Speculative Client Execution in Deferred Update Replication

Balaji Arun, Sachin Hirve, Roberto Palmieri, Sebastiano Peluso, Binoy Ravindran
Virginia Tech
{ba2669,hsachin,robertop,peluso,binoy}@vt.edu

ABSTRACT

Deferred Update Replication (DUR) is a powerful replication technique that allows parallelism of clients' execution while a global certification phase checks the validity of the transactional execution against workloads running on remote nodes. The well-known favorable scenario of DUR is when remote transactions rarely conflict with each other. In this paper we show that, even in this case, the conflicts happening among local application threads can significantly decrease performance. We address this problem by using speculation. We let local transactions propagate their post-execution snapshot to other local transactions before the outcome of the global certification is notified. This way, in scenarios where accesses are partitioned across nodes, we prevent local transactions from aborting each other. Through experimental study based on well-known transactional benchmarks we assess the effectiveness of the approach, gaining more than 10× using TPC-C benchmark.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

Deferred Update Replication, Speculation, Fault Tolerance

1. INTRODUCTION

Replication is a widely used technique to ensure fault-tolerance of transactional systems. When *full-replication* is adopted (i.e., all shared objects are replicated across all nodes), the correctness of transactions started on remote nodes is usually guaranteed through a protocol that relies on a *total order* layer (e.g., Paxos [12]). This layer ensures that each sent message is delivered by all correct nodes (i.e., nodes that did not fail) in the same order. This way each node can locally test the correctness of its executed transactions against all concurrent transactions in the system.

The *deferred update replication* (DUR) [15] is a well-known scheme where transactions execute locally and their commit phase (including the transaction validation procedure) is deferred until a total order [12] among all nodes is established. This total order is required because it imposes a common serialization order among all transactions in the system, which is used to verify the global correctness of transactions' execution. In fact before commit, each transaction has to undergo a *certification phase* where the transaction validates the consistency of its read operations, performed during the execution, against write operations done by other concurrent transactions in the system. To accomplish this task, a total order is leveraged so that all nodes know a unique order to follow while performing the certification. If the snapshot observed is still consistent, then the transaction can safely commit by updating the shared state with its written objects. The sequence of commit necessarily matches the global total order.

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

As an example, consider TPC-C [6], the classical transactional benchmark widely used for evaluating distributed synchronization protocols. Most TPC-C transactions access a `warehouse` before performing other operations. The usual deployment of TPC-C is to pin one (or a set of) `warehouse` to each node and let transactions generated on that node to likely request that warehouse. This configuration, which is representative of several applications with well-partitioned accesses, matches DUR's needs in terms of few remote aborts, but it also reduces the parallelism of local application threads due to conflicts.

A generally adopted technique for preventing a local trans-

.

action T_2 from invoking a certification phase if it conflicts with another local transaction T_1, which is already completed, is to validate T_2 locally after its completion. This way, T_2 is able to detect the conflict with T_1 and thus it can abort immediately without burdening the global certification layer (i.e., the distributed software component that provides the total order and validates/commits transactions) with additional (and futile) work. In fact, T_2 is already doomed even if no remote transaction conflicts with it. In this paper we propose a different approach to solve the above problem. We do not abort T_2 but rather we allow it to speculatively read [9, 14, 13] from the snapshot generated by the execution of T_1. Clearly the whole execution of T_2 depends on the eventual commit of T_1 before T_2 itself, however, in case of well-partitioned accesses, this will likely be the case.

In other words, we define an execution order of local transactions and we propagate the state changes made by one execution to another along the chain of subsequent conflicting transactions, according to the predefined execution order. In addition, transactions from one node are submitted to the global certification layer in the same order as they are executed and we do not allow the final agreed total order to subvert that order. According to the aforementioned example, T_2 will be successfully certified and thus committed because the T_1's snapshot, which T_2 speculatively accessed, will be committed before T_2's certification.

Our approach can be summarized in two high-level guidelines, which can be successfully applied to existing DUR-based protocol for increasing their performance further:

- *Local Transaction Ordering.* All transactions executing on one node should be processed according to a local order. It is worth to note that this order is not necessarily known (or pre-determined) before starting the transaction execution, rather it could be determined while transactions are executing taking into account their actual conflicts.
- *Local Certification Ordering.* Each node should submit completed transactions to the certification layer in the same order as they are speculatively (locally) processed and it should not allow the global ordering layer to change this (partial) order. This way, the local transaction ordering is always compliant with the final commit order, thus making the speculative execution effective.

