On the Viability of Speculative Transactional Replication in Database Systems: a Case Study with PostgreSQL

Sebastiano Peluso*, Roberto Palmieri†, Francesco Quaglia*, Binoy Ravindran†
Sapienza University of Rome (*), Virginia Tech (†)

***Abstract*—We investigate the feasibility of systematic speculative processing in the context of Optimistic Atomic Broadcast (OAB) based replication of database systems. Specifically, we present the design and prototypal implementation of a fully speculative version of the PostgreSQL open source relational database, together with experimental results showing performance advantages over non-speculative replication.**

I. Introduction

Active replication is a classical means for providing fault-tolerance and high availability. It is based on the enforcement of consensus among the replicas on a common total order for request processing. The establishment of the agreed upon total order is typically demanded to an Atomic Broadcast (AB) group communication primitive, representing a convenient abstraction of consensus [3].

For replication in transaction processing systems, a key optimization technique has been presented in [5]. It is based on the observation that replicas can use the spontaneous network delivery order as an early, although possibly erroneous, guess of the total delivery-order of messages eventually defined via AB. This idea is nicely encapsulated by the Optimistic Atomic Broadcast (OAB) primitive [5], representing a variant of AB where the notification of the final message delivery order is preceded by an early *optimistic message delivery* indication. By activating transaction processing upon the optimistic message delivery event (rather than waiting for the final order to be established), OAB-based replication techniques allow overlapping the (otherwise sequential) replica synchronization and local computation phases.

Early solutions along this path [5] admitted speculative processing of a single optimistically delivered transaction along any chain of conflicting transactions. Hence, no propagation of the output of speculatively processed transactions was allowed. More recent advances have instead been based on the idea to increase the actual level of speculation by (A) allowing propagation of the output of speculatively processed transactions along conflicting-transaction chains (see [7]), and (B) speculatively exploring multiple serialization orders so to increase the likelihood of materializing in-advance the order actually matching the OAB outcome (see [8]). However, at current date, the assessment of replication protocols making systematic use of speculation has been limited to the context of in-memory transactional systems, such as Software Transactional Memories, hence not covering the case of traditional database systems. Also, in most cases, the assessment has been based on simulation studies (see [8], [9]), rather than real implementations.

In this article we complement the aforementioned results by presenting the design and implementation of a speculative version of the well known PostgreSQL open source relational database [1]. This version supports all the capabilities required to put in place the two strategies depicted in the above points (A) and (B). In particular, it allows speculatively produced post-images of the data-items, which are still uncommitted, to be visible to concurrent transactions, thus supporting speculation along chains of conflicting transactions. Also, it allows for concurrently running multiple speculative instances of a same transaction, each one accessing a different speculative snapshot of the database. Hence, multiple serialization orders can be concurrently explored in a speculative fashion. We also report the results of an experimental study, based on TPC-C [11], which quantify the performance improvements wrt replication schemes not making systematic use of speculation.

The remainder of this paper is structured as follows. In Section II literature results related to our study are discussed. In Section III our design choices and the modifications made to the kernel of PostgreSQL are described. Section IV presents the experimental study.

II. Related Work

The work in [2] explores the idea of speculatively executing transactions by exploiting the notion of save-point. Specifically, upon the detection of a conflict, a copy of the current transaction is forked and remains idle, thus acting as the save-point to reduce the cost of transaction aborts. This work is targeted to non-replicated real-time databases, while our focus is on the design/implementation of core mechanisms in support of distributed protocols for replicated transactional systems.

In [10] the post-images of data, whose locks are held by prepared distributed transactions are allowed to be accessed by conflicting transactions, thus reducing the lock-wait time. Differently from this approach, we target replicated transactional systems not relying on distributed locking protocols (rather on the usage of OAB group communication primitives and purely local concurrency control mechanisms).

The proposals in [4], [12] extend the kernel of PostgreSQL by supporting replica control and concurrency control in an integrated manner. However, they do not cope with

inner database supports for speculation, which is instead the target of our work.

The approaches in [7], [8], [9] make systematic use of speculative schemes for local transaction processing activities at each replica, thus aiming at maximizing the overlap between processing and distributed coordination. However, as already hinted, they have been evaluated limitedly to the context of Software Transactional Memory systems. Instead, our focus is on the design/implementation (and evaluation) of speculative replication in the context of relational database systems.

