter), with a significant impact on the total transaction execution time because any task done after the establishment of the global order increases latency perceived by clients. In addition, in such systems, the throughput is always bound by the performance of the (usually single) committer thread.

One approach that overcomes this limitation consists of relaxing the total order while providing a set of partial orders where only conflicting transactions are commonly serialized [16]. This allows the parallelization of the serial phase and also of the ordering process but requires additional information from the application to classify submitted transactions. However, in the presence of write-intensive workload with medium/high conflicts, establishing the total order still represents the most effective design choice.

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

In order to accomplish the above goal, we designed *MiMoX*, an optimized sequencer-based total order layer which inherits the advantages of two well-known mechanisms: the *optimistic notification* [12, 17], issued to nodes prior to the establishment of the total order; and *batching* [21, 6], as a means for improving the throughput of the global ordering process. MiMoX proposes an architecture that mixes these two mechanisms, thus allowing the anticipation (thanks to the optimistic notification) of a big amount of work (thanks to batching) before the total order is finalized. Nodes take advantage of the time needed for assembling a batch to compute a significant amount of work before the delivery of the order is issued. This anticipation is mandatory in order to minimize (and possibly eliminate the need for) the transaction's serial phase. As originally proposed in [17], MiMoX guarantees that if the sequencer node (*leader*) is not replaced during the ordering process (i.e., either suspected or crashed), the sequence of optimistic notifications matches the sequence of final notifications. As a distinguishing point, the solution in [17] relies on a ring topology as a means for delivering transactions optimistically, whereas MiMoX does not assume any specific topology.

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

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

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

The principles at the base of ARCHIE can be applied in both DUR- and DER-based systems. For the purpose of this paper, we optimized ARCHIE to cope with the DER model. This is because DER has three main benefits over DUR. First, it makes application behavior independent of failures. When a node in the system crashes or stops serving incoming requests, other nodes are able to transparently service the same request, process the transaction, and respond back to the application. Second, it does not suffer from aborts due to contention on shared remote objects because a common serialization order is defined prior to starting transaction (local) execution, thus yielding high performance and better scalability in medium/high contention scenarios [13]. Third, with DER, the size of network messages exchanged for establishing the common order does not depend on the transaction's logic (i.e., the number of objects accessed). Rather, it is limited to the name of the transaction and, possibly, its input parameters, which reduces network usage and increases the ordering protocol's performance.

As commonly adopted in several SMA-based transactional systems [13, 10, 18] and thanks to the full replication model, ARCHIE does not broadcast read-only workloads through MiMoX; read-only requests are handled locally, in parallel with the speculative execution. Processing write transactions (both conflicting and not conflicting) in the same order on all nodes allows ARCHIE to guarantee 1-copy-serializability [2].

We implemented ARCHIE in Java and we conducted a comprehensive experimental study using benchmarks including TPC-C [5], Bank and a distributed version of Vacation [4]. As competitors, we selected one DUR-based: PaxosSTM [26] – a high-performance open source transactional system; and two DER-based: one non-speculative (SM-DER [23]) and one speculative (HiperTM [10]) transactional system.

Our experiments on *PRObE* [7], a state-of-the-art public cluster, reveal ARCHIE's high-performance and scalability. On up to 19 nodes, ARCHIE outperforms all competitors in most of the tested scenarios. As expected, when the contention is very low, PaxosSTM behaves better than ARCHIE.

The paper makes the following contributions:

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

2. ARCHIE

Assumptions. We assume a distributed system, where a set of processes $\Pi = \{p_1, \dots, p_n\}$ communicate using message passing links. 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 [8] that is necessary to implement a leader election.

Nodes may fail according to the fail-stop (crash) model [2]. We assume $2f + 1$ nodes where at most f nodes are simultaneously faulty. In any communication step, a node contacts all other nodes and waits for a quorum Q of replies. We assume the classical quorum [15] $Q{=}f+1$ 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 node with the last updated information that we can use for recovering the system. Further, we consider only non-byzantine faults.

For the sake of generality and following the trend of [13, 10, 16], we adopt the programming model of software transactional memory (STM) [24] and its natural extension to distributed systems (i.e., DTM). DTM allows the programmer to simply mark a set of operations with transactional requirements as an "atomic block". The DTM framework transparently ensures the block's transactional properties while executing it concurrently with other blocks. There exists many frameworks that integrate STM-like atomic block abstraction with different types of transactional systems, such as *Key-Value* store [19, 22]. The adoption of this model does not restrict the applicability of our approach. Archie only assumes the existence of transactional read and write operations to instrument. As all DER-based systems, *snapshot-deterministic* [18, 16] transactions are assumed. This excludes any forms of non-determinism.

