Boosting Locality in Multi-version Partial Data Replication

João A. Silva, João M. Lourenço and Hervé Paulino
NOVA LINCS/Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade NOVA de Lisboa, 2829-516 Caparica, Portugal
jaa.silva@campus.fct.unl.pt {joao.lourenco,herve.paulino}@fct.unl.pt

ABSTRACT

Partial data replication protocols for transactional distributed systems present a high scalability potential, but suffer from a shortcoming of the utmost importance: data access locality. In a partial data replication setting, performance can be boosted by serving transactional read operations locally and preventing the expensive overhead of inter-node communication. In this paper we address this concern by proposing a generic caching mechanism directed towards multi-version partial data replication protocols and illustrate its application in a specific protocol, namely SCORe. Experimental results corroborate the effectiveness of the proposed caching mechanism in read-dominated workloads, where it clearly improves the system's overall throughput.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*

General Terms

Algorithms, Design, Experimentation

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 data distribution and replication [3, 6, 8, 17]. When using full data replication [3, 6, 11, 12] each node replicates the entire system's data. This replication strategy allows all data accesses to be served locally, but in large scale systems it induces undesirable overheads by requiring the coordination of *all* the system's nodes when propagating updates.

An alternative approach to full replication is partial replication [17,18], whereby each node replicates only a subset of the system's data. Partial replication provides some (configurable) fault tolerance while achieving high scalability due to its *genuineness* property. In genuine partially replicated systems, committing a transaction involves only the nodes replicating data items that were accessed by the committing transaction [17].

The high scalability potential of partial replication protocols is bounded by the average latency of read operations. As each node in the system only replicates a subset of the entire system's data, some read operations are served locally and have low latency, while others must entail inter-node communication to access remote data and have high latency. Moreover, when the number of nodes in the system increases the number of remote read operations increases as well, which will eventually hamper the system's performance if no techniques are employed to improve the locality of data access.

Data access locality can be improved by using caching techniques, which create and manage local copies of frequently accessed remote data items, and are known to be effective in read-dominated workloads. However, integrating a caching mechanism in a system that makes use of partial replication while keeping strong consistency guarantees is a challenging task. It requires a lightweight consistency algorithm able to maintain the original protocol's correctness properties, while serving read operations based on cached data. Leveraging on the caching mechanism to improve the replicated system's performance raises two main problems, namely: (i) it shall not affect the correctness of the algorithm, i.e., shall not affect the originally provided consistency guaranties; and (ii) it shall not have a negative impact in the freshness of the data observed by the transactions.

This paper addresses the challenges of designing an efficient caching mechanism supporting multi-version partial replication protocols, and have the following contributions: (1) the design of a caching mechanism (discussed in §3) targeting replication protocols providing strong consistency guarantees—which generalizes previous work by Pimentel et al. [15]—and is *malleable* to the point of being used with multi-version partial replication protocols that use scalar logical timestamps; (2) a concrete implementation of the proposed caching mechanism (discussed in §4) developed on top of the ReDstm [19] distributed transactional memory (DTM) framework and applied to SCORe [13]; and (3) the characterization of which transactional workloads may benefit from the proposed caching system (discussed in §5).

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.

In partial replication scenarios, each node $n_i \in \Pi$ 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) [1], 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 monotonically increasing scalar logical timestamp that identifies (and totally orders) the versions of d.

We abstract over the data placement strategy by assuming that the 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. Nodes may be shared by different groups. 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.

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. We also assume that transactions are dynamic, i.e., we have no prior knowledge on the set of data items accessed (read and written) by transactions. 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, in each node, the replication protocol (or the MVCC algorithm) keeps a scalar timestamp, $mostRecentTS_i$, 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 challenges identified in §1 and propose a generic caching 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.

Algorithm 1 Read operation on the local node.

```
1: function READCACHE(Key $k$, Timestamp $ts$)
2:   Version $v \leftarrow$ GETVISIBLE($k$, $ts$)
3:   if $v \neq null$ then
4:     if $ts < v.validity$ then return $v$
5:   return $null$

6: function GETVISIBLE(Key $k$, Timestamp $ts$)
7:   Versions $vers \leftarrow cache$.GETVERSIONS($k$)
8:   if $vers \neq null$ then
9:     Version $v \leftarrow vers.mostRecentVersion$
10:    while $v \neq null$ do
11:      if $v.version \leq ts$ then return $v$
12:      else $v \leftarrow v.prev$          ▷ $v.prev$ is $v$'s previous version
13:  return $null$
```

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 enabling the operation of the consistency algorithm. A version of a cached data item d is a sequence of tuples $\langle k, val, ver, validity\rangle$, where k is a unique identifier for d; val is a value for d; ver is the timestamp of this value for d; and $validity$ is the timestamp 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.