III. The Speculative Version of PostgreSQL

A. Overview of PostgreSQL Internals

PostgreSQL relies on a traditional multiprocess architecture based on a front-end server, called *Postmaster*, which is in charge of accepting client connections. Each incoming connection triggers a new transaction activation, which takes place by spawning a so called *Backend* transactional process that takes care of interacting with the client and of performing all the tasks associated with transaction processing. Each transaction statement is executed by accessing all the tuples matching the statement parameters. This happens by ultimately caching these tuples into an in-memory shared heap, concurrently accessible by all the Backend processes. Each transactional process keeps an in-memory data structure, called TupleTable, which caches the references to the accessed tuples within the shared heap.

PostgreSQL implements a multi-version concurrency control scheme providing Snapshot-Isolation (SI) semantic. This is achieved by creating a new version of a tuple whenever a write operation is executed on it, and by letting any read operation access the most recent version of the target tuple that was already committed at the time the transaction started. At most one uncommitted version of each tuple can exist at any time, which we refer to as the *active* version. Instead, the most recent version generated by a committed transaction is referred to as the *valid* version. To determine tuple visibility, and to detect conflicts, the concurrency control scheme maintains per-tuple meta-data including a couple of transaction identifiers, namely $< t\_xmin, t\_xmax >$. They represent the identifiers of the transactions that created and updated the tuple version, respectively. Accordingly, when a transaction T creates an active version of a tuple, the t_xmax value is set to the special value $null$, while the t_xmax value associated with the valid tuple version is set to T's identifier.

At startup time, each transaction T determines its database *snapshot* as the set including T's own identifier and the identifiers of transactions concurrent with T. The latter include transactions that were already active upon starting up T, plus any transaction possibly activated after T's startup.

The concurrency control mechanism handles read/write operations as follows. Upon read access to a tuple by transaction T, the tuple-versions set is used to retrieve the most recent tuple version committed by a transaction not concurrent with T. This corresponds to the version having maximum t_xmin value among the versions created by committed transactions not concurrent with T. Clearly, the selected tuple might correspond to a version older than the valid one. On the other hand, if the read request is for a tuple previously written by T, it is served by accessing the active version previously created by the same transaction T. Upon write access to a tuple by transaction T, the following version checks are performed: if the valid version was created by a transaction concurrent with T (i.e. t_xmin of the valid version is the identifier of a transaction concurrent with T), the abort of T is immediately forced. Otherwise, if there is no currently active version of the target tuple, T requests an exclusive lock on the valid version, which might lead to a wait phase. On the other hand, in case an active version of the tuple exists, T is queued for future access to the exclusive write lock associated with the tuple. An additional background process, called *Vacuum*, takes care of deleting obsolete tuple versions.

B. Details on the Added Facilities

1) Enhanced Multiversioning: Speculation requires non-blocking tuple access. Also, for speculation along chains of conflicting transactions, the post-images of inserted/updated tuples must be visible before the writing transaction is committed. They must become visible right upon a so called *complete* event [8]. Completion implies that the transaction already issued its read/write operations, and has logically issued the commit command, which needs to be mapped to a non-classical execution semantic, where no immediate commitment of the updated tuples must be performed.

To support both non-blocking tuple access and the completion event of speculatively executed transactions, we have reorganized the data structures implementing the multiversioning scheme by allowing the existence of multiple active versions of each individual tuple. In particular, in the enhanced multiversioning scheme, write-write conflicts, which originally lead to transaction block or abort, are handled as insertions of additional active tuple-versions within the shared heap. Overall, the existence of an active version of a tuple becomes the expression of a speculative INSERT/UPDATE operation performed by a transaction in relation to that tuple. Differently from the original PostgreSQL architecture, where locks and process-wait were adopted in case of conflicting access to the unique active version of a tuple, our pool of multiple active versions is accessed within critical sections relying on spin-locks.

The support for (i) visibility of speculative tuple versions produced by transactions that have reached the complete stage and (ii) speculative deletion of tuple versions, has been based on the introduction of a couple of additional 1-bit flags $< SPEC\_VISIBLE, SPEC\_DELETED >$

within the header of each tuple. Right upon a speculative INSERT/UPDATE of a tuple, the corresponding version added to the pool is marked as $SPEC\_VISIBLE = 0$, with the meaning that the tuple is not yet visible to other transactions. If, and when, the creator transaction reaches the complete stage, the flag $SPEC\_VISIBLE$ is raised, with the meaning that the written version becomes speculatively visible to other transactions. In case a DELETE operation is speculatively executed, the tuple is simply marked with the flag $SPEC\_DELETED$ raised. Hence multiple speculative deletions of the same tuple version will all be collapsed into a single signalling action, and will all be reversible by resetting the $SPEC\_DELETED$ flag. In order to correctly handle the rollback of speculative deletions, a deletion counter has been added to the tuple header. When a speculative deletion occurs, the counter is incremented. When the speculative transaction that deleted the tuple is aborted the counter is decremented, and, if the value zero is reached, the $SPEC\_DELETED$ flag is reset, which expresses that all the speculative deletions have been undone.

Overall, a real deletion of a tuple version will be actuated only in case the corresponding speculatively executed transaction is eventually committed. Also, a speculative version of whichever tuple x is immediately removed by the shared heap upon the abort of the creator transaction. This may cause cascading abort of other speculatively executed transactions as it will be discussed later on.

As for visibility rules over the pool of active versions upon a read operation by a transaction, we have decided not to adopt restrictive approaches, thus not limiting the possibility to speculate by a-priori excluding specific versions. Hence, the visible versions of a tuple x upon read access by transaction T (e.g. via the SELECT statement) include both the valid version (i.e. the latest committed one) and the active versions (not speculatively deleted) that have been written by transactions concurrent with T ([1]). The choice towards maximizing tuple visibility across concurrent speculative transactions is motivated by the aim at providing a general architecture for speculative processing, that could be complemented by plug-in modules implementing specific (more restrictive) visibility rules. As an example, the protocol in [7], which limits speculation to a single serialization order (the one compliant with the current sequence of optimistic deliveries by the OAB service), may impose that each speculative transaction T accesses a single committed/speculative version of a tuple x. This tuple version should be the one produced by the latest transaction T' preceding T within the optimistic delivery sequence, which also wrote x during its execution.

[1]The committed versions preceding the valid version are assumed not to be visible since, as we will point out in Section III-B3, we employ a validation scheme where a speculative transaction is committable only in case it read the valid tuple version.

2) Speculative Transactions Forking: As hinted, the materialization of speculative transaction processing requires non-blocking access to active tuple versions. However, in order to permit the exploration of alternative serialization orders, an additional *forking* mechanism is needed to allow a transaction T, which finds n visible versions of a tuple x, to really explore all the n possibilities, in terms of returned value of the corresponding read operation. The implementation of such a forking mechanism represents an additional modification we made to PostgreSQL.

One problem we had to face is that each child transaction needs to be a logical clone of the parent one, in terms of data access operations executed up to the forking point. However, each of the children needs to figure as a truly independent transaction, with the meaning that the tuples written by the parent cannot be considered as representative copies of the tuples that would have been written by the children up to the forking point. This is because each written version of a tuple by a transaction T is kept within the shared heap as a single copy, whose header only expresses information in relation to the identity and the state of the creating transaction, namely T. When T reaches the complete stage, the tuple version can be marked as $SPEC\_VISIBLE$ independently of whether the children of T are still ongoing transactions. Overall we had to face the problem of enforcing Piece-Wise-Deterministic behavior in terms of tuples read by the children of T, while allowing independent write operations.

To address the above problem, we have augmented the transactional context natively kept by PostgreSQL by introducing a so called *statementSet*. This is a data structure used for logging, in main-memory, all the commands executed by a transaction, and the references to all the tuples that have been accessed while executing each single command. The statementSet is formed by nodes (one per executed command) linked together within a linear list. In practice, the statementSet represents an implementation of an in-memory logging structure for the transaction read and write sets. In fact, each node identifies which tuples have been accessed in either read or write mode within the shared heap as a result of the execution of a given command. Upon forking a new child transaction, the statementSet of the parent is inherited by the child, which uses the inherited references to the tuple versions read by the parent in order to execute a validation phase. In particular, the child transaction checks whether these references are still valid, which means that the tuple versions read by the parent are still present (in the same original state, namely committed vs active) within the shared heap. In the positive case, the child transaction uses the inherited references to the tuples written by the parent in order to make its own copies of these tuple versions (which will be reflected within the shared heap as if they were actually written by the child transaction). After, the statementSet of the child is updated in order to correctly point to the new copies of the tuples logically associated with

those write operations. Overall, real inheritance is actuated only in relation to the read set of the parent transaction. Instead, the child transaction re-delivers its own copies of the written tuples, thus making its write set independent of the parent one. Also, the above validation phase allows avoiding to re-process the statements logged by the inherited statementSet at the side of the child transaction.

On the other hand the validation phase of the inherited read set may fail since some discrepancy may arise while determining whether the tuple versions read by the parent are still actual. In case of discrepancy, it means that the state of the database has changed such in a way that the currently explored serialization order, as seen by the child transaction, is meaningless. This may occur since (A) some other speculative transaction, from which the current child transaction should depend on, has been aborted, thus erasing visibility of its written-tuple versions from the shared heap, (B) some tuple version that should belong to the read set has been speculatively deleted, or (C) some tuple version referenced by the commands in the statemenSet is no more the most recent committed version.

We have also implemented a mechanism for explicitly dealing with cases where the above discrepancy is verified. Specifically, upon validation failure of the read set, the child transaction resets the references currently kept by the statementSet, and speculatively re-processes all the commands that are found to be already present within the inherited statementSet. This will allow the child transaction to speculate along a different serialization order on the basis of the changed state of the database. We note that the re-execution of the commands belonging to the statementSet will allow the child transaction to execute its own write operations (if any), thus generating speculative tuple versions within the shared heap which are explicitly flagged as being created by such child transaction instance.

The statementSet records already executed commands. However, additional commands might be issued to the parent transaction by the client application after the children have already been forked. These commands must be transferred to the children, since they were not yet logged within the statementSet at the time of the fork operation. This has been done via a push mechanism where the parent transfers each new transaction command to the children via socket-pairs.

As a final point, we allow each transaction (the original parent or the forked children) to further actuate forking whenever more than one version is visible upon successive read operations. This approach could be complemented by plug-in modules aimed at controlling (e.g. limiting) the number of speculative instances of each transaction depending on, e.g., the amount of available hardware resources or the current system load.

3) Transaction Demarcation and Commit: As the literature on speculative transactional replication protocols indicates, all the speculative instances of a same transaction T are said to belong to the same *family*. In the extended version of PostgreSQL we had to face the need for supporting families while performing transaction demarcation, and to include a support for committing families of transactions, which means committing a single transaction instance (the one that has been executed along the serialization order that ultimately complies with the OAB finalization).

As for transaction demarcation, we have associated each speculative transaction with a couple of identifiers $< FAMILY\_ID, INSTANCE\_ID >$, indicating the family to which the transaction belongs, and the specific identification code of that transaction within the family. In order to support this choice we had to maintain compliance with the demarcation and transaction identification rules natively adopted by PostgreSQL, since a lot of core sub-systems within the database kernel rely on the traditional, single-identifier rule. To support transaction families transparently to core sub-systems of the database kernel, we have adopted the following rules. As soon as the first instance T of a given transactional request is activated, $INSTANCE\_ID$ is populated via the traditional transaction identifier assigned by PostgreSQL, say y. Given that this identifier is unique, we have decided to use it also to determine the family identifier for all the transactions that will be forked by T. Hence T is identified as $< y, y >$ and any other forked transaction deriving (directly or indirectly) from T will be identified as $< y, z >$, where z is again the native identifier assigned by PostgreSQL upon starting up the transaction. Hence, a commit operation involving the speculative transaction $< y, z >$ will ultimately result in triggering the core database modules by using the transaction identifier z as the commit parameter. We have also added a shared table (implemented as a hash table) maintaining the meta-data related to all the active/committed families. In particular, a flag within the table entry associated with any family indicates whether the family has been already committed, in which case no other speculative transaction instance belonging to that family could ever be activated/committed.

Let us now address the problem of how to determine that a given speculative transaction is committable. This problem comes out since transactions may have observed either committed or active (speculative) data versions during their execution, which is instead not allowed by the blocking SI concurrency control algorithm natively adopted by PostgreSQL. To detect whether a speculatively executed transaction is committable, we have further extended the kernel of PostgreSQL in order to implement a validation scheme where the set of tuple versions read by the transaction is checked to determine whether they currently correspond to the valid versions within the database. Hence, a speculative transaction T is identified as committable if it is determined that each tuple version accessed by the transaction in read mode was (or has become) the latest committed version. We note that this validation rule determines a transaction

commit sequence that mimics a locking-protocol schedule. In fact, if a transaction T reads a tuple version that becomes the valid version prior T reaches the commit point, then the schedule is equivalent to one where T takes a lock on that version, thus excluding the possibility for other transactions to update that version concurrently with the execution of T. However, in our solution, such a schedule is determined a-posteriori of the speculative execution of transactions, with no blocking phase imposed to the execution. When committing a transaction $T =< y, z >$, all the other speculative transactions belonging to the same family are aborted, and the family meta-data are permanently updated.

IV. Experimental Study

PostgreSQL Version 8.1.15, augmented with speculative capabilities, has been run on top of a 64-bit NUMA HP ProLiant server equipped with four 2GHz AMD Opteron 6128 processors (for a total of 32 CPU-cores), 64GB of RAM, and 2 SSD disks, each providing 500 GB storage, organized as a RAID-1 array. The operating system is 64-bit Debian 6, with Linux kernel version 2.6.32.5.

Our test-bed configuration entails a set of TPC-C terminals running transactions via JDBC. The set of tables for one warehouse are maintained at the database side according to TPC-C specifications. Each terminal continuously executes transactions by selecting the business logic from the set of profiles specified by the benchmark. The only relevant difference, with respect to a classical execution of TPC-C (e.g. tailored for the evaluation of stand alone database systems) is that the test-bed has been augmented with an OAB emulation sub-system, which mimics group communication dynamics in a high fidelity fashion. Particularly, this sub-system is in charge of determining the actual OAB delays (between the arrival of the transactional request and its ordering finalization), as seen by the instance of the speculative PostgreSQL database running on top of the used 32-core machine. Thanks to the OAB emulator, the test-bed allows reliable assessment of the performance that would be achieved in case of execution in a real distributed/replicated environment, while enabling experimentation on top of the available multi-core machine.

The OAB emulator determines optimistic and final delivery delays for transactional requests by exploiting available OAB traces related to executions of the Appia GCS Toolkit [6] on a cluster of 4 quad-core machines (2.40GHz - 8GB RAM) connected via a Gigabit Ethernet and using TCP at the transport layer. By the traces, the average delay for the final delivery of a message (which definitely establishes its final order) is about 400 millisec. TPC-C terminals have been equipped with an OAB-stub for integration with the OAB emulator, which allows them to perform the following additional actions (beyond issuing actual database commands): **A1)** Upon starting up a new transaction, the terminal mimics the transmission of a corresponding transactional request via OAB across replicated nodes by delivering a fictitious message to the OAB emulator component. This component immediately replies to the terminal, as if the optimistic delivery of the message was immediately raised to the local sender (which is typical of real OAB implementations), and then schedules the final delivery event for the same message, which mimics the final delivery within OAB and leads the message to be enqueued into a TO (Total Order) queue after a given amount of time. **A2)** Before issuing the commit command, the complete command is raised towards the database. Then the terminal awaits that the corresponding transactional request becomes the top standing one within the TO-queue. When this happens, the commit command is issued. In case of positive outcome, the request gets dequeued from the TO-queue, and the terminal starts processing a subsequent transaction.

To support this test-bed, PostgreSQL has been further modified to allow reporting to the terminal the outcome associated with the whole set of speculatively processed instances of the transaction activated by the terminal. In particular, if at least one speculatively processed instance commits successfully, then the terminal will observe positive outcome, otherwise the observed outcome will be negative and the terminal will restart issuing the same transaction. Also, we extended JDBC drivers for PostgreSQL in order to allow them to correctly treat the complete command.

A similar test-bed, only entailing a few modifications, has been used for determining performance results for the case of traditional non-speculative execution on the original version of PostgreSQL. Particularly, the business logic associated with the terminal is activated only upon the TO-deliver of the corresponding request (namely when this request becomes the top standing one into the TO-queue). Also, upon starting the processing of the transactional logic, the terminal atomically removes the request from the TO-queue (hence allowing a transaction on another terminal to start processing, if the corresponding request becomes the top standing within the TO-queue) and inserts the request into a COMMIT-queue. Right before attempting to commit, the terminal synchronizes with the COMMIT-queue by awaiting that the corresponding request becomes the top standing one. After this occurs, the commit command is actually issued. This leads to scenarios where a kind of concurrency is allowed, since transactions on different terminals are allowed to be activated independently of whether already active transactions exist. On the other hand, they must be committed in a sequentialized fashion, according to the insertion order within the COMMIT-queue, which reflects the TO-deliver order. We note that in case a transaction is aborted, e.g., due to conflict with a concurrently activated transaction, then it is simply retried, which will anyhow lead to eventually commit the transaction according to the TO defined order (since the required tuples are released by the conflicting transactions on the basis of timeouts). Overall,

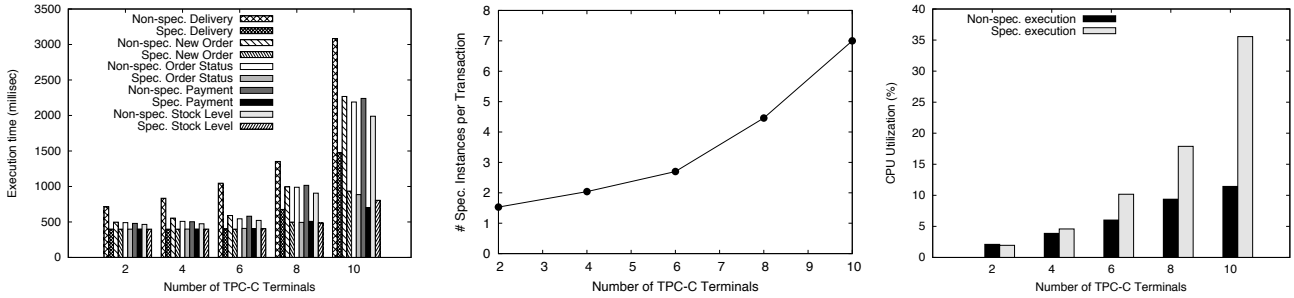

Figure 1: Experimental Results.

this variant of the test-bed organization mimics a typical State Machine (SM) replication approach where transaction commits are sequentialized according to the outcome of the total order group communication scheme, but concurrency is allowed while processing non-conflicting transactions.

In Figure 1 (left) we show the average response time for both speculative and non-speculative execution of the five transaction profiles of TPC-C. The data are reported while varying the number of active terminals from 2 to 10 (we recall that 10 is the maximum number of concurrent terminals per warehouse admitted by TPC-C). We can observe how the speculative architecture always provides reduced response time independently of the considered configuration. Further, the response time reduction increases while the system load (i.e. the number of active terminals) is increased. In such a scenario, the likelihood of transaction conflict increases, hence leading our speculative approach, which does not induce transaction-block thanks to the enhanced multiverisoning scheme, to improve its effectiveness. We also note that, unless for the case of 8/10 terminals and for the heavy weight TPC-C Delivery profile, the speculative architecture provides average response time comparable with the average latency of the OAB service (namely 400 millisec), which is instead not guaranteed by the non-speculative architecture. This phenomenon is due to the fact that upon the final delivery of the corresponding transactional requests, the speculative architecture has already been able to process the whole set of speculative transaction instances covering the different serialization orders that include the one established by the OAB service. This does not occur in scenarios with more terminals due to the increased level of transaction conflict, which increases the number of serialization orders to be speculatively explored (given the increased amount of active tuple versions upon any read).

Figure 1 (center) shows the average number of speculative transaction instances activated per each transactional request handled by the terminal. This value tends to increase linearly up to 6 terminals, while it exhibits a super-linear increase with 8 or 10 terminals. However, such an increase creates a balance favorable to response time since the speculative scheme increases its gain over the non-speculative one right when the number of terminals is increased to 8/10.

In Figure 1 (right), we report the CPU usage for both speculative and non-speculative execution, which has been computed by only considering the usage of the CPU-cores not reserved for the terminals (one per each terminal) or for the OAB emulator (5 in total, since this emulator has been run by relying on a pool of 5 threads). In other words, it refers to the CPU-cores available for running PostgreSQL backend instances. By the data, the CPU usage increases with the same factor as the number of speculative instances to be executed per each transactional request handled by the terminal. This is an indication of reduced overhead from the support for systematic speculation.

ACKNOWLEDGEMENTS

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385. The authors thank Alessio Manzo and Gabriele Ricciardi for their help in the early software development phase.

REFERENCES

[1] Postgresql. http://www.postgresql.org/.

[2] A. Bestavros and S. Braoudakis. Value-cognizant speculative concurrency control. In *Proc. of VLDB*, pages 122–133, 1995.

[3] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[4] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proc. of VLDB*, pages 134–143, 2000.

[5] 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.

[6] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. of ICDCS*, pages 707–710, 2001.

[7] R. Palmieri, F. Quaglia, and P. Romano. Aggro: Boosting stm replication via aggressively optimistic transaction processing. In *Proc. of NCA*, pages 20–27, 2010.

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

[9] R. Palmieri, F. Quaglia, and P. Romano. ASAP: An aggressive speculative protocol for actively replicated transactional systems. In *Proc. of NCA*, pages 203–211, 2012.

[10] P. K. Reddy and M. Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *IEEE TKDE*, 16(2):154–169, 2004.

[11] TPC Council. TPC-C Benchmark, Revision 5.11. Feb. 2010.

[12] S. Wu and B. Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *Proc. of ICDE*, pages 422–433, 2005.