Boosting Locality in Multi-version Partial Data Replication

João A. Silva, João M. Lourenço and Hervé Paulino
CITI, Universidade NOVA de Lisboa
jaa.silva@campus.fct.unl.pt {joao.lourenco,herve.paulino}@fct.unl.pt

August, 2014

Abstract

Partial data replication protocols for transactional distributed systems present a high scalability potential. But they present a shortcoming of the utmost importance: data locality, i.e., serving transactional reads locally, avoiding inter-node communication. In this paper we tackle this problem proposing a generic cache mechanism adaptable to multi-version partial data replication protocols. We present the design of a generic cache mechanism and apply it to a specific partial data replication protocol, namely SCORe. We perform some experimental evaluation in order to evaluate the effectiveness of the proposed mechanism. The results show some improvement in the system's overall throughput in read dominated workloads.

Keywords: Cache; Multi-version; Partial Data Replication; Concurrency Control; Distributed Systems

1 Introduction

A popular approach for addressing the requirements of high availability and scalability in transactional systems is the employment of both data distribution and replication [3, 8, 9, 17]. Many of the systems that support these strategies make use of full data replication [3, 8, 11, 12], whereby each node replicates the entire system's data. This strategy allows all data accesses to be served locally, but induces undesirable overheads in large scale systems due to the need of coordination among *all* the nodes of the system when propagating updates.

An alternative approach is partial data replication [17, 18], whereby each node replicates only a subset of the system's data. This strategy still presents some (configurable) fault tolerance, while achieving high scalability due to its genuine property, according to which the commitment of a transaction only involves nodes that replicate data items accessed by the commiting transaction [17].

Although partial replication protocols have the potential for high scalability, they remain behind due to the high latency when accessing remote data. In this context of partial replication, where data is distributed among the system's nodes and is replicated only by some of them, there will be cases where some

of the nodes need to access data items which are not held in a local copy. In these situations the only option is to request the required data item from a remote node that holds a copy of the needed item, which requires inter-node communication, imposing non-negligible overheads. In other words, unless some techniques are employed to improve data access locality, the amount of remote read operations increases with the size of the system, eventually hampering its performance.

One possible strategy to address the performance penalty of accessing remote data is to minimize inter-node communication by using caching techniques, which create and manage local copies of frequently accessed remote data items. Such caching techniques are particularly effective in read dominated workloads. However, integrating a cache mechanism in a system that makes use of partial replication and still offers strong consistency guarantees is a challenging task, as it requires a lightweight consistency algorithm that is able to maintain the original protocol's correctness properties while serving read operations based on the locally cached data.

The caching of remote data items raises two main problems, namely: (i) leveraging the caching mechanism to improve the performance of the system shall not affect it correctness, namely shall not affect the consistency guaranties provided by the original algorithm; and (ii) the usage of the caching mechanism shall not have a negative impact in the freshness of the data observed by the transactions, otherwise it will hamper the performance of the system. In this paper, we address the challenges of designing an efficient cache of remote data applicable to multi-version partial data replication protocols. Thus, the contributions of this paper are the following:

(1) we generalize the work of Pimentel et al. [15] by proposing a cache mechanism targeted towards replication protocols providing strong consistency guarantees, like 1-copy-serializability (1CS), and make it malleable to be used in multi-version partial data replication protocols that use scalar logical timestamps;

(2) the proposed caching mechanism is generic and may be customized for a vast set of multi-version partial data replication transactional systems, and we discuss a concrete implementation for the SCORe protocol [13] when used on top of ReDstm [**?**], a distributed transactional memory (DTM) framework; and

(3) the characterization of which transactional workloads may benefit from the proposed caching system.

TODO something about the results

The remaining of this paper is organized as follows. §2 presents our target system model. Our proposed cache mechanism is discussed in §3. In §4 we exemplify how this cache mechanism can be applied to a specific partial replication protocol, namely SCORe [13]. The results of the experimental evaluation are presented and discussed in §5. §6 discusses related work, and we conclude this paper with §7.

2 System Model

We consider a classical asynchronous distributed system comprised by $\Pi = \{n_1, \ldots, n_k\}$ nodes. We assume nodes communicate only through message passing, thus having no access to any kind of shared memory or global clock. Messages can experience arbitrarily long (but finite) delays and we assume no bound on the nodes' computational speed nor on their clocks skews. We consider the classical crash-stop failure model, whereby nodes can fail by crashing but do not behave maliciously.