In a read operation, the *local node* is the one that requests the data item and which originates a remote read operation if the data item is not replicated locally. In turn, the *remote node* is the one that receives the remote read request and replies with the requested data item.

Read operation in the remote node. When a remote node n_j receives a remote read request for a data item d, it responds with the value of d, the data item version v, and the respective version's validity timestamp $v.validity$. 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 d, it first looks up for the data locally. If the 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 the latter case, with the addition of the cache consistency algorithm, now T first inquires the cache data container about d and only then, if d is not found, it issues a remote read request for d. A cache value for d can only be used if there is some version v of d that was created before T began. This validity check works by checking $v.version$ and $T.ts^S$: it selects the most recent version having $version$ less than or equal to $T.ts^S$ (Lines 10–12).

When v is found (Line 3), an additional check is still required to ensure that it is safe for T to read v: $T.ts^S$ is compared against $v.validity$ (Line 4), and if the check fails, v is considered obsolete because there may exist a newer version on the remote node that is still unknown in the local node, i.e., a fresher version may have been committed by some transaction that should be serialized before T, and

Algorithm 2 Extension operation on the sender node n_i.

```
1: function GETMODIFIEDSET(NodeId $j$)
2:   Timestamp $mostRecent \leftarrow mostRecentTS_i$
3:   Timestamp $lastSent \leftarrow lastSentValue[j]$
4:   Set $mSet \leftarrow \emptyset$
5:   if $mostRecent > lastSent$ then
6:     for all $txn \in committedTransactions$ do
7:       if $txn.ts^C > lastSent$ then
8:         for all $item \in txn.writeSet$ do
9:           if ISPRIMARYOWNER($item$) then $mSet \leftarrow mSet \cup \{item\}$
10:    return $[mSet, mostRecent]$
11:    $lastSentValue[j] \leftarrow mostRecent$
12:  return $null$
```

Algorithm 3 Extension operation on the receiver node.

```
1: Validity[] $mostRecentValidities \leftarrow$ Validity[$\Pi$]

2: function EXTVALS(Set $mSet$, Timestamp $mostRecent$, NodeId $i$)
3:   for $modifiedItem \in mSet$ do
4:     Versions $vers \leftarrow cache$.GETVERSIONS($modifiedItem$)
5:     if $vers \neq null$ then
6:       Version $v \leftarrow vers.mostRecentVersion$
7:       if $v.validity$.ISSHARED( ) then
8:         $v.validity \leftarrow [v.validity.validity, false]$
9:   Validity $mrv \leftarrow mostRecentValidities[i]$
10:  if $mrv = null$ then
11:    $mostRecentValidities[i] \leftarrow [mostRecent, true]$
12:  else $mrv.validity \leftarrow mostRecent$
```

whose updates T should observe. Otherwise, T can safely read v. If any of the checks fail, a cache miss is forced (by returning a null value) and a remote read request for d is issued.

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 timestamp 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.

The timestamps of transactions ts^S increase monotonically to reflect the data modifications in the system. Hence, the validity timestamp of a cached version needs to be refreshed (extended) to maximize the chance of success when serving read requests from cached data. This is achieved by the validity extension mechanism described in Algorithms 2 and 3.

Extension operation in the sender node. The validity extension mechanism consists in strategically broadcasting extension messages, and appropriately updating the validities of the data items referenced by the extension message.

The GETMODIFIEDSET function (in Algorithm 2) describes the operations performed by a node n_i to build a modifications 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 the primary owner (i.e., $\cup_{d \in partition_j} n_i = master(g_j)$) 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 participates (denoted as $committedTransations$), i.e., $\cup_{txn \in \mathcal{T}} n_i \in participants(txn)$ (where $\mathcal{T}$ is the set of all transactions). For each other node n_j it also maintains the timestamp ts^C of the last committed transaction when 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 $ts^C = lastSentValue[j] + 1$, and merging the write-sets of all the corresponding transactions.

Extension operation in 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 defined 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 a *mSet*, and the timestamp of the most recent committed update transaction at the time the *mSet* was built (denoted *mostRecent*).

Function EXTVALS is triggered when an extension message is received. The most recent version of each cached data item owned by n_i that is not included in the *mSet* (i.e., it was not updated meanwhile) may have its validity extended to *mostRecent* timestamp. The process of extending the validities could be achieved by iterating all the cached versions to identify and update the validity of the appropriate cached items. To do this more 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. Hence, each shared validity is comprised by the validity value itself (denoted *v.validity.validity*), and a boolean value representing if the validity is in fact shared or not (accessible through the ISSHARED function).