In this paper we applied our speculative approach to PaxosSTM [19], a state-of-the-art, high performance and open-source DUR protocol, presenting X-DUR. X-DUR embeds a set of design choices for simplifying its implementation, while still showing significant and promising gains with respect to the original, non-speculative version.

We evaluate X-DUR using three well-known transactional benchmarks such as Bank (a monetary application), TPC-C [6] (popular on-line transaction processing benchmark), and Vacation (a distributed version of the famous application included in the STAMP suite [4]). As testbed we use up to 23 nodes available on PRObE [7], a state-of-the-art public cluster. Results reveal X-DUR's benefits, especially when the contention in the system is high, thus saving local aborts. As an example of our findings, the maximum speed-up observed when running TPC-C is higher than one order of magnitude against the original PaxosSTM.

X-DUR is the first protocol that applies speculation to clients' transactions for handling local contention in a DUR-based scheme.

2. RELATED WORK

The DUR model has been proposed in [15] and further investigated in a number of subsequent works [17, 19, 10].

Speculation is a widely used technique for anticipating work based on uncertain inputs [3, 14, 16, 16, 1]. Some of these works process transactions speculatively along different serialization orders [14, 3], whereas others [16, 1] fix a single order at the beginning and follow it. This intuition is the same as X-DUR but the deployment is different. X-DUR speculates on the processing of client transactions before invoking the global certification layer (or the ordering layer as in [16, 14]). Finally [1] applies speculation to parallelize different parts of a transactions.

The protocols presented in [18, 5] improve the performance of DUR-based protocols by leveraging well-partitioned accesses for speeding up the certification phase. This way, if conflicts are rare, transactions can be validated and committed in parallel, improving performance significantly. On the other hand, X-DUR provides a solution only for optimizing contention among local transactions, thus can be applied on top of [18, 5] to further enhance their performance.

Finally, we consider Specula [16] as the closest approach to X-DUR. In Specula each application thread commits speculatively and goes ahead executing next transactions assuming the success of the certification of the previous speculatively committed transactions. Specula differs from X-DUR because it does not enforce an order between transactions speculatively committed by different threads, thus if they conflict each other, one of the executions is still aborted. X-DUR's approach can enhance Specula by introducing some coordination among application threads so that an order can be imposed and those aborts can be avoided.

Another approach pursuing the same goal of X-DUR is in [20], where transactions are assigned to nodes in a way such that conflicting transactions are likely delivered on the same node. Subsequently, within a node transactions are ordered using a lock-based concurrency control.

3. SYSTEM AND EXECUTION MODEL

We assume a distributed system, where a set of processes 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 [12] $Q{=}f+1$. Further, we consider only non-byzantine faults.

The transactional application executing on top of our system is composed of multiple threads balanced on all nodes. According to the certification-based replication scheme [15], each thread activates and executes transactions at the same node where it is running, recording objects read from and written to in private spaces called the *read-set* and the *write-set*, respectively. When a transaction reaches the stage where all of its operations have been executed, the executing thread simply waits until the transaction is globally validated and then is either committed or aborted. This decision is deterministic on all the nodes, including the node where the

transaction started and executed, due to the total order enforced by the certification layer. Once the node recognizes the transaction's result, it informs the relative application thread. If the outcome is *commit*, then the thread can go ahead and perform subsequent work. On the other hand, if the outcome is *abort*, then the application thread has to re-issue the transaction from its very beginning.

4. X-DUR

In this section we detail X-DUR, our speculative DUR protocol, which allows client transactions to speculatively read from uncommitted (but completed) snapshots[1] generated by other local transaction executions already submitted to the certification layer. This way, concurrent transactions running on a single node are already serialized with the same order they are certified, thus canceling any possible abort due to local contention. This approach becomes particularly effective in case of well-partitioned accesses, where transactions submitted by different nodes are unlikely to conflict with each other.

4.1 Speculative Execution

X-DUR implements the local transaction ordering by forcing a predefined order of client transactions. A number of application threads execute in parallel on each node and their issued transactions are scheduled by a single-threaded X-DUR executor. Its role is twofold: *i)* it assigns the speculative serialization order to new transactions according to their arrival order; *ii)* it executes them enforcing the local transaction order.

When a transaction completes its speculative execution, it makes its written objects available as speculative committed versions to other subsequent (according to the local transaction ordering) local speculative transactions. In order to accomplish this task, each shared object includes the last speculatively written value besides the last committed one. We do not need to keep more than one speculative value per object because the execution is single threaded thus only the value written by the last speculative execution needs to be visible. Other speculatively modified objects are kept in the write-set of those already completed transactions that are waiting for their global certification phase.

X-DUR's speculative execution is effective if the certification layer does not subvert the local transaction order used for the speculative execution. In case this happens, then transactions will be serialized by the certification layer in a different way, thus the speculative execution will likely not be committed and an abort signal will be issued to the application thread. In order to prevent this scenario, each node enqueues certification requests in a batch such that the order in which they appear in the batch matches the local transaction execution order. This batch is then sent to all nodes through the global certification layer. This way, the total order layer cannot arbitrarily decide to re-order certification requests coming from a node because the batch becomes the ordering unit, rather than a single message. Batches sent from different nodes are ordered enforcing no specific rule because X-DUR relies on the assumption that accesses are well-partitioned across nodes.

[1]We name *uncommitted snapshot* the whole shared state including uncommitted modifications made by the commit of a speculative transaction.

4.2 Handling Certification Phase

The global certification layer is the distributed component in charge of total ordering certification requests and validating/committing transactions locally. X-DUR inherits the technique for certifying transactions from the DUR model. Specifically, when a batch is delivered, each transaction's read-set and write-set is extracted and certified. The certification consists of validating the read-set against the current (non-speculative) committed versions available and, in case of a successful validation, all speculative written objects are made available to all non-speculative transactions (including the next certification requests).

If on the one hand the speculative execution allows to move forward the transactions' progress in case of partitioned accesses, then on the other hand a remote conflict could inevitably force a possible long chain of speculative conflicting transactions to abort. X-DUR solves this problem by stopping the speculative execution of incoming transactions as soon as a remote abort is detected during the certification phase. In practice, when an abort happens the speculative execution handler forces all new transactions to read from the committed values and start over with a new speculative snapshot.

4.3 Example

In the following we provide an example of how X-DUR works. Consider a replicated system composed of five nodes $N_1, \ldots, N_5$. On top of each node there are three threads deployed for running one transaction each. Say node N_i generate T_1^i, T_2^i, and T_3^i. Assume now that all transactions coming from one node are conflicting with each other and they are speculatively processed following the above order. We can now distinguish between two cases: *(A)* one that implements the fully partitioned accesses, thus no transaction conflicts across different nodes; the other, *(B)*, where there are at least two transactions started on different nodes that conflict (let us call them T_2^i and T_3^k).

The case *(A)* is the sweet spot for X-DUR because each node N_i submits a batch $B_i = \{T_1^i, T_2^i, T_3^i\}$ where any possible order among B_1, B_2 and B_3 is acceptable. In fact, independently from the batches' order, the order of transactions T_1^i, T_2^i, and T_3^i (which are conflicting) will be always preserved. Exploiting this fact, once a batch is received, the validation certainly will succeed and the speculative transaction can be committed. In *(B)*, the batches' order matters because if B_i is ordered before B_k, then T_3^k cannot pass the certification phase and will be aborted. On the contrary, if B_k is serialized before B_i, then the designated victim is T_2^i and, as a consequence, also T_3^i. This is because they conflict with each other and T_3^i speculatively executed a read operation from the modifications made by T_2^i, which is now aborted. In the latter case, on node N_i other conflicting transactions could possibly have already been speculatively committed. If so, T_2^i and T_3^i's abort could have already made their read-set invalid. To prevent new transactions from reading a possible invalid speculative snapshot, the abort of T_2^i forces all new transactions to read from the committed snapshot.

4.4 Correctness

X-DUR ensures 1-Copy Serializability (1CS) [2] as global property. The proof is straightforward because each transaction is deterministically certified in the same order on all

nodes. This means that all transaction executions are validated and committed in the same order on all nodes, even in presence of failures. The speculative execution of X-DUR does not hamper 1CS because during the certification phase all speculative transactions are validated before being committed. If their execution is not consistent (e.g., due to remote conflicts), they are aborted and restarted by reading from the last committed shared state.

5. EVALUATION

We implemented X-DUR in Java, inheriting the PaxosSTM's software architecture [19]. PaxosSTM [19] processes transactions locally, and relies on JPaxos [11] as a total order layer for their global certification across all nodes.

We used the *PRObE* testbed [7], a public cluster that is available for evaluating systems research. Our experiments were conducted using 23 nodes (tolerating up to 11 failures) in the cluster. Each node is equipped with a quad socket, where each socket hosts an AMD Opteron, 16-core, 2.1 GHz CPU. The memory available is 128GB, and the network connection is a high performance 40 Gigabit Ethernet.

As transactional application we leverage Bank, a common benchmark that emulates bank operations, TPC-C [6], a popular on-line transaction processing benchmark, and Vacation, a distributed version of the famous application included in the STAMP suite [4]. Each application is configured for avoiding transactions that remotely conflict with each other (well-partitioned accesses across nodes). On the other hand, within each node we define three configurations, which reflect three different local contention levels: low, medium, high. Each of them differs from others for the total number of shared objects available. Table 1 summarizes all the configurations used. We enforce the well-partitioned accesses by equally dividing the total number of shared objects per node (e.g., with Bank, medium contention and 10 nodes, application threads on one node are allowed to access 200 accounts). Within a node, local accesses are uniformly distributed (not skewed).

Application	Low contention	Medium contention	High contention
Bank (`accounts`)	5000	2000	500
TPC-C (`warehouses`)	230	115	23
Vacation (`relations`)	1000	500	250

Table 1: The details of the Low/Medium/High contention configurations used per benchmark.

Read-only profiles are excluded from the evaluation because those transactions can be run locally exploiting a multiversioned repository and concurrency control, as in [19, 9, 10, 14]. As a result they never abort thus in this case the speculative execution of X-DUR is not required.

All clients (i.e., application threads) in the system are balanced among deployed nodes. In order to avoid changing the load of the system as we increase the number of nodes, we keep the number of clients fixed throughout all experiments. This explains the common shape of lines in the plots. Increasing the size of the system does not change the overall load, thus the performance of all tested configurations degrade due to the higher overhead of the certification layer (e.g., longer broadcast phase, higher number of messages). The best throughput is often reached in the range of 7-15 nodes deployed, where the system is not so large and clients are properly balanced so that local nodes' resources are not saturated (as for the case of 3 nodes). In all experiments (except for Figure 4) the following total numbers of clients are used: 920 for Bank and 550 for TPC-C and Vacation.

For each benchmark we compared the performance of X-DUR and PaxosSTM. Given the well-partitioned accesses, both the systems show high-performance. However, the impact of X-DUR's speculative execution becomes clear when the number of shared objects decreases (i.e., medium/high contention). In these cases, even with the simple speculative single-threaded execution, local transactions are prevented from aborting thus substantially increasing performance.

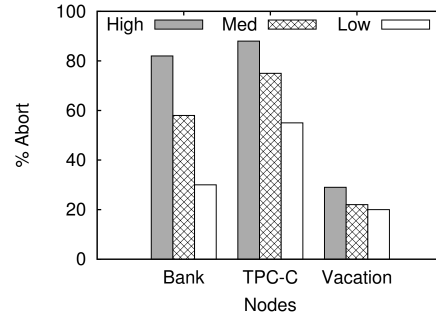

Figure 1: % of aborted transactions on 11 nodes varying the contention level and different benchmarks using PaxosSTM.

In Figure 1 we provide an evidence of how the local conflicts can impact the overall system progress even in case of well-partitioned accesses across nodes. Here we report the percentage of aborted transactions measured using PaxosSTM for different benchmarks and local contention levels. Nodes are fixed to 11. We recall that accesses are well-partitioned thus aborts cannot be due to remote conflicts, rather they only count as local aborts. Among all the benchmarks, TPC-C reveals a higher number of aborted transactions in the high contention scenario (i.e., total of 23 shared `warehouses`, thus the clients of each node are allowed to access 2 (local) `warehouses`).

Figure 2 shows the results using Bank benchmark. As expected under the low contention scenario X-DUR behaves generally worse than PaxosSTM because the number of PaxosSTM's aborted transactions is low thus its parallelism is more effective than X-DUR's single-threaded execution. However, due to the nature of Bank's small transactions, the gap between them is limited to 43%. On the other hand, when we move on the medium and high contention experiments, then X-DUR effectively exploits its speculative execution preventing local transactions from conflicting with each other, showing better performance up to 44% and 2.1× in the medium and high contention, respectively (if we exclude the case of 3 nodes where PaxosSTM's is penalized due to many clients deployed on few nodes). Performance of X-DUR does not significantly vary when we change the number of shared objects due to the serial execution.

Figure 3 shows the average latency perceived by clients measured during the experiments in Figure 2. While the

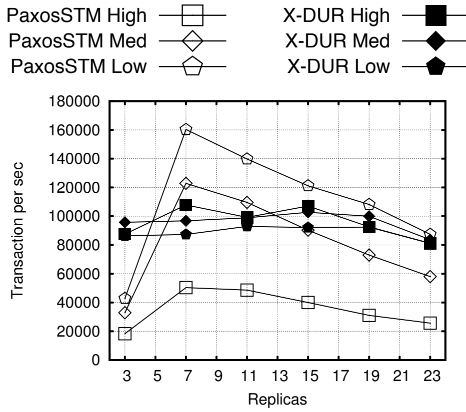

Figure 2: Throughout of PaxosSTM and X-DUR using Bank benchmark.

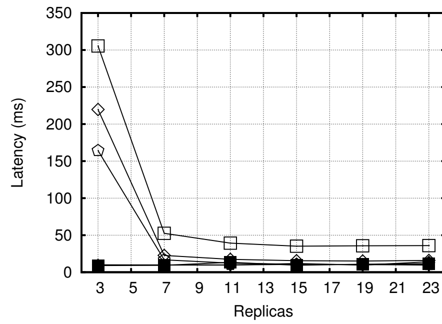

Figure 3: Client perceived latency of PaxosSTM and X-DUR using Bank benchmark.

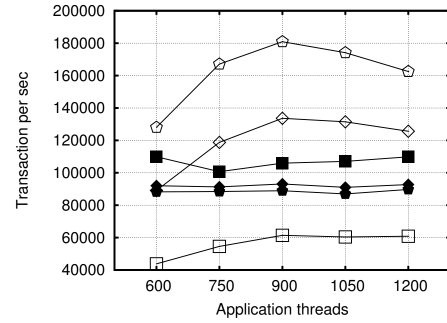

Figure 4: Throughput of PaxosSTM and X-DUR varying the number of clients using 7 nodes running Bank benchmark.

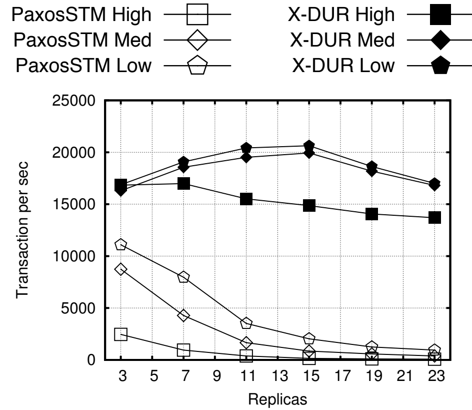

Figure 5: Throughout of PaxosSTM and X-DUR using TPC-C benchmark.

performance of all configurations almost reflect the same (inverse) trend as the throughout reported in Figure 2, PaxosSTM with high contention behaves the worst. Interestingly, the average latency with 3 nodes for PaxosSTM is much higher than X-DUR. This is because the number of clients are fixed thus with a small system size there are too many clients operating per node and this causes repeated conflicts and (local) aborts. X-DUR does not suffer from such a case because clients cannot conflict with each other.

We further leveraged Bank for showing the throughput of PaxosSTM and X-DUR when we fix the number of nodes and we vary the number of application threads (Figure 4). We selected 7 nodes because it represents the highest datapoint in Figure 2. Clients are set in the range of 600 to 1200 (steps of 150). The plot confirms how PaxosSTM's behavior changes when different number of clients are deployed. X-DUR is less affected than its competitor due to the singled-threaded handler for executing incoming transactions. We observe 900 as configuration where all configurations provide the best performance. This is the reason that motivated us to use 900 as total number of clients for Bank.

Throughput and latency of competitors running TPC-C benchmark in all the configurations are shown in Figures 5 and 6, respectively. TPC-C is an application with longer transactions than Bank and with an average higher contention level. This is because the usual deployment of TPC-C suggests to let clients running on a node to likely access only one `warehouse`[2] (in our experiments this happens when we deploy 23 nodes in the high-contention case). In this scenario, the speculation becomes particularly effective because almost all transactions access to the same `warehouse` (or few of them when the size of the system is less than 23). With medium and high contention we allowed clients to access at least five and ten `warehouses` respectively, but still this is not enough for sensibly reducing conflicts among local threads, thus X-DUR gains up to more than one order of magnitude against PaxosSTM in the high conflict scenario.

The last application we used for evaluating our proposal is Vacation. The collected throughput is shown in Figure 7. Vacation is more similar to TPC-C than Bank in terms of composition of transactions, but the overall contention is lower (as in Bank). In fact, as in Bank, the number of shared objects (i.e., `relations`[3]) accessed per node in the well-partitioned accesses configuration is higher than the number of `warehouses` in TPC-C. X-DUR consistently beats the competitor throughout all the tested cases. When the system is more loaded and the network overhead has still a limited impact on performance (e.g., nodes less than 15), we observe the maximum gain of X-DUR against PaxosSTM (i.e., 3.36×) when we configure the benchmark with high contention. With medium and low contention we still

[2]The most important object in TPC-C.

[3]The most important object in Vacation.

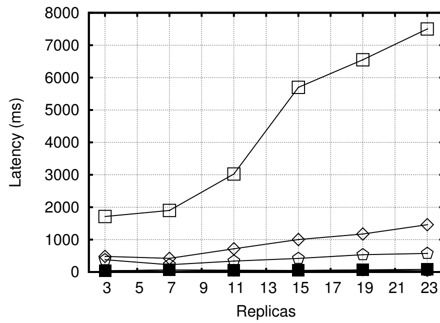

Figure 6: Client perceived latency of PaxosSTM and X-DUR using TPC-C benchmark.

gain up to 2.9× and 2.1× respectively.

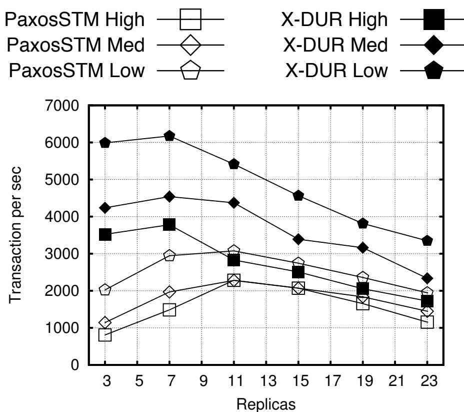

Figure 7: Throughout of PaxosSTM and X-DUR using Vacation benchmark.

6. CONCLUSIONS

In this paper we present an approach to use speculation for sparing DUR-based protocols from aborts due to local contention. According to the DUR model, if transactional accesses are partitioned across nodes, then only local contention can cause a transaction to abort. We propose the usage of speculation for allowing transactions to execute starting from uncommitted snapshots produced by completed transactions waiting for their global certification. In addition, we also make sure that the total order established by the certification layer does not contradict the local speculative order. The peculiarity of our approach is that it is general and fits in several existing DUR-based protocols, thus contributing to further enhance their performance.

7. ACKNOWLEDGMENTS

This work is supported in part by US National Science Foundation under grant CNS-1217385.

8. 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] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber. Speculative out-of-order event processing with software transaction memory. In *DEBS*, pages 265–275, 2008.

[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] N. Carvalho, P. Romano, and L. Rodrigues. Scert: Speculative certification in replicated software transactional memories. In *SYSTOR*, page 10, 2011.

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

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

[10] 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.

[11] 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.

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

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

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

[15] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, July 2003.

[16] S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues. SPECULA: speculative replication of software transactional memory. In *SRDS*, pages 91–100, 2012.

[17] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS*, 2012.

[18] D. Sciascia, F. Pedone, and F. Junqueira. Scalable deferred update replication. In *DSN*, pages 1–12, 2012.

[19] 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.

[20] V. Zuikeviciute and F. Pedone. Conflict-aware load-balancing techniques for database replication. In *SAC*, pages 2169–2173, 2008.