Following the partial replication strategy, each node n_i replicates only a subset of the system's data and we assume each data item is tagged with an unique identifier. As in classical multi-version concurrency control (MVCC), each data item d is represented by a sequence of versions $\langle k, val, ver \rangle$, where k is the unique identifier of d, val is its value and ver is a scalar, monotonically increasing logical timestamp that identifies (and totally orders) the versions of d.

We abstract over the data placement strategy by assuming data is divided across p data partitions, and that each partition is replicated across r nodes, i.e., r represents the replication factor of each data item. We represent by g_j the group of nodes that replicate partition j, and we also say that g_j is the owner of partition j. We assume that for each partition there exists a single node called master, represented by $master(g_j)$. Each group is comprised by exactly r nodes (to guarantee the target replication factor), and we assume that not all of them crash simultaneously. Also, we do not require groups to be disjoint, i.e., different groups can have nodes in common, and a node may participate in multiple groups as long as $\bigcup_{j=1\ldots p} g_j = \Pi$. Given a group g_j, we represent as $data(g_j)$ the set of data items replicated by the nodes in g_j.

We model transactions as a sequence of read and write operations on data items, encased in an atomic block. Each transaction originates on a node $n_i \in \Pi$, and can read and/or write data items in any replicated partition. Also, we assume no prior knowledge on the set of data items accessed (read or written) by a transaction. Given a data item d, we represent as $replicas(d)$ the set of nodes that replicate d, i.e., the nodes of group g_j that replicate the data partition containing d. Given a transaction T, we define $participants(T)$ as the set of nodes which took part in a transaction, namely $\bigcup_{d \in \mathcal{F}} replicas(d)$, where $\mathcal{F} = readSet(T) \cup writeSet(T)$ (i.e., $\mathcal{F}$ is the set of data items read or written by T). We also assume that every transaction T has two scalar timestamps: a start timestamp, called $T.ts^S$, which represents the transaction's snapshot, and a commit timestamp, called $T.ts^C$, which represents T's serialization point.

Finally, we assume that the replication protocol (or the MVCC algorithm) keeps a scalar timestamp, henceforth called $mostRecentTS_i$, that represents the timestamp of the most recent committed update transaction in node n_i.

3 Caching in Multi-version Partial Data Replication

In this section, we address the two main challenges identified in §1 and propose a generic cache mechanism targeting multi-version partial replication environments. Our solution is two-fold: (i) a cache consistency algorithm, which allows

to determine if a transaction can safely read cached data items while still preserving the replication protocol's correctness properties; and (ii) an asynchronous validity extension mechanism, aimed at maximizing the freshness of the data items maintained in cache.

3.1 Ensuring Data Consistency

To ease implementation, both cached and non-cached data items are maintained in multi-version data containers. However, unlike the versions of non-cached data, the versions of cached data items are augmented with additional information in order to enable the operation of the consistency algorithm. A version of a cached data item d is a sequence of tuples $\langle k, val, version, validity \rangle$, where k is a unique identifier for d; val is a value for d; $version$ is the timestamp/version of this value for d; and $validity$ is the timestamp/version up to when this value is valid, i.e., the timestamp that represents the most recent snapshot in which this version represented the freshest value for d.

Read operation in the remote node Since we augmented the versions of cached data items with validity timestamps, the cache consistency algorithm requires more information to be sent when a remote node responds to a remote read request. When a remote node n_j receives a remote read request for data item d, it responds with the requested data item's version v, and also with the respective version's validity timestamp. If v is the most recent version of d, its validity is the timestamp of the last update transaction to have committed on node n_j, i.e., $mostRecentTS_j$. Otherwise, $v.validity$ is set to the timestamp of the last transaction to have committed on node n_j before the transaction that overwrote v, i.e., $v.validity$ is set to the most recent committed snapshot on n_j in which v was still the most up to date version of d.

Read operation in the local node Algorithm 1 describes the behaviour of a read operation in the local node n_i. When a transaction T needs to read a data item, it first checks for the data locality. If a data item d is replicated in the local node (i.e., $n_i \in replicas(d)$), the read operation can be satisfied locally. Otherwise, d is considered to be remote. In this last case, with the addition of the cache consistency algorithm, now T first inquires the cached data container about d and only then, if d is not found, it issues a remote read request for d. If d is found in the cache (Line 9), it can only be used if there is some version v of d that was created before T began. This is achieved by simply checking $v.version$ and $T.ts^S$: as in classical MVCC, it selects the most recent version having $version$ less than or equal to $T.ts^S$ (Lines 11–15).

When v is found (Line 3), an additional check is still required to ensure that is safe for T to read v. The $T.ts^S$ is compared with $v.validity$ (Line 4), and if the check fails v is considered too old because it may exist a newer version on the remote node that is not known on the local node yet, i.e., a fresher version may have been committed by some transaction that should be serialized before T, and whose updates T should observe. Otherwise, T can safely read v. If some of the checks fails a cache miss is forced (by returning a null value) and the remote read request for d is executed.

Algorithm 1 Read operation on the local node.

```
1: function readCache(Key k, Timestamp ts)
2:     Version v ← getVisibleVersion(k, ts)
3:     if v ≠ null then
4:         if ts < v.validity then
5:             return v
6:     return null

7: function getVisibleVersion(Key k, Timestamp ts)
8:     Versions vers ← cache.getVersions(k)
9:     if vers ≠ null then
10:        Version v ← vers.mostRecentVersion
11:        while v ≠ null do
12:            if v.version ≤ ts then
13:                return v
14:            else
15:                v ← v.prev
16:    return null
```

3.2 Maximizing Cache Effectiveness

According to Algorithm 1, a transaction T can safely read a cached version v only if $v.validity$ ensures that it is sufficiently fresh given $T.ts^S$. On the other hand, as usual in MVCC, at the beginning of a transaction (in node n_i) its ts^S is set to $mostRecentTS_i$, i.e., the timestamp of the last update transaction to have committed in n_i. This ensures that any freshly started transaction T will necessarily observe the updates produced by any committed transaction involving T's originating node.

Since transactions' ts^S are monotonically increasing to reflect the data modifications in the system, the validity timestamp of a cached versions needs to be refreshed if one wants to maximize the chance of being able to serve transaction's read requests using cached data. This is achieved by a validity extension mechanism described in Algorithms 2 and 3.

Extension operation on the sender node In a simple way, the validity extension mechanism consists in broadcasting extension messages, and according to them updating the required validities.

The getModifiedSet function (Algorithm 2) describes the operations performed by a node n_i to build a modified set ($mSet$) intended for another node n_j. A $mSet$ is a set that contains the identifiers of all the data items of which the sender node n_i is primary owner (i.e., $\bigcup_{d \in data(g_j)} n_i = master(g_j)$) and that were modified since the last time a $mSet$ was built for n_j. To allow this, each node n_i keeps track of all the transactions in which it participated (denoted as $committedTransations$), i.e., $\bigcup_{txn \in \mathcal{T}} n_i \in participants(txn)$ (where $\mathcal{T}$ is the set of all transactions). It also maintains for each other node n_j the commit timestamp of the last committed transaction at the moment for which n_i sent an extension message to n_j (denoted as $lastSentValue[j]$).

Logically, a $mSet$ is built by iterating through all the committed transactions in node n_i starting from the most recent transaction to the transaction with commit timestamp equal to $lastSentValue[j] + 1$, and merging the write-sets of all the corresponding transactions.

Extension operation on the receiver node Extension messages can be disseminated asynchronously across the system using various propagation strategies (provided that the dissemination is done with FIFO ordering). We devised

Algorithm 2 Extension operation on the sender node n_i.

```
1: function getModifiedSet(NodeId j)
2:     Timestamp mostRecent ← mostRecentTS_i
3:     Timestamp lastSent ← lastSentValue[j]
4:     Set mSet ← ∅
5:     if mostRecent > lastSent then
6:         for all txn ∈ committedTransactions do
7:             if txn.ts^C > lastSent then
8:                 for all item ∈ txn.writeSet do
9:                     if isPrimaryOwner(item) then
10:                        mSet ← mSet ∪ {item}
11:        return [mSet, mostRecent]
12:        lastSentValue[j] ← mostRecent
13:    return null
```

three basic dissemination strategies:

Eager An extension message is broadcast whenever a transaction commits at some node;

Batch Each node broadcasts an extension message with a fixed (configurable) frequency; and

Lazy An extension message is only disseminated when a node receives a remote read request, by piggybacking it in the remote read response.