By executing function EXTVALS, 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 9–12), instantly extending the validities of the most recent cached versions.

4. IMPLEMENTATION

We applied the caching mechanism described in §3 to a specific partial replication protocol, namely SCORe [13]. SCORe is a multi-version partial replication protocol providing 1-copy-serializability (1CS). 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.

Read operations require the definition of which of the existing versions 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

Algorithm 4 Adaptation of Algorithm 1 for SCORe.

```
1: function readCache(Key $k$, Timestamp $ts$, boolean $firstRead$)
2:   Version $v \leftarrow$ getVisible($k$, $ts$, $firstRead$)
3:   ...
4: function getVisible(Key $k$, Timestamp $ts$, boolean $firstRead$)
5:   Versions $vers \leftarrow cache$.getVersions($k$)
6:   if $vers \neq null$ then
7:     Version $v \leftarrow vers.mostRecentVersion$
8:     if $firstRead$ then return $v$
9:     ...
10:  return $null$
```

Algorithm 5 Adaptation of Algorithm 2 for SCORe.

```
1: function getModifiedSet(PartitionId $j$)
2:   Timestamp $mostRecent \leftarrow mostRecentTS_i$
3:   Timestamp $lastSent \leftarrow lastSentValue[j]$
4:   Set $mSet \leftarrow \emptyset$
5:   if $mostRecent > lastSent$ then
6:     for all $item \in committedItems$ do
7:       if isLocal($item$) then $mSet \leftarrow mSet \cup \{item\}$
8:     return $[mSet, mostRecent, j]$
9:     $lastSentValue[j] \leftarrow mostRecent$
10:  return $null$
```

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 is the most recent one that has a timestamp less than or equal to $T.ts^S$.

We applied our cache mechanism to SCORe, on top of the ReDstm framework. ReDstm allows the implementation of multiple replication protocols, and it follows the system model described in §2, except that each group of nodes is comprised by exactly r nodes, groups are *disjoint*, and each data partition is replicated by only one group.

Cache Consistency Algorithm. The cache consistency algorithm presented in §3.1 was kept almost untouched when applied to SCORe. The only change was an (optional) optimization in the getVisible function (see Algorithm 4). 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 (Line 8). Thus, allowing SCORe to advance the transaction's ts^S.

Validity Extension Mechanism. The extension mechanism suffered some mild changes, but these were optimizations as well. 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 shows the modifications. 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 modified. Instead of keeping the most recent validities per node it keeps them in a per partition basis.

5. EXPERIMENTAL EVALUATION

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 experiments were conducted on a cluster with 8 nodes: 5 comprising two quad-core AMD Opteron 2376 2.3 GHz and 16 GB of RAM, plus 3 comprising two dual-core Intel Xeon X3450 2.66 GHz (hyper-threaded) and 8 GB of RAM. The operating system is Debian 5.0.10 with Linux kernel 2.6.26-2-amd64, and the nodes are interconnected via a private Gigabit Ethernet. The installed Java platform is OpenJDK 6. The replication factor of each data item was set to *two*. The Batch strategy was configured with a period of 50 milliseconds.

Benchmarks. The Red-Black Tree (RBT) benchmark [7] is composed by three transactions: *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 looks up for a specific element. Insertions and deletions are *update* transactions. This benchmark is characterized by very short and fast transactions. Elements are chosen and partitioned at random, keeping contention at very low levels.

The Vacation benchmark is part of the STAMP suite [9]. It emulates a travel reservation system implemented as a set of binary trees tracking customers and their reservations for various travel items. It has 3 *write* transactions: reservations, cancellations, and updates. We modified Vacation to add a *read-only* transaction that consults reservations.

5.1 Results

The best scenario for a caching technique is a read-dominated workload, with few update transactions. In this case, the cache can serve more read operations from the cached values. Fig. 1 shows measurements from running the RBT benchmark with and without cache (NoCache in the plots). Fig. 1a displays the throughput of the system for 3 workloads, respectively from left to right, 0%, 10% and 50% updates. Then, Fig. 1b exhibits the percentage of remote read operations requested by the system in the same 3 workloads (regarding the total percentage of read operations).

With 0% updates—the most favorable scenario—we can see that the system with cache, for all 3 dissemination strategies, performs much better than the system without cache and scales with the number of nodes. In fact, since data is not updated, the extension mechanism is only executed once in the first read of each needed data item, and from there on all the read operations of that data item are served locally. From this behavior follows that all 3 dissemination strategies perform exactly in the same way. Accordingly, in Fig. 1b we can see that all 3 dissemination strategies greatly reduce the amount of remote read operations, while in the system without cache it follows the growth of the system.

When we increase the amount of updates, we start to see the cache mechanism being less effective. With 10% updates the Lazy dissemination strategy has the best performance. This is mainly due to the fact that from all the dissemination strategies, the Lazy variant produces the least amount

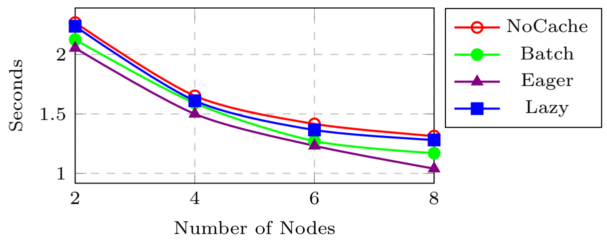


Figure 2: Execution time for the Vacation (low) benchmark with 10% updates.

of network communications. The overhead produced by the other strategies in exchanging a great amount of extension messages has a high impact in performance, which is aggravated by the fact that this benchmark has very short transactions, and in the workload the elements are chosen at random inducing very low contention.

With a workload of 50% updates, the performance of the system with and without cache is very similar due to the overhead generated by the amount of extension messages exchanged by the Batch and Eager strategies. In turn, the Lazy variant begins to be affected by the costs of building and transmitting the extension messages along the critical path of transaction execution. In this update intensive workload, caching techniques are expected to be less profitable because the intense flow of update transactions makes this workload intrinsically hard to be cached.

The Vacation benchmark has longer transactions than the RBT benchmark, hence we expect to see improvements in the performance of the Batch and Eager variants. In fact, this is exactly what we observe in Fig. 2. Since transactions are longer and have larger write-sets, the fact that the Lazy strategy introduces added costs along the critical path of transaction execution makes it perform worst than the other two variants. In turn, the Eager strategy presents the best performance out of the 3, accompanied by a substantial decrease in the amount of remote read operations.

In sum, the Lazy dissemination strategy is the one that produces the least amount of extension messages, being attractive mainly for communication intensive workloads and for workloads that have short transactions. But, since it introduces the assemble and transmission of extension messages along the critical path of transaction execution, it can cause non-negligible overheads in workloads that generate large extension messages. In turn, the Eager variant is more effective in workloads with long read-only transactions and/or big update transactions. The Batch variant is in-between the other two, since it presents a trade off between producing a reduced number of extension messages and the delays in batching them.

6. RELATED WORK

In replicated transactional systems, replication strategies may range from full replication, where all nodes replicate all the system's data, to data distribution, where each data item resides in a single node. In the middle, fit solutions exploring partial replication, where each node replicates only a subset of the system's data, which we are targeting.

Most proposals of replicated transactional systems aim for full replication [3, 11, 12], having the advantage of being able to serve every data access locally, hence reducing inter-node communication. However, these systems require every node to participate in the commit phase, no mater which data was modified by a transaction, hence requiring more synchronization and hampering scalability. The data distribution approach has also been explored [8, 16], ranging from master/slave to control/data-flow techniques.

When considering the partial replication approach, solutions can be grouped according to which nodes are involved in the commitment of transactions, and to which consistency guarantees are provided. Non-genuine protocols were introduced in [18], whereby all the nodes in the system are necessarily involved in the commit process. Against this approach, genuine protocols (i.e., where the commit of a transaction involves only the nodes replicating data items modified by that transaction) have shown to 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 non-replicated system. However, either for performance or by the implications of the CAP theorem [2], it is common for transactional systems resort to weaker consistency levels, such as snapshot isolation [18] or eventual consistency [4].

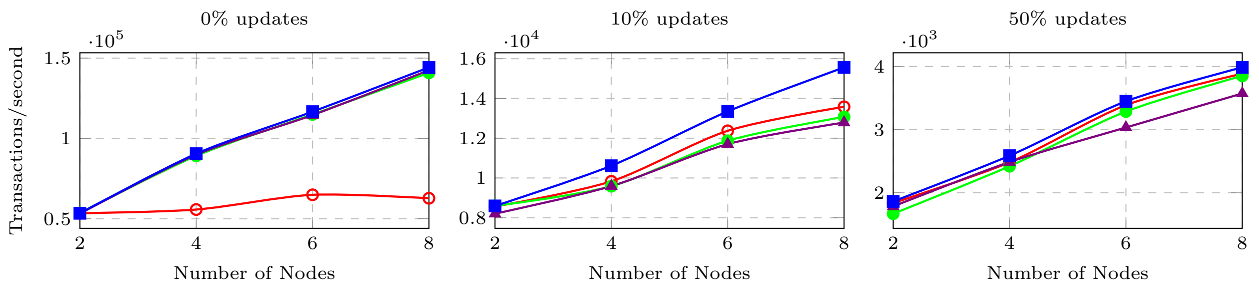
In the context of databases, Serrano et al. [18] argue that 1CS imposes strong scalability limitations and propose a non-genuine protocol with an alternative consistency level called 1-copy-snapshot-isolation (1SI), which explores snapshot isolation for managing consistency between replicas. In turn, Schiper et al. propose P-Store [17], an efficient solution ensuring 1CS for databases, proposing the first genuine protocol. However, it requires read-only transactions to undergo a distributed validation phase. GMU [14] was the first proposal of a genuine protocol ensuring read-only transactions never abort nor are forced to undergo distributed validation. SCORe [13] is very similar to GMU and may be seen as an evolution of the latter offering 1CS (instead of extended update serializability (EUS) offered by GMU).

Caching frequently accessed remote data 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 the consistency model when read operations are served from (asynchronously replicated) cached data. Pimentel et al. [15] present a cache mechanism for the GMU protocol [14]. GMU offers EUS as consistency guarantee and its distributed multi-version concurrency control algorithm relies on a vector clock-based synchronization mechanism. We build on that work and generalize it by removing protocol-specific implementation details and optimizations, and make it malleable to be used by different multi-version partial replication protocols that use *scalar* logical timestamps (instead of GMU's vector clocks).

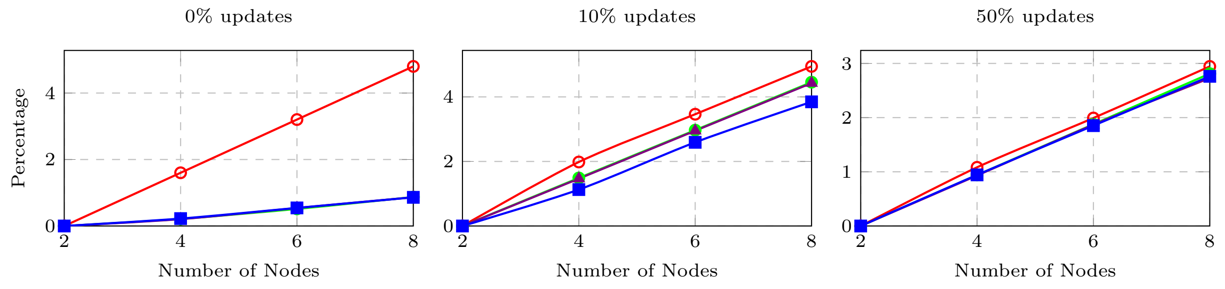
Finally, other orthogonal techniques exist that are related to caching, such as Tashkent [5] or AutoPlacer [10], which attempt to dynamically tune the mapping of data to nodes, in order to minimize the frequency of remote data accesses.

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, which becomes more notorious as the system grows. A possible solution to tackle this problem is the use of caching techniques. These techniques create local replicas of remote data that is frequently accessed, in order to serve transactional read operations locally. Since caching is an orthogonal technique



(a) Throughput in the RBT benchmark.



(b) Percentage of remote read operations in the RBT benchmark.

Figure 1: The impact of the caching mechanism in the RBT benchmark.

to transactional systems themselves, it can be used together with other mechanisms to improve data access locality.

In this paper, we propose a generic cache mechanism adaptable to multi-version partial replication protocols, and apply it to a specific partial replication protocol, namely SCORe, as a proof of concept. The evaluation of our implementation showcases encouraging, though modest, improvements. Further fine-tuning of implementation-specific details is still required for our implementation of the cache mechanism to display its full potential.

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

Acknowledgments

This work was partially funded by FCT-MEC, in the context of the research project PTDC/EIA-EIA/113613/2009 and the strategic project PEst-OE/EEI/UI0527/2014.

8. REFERENCES

[1] P. A. Bernstein et al. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

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

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

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

[7] G. Korland et al. Deuce: Noninvasive software transactional memory in Java. *Transactions on HiPEAC*, 2010.

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

[9] C. C. Minh et al. Stamp: Stanford transactional applications for multiprocessing. In *IISWC*, 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.

[19] J. A. Silva et al. Supporting multiple data replication models in distributed transactional memory. In *ICDCN*, 2015.