Transaction Processing Model. Archie defines application threads that invoke transactions (also called clients), and service threads that process transactions. These two groups of threads do not necessarily run on the same physical machine. Our transaction processing model is similar to the multi-tiered architecture that is popular in relational databases and other modern storage systems, where dedicated threads (different from threads that invoke transactions) process transactions.

Archie provides an application-level library that contains interfaces for interacting with the transactional system such as the `invoke` procedure. We adopt the store-procedure abstraction, common in DBMS, where the client does not send all the transactional operations to the service threads. When a client performs a transaction T_x, it is wrapped into a transaction request $REQ(T_x)$ and marked as a *read-only* or *write* transaction depending on T_x's operations. If all operations are reads, T_x is marked as read-only (T_r); otherwise as a write (T_w). For a write transaction, $REQ(T_w)$ is passed to MiMoX, which is responsible for ordering $REQ(T_w)$ among all the other requests submitted concurrently.

MiMoX delivers each transaction request (or a batch of them) twice, once optimistically and once finally. These two events are handled at each node by ParSpec, the local concurrency control protocol. When $REQ(T_w)$ is optimistically delivered, ParSpec extracts T_w from $REQ(T_w)$, retrieves T_w's business logic, and starts executing it speculatively. When $REQ(T_w)$ is finally delivered, ParSpec commits T_w if T_w executed in an order compliant with the final delivery order. Otherwise, T_w is aborted and restarted. ParSpec works completely locally; no network interaction is needed.

When a client issues a read-only transaction T_r, the library targets one node in the system and delivers $REQ(T_r)$ directly to that node, without ordering the request globally.

3. MIMOX

MiMoX is a network system that ensures total order of messages across remote nodes. It relies on Multi-Paxos [15], an algorithm of the Paxos family, which guarantees agreement on a sequence of values in the presence of faults (i.e., total order). MiMoX is sequencer-based – i.e., one elected node in the system, called the *leader*, is responsible for defining the order of the messages.

MiMoX provides the APIs of Optimistic Atomic Broadcast [12]: `broadcast(m)`, which is used by clients to broadcast a message m to all nodes; `final-delivery(m)`, which is used for notifying each replica on the delivery of a message m (or a batch of them); and `opt-delivery(m)`, which is used for early-delivering a previously broadcast message m (or a batch of them) before the `final-delivery(m)` is issued.

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

3.1 Ordering Process

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

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

Each node, upon receiving the opt-batch, immediately triggers the optimistic delivery for it. As in [17], we believe that within a data-center the scenarios where the leader crashes or becomes suspected are rare. If the leader is stable for at least the duration of the final-batch's agreement, then even if the opt-batch is received out-of-order with respect to other opt-batches sent by the leader, this possible reordering is still nullified by the ordering information (i.e., #Seq) stored within each opt-batch.

After sending the opt-batch, MiMoX loops again serving the next opt-batch, until the completion of the final-batch. When ready, MiMoX uses the Multi-Paxos algorithm for establishing an agreement among nodes on the final-batch.

The leader *proposes* an order for the final-batches, to which the other replicas reply with their agreement – i.e., *accept* messages. When a majority of agreement for a proposed order is reached, each replica considers it as *decided*.

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

3.2 Handling Faults and Re-transmissions

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

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

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

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

3.3 Evaluation

We evaluated MiMoX's performance by an experimental study. We focused on MiMoX's scalability in terms of the system size, the average time between optimistic and final delivery, the number of requests in opt-batch and final-batch, and the size of client requests. We used the *PRObE* testbed [7], a public cluster that is available for evaluating systems research. Our experiments were conducted using 19 nodes (tolerating up to 9 failures) in the cluster. Each node is equipped with a quad socket, where each socket hosts an AMD Opteron 6272, 64-bit, 16-core, 2.1 GHz CPU (total 64-cores). The memory available is 128GB, and the network connection is a high performance 40 Gigabit Ethernet.

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

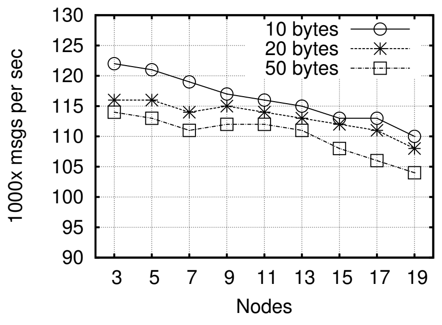


Figure 1: MiMoX's message throughput.

Figure 1 shows MiMoX's throughput in requests ordered per second. For this experiment, we varied the number of nodes participating in the agreement and the size of each request. Clearly, the maximum throughput (122K requests ordered per second) is reached when the node count is low (3 nodes). However, the percentage of degradation in performance is limited when the system size is increased: with 19 nodes and request size of 10 bytes, the performance decreases by only 11%.

Figure 1 shows also the results for request sizes of 20 and 50 bytes. Recall that ARCHIE's transaction execution process leverages the ordering layer only for broadcasting the transaction ID (e.g., method or store-procedure name), along with its parameters (if any), and not the entire transaction business logic. Other solutions, such as the DUR scheme, use the total order layer for broadcasting the transaction read- and write-set after a transaction's completion, resulting in larger request size than ARCHIE's. In fact, our evaluations with Bank and TPC-C benchmarks revealed that almost all the transaction requests can be compacted between 8 and 14 bytes. MiMoX's performance for a request size of 20 bytes is quite close to that for 10 byte request size. We observe a slightly larger gap with 19 nodes and 50 byte request size, where the throughput obtained is 104K. This is a performance degradation lesser than 15% with respect to the maximum throughput. This is because, with smaller requests (10 or 20 bytes), opt-batches do not get filled to the maximum size allowed, resulting in smaller network messages. On the other hand, larger requests (50 bytes) tend to fill batches sooner, but these bigger network messages take more time to traverse.

Figure 2 shows MiMoX's delay between the optimistic and the relative final delivery, named overlapping time. This experiment is the same as that reported in Figure 1. MiMoX achieves a stable overlapping time, especially for a request size of 10 bytes, of ≈8 *msec*. This delay is non-negligible if we consider that ARCHIE processes transactions locally. Using bigger requests, the final-batch becomes ready sooner because less requests fit in one final-batch. As a result, the

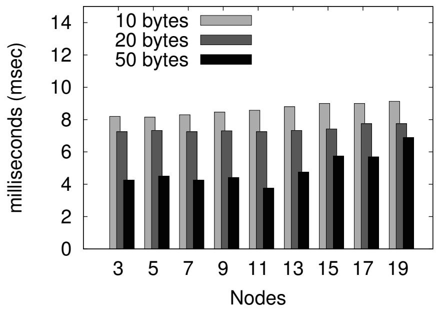


Figure 2: Time between optimistic/final delivery.

time between optimistic and final delivery decreases. This is particularly evident with a request size of 50 bytes, where we observe an overlapping time that is, on average, 4.6 *msec.*

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

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

Table 1: Size of requests, batches, and % reorders.

Table 1 shows other information collected from the previous experiments. It is interesting to observe the number of opt-batches that makes up a final-batch (5 on average) and the number of client requests in each opt-batch (varies from 90 to 230 depending on the request size). This last information confirms the reason for the slight performance drop using requests of 50 bytes. In fact, in this case each opt-batch transfers ≈ 4500 bytes in payload, as compared to ≈ 2300 bytes for request size of 10 bytes.

In these experiments, we used TCP connections for sending opt-batches. Since MiMoX uses a single thread for sending opt-batches and for managing the final-batch, reorders between optimistic and final deliveries cannot happen except when the leader crashes or is suspected. Table 1 supports this. It reports the maximum reordering percentages observed when leader is stable (column *Re NF*) and when the leader is intentionally terminated after a period of stable execution (column *Re F*), using 19 nodes.

4. PARSPEC

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

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

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

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

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

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

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

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

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

Speculative active transactions make available new versions of written objects only when they x-commit. This way, other speculative transactions cannot access intermediate snapshots of active transactions. However, when `MaxSpec` transactions are activated in parallel, multiple concurrent writes on the same object could happen. When those transactions reach their x-commit phase, different speculative versions of the same object could be available for readers. As an example, consider four transactions $\{T_1,T_2,T_3,T_4\}$ that are optimistically delivered in this order. T_1 and T_3 write to the same object O_a, and T_2 and T_4 read from O_a. When T_1 and T_3 reach the speculative commit phase, they make two speculative versions of O_a available: $O_a^{T_1}$ and $O_a^{T_3}$. According to the optimistic order, T_2's read should return $O_a^{T_1}$ and T_4's read should return $O_a^{T_3}$. Even though this approach maximizes concurrency, its implementation requires traversing the shared lists of transactional meta-data, resulting in high transaction execution time and low performance [1].

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

In the following, ParSpec's operations are detailed.

4.1 Transactional Read Operation

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

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

4.2 Transactional Write Operation

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

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

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

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

4.3 X-Commit

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

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

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

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

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

For each written object, a new committed, but not yet visible, version is added to the object's version list. The visibility of this version is implemented leveraging *SCTS*. Specifically, *SCTS* is assigned to the *VCTS* of the version. *SCTS* is always greater than *CTS* because the speculative execution always precedes the final commit. This way (as we will show in Section 4.6) no non-speculative transaction can access that version until *CTS* is equal to *SCTS*. If the MiMox's leader is stable in the system (i.e., the optimistic order is reliable), then when *CTS* reaches the value of *SCTS*, then the speculative transaction has already been executed, validated and all of its versions are already available to non-speculative transactions.

4.4 Commit

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

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

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

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

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

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

4.5 Abort

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

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

4.6 Read-Only Transactions

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

5. CONSISTENCY GUARANTEES

Archie ensures 1-Copy Serializability [2] as a global property, and it ensures also that any speculative transaction (active, x-committed and aborted) always observes a serializable history, as a local property.

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

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

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

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

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

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

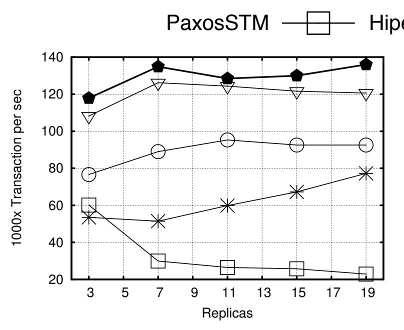
6. IMPLEMENTATION AND EVALUATION

We implemented ARCHIE in Java: MiMoX's implementation inherited JPaxos's [21, 14] software architecture, while ParSpec has been built from scratch. As a testbed, we used PRObE [7] as presented in Section 3. ParSpec does not commit versions on any stable storage. The transaction processing is entirely executed in-memory while fault-tolerance is ensured through replication.

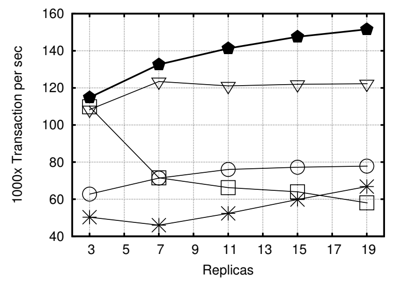
We selected three competitors to compare against ARCHIE. Two are state-of-the-art, open-source, transactional systems based on state-machine replication. One, PaxosSTM [26] implements the DUR model, while the other, HiperTM [10], complies with the DER model. As the third competitor, we implemented the classical DER scheme (called SM-DER) [23], where transactions are ordered through JPaxos [14] and processed in a single thread after the total order is established.

PaxosSTM [26] processes transactions locally, and relies on JPaxos [14] as a total order layer for their global certification across all nodes. On the other hand, HiperTM [10], as ARCHIE, exploits the optimistic delivery for anticipating the work before the notification of the final order, but it processes transactions on single thread. In addition, HiperTM's ordering layer is not optimized for maximizing the time between optimistic and final delivery.

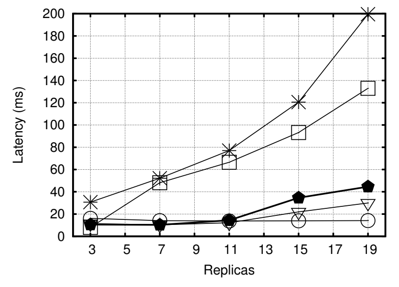
Each competitor provides its best performance under different workloads, thus they represent a comprehensive selection to evaluate ARCHIE. Summarizing, PaxosSTM ensures high-performance in workloads with very low contention, such that remote aborts do not kick-in. HiperTM, as well as SM-DER, are independent from the contention because they process transactions using a single thread but their



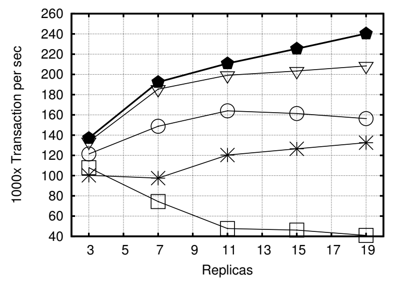
(a) Throughput - High - 90%.



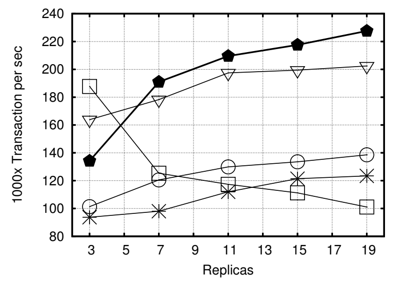
(b) Throughput - Medium - 90%.



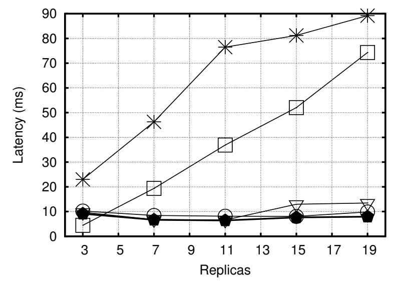
(c) Latency - High - 90%.



(d) Throughput - High - 50%.

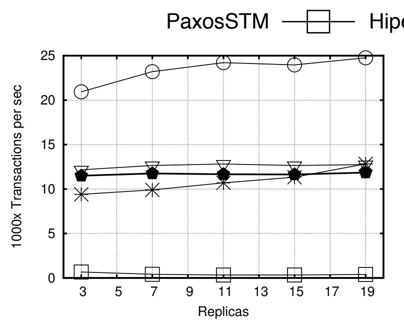


(e) Throughput - Medium - 50%.

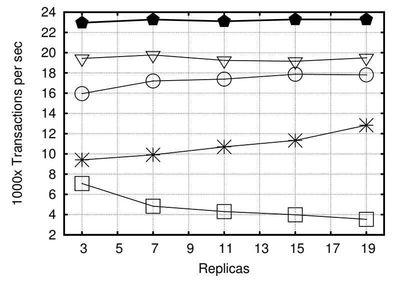


(f) Latency - High - 50%.

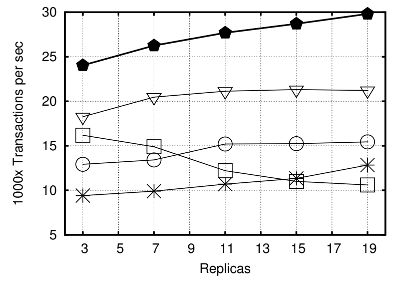
Figure 3: Performance of Bank benchmark varying nodes, contention and percentage of write transactions.



(a) #1 warehouse.



(b) #19 warehouses.



(c) #100 warehouses.

Figure 4: Performance of TPC-C benchmark varying nodes and number of warehouses.

performance is significantly affected by the length of transactions (any operation is on the transaction's critical path). This way, workloads composed of short transactions represent their sweet spot. In addition, SM-DER excels for workloads where contention is very high. Here the intuition is that, if only few objects are shared, then executing transactions serially without any overhead is the best solution.

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

The benchmarks adopted in this evaluation include Bank, a common benchmark that emulates bank operations, TPC-C [5], a popular on-line transaction processing benchmark, and Vacation, a distributed version of the famous application included in the STAMP suite [4]. We scaled the size of the system in the range of 3-19 nodes and we also changed the system's contention level by varying the total number of shared objects available. All the competitors benefit from the execution of local read-only transactions. For this reason we scope out read-only intensive workloads. Each node has a number of clients running on it. When we increase the nodes in the system, we also slightly increase the number of clients accordingly. This also means that the concurrency and (possibly) the contention in the system moderately increase. This is also why the throughput tends to increase for those competitors that scale along with the size of the system. In practice, we used on average the following total number of application threads balanced on all nodes: 1000 for TPC-C, 3000 for Bank, and 550 for Vacation.

6.1 Bank Benchmark

We configured the Bank benchmark for executing 10% and 50% of read-only transactions, and we identified the high, medium and low contention scenarios by setting 500, 2000, and 5000 total bank accounts, respectively. We report only the results for high and medium contention (Figure 3) because the trend in low contention scenario is very similar to the medium contention though with higher throughput.

Figure 3(a) plots the results of the high contention scenario. PaxosSTM suffers from a massive amount of remote aborts (≈85%), thus its performance is worse than others and it is not able to scale along with the size of the system. Interestingly, SM-DER behaves better than HiperTM because HiperTM's transaction execution time is higher than SM-DER's due to the overhead of operations' instrumentation. This is particularly evident in Bank, where transactions are short and SM-DER's execution without any overhead provides better performance. In fact, even if HiperTM anticipates the execution leveraging the optimistic delivery, its validation and commit after the total order nullify any previous gain. We observed also the time between the optimistic and final delivery in HiperTM to be less than 1 *msec*, which limits the effectiveness of its optimistic execution.

The two versions of ARCHIE perform better than others but still ARCHIE-FD, without the speculative execution, pays a penalty in performance around 14% against ARCHIE. This is due to the effective exploitation of the optimistic delivery. Consistently with the results reported in Section 3, we observed an average time between optimistic and final delivery of 8.6 *msec*, almost 9× longer than HiperTM. However, as showed in Figure 3(c), ARCHIE's average transaction latency is still much lower than others. The peak throughput improvement over the best competitor (i.e., SM-DER) is 54% for ARCHIE and 41% for ARCHIE-FD.

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

Figures 3(d), 3(e), 3(f) show the results with higher percentage of read-only transactions (50%). Recall that all protocols exploit the advantage of local processing of read-only transactions but absolute numbers are higher than before, as well as latency is reduced, but the trends are still similar.

6.2 TPC-C Benchmark

TPC-C is characterized by transactions accessing several objects and the workload has a contention level usually higher than other benchmarks (e.g., Bank). The mix of TPC-C profiles is the same as the default configuration, thus generating 92% of write transactions. We evaluated three scenarios, varying the total number of shared warehouses (the most contented object in TPC-C) in the range of {1,19,100}. With only one warehouse, all transactions conflict each other (Figure 4(a)) thus SM-DER behaves better than other competitors. In this case, the parallel execution of ARCHIE is not exploited because transactions are always waiting for the previous speculative transaction to x-commit and then start almost the entire speculative execution from scratch. Increasing the number of nodes, HiperTM behaves better than ARCHIE because of minimal synchronization required due to the single thread processing. However, when the contention decreases (Figure 4(b)), ARCHIE becomes better than SM-DER by as much as 44%. Further improvements can be seen in Figure 4(c) where contention is much lower (96% of gain).

ARCHIE is able to outperform SM-DER when 19 warehouses are deployed, because it bounds the maximum number of speculative transactions that can conflict each other (i.e., `MaxSpec`). We used 12 as `MaxSpec`, thus the number of possible transactions that can conflict with each other is less than the total number of shared objects, thus reducing the abort percentage from 98% (1 warehouse) to 36% (19 warehouses) (see also Figure 6). Performance of SM-DER worsens from Figure 4(a) to Figure 4(b). Although it seems counterintuitive, it is because, with more objects, the cost of looking up a single object is less than with 19 objects.

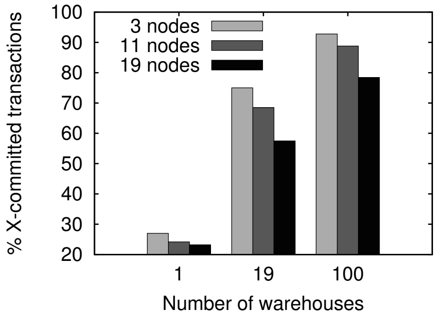


Figure 5: % of x-committed transactions before the notification of the total order.

Figure 5 shows an interesting parameter that helps to understand the source of ARCHIE's gain: the percentage of speculative transactions x-committed before their total order is notified. It is clear from the plot that, due to the high contention with only one warehouse, ARCHIE cannot exploit its parallelism thus almost all transactions x-commit after their final delivery is issued. The trend changes by increasing the number of warehouses. In the configuration with 100 warehouses, the percentage of x-committed transactions before their final delivery is in the range of 75%-95%. The performance related to this data-point is shown in Figure 4(c) where ARCHIE is indeed the best, and the gap with respect to ARCHIE-FD increased up to 41%.

Figure 6 reports the percentage of aborted transactions of the only two competitors that can abort: PaxosSTM and ARCHIE. PaxosSTM invokes an abort when a transaction does not pass the certification phase, while ARCHIE aborts a transaction during the speculative execution. Recall that, in PaxosSTM, the abort notification is delivered to the client, which has to re-execute the transaction and start again a new global certification phase. On the other hand, ARCHIE's abort is locally managed and the re-execution of the speculative transaction does not involve any client operation, thus saving time and network load. In this plot, we vary the number of nodes in the system and, for each node,

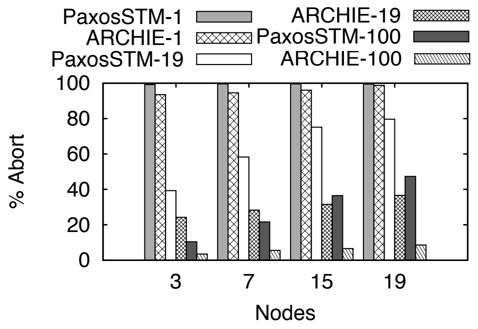


Figure 6: Abort % of PaxosSTM and Archie.

we show the observed abort percentage changing with the number of warehouses as before. The write intensive workload generates a massive amount of aborted transactions in PaxosSTM while in Archie, thanks to the speculative processing of `MaxSpec` transactions at a time, the contention does not increase significantly. The only case where Archie reaches 98% is with only one shared warehouse.

6.3 Vacation Benchmark

The Vacation Benchmark is an application originally proposed in the STAMP suite [4] for testing centralized synchronization schemes and often adopted in distributed settings (e.g., [26]). It reproduces the behavior of clients that submit booking requests for vacation related items.

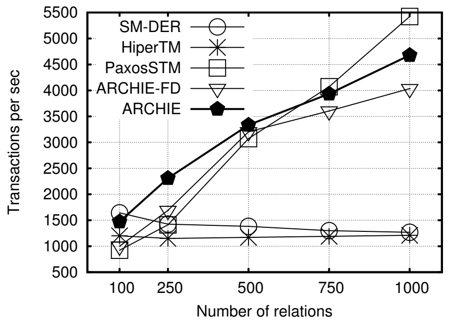


Figure 7: Throughput of Vacation benchmark.

Vacation generates longer transactions than the other considered benchmarks, thus also its total throughput is lower. Figure 7 shows the results. In this experiment we varied the total number of *relations* (object used for defining the contention in the system) and we fixed the number of nodes to 11. Vacation's clients do not perform any read-only transaction, however those transactions can still occur as a result of unsuccessful booking requests. However, the actual number of read-only transactions counted is less than 3%, thus their impact on performance is very limited.

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

7. RELATED WORK

The original Paxos algorithm for establishing agreement among nodes in the presence of failures was presented in [15], and later optimized in several works, e.g., [21, 14, 3].

Optimistic delivery has been firstly presented in [12], and later investigated in [18, 10, 17, 20]. The work in [18] exploits optimistic delivery and proposes an adaptive approach to different networks models. The ordering protocol proposed in [17] is the first that ensures no-reordering between optimistic and final delivery in case of stable leader by relying on a network with a ring topology. Archie provides the same property but with a generic network.

S-Paxos [3] introduced the idea of offloading the work for creating batches from the leader and distributing it across all nodes. In contrast, MiMoX is the first to exploit the time needed for creating the batch for maximizing the delay between the optimistic and final deliveries; and to take advantage of the process of breaking down a single batch into multiple batches for ensuring a reliable optimistic delivery.

Transactional replication based on atomic primitives has been widely studied in the last several years [12, 10, 13, 26, 22]. Some of them focus on partial replication [22], while others target full replication [10, 13, 26]. Archie guarantees higher performance than partial replication approaches by avoiding remote communication in the transaction's critical path. In addition, partial replication protocols are affected by application locality: when transactions are mostly accessing remote objects instead of local, the performance of the local concurrency control becomes negligible when compared with network delays. By exploiting full replication, Archie's behavior is independent of application locality.

Calvin [25] provides a transactional sequencing and replication layer over a partitioned storage system. Calvin defines the order of transactions and enforces this order using single-threaded lock acquisition over objects. Calvin waits for the final order of requests before starting execution on requests. In contrast, Archie uses optimistic delivery to execute transactions in parallel with the coordination phase.

Eve [11] is a replicated transactional system that proposes the *execution-verify* approach. Roughly, Eve inherits the benefits from the DUR model, while falling back to the DER approach when the result of the speculative execution is not compliant with other remote executions. In this case, transactions are re-executed with a predefined order and serially committed. In high contention scenarios, most speculative executions could be irreconcilable and Eve does not provide s specific solution to preserve high performance. ParSpec solves this issue and could be integrated with Eve for speeding up the performance of this fall-back case.

Recently, a redesign of the state-machine approach, called P-SMR, has been proposed in [16]. The motivation of both Archie and P-SMR is similar: both want to increase the parallelism in state-machine replication. According to [16], Archie can be classified as sP-SMR (*semi-parallel state-machine replication*) because the total order deliveries are sequential, even though the processing of independent transactions is concurrent. P-SMR provides its best performance when clients submit independent transactions (or commands), mostly read-only, while Archie is specifically designed for write-intensive scenarios where conflicts are not rare but, thanks to the `MaxSpec`, the contention is still kept limited. In addition, P-SMR partitions objects, thus transactions spanning on multiple partitions could hamper performance.

Archie does not suffer from those transactions because, even though they could introduce a serialization point, it is limited to only `MaxSpec` threads. On the other hand, in P-SMR read-only transactions read "fresher" versions of accessed data than in Archie because they are linearized among other conflicting transactions. P-SMR, in order to implement parallelism, involves the programmer in the definition of conflicting transactions; Archie does not require any application knowledge because ParSpec figures out conflicts at run time, keeping the programmer out of the loop.

Rex [9] is a fault-tolerant replication system where all transactions are executed on a single node, thus retaining the advantages of pure local execution, while traces are collected reflecting the transaction's execution order. Then, Rex uses consensus to make traces stable across all replicas for fault-tolerance (without requiring a total order).

8. CONCLUSION

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

Acknowledgment

The authors thank Fernando Pedone and all anonymous reviewers for their important comments. This work is supported in part by US National Science Foundation under grant CNS-1217385.

9. REFERENCES

[1] J. a. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui. Unifying thread-level speculation and transactional memory. In *Middleware*, pages 187–207, 2012.

[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*, pages 111–120, 2012.

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, pages 35–46, Sept 2008.

[5] T. Council. TPC-C benchmark. 2010.

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

[7] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. Probe: A thousand-node experimental cluster for computer systems research. volume 38, June 2013.

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

[9] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: replication at the speed of multi-core. In *EuroSys*, pages 11:1–11:14, 2014.

[10] S. Hirve, R. Palmieri, and B. Ravindran. HiperTM: High Performance, Fault-Tolerant Transactional Memory. In *ICDCN*, pages 181–196, 2014.

[11] M. Kapritsos, Y. Wang, V. Quema, A. Clement, Alvisi, and Dahlin. All about Eve: execute-verify replication for multi-core servers. In *OSDI*, pages 237–250, 2012.

[12] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE TKDE*, 15(4):1018–1032, 2003.

[13] T. Kobus, M. Kokocinski, and P. T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *ICDCS*, pages 286–296, 2013.

[14] 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.

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

[16] P. J. Marandi, B. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *ICDCS*, pages 368–377, 2014.

[17] P. J. Marandi, M. Primi, and F. Pedone. High performance state-machine replication. In *DSN*, pages 454–465, 2011.

[18] R. Palmieri, F. Quaglia, and P. Romano. OSARE: Opportunistic speculation in actively replicated transactional systems. In *SRDS*, pages 59–64, 2011.

[19] S. Peluso, P. Romano, and F. Quaglia. SCORe: A scalable one-copy serializable partial replication protocol. In *Middleware*, pages 456–475, 2012.

[20] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues. An optimal speculative transactional replication protocol. In *ISPA*, pages 449–457, 2010.

[21] N. Santos and A. Schiper. Tuning paxos for high-throughput with batching and pipelining. In *ICDCN*, pages 153–167, 2012.

[22] N. Schiper, P. Sutra, and F. Pedone. P-Store: Genuine Partial Replication in WAN. In *SRDS*, pages 214–224, 2010.

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

[24] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[25] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

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