Finally, Algorithm 3 presents the extension process executed when a node receives an extension message (from node n_i). An extension message is comprised by two pieces of information: a $mSet$, and the timestamp of the most recent committed update transaction at the time the $mSet$ was built (denoted $mostRecent$).

When an extension message is received, the execution of function extendValidities is triggered. Extending the validities of the cached versions logically means that the validities of all (the most recent) cached versions owned by n_i that are not included in the $mSet$ (i.e., that have not been updated since the last extension message) may be extended. This could be achieved by iterating all the cached versions to identify which validities should be extended. But, in order to do this efficiently, the extension process associates a single shared validity (denoted as $mostRecentValidities[i]$), to all the data items of node n_i whose cached versions are known to be up to date at the time the $mSet$ was built.

By executing function extendValidities, two operations are performed: all data items contained in the $mSet$ are detached from the shared validity (Lines 3–8), by cloning its value into a private validity; and the value of the shared validity for node n_i, $mostRecentValidities[i]$, is updated to $mostRecent$ (Lines 9–13), instantly extending the validities of the most recent cached versions.

Note that, depending on the replication protocol and on the MVCC algorithm, some slight modifications/adaptations may be required to both the cache consistency algorithm and the validity extension mechanism described above.

4 Implementation

Since the mechanism presented in §3 is generic, as a proof of concept, we applied it to a specific partial replication protocol, namely SCORe [13]. In this section,

Algorithm 3 Extension operation on the receiver node.

```
1: Validity[] mostRecentValidities ← Validity[Π]

2: function extendValidities(Set mSet, Timestamp mostRecent, NodeId i)
3:     for modifiedItem ∈ mSet do
4:         Versions vers ← cache.getVersions(modifiedItem)
5:         if vers ≠ null then
6:             Version v ← vers.mostRecentVersion
7:             if v.validity.isShared( ) then
8:                 v.validity ← [v.validity.validity, false]
9:     Validity mrv ← mostRecentValidities[i]
10:    if mrv = null then
11:        mostRecentValidities[i] ← [mostRecent, true]
12:    else
13:        mrv.validity ← mostRecent
```

we present a brief overview of the protocol and describe the adaptation of the proposed cache mechanism.

4.1 Overview of the SCORe Protocol

SCORe is a multi-version partial replication protocol providing 1CS as consistency guarantee, and is in accordance with the system model presented in §2. As usual in MVCC, in SCORe each node maintains a list of versions for each replicated data item. The versions that are visible to a transaction T are determined via $T.ts^S$, which is established upon its first read operation. We omit a description of SCORe's commit phase, which is not required for the understanding of the operation of the cache mechanism. It suffices to say that is based on a variant of the two phase commit protocol (2PC).

Read operations require the determination of which version among the maintained ones should be visible to a transaction. This is achieved using the following three rules:

R1 Snapshot lower bound In every read operation on a node n_i, SCORe verifies that n_i is sufficiently up to date to serve transaction T, i.e., whether it has already committed all the transactions that have been serialized before T according to $T.ts^S$. This is achieved by blocking T until $T.ts^S$ is greater or equal than $mostRecentTS_i$;

R2 Snapshot upper bound In order to maximize data freshness, on the first read operation of transaction T, $T.ts^S$ is set to the timestamp of the most recent version of the data item being read; and

R3 Version selection As usual in MVCC, whenever there are multiple versions for some data item, the selected version will be the most recent one that has a timestamp less than or equal to $T.ts^S$.

4.2 Caching in the SCORe Protocol

We applied the cache mechanism in SCORe, on top of the ReDstm framework [**?**]. This framework allows the implementation of multiple replication protocols, and it follows the system model described in §2, except for one key difference. Each group of nodes is comprised by exactly r nodes, but they are disjoint, and each data partition is replicated by only one group.

Algorithm 4 Adaptation of Algorithm 1 for SCORe.

```
 1: function readCache(Key k, Timestamp ts, boolean firstRead)
 2:     Version v ← getVisibleVersion(k, ts, firstRead)
 3:     ...

 4: function getVisibleVersion(Key k, Timestamp ts, boolean firstRead)
 5:     Versions vers ← cache.getVersions(k)
 6:     if vers ≠ null then
 7:         Version v ← vers.mostRecentVersion
 8:         if firstRead then
 9:             return v
10:         ...
11:     return null
```

4.2.1 Cache Consistency Algorithm

The cache consistency algorithm presented in §3.1 was kept almost untouched when we applied it to SCORe. The only modification introduced was in the getVisibleVersion function (see Algorithm 4), and in fact this modification is an optional optimization.

SCORe's reading rule R2 determines that in the first read operation of every transaction T, $T.ts^S$ is advanced in order to maximize data freshness. So, when reading cached data, we can apply the same rule and return the most recent version of a data item when a transaction is doing its first read operation (Lines 8–9). Thus, allowing SCORe to advance the transaction's ts^S.

4.2.2 Validity Extension Mechanism

Contrary to the cache consistency algorithm, which was left almost untouched, the extension mechanism suffered some mild changes.

In ReDstm, each group of nodes replicates only one data partition and each data partition is replicated by exactly one group of nodes. Since each group replicates the same data items, all the nodes in a group will be aware of the modifications performed to the data items they replicate, thus all of them will build equal *mSets*. Therefore, we adapted the extension mechanism and only one node per group, i.e., the group master, builds and broadcasts extension messages to the other nodes (for the eager and batch strategies).

Algorithm 5 displays the adaptations done to function getModifiedSet. Here, each node keeps track of which data items (that are replicated locally) were updated since the last *mSet* was sent.

The part of the extension mechanism presented in Algorithm 3 was slightly changed. Instead of keeping the most recent validities per node it keeps them in a per partition basis.

Algorithm 5 Adaptation of Algorithm 2 for SCORe.

```
 1: function getModifiedSet(PartitionId j)
 2:     Timestamp mostRecent ← mostRecentTS_i
 3:     Timestamp lastSent ← lastSentValue[j]
 4:     Set mSet ← ∅
 5:     if mostRecent > lastSent then
 6:         for all item ∈ committedItems do
 7:             if isLocal(item) then
 8:                 mSet ← mSet ∪ {item}
 9:         return [mSet, mostRecent, j]
10:         lastSentValue[j] ← mostRecent
11:     return null
```

5 Experimental Evaluation

In this section, we present the results of an experimental evaluation of the proposed cache mechanism. In this evaluation we address two questions: (i) *what is the impact of the cache mechanism in the system's overall throughput?* and (ii) *what is the impact of the cache mechanism in the amount of remote read operations?*

Experimental setup All the experiments were conducted in a heterogeneous cluster with 8 nodes. The first 4 nodes have 2 × Quad-Core AMD Opteron 2376 2.3 GHz and 16 GB of RAM. The other 4 have 1 × Quad-Core Intel Xeon X3450 2.66 GHz (with Hyper-Threading) and 8 GB of RAM. The operating system is Debian 5.0.10 with the Linux kernel 2.6.26-2-amd64, and the nodes are interconnected via private Gigabit Ethernet. The installed Java platform is OpenJDK 6. The replication factor of each data item was set to two.

Benchmarks TODO

We used the Red-Black Tree micro-benchmark to run some experiments. The benchmark is composed of three types of transaction: *insertion*, which add an element to the tree (if not yet present); *deletion*, which remove an element from the tree (if present); and *searching*, which search the tree for a specific element. Insertions and deletions are said to be *write* transactions. This benchmark is characterized very small and fast transactions that perform little work, keeping contention at very low levels. AND LOW LOCALITY in data access. random elements.

5.1 Results

TODO

6 Related Work

We can identify a set of variables that directed the research conducted in the context of replicated transactional systems, such as: the nature of the data's physical storage medium; the replication level; and the consistency guarantees between replicas.

Considering the nature of the data's physical storage medium, we can find research directed to database management systems (DBMSs) in persistent storage [7] (like hard disks), to DBMSs in volatile storage [4] (like RAM), and also to transactional memory systems [3, 16].

The replication level may range from full data replication, where all nodes replicate an entire copy of the system's data, to data distribution, where each data item resides in a single node. In the middle, fits the solutions exploring partial data replication, where each node replicates only a subset of the system's data.

Most proposals of replicated transactional systems [2, 3, 11, 12] aimed at full replication scenarios. This approach of full replication has the advantage of being able to serve every data access locally, hence reducing the inter-node

communication. However, it requires every node to participate in the commit phase, no mater which data was modified by the transaction, hence requiring more synchronization and reducing the potential scalability of this approach.

The data distribution approach was also explored [9,16], ranging from master/slave to control and data-flow techniques.

When considering the partial replication approach, solutions in the literature can be grouped according to which nodes are involved in the commitment of transactions; and according to which consistency guarantees are provided. The work in [18] introduces non-genuine protocols, in which all the nodes in the system are necessarily involved in the commitment of a transaction. Against this kind of approach, genuine protocols (i.e., the commitment of a transaction only involves the nodes replicating data items modified by that transaction) have shown that can achieve better scalability [13,14,17]. Regarding the offered consistency guarantees, the strongest level of consistency is 1CS, which ensures that a system with multiple replicas behaves like a centralized (non-replicated) system. However, either for performance optimization or by the natural implications of the CAP theorem [1], it is common for transactional systems with data replication resort to weaker consistency levels, such as snapshot isolation [18] or eventual consistency [4].

In the context of DBMSs, Serrano et al. [18] argue that 1CS imposes strong limitations on the scalability of replicated solutions and proposes a non-genuine protocol with an alternative consistency level called 1-copy-snapshot-isolation (1SI), which explores snapshot isolation for managing consistency between replicas. In turn, Schiper et al. propose P-Store [17], an efficient solution ensuring 1CS for DBMSs, proposing the first genuine protocol. However, P-Store imposes that read-only transactions also undergo a distributed validation phase. More recently, some more genuine protocol have been proposed. GMU [14] was the first proposal of a genuine protocol ensuring that read-only transactions are never aborted or forced to undergo a distributed validation phase. SCORe [13] is very similar to GMU and may be seen as an evolution of it, but it offers 1CS (instead of extended update serializability (EUS) offered by GMU).

Doing cache of remote data that is accessed frequently is an orthogonal technique to all these systems, and it can be adopted to improve the efficiency of data accesses. Here, the main challenge is how to preserve consistency when cached data (which is replicated asynchronously) is read.

Finally, other techniques exist that are related to caching, such as Tashkent [6] or AutoPlacer [10], which attempt to dynamically tune the mapping of data to nodes, in order to minimize the frequency of remote data accesses. These techniques are orthogonal to cache mechanisms and may even be used together.

7 Conclusions

Partial replication systems present a high scalability potential due to their genuineness property, but they can be severely affected by the inefficient placement of data. If nothing is done improve data access locality, as the system grows. A possible solution to tackle this problem is the use of caching techniques. These techniques replicate remote data that are frequently accessed, in order to serve read operations locally.

In this paper, we proposed a generic cache mechanism adaptable to multi-version partial data replication protocols.

something about the results

As future directions for this work, we highlight the development of a garbage collection mechanism for old cached data, and an extensive experimental evaluation using various benchmarks with different workloads and data access patterns.

References

[1] E. A. Brewer. Towards robust distributed systems. In *PODC*, 2000.

[2] N. Carvalho et al. A generic framework for replicated software transactional memories. In *NCA*, 2011.

[3] M. Couceiro et al. D2stm: Dependable distributed software transactional memory. In *PRDC*, 2009.

[4] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. *SIGOPS*, 2007.

[5] R. J. Dias et al. Efficient support for in-place metadata in transactional memory. In *Euro-Par*. 2012.

[6] S. Elnikety et al. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys*, 2006.

[7] J. Gray et al. The dangers of replication and a solution. In *ACM SIGMOD*, 1996.

[8] B. Kemme et al. A suite of database replication protocols based on group communication primitives. In *ICDCS*, 1998.

[9] C. Kotselidis et al. Distm: A software transactional memory framework for clusters. In *ICPP*, 2008.

[10] J. Paiva et al. Autoplacer: Scalable self-tuning data placement in distributed key-value stores. In *ICAC*, 2013.

[11] R. Palmieri et al. Aggro: Boosting stm replication via aggressively optimistic transaction processing. In *NCA*, 2010.

[12] F. Pedone et al. The database state machine approach. *Distributed and Parallel Databases*, 14, 2003.

[13] S. Peluso et al. Score: A scalable one-copy serializable partial replication protocol. In *Middleware*. 2012.

[14] S. Peluso et al. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS*, 2012.

[15] H. Pimentel et al. Enhancing locality via caching in the GMU protocol. In *CCGRID*, 2014.

[16] M. M. Saad et al. Hyflow: A high performance distributed software transactional memory framework. In *HPDC*, 2011.

[17] N. Schiper et al. P-store: Genuine partial replication in wide area networks. In *SRDS*, 2010.

[18] D. Serrano et al. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *PRDC*, 2007.