Enhancing locality via caching in the GMU protocol

Hugo Pimentel
IST/INESC-ID
hugo.pimentel@ist.utl.pt

Paolo Romano
IST/INESC-ID
romano@inesc-id.pt

Sebastiano Peluso
Sapienza University of Rome/INESC-ID
peluso@gsd.inesc-id.pt

Pedro Ruivo
Red Hat, Inc.
pruivo@redhat.com

***Abstract*—GMU is a recently proposed genuine partial replication protocol for transactional systems that relies on an innovative, fully-decentralized multiversioning scheme to maximize efficiency and scalability. In this paper we tackle one of the key issues that affect the efficiency of GMU-based applications: enhancing their data locality, i.e. the ability to serve transactional reads locally, avoiding remote inter-node communication. To this end we introduce a lightweight caching mechanism that allows for safely accessing *asynchronously replicated* copies of remote data items while preserving GMU's original consistency criterion and its scalability.**

We assess the efficiency and effectiveness of the presented solution by means of an extensive experimental analysis, evaluating deployments on large scale public and private cloud infrastructures, and using well known benchmarks for transactional platforms. The results show striking speedups of up to 14 times in read dominated workloads, and a reduction of the network bandwidth by up to one order of magnitude.

***Keywords*-Distributed Transactional Platforms, Key-Value Stores, Partial Replication, Caching**

I. Introduction

Context. The advent of the cloud computing paradigm has empowered programmers with the ability to scale out their applications easily to hundreds of nodes, fostering research in the area of highly scalable, elastic distributed data platforms (DTP). The first generation of cloud DTPs [8], [5] opted for maximizing scalability by embracing very weak consistency models, such as eventual consistency. However, by relaxing consistency, these systems shift a heavy burden from the platform architects to the application developers, who are required to cope with complex issues associated with concurrency and failures. Indeed, the inherent complexity of building applications on top of weakly consistent systems has been recently recognized by some of the pioneers of eventual consistency [5], and motivated several works aimed to guarantee strong consistency semantics in large scale cloud platforms [7], [20], [21].

By relying on multi-version concurrency control algorithms, these solutions [20], [21], [7] allow for a very efficient management of read-only transactions, sparing them from the possibility of aborting as well as from the costs of any validations. Another key property of these systems, aimed precisely to maximize their scalability is the, so called, *genuine* partial replication property, according to which the execution of a transaction can only involve nodes that replicate data items it accessed [24]. This property is of the utmost importance to enable high scalability, as it rules out non-scalable solutions based either on centralized components (which may turn into bottlenecks/single point of failures) or on full-replication (which induces unacceptable overheads in large scale platforms due to the need of propagating updates across the entire set of nodes of the system).

Motivation. While partial replication protocols possess high potential for scalability, the actual efficiency of these systems can be seriously hindered if the underlying data store does not achieve a good degree of locality when serving applications' data access patterns. In fact, as in these systems data is distributed across the entire set of nodes in the platform and replicated only on relatively small number of them, it follows that, in order to process a transaction's read request, it may be necessary to fetch data remotely. This problem is particularly exacerbated since many popular key-value stores (transactional or not), such as Dynamo [8] or Infinispan [15], use random placement based on consistent hashing. By relying on random hash functions to determine the location of data across nodes, these solutions allow lookups to be performed locally, in an very efficient manner [8]. Furthermore, consistent hashing guarantees that the join/leave of a node incurs in a limited change in the mapping of keys to buckets. However, due to the random nature of data placement (oblivious to the access frequencies of nodes to data), solutions based on consistent hashing may result in sub-optimal data placements. Indeed, assuming random placement of data, it is easy to see that the probability of finding a requested data item locally is inversely proportional to the number of nodes in the system. In other words, unless appropriate techniques are employed to enhance locality in the access to data, as the scale of the system grows, the number of remote read operations is destined to grow linearly, eventually saturating the network's capacity and hindering scalability.

A possible approach to maximize the efficiency of these protocols is to rely on caching techniques, which repli-

This work was supported by national funds through FCT — Fundação para a Ciência e Tecnologia — under project PEst-OE/EEI/LA0021/2013, by specSTM project (PTDC/EIA-EIA/122785/2010), and by National Science Foundation under Grant No. 0910812.

cate remote data items that are frequently accessed in an asynchronous (and hence lightweight) fashion, in order to minimize the frequency of inter-node communication. However, integrating a caching mechanism in strongly consistent genuine partial replication protocols is far from being an obvious task, as it requires designing highly scalable cache consistency protocols capable of preserving transactional consistency while allowing read operations targeting data items not owned by the current node to be performed locally (based on cached data), i.e. without contacting the actual data owner.

Contributions. In this paper we introduce GMU-C, a distributed caching scheme for GMU [20], a recent, genuine partial replication protocol that employs a fully distributed multi-versioning scheme based on vector clocks. GMU has several noteworthy properties: i) it spares read-only transactions from the risk of aborts, as well as from the cost of expensive remote validations, ii) it ensures that every transaction (including update transactions that have to be aborted eventually) always observes a consistent snapshot of data, that is a snapshot producible by some linearization of a prefix of the history of committed transactions, a property called Extended Update Serializability [1] (EUS). Developed in the context of the EU project Cloud-TM [1], GMU is planned to be integrated in the official version of Infinispan [15], a mainstream open-source distributed transactional platform developed by Red Hat that is at the basis of a number of highly visible open-source projects (like Hibernate and JBoss Application Server). The integration of the GMU protocol in a highly popular product like Infinispan amplifies significantly its practical relevance, and that of the current work.

The presented protocol consists of two main innovative building blocks:

- a novel cache consistency algorithm, which allows transactions to read asynchronously replicated copies of data items, without compromising the consistency criterion abided by GMU and preserving its genuineness property;
- a lightweight cache invalidation mechanism aimed at maximizing the freshness of the data maintained in cache (and hence the profitability of the caching scheme), and designed to support pluggable dissemination and eviction strategies.

We integrated the proposed distributed caching scheme into Infinispan, and assessed the efficiency and effectiveness of the presented solution by means of an extensive experimental analysis, based on both synthetic and well known benchmarks for transactional platforms and on large scale deployments on public and private cloud infrastructures. The experimental data show striking speedups of up to 14 times in read dominated workloads, and a reduction of the network bandwidth by up to one order of magnitude.

[1] http://www.cloudtm.eu

Structure of the document. The rest of this document is organized as follows. Section II discusses related work. GMU and the proposed caching protocol are presented in Sections III and IV, respectively. The results of the experimental evaluation study are presented in Section V. Finally, Section VI concludes this paper.

II. Related Work

The problem of how to replicate transactional systems has been long investigated in the literature. Most of the existing proposals [6], [19], [18], [4] have been targeted at the case of full replication, in which every node maintains a copy of each data in the system. Full replication techniques have the key attractive characteristic of ensuring that any data access can be locally served, which allows for avoiding any inter-node communication till the commit phase. On the other hand, the need to involve all nodes in the system whenever a transaction updates any data item limits the inherent scalability of these protocols, making them suitable for relatively small-scale systems (on the order of a few tens of nodes).

When considering partial replication schemes, literature proposals can be grouped depending on (i) whether they can be considered genuine, and on (ii) the consistency guarantees that they provide. The works in [25], [26] introduce non-genuine protocols. in which the commitment of a transaction requires interactions with all the sites within the replicated system. Compared to these approaches, genuine partial replication schemes have been shown to achieve significantly higher scalability levels [20], [21], [24]. The P-Store protocol [24] provides a genuine solution that supports 1-copy serializability (1CS) [2]. However, P-Store imposes that read-only transactions undergo a distributed validation phase (taking place at commit time) and that are potentially subject to rollback/retry. Conversely, GMU, the target protocol of this work, never aborts read-only transactions, since it guarantees that transactions always observe a consistent snapshot of data, and consequently spare them from expensive remote validations. Unlike P-Store, GMU does not ensure 1CS, but a consistency criterion called Extended Update Serializability (EUS). EUS is incomparable with 1CS as it ensures that read-only transactions can observe the commit event of two non-conflicting update transactions in different order (which is prohibited by 1CS. On the other hand, EUS ensures that even transactions that abort must observe a state of the system that could be producible by a serial transaction execution history — whereas 1CS allows transactions that abort to observe arbitrary snapshots. Another recent and well-known transactional replication protocol is Spanner [7] by Google. Analogously to GMU, Spanner implements a distributed multi-versioned concurrency control algorithm that allows

to execute read-only transactions in a lock-free way and on consistent snapshots. The essential difference between GMU and Spanner is in that the latter relies on specialized hardware (atomic clocks) to serialize transactions, whereas GMU uses a distributed multi-versioning scheme based on the usage of vector-clocks. Additional details on GMU's functioning will be provided in Section III.

As already mentioned, in these platforms the issue of how to partition data across the nodes of the system is of paramount importance. Solutions in this area can be coarsely classified in two groups: solutions based on consistent hashing [8], [15], [13], and directory-based approaches [5]. Solution based on consistent hashing disseminate data in random fashion, and have the key advantage of allowing for very efficient lookups and of minimizing the amount of data transferred upon the join/leave of nodes from the system. On the down side, the random nature of the hashing functions used to determine data ownership can lead to sub-optimal data placements and poor data locality. Directory-based schemes, as used for instance in BigTable [5] and Spanner [7], provide programmers with fine-grained control on the mapping of data to nodes in the system, but incur the additional costs of i) contacting remote directory services to lookup data, and ii) ensuring the high availability of the directory service. Caching of frequently accessed remote data is an orthogonal approach that could be adopted to maximize the efficiency of both consistent hashing-based and directory-based systems. The key challenge to implement a caching scheme for a distributed transactional platform is how to preserve ACID consistency when reading cached data that is only asynchronously replicated. The cache consistency protocol presented in this paper tackles precisely this issue.

Finally, this work is related to techniques, such as Tashkent [9] or AutoPlacer [17], that aim at dynamically tuning the mapping of data to the nodes in the system to minimize the frequency of remote data accesses. These approaches are orthogonal to and can be used in conjunction with caching.

III. System model and Overview of GMU

We consider a classic asynchronous distributed system model composed of $\Pi = \{p_1, \ldots, p_n\}$ nodes (also called processes). Nodes communicate through message passing and do not have access to a shared memory nor a global clock. Messages may experience arbitrarily long (but finite) delays and we assume no bound on relative site speeds or clock skews. We consider the classic crash-stop failure model: sites may fail by crashing, but do not behave maliciously.

Each node p_i stores a partial copy of data, for which we assume a simple key-value model. Each data item d is a triple (k, val, ver), where k is a key, val its value and ver is a scalar, monotonically increasing timestamp that identifies (and totally orders) the versions of a data item d. For the sake of brevity, we will use the notation (k_{ver}) to denote the ver-th version of the value associated with key k.

We abstract over the data placement policy by assuming that data is subdivided across m partitions, and that each partition is replicated across r processes (in other words, r represents the replication degree for each data item). We denote with $\Gamma = \{g_1, \ldots, g_m\}$ the set of groups of processes g_j that replicate the j-th data partition. We also say that a group of process g_j owns the j-th data partition, and assume that for each partition there exists a single process called primary owner. Each group is composed of exactly r processes (to ensure the target replication degree), of which at least one is assumed to be correct. In order to maximize flexibility of the data placement strategy, we do not require groups to be disjoint (they can have nodes in common), and assume that a process may participate to multiple groups, as long as $\bigcup_{j=1\ldots m} g_j = \Pi$. Note that this model allows us to capture a wide range of data distribution algorithms, such as schemes, currently very popular in NoSQL transactional data stores, which rely on consistent hashing [8], [15], [13] based distribution policies to: i) minimize data transfer upon joining/leaving of nodes; ii) ensure the achievement of predetermined replication degree; iii) avoid distributed lookups to retrieve the identities of the group of processes storing replicas of the requested data items.

We model transactions as a sequence of read and write operations on data items, preceded by a begin, and followed by a commit or abort operation. Transactions originate on a process $p_i \in \Pi$, and can read/write data stored belonging to any partition. Also, we do not assume any a-priori knowledge on the set of data items read or written by transactions. Given a data item d, we denote as $Replicas(d)$ the set of processes that maintain a replica of d (namely the nodes of the group g_j that replicate the data partition containing d).

A. GMU Overview

Before presenting GMU-C, we provide a brief overview of the base protocol that it extends, namely GMU. We remind to [20] for an extensive discussion of the GMU protocol, and provide, in the following, a succinct description aimed at providing sufficient information to understand the functioning of GMU-C.

As classical multi-version concurrency control schemes GMU maintains a chain of totally ordered committed versions for each data item on every node. The version order is determined by the scalar timestamps that follow the order of commits on a given node, hence determining a relationship between the event of a commit of a transaction T on a node p_i and the versions of all the data items updated by T. As transactions can, in general, read/write data maintained by different nodes, GMU associates with the commit event of a transaction T a vector clock (with size equals to the

cardinality of Π) that univocally determines the versions produced by T, and that keeps track of the dependencies with the other commits on the involved nodes. In addition each node p_i maintains track of vector clocks associated with all update transactions that committed involving node p_i into a list named $CLog$ that is ordered in accordance with the order of commits on p_i.

During its execution, a transaction T maintains:

- A reading vector clock, noted $T.VC$, used as a visibility reference during read operations, that keeps track of the causal and data dependencies created by T during its execution.
- A vector of boolean values, noted $T.hasRead$, that maintains, in its i-th entry, the information on whether the current transaction has already read on node p_i or not.

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

R1 (Snapshot lower bound). On the first read operation on a node N_i, GMU verifies that N_i is sufficiently up to date to serve the transaction, i.e. whether it has already committed all the transactions that have been serialized before T according to T's current reading VC. This is achieved by blocking T until the i-th entry of $T.VC$ is strictly larger than the i-th entry of the most recent vector clock in N_i's $CLog$.

R2 (Snapshot upper bound). On the first read operation on a node N_i, GMU determines which is the most recent snapshot committed on N_i that T can observe, taking into account the set of read operations already issued by the transaction. Such snapshot, which is uniquely identified by an entry of N_i's $CLog$, and denoted as $MaxVC$, is the most recent entry in N_i's $CLog$ such that all $MaxVC$'s entries associated with nodes from which T has already read are not higher that $T.VC$. This rule allows for ensuring that T never observes "too fresh" data snaphots, i.e., the commit events of transactions T' that committed on a node from which T has already read, and whose updates T has "missed" (having T been serialized before T'). The reading vector clock of T is then updated by computing the max between it and $MaxVC$, hence reflecting the fact that the transaction has established a new upper bound on the freshness of the snapshot that it can observe.

R3 (Version selection). Once the lower bound on data freshness has been guaranteed, and that $T.VC$ has been updated to reflect the upper bound on the snapshot observable by T, the version of the requested key is determined by extracting the local version number identifier v from $T.VC$ and getting the most recent version among the ones committed in the snapshots observable by the transaction.

We omit a detailed description of GMU's commit phase, as this is irrelevant for understanding the functioning of GMU-C. It suffices to say that it is based on a variant of the Two-Phase Commit protocol during which i) the transaction's readset is validated, and, ii) in case this is found to be up to date, a new commit VC for the transaction is established (using a voting mechanism inspired to the Skeen total order multicast scheme [28]) and inserted in the $CLog$ of the participating nodes.

Consistency criterion. As already mentioned in Section I, the target consistency criterion of GMU is Extended Update Serializability (EUS) [1], whose specification we report in the following.

Like 1CS, EUS requires that the conflict graph including all committed update transactions is acyclic. Differently from 1CS, however, EUS allows two read-only transactions to observe the commit events of two non-conflicting update transactions in different orders. Further, EUS imposes additional guarantees on the snapshots observed by transactions that abort, mandating that these must be equivalent to those producible by *some* serial execution of the set of committed update transactions (and not arbitrary, as with 1CS). This sort of guarantees may be necessary to ensure that the application does not behave in an unexpected manner (e.g., crashing, or being trapped in infinite loops) due to the observation of non-serializable snapshots, which would be detectable only at commit time using 1CS.

IV. Caching in the GMU protocol

GMU-C is composed of two main sub-components:

1) A cache consistency algorithm, which allows transactions to determine whether it is safe to read locally cached copies of remote data items.
2) A cache invalidation mechanism, which aims at maximizing the freshness of the data maintained in cache (and hence the profitability of the caching scheme), and was designed to support arbitray dissemination and eviction strategies.

In the following we discuss these two components individually.

A. Ensuring data consistency

Cached data is maintained in a multi-versioned data container. Each version of a cached data item is a tuple $\langle k, val, creationVC, validityVC \rangle$, where k and val are its key and value, $creationVC$ is the VC of the transaction that committed this version, and $validityVC$ is a VC that represents a lower bound on the most recent snapshot in which this version represented the freshest value of this key.

The pseudo code describing the behavior of a read from the cached data container is reported in Algorithm 1. When a transaction T needs to read a data item d that is not local, it triggers the `readCache` function to determine whether it is possible to serve the read from the cache. To this end, it first determined whether any version of d is present

Algorithm 1: Read operations on local node (p_i)

```
1  [Key, Ver, VC] readCache(Key k, VC xactVC, bool[] hasRead)
2  Versions vers ← getAllVersions(k);
3  if vers = null then
4      return null;
5  Ver v ← precVer(vers, xactVC, hasRead);
6  if v ≠ null then
7      VC validityVC ← v.validityVC;
8      if validityVC[owner(k)] ≥ xactVC[owner(k)] then
9          VC updatedVC ←xactVc;
10         if ¬hasRead[p_i]) then
11             if checkMaxVC(validityVC, xactVC, hasRead) then
12                 updatedVC ←
                   mergeAndMax(validityVC, xactVC);
13             else
14                 updatedVC ←
                   mergeAndMax(v.creationVC, xactVC);
15     return [key, v, updatedVC];

16 Ver precVer(Versions vers, VC xactVC, bool[] hasRead)
17 Ver v ← vers.mostRecent;
18 if ∄j s.t. hasRead[j] = true then
19     return v; // 1st read by the transaction
20 while v ≠ null do
21     if ∀j s.t. hasRead[j] = true : v.creationVC[j] ≤ xactVC[j]
       then
22         return v;
23     else
24         v ← v.prev;
25 return null;

26 bool checkMaxVC(VC validityVC, VC xactVC, bool[] hasRead)
27 if ∄j s.t. hasRead[j] = true then
28     return true;
29 // In this case it is not safe to mergeAndMax(validityVC[j],xactVC[j])
30 if ∃j s.t. validityVC[j] > xactVC[j] ∧ hasRead[j] = true then
31     return false;
32 return true;
```

in cache (lines 2-3). If at least some version exists, it is first determined the most recent one (if any) to have been committed by a transaction T' that is serialized before T. To this end (see `precVer` function), the $creationVC$ of the versions and the vector clock of T, $xactVC$ are compared. T' is considered serialized before T if $creationVC$ does not follow $xactVC$ in all the entries associated with nodes from which T has already read. These are nodes for which, according to rule R2 of GMU (see Section III), the reading VC of T can no longer be advanced. Hence, this rule guarantees that the transaction can never observe versions created by transactions that have been serialized after T. As a consequence, if T has not read from any node yet, it can simply return the freshest cache version of v (lines 17-18). On the other hand, if no such version exists (e.g, because it has been purged from cache by the cache eviction mechanism), a $null$ value is returned, notifying a cache miss.

In order to determine whether the version v returned by the `precVer` function is actually visible by a transaction T, it is necessary to further verify whether v is sufficiently fresh based on T's reading VC. This is true if it can be safely excluded that some transaction committed a (fresher) version of d, being serialized before T and after the transaction that committed v, say T_v. As the commit events on a node are totally ordered in GMU, this can be guaranteed by simply comparing if the $validityVC$ of v stores, in the entry associated with the owner of d, a value larger than the corresponding entry in $xactVC$. The reason why this condition suffices will become clearer shortly, after having described the logic used to compute and update the $validityVC$s of cached data items.

Finally, in case transaction T is being served a cached item, from whose owner T had not read yet, as in rule R2 of GMU, also in GMU-C, T's VC is updated in order to a) capture the fact that, if T could observe version v, then T has to be serialized after T_v; and to b) possibly allow T to observe snapshots created by transactions serialized after T_v, hence maximizing data freshness. In order to ensure that the reading VC of T, $xactVC$, correctly embeds the causality relationship $T_v \rightarrow T$ it is sufficient to update $xactVC$ with the maximum between itself and $creationVC$. On the other hand, the logic used in GMU to determine the upper bound the visible snapshot of a transaction (see rule R2 in Setion III) requires iterating over the $CLog$ of the node responsible for maintaining the requested data item. This is clearly not feasible in case we are serving the read operation from the cache, or one should fully replicate the $CLog$ of every node, compromising the genuine partial replication property of the GMU protocol and mining its scalability. We note however that we can still exploit the $validityVC$ and check whether it does not include the commit events of transactions that were serialized after T in any node from which it has already read (lines 29-30). If this is not the case, $xactVC$ can be safely *mergeAndMaxed* with $validityVC$, as the same would have happened if the read request had been served remotely, by the node n_d owning d, at a time in which the latest entry in n_d's $CLog$ was $validityVC$. The *mergeAndMax* primitive is the classic update rule for two vector clocks, according to which the resulting vector contains, in each entry, the maximum between the corresponding entries of the two vector clocks passed as input parameters.

Algorithm 2 describes the remote operations introduced, during the processing of a remote read request from node p_i on node p_j, in order to compute the additional meta-data required by the cache consistency protocol, namely $creationVC$ and $validityVC$. First, the data item version to be returned to p_i is retrieved using the classical GMU reading logic. This logic is encapsulated by the $GMURead$ primitive, which returns, in addition to the version v, also the new transaction VC, updated to reflect the read of v according to GMU's rule R2.

Next the $creationVC$ is determined by looking up in the $cLog$ for the transaction that committed v (lines 15-

Algorithm 2: Read operations on remote node (node p_j)

```
1  on receive READREQ[Key k, VC T.VC, bool[] T.hasRead] from p_i
2  [Ver v, VC newXactVC] ← GMURead(k, T.VC, T.hasRead);
3  VC creationVC ← getCreationVC(v);
4  VC validityVC] ← getValidityVC(v, v.getNextVers());
5  send [readV, newXactVC, validityVC, creationVC] to p_i

6  VC getValidityVC(Ver readV, Ver nextV)
7  if nextV = null then
8      return CLog.mostRecentVC;
9  else
10     foreach vc ∈ CLog do
11         if vc[j] < nextV.value then
12             return vc;
13 return null;

14 VC getCreationVC(Ver readV)
15 foreach vc ∈ CLog do
16     if vc[j] = readV.value then
17         return vc;
18 return null;
```

17). The $validityVC$ is calculated by using the most recent entry in $CLog$, in case v is the freshest version of that data item (lines 7-8). Otherwise, the $validityVC$ is set to the vector clock of the last transaction to have committed on node p_j before the transaction that overwrote v (lines 9-12). In other words, $validityVC$ is set to the most recent committed snapshot on p_j in which v was still the most up to date version of k.

B. Maximizing cache effectiveness

According to Algorithm 1, a transaction T can safely access a cached version v only if the $validityVC$ of v ensures that this version is sufficiently fresh given T's reading VC, i.e. if v's $validityVC$ is larger or equal than T's reading VC in the entry associated with the node that owns the corresponding data item. In the negative case, a cache miss has to be forced in order to ensure that no fresher version of v has been committed by some transaction that should be serialized before T, and whose updates T should observe.

On the other hand, in order to ensure that transactions have a chance to observe the latest versions of data, the reading VC of executing transactions is advanced in two occasions: i) upon activation of a new transaction, its $xactVC$ is set equal to the most recent entry in the local $CLog$ — this ensures that any freshly started transaction T will necessarily observe the updates produced by any transaction that committed involving T's originating node; ii) upon every first read of the transaction on a data item owned by a node n, as specified by GMU's rule R.2, in case the read is not served from the cache, or according to the behavior formalized by the pseudo-code of Algorithm 1, if the read returns cached data.

Since the reading VCs of transactions are constantly advanced in order to maximize the freshness of data observed by transactions, the $validityVC$ of cached data versions need to be refreshed if one wants to maximize the chances of being able to serve transaction's read requests using cached data. This is precisely the purpose of the cache invalidation mechanism described in Algorithms 3 and 4.

Let us start by analyzing the pseudo-code of `getInvalidationSet` (see Algorithm 3), which describes the actions performed by node p_i to build a, so called, $iSet$, destined to a node p_j. Node p_i maintains for each other node p_j in the system, an index (noted $CLog.lastSentValue$) that points to the last position of its own $CLog$ for which p_i sent information to p_j. An $iSet$ contains the identifiers of all the keys of which p_i is primary owner and that were updated since the last time that an $iSet$ was built for p_j. The $iSet$ for node p_j is built by iterating of the local $CLog$ starting from the most recent entry to the entry following $CLog.lastSentValue[j]$, and merging the writesets of all the corresponding committed transactions.

Algorithm 3: Invalidation operations on sender node(node p_i)

```
1  [long, Set, VC] getInvalidationSet(nodeID j)
2  VC mostRecentVC ← CLog.mostRecentVC;
3  long lastSentValue ← CLog.lastSentValues[j];
4  long mostRecentValue ← mostRecentVC[i];
5  Set iSet ← ∅;
6  atomically do
7      if mostRecentValue > lastSentValue then
8          foreach vc ∈ CLog do
9              if vc[i] > lastSentValue then
10                 foreach w ∈ vc.keysCommitted do
11                     if isPrimaryOwner(i, w) then
12                         iSet.add(w);
13             return [lastSentValue, iSet, mostRecentVC];
14 return null;
```

$iSet$s can be disseminated asynchronously across the nodes in the system, using alternative data propagation strategies, such as pull vs push based [3], gossip-based [27] vs piggyback-based. The only requirement for the communication layer used to disseminate the $iSet$s is FIFO ordering guarantee (e.g., as provided by TCP channels). We integrated in GMU-C's prototype the following three different dissemination strategies, which will be evaluated in Section V:

- EAGER: an $iSet$ is broadcast whenever time a transaction commits at some node;
- BATCH: each node broadcasts an $iSet$ with a fixed, configurable frequency (in case the $iSet$ is non-empty);
- LAZY: an $iSet$ is only disseminated when a node receives a remote request, by piggybacking it in the remote request response.

Algorithm 4: Invalidation operations on receiver node(node p_j)

```
1 Validity[] mostRecentValidities=new Validity[|Π|];
2 upon receive Invalidate[iSet, mostRecentVC] from p_i
3 foreach invalidKey ∈ iSet do
4     Version v ← getMostRecentCachedVersion(invalidKey);
5     if v ≠ null ∧ v.validity = mostRecentValidities[i] then
6         // Detach v.validity from mostRecentValidities[i]
7         v.validity ← mostRecentValidities[i].clone();
8 mostRecentValidities[i].set(mostRecentVC);
```

Finally, let us present the invalidation logic executed upon reception of an *Invalidate* message from node p_i, described by the pseudo-code of Algorithm 4. An *Invalidate* message carries an $iSet$, and the VC of the most recent committed transaction at the time in which the $iSet$ was built, denoted as $mostRecentVC$. Applying an *Invalidate* message from node p_i implies logically extending the $validityVC$s of all (the most recent) cached data item versions owned by p_i that are not included in the $iSet$, i.e. that have not been updated since the last $iSet$ message.

In order to implement such operation efficiently, i.e. while avoiding to iterate over the entire cache data container to identify the keys whose $validityVC$ should be updated, the invalidation logic performs the dual operation. It associates a single, shared VC, denoted as $mostRecentValidities[i]$, to all the keys of node p_i whose cached version are known as to be up to date at the time in which an $iSet$ was generated. Whenever a new *Invalidate* message is received from node p_i, two operations are performed: i) all keys contained in the $iSet$ are detached from the shared VC, $mostRecentValidities[i]$, by cloning its value in a private $validityVC$; ii) next, the value $mostRecentValidities[i]$ is updated to the value of $mostRecentVC$ specified in the message.

V. Evaluation

In this section, we present the results of an experimental study of the proposed caching scheme for GMU integrated in Infinispan that aims (i) to assess its effectiveness by evaluating which speed-ups can be achieved thanks to its usage, (ii) to compare the performance of the three invalidation dissemination mechanisms described in Section IV, and (iii) to evaluate the overhead introduced by the caching protocol in unfavourable, update intensive workloads.

The experimental study has been performed on two different testbeds: the CloudTM and FutureGrid infrastructures.

CloudTM is a private OpenStack-based cloud computing infrastructure, deployed in a dedicated cluster. Each machine is equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors, 40 GB of RAM and interconnected via a private Gigabit Ethernet. The VMs are instantiated with one processing core and 4 GB of RAM and the virtualization took advantage of the hardware support provided by the Intel(R) processors.

FutureGrid [10], [14] is a public cloud infrastructure, typically characterized by more competitive resource sharing, ample usage of virtualization technology, and equipped with a high-performance communication infrastructure. In the evaluation we used from 16 to 80 virtual machines, equipped with 4GB RAM and one 2.93GHz core Intel Xeon CPU X5570, and running CentOS 5.5 x86_64. All the VMs were deployed in the same physical data-center and interconnected via InfiniBand, a switched fabric computer network communications link used in high-performance computing and enterprise datacenters.

The benchmark applications adopted for the assessment of the work are (a porting of) TPC-C [23], Vacation and a synthetic transactional benchmark.

TPC-C is an On-Line Transaction Processing (OLTP) benchmark that simulates the activities of a wholesale supplier that operates out of a set of sales districts clustered in a certain configurable number of warehouses. The TPC-C benchmark requires to scale out the amount of data stored (proportional to the number of warehouses) in the system as its scale grows. The TPC-C version used in this experimental study was adapted to run on top of transactional key-value stores (and used, in previous works to evaluate the performance of strongly consistent partial replication protocols [20], [21]).

Vacation is a benchmark from the STAMP [16] suite that simulates an online travel agency in which several types of resources can be manipulated by customers or by the agency. There are three distinct types of sessions: reservations, cancellations, and updates. As in TPC-C, we used a port of this benchmark (which was originally conceived to evaluate Transactional Memory systems [12]) for distributed key-value stores that is also designed to use larger data set (i.e., a higher number of agencies) when we scale out the system.

The synthetic benchmark executes transactions with a very reduced number of data accesses, i.e., up to 5 read/write operations, which allows to generate workloads containing very fast and lightweight transactions. The rationale for including, in the suite of benchmarks used in this evaluation study, also this synthetic benchmark is that it allows for shaping more easily workloads capable of highlighting relevant trade-offs in the design of the proposed solution.

For all the benchmarks we adopted two base configurations in order to generate two types of workloads: Workload A, which is a read dominated workload with 90% of read-only transactions and 10% of update transactions; Workload B, which is a mixed workload containing 50% of read-only transactions and 50% of update transactions. For all benchmarks we configured the default broadcast rate of the BATCH strategy at 50 ms, which, after a preliminary experimental phase, turned out to be always very close to optimum across all the tested workloads.

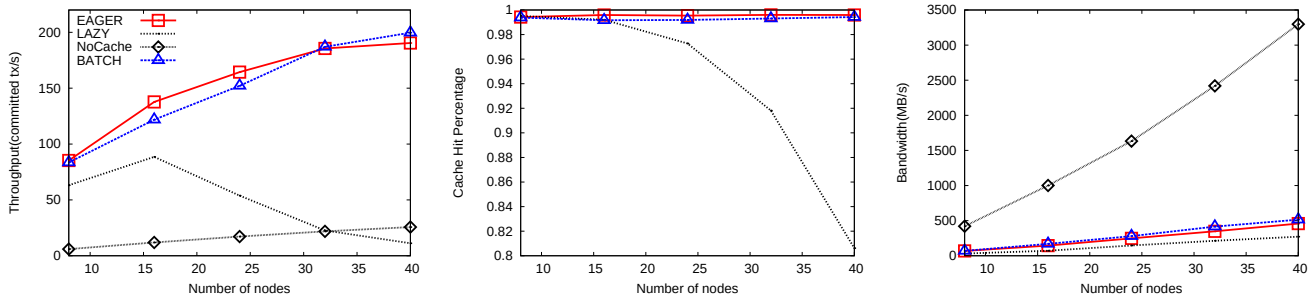

Figure 1. TPC-C benchmark deployed on CloudTM for Workload A. Left to Right: Throughput, Cache Hit Percentage, Total Bandwidth.

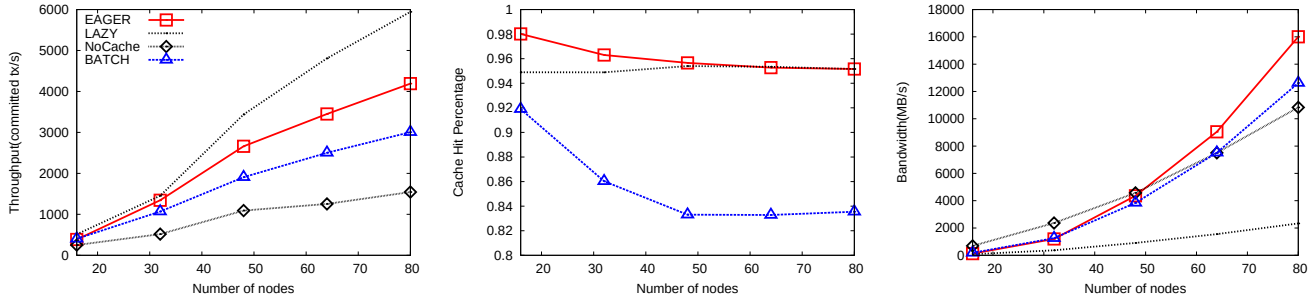

Figure 2. Synthetic benchmark deployed on CloudTM for Workload A. Left to Right: Throughput, Cache Hit Percentage, Total Bandwidth.

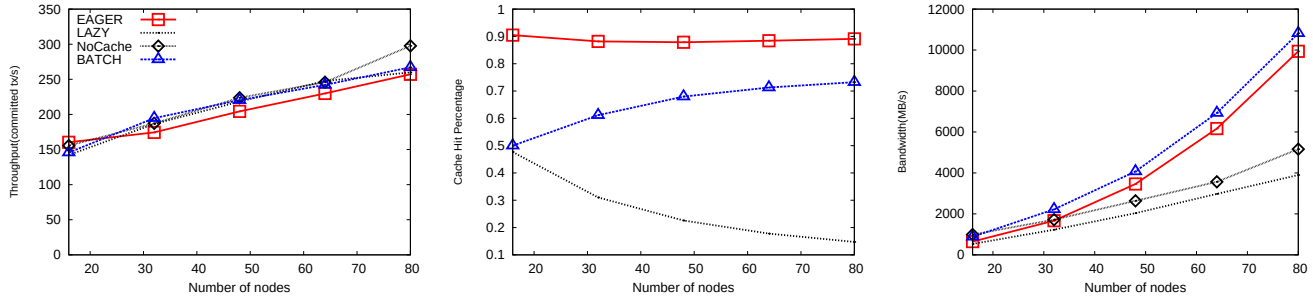

Figure 3. Vacation benchmark deployed on FutureGrid for Workload B. Left to Right: Throughput, Cache Hit Percentage, Total Bandwidth.

We evaluate effectiveness of GMU-C using three key performance indicators, i.e. throughput, cache hit percentage and total consumed bandwidth (across all nodes in the system). We compare the performance of the three invalidation strategies described in Section IV and the one of the base GMU protocol, i.e. without caching support, while varying, on the x-axis, the size of the platform, i.e. number of machines. Among all the results obtained in the experimental phase (which can be found here [22]) we present, for space constraints, a selection of the most representative ones, which allow to evaluate the performance of GMU-C with very different workloads, and both in favourable and adverse conditions.

Figure 1 shows the results obtained with the execution of the Workload A configuration of TPC-C on the CloudTM platform. In this configuration, the EAGER and BATCH variants achieve an extremely high cache hit rate, of about 99%. This results in impressive speedups vs the baseline (NoCache in the plots), which amount to up to 14x. This is because TPC-C includes very long read-only transactions that generate a large network traffic as the system scale grows, and the probability of accessing data locally decreases. This is avoided by the EAGER and BATCH strategies that manage to serve almost the totality of remote read requests. This is not the case for the LAZY strategy, whose performance plunge beyond 16 machines, accompanied by a significant drop of the cache hit rate. This is because TPC-C exhibits a very skewed data popularity, with a very small set of frequently read and accessed data items (e.g., warehouses and districts), which are the cause of a large number of cache misses with the LAZY strategy. Further, the LAZY strategy introduces the costs of building and transmitting

the *Invalidate* messages along the critical path of transaction execution. In this benchmark, this cost is relatively high as the writesets of committed transactions can contain a large number of data items.

Figure 2 shows the results obtained with the execution of the Workload A configuration of the synthetic benchmark on the CloudTM platform. In this case the LAZY variant has its best performance (about 4x speed-up vs the baseline not using caching) if compared to the other approaches mainly because (i) the network traffic produced by all the other strategies has a higher impact due to the very small transaction execution time and (ii) the benchmark does not generate very skewed accesses thus not inducing a significant contention like in TPC-C.

Figure 3 shows the results obtained with the execution of the Workload B configuration of the Vacation benchmark on the FutureGrid platform. In this update intensive scenario, caching is not expected to pay off, since i) update transactions spend most of their execution time in the commit phase, and given that ii) the intense stream of update transactions make this workload intrinsically very hard to be cached. Interestingly, however, even in this adverse scenario the usage of caching mechanism does not result in a reduction of performance with respect to the baseline GMU protocol that does not employ caching.

Summing up the evaluation study it is clearly visible that GMU benefits significantly from the introduction of the proposed caching mechanism, especially, as expectable, in read-dominated workloads. However, the various presented invalidation schemes used in the caching mechanism exhibit different trade-offs.

- LAZY is attractive in communication intensive workloads, where it allows for effectively saving bandwidth. However, by placing the construction and transmission of the invalidation messages in the critical path of transaction execution, it can incur in non-negligible overheads in workloads that generate large iSet messages.
- EAGER is more effective in ensuring high hit rate, especially for applications where read-only transactions are long and the number of keys in update transactions is big, but for smaller transactions the overhead introduced can be detrimental even when having a very good cache hit rate.
- BATCH, despite having the potential for reducing communication and, hence, enhance efficiency, did not prove to be particularly beneficial in any of the considered workloads. The gains in terms of reduced bandwidth with respect to eager, in fact, are normally outweighed by the drop in the cache hit rate imputable to the delays induced by batching the cache invalidation messages.

VI. Conclusions

The potential for scalability of partial replication protocols can be severely hampered by inefficient data placement policies, which can induce poor data access locality as the system scale grows. A possible approach to maximize the efficiency of these protocols is to adopt caching techniques and replicate remote data items that are frequently accessed in an asynchronous (and hence lightweight) fashion.

In this paper, we introduced GMU-C, a distributed caching mechanism for GMU, a recently proposed, highly scalable partial replication protocol for transactional systems. At the time of writing, GMU is being integrated in a mainstream distributed transactional key-value store, Infinispan by Red Hat, greatly amplifying its practical relevance and that of the current work.

We conducted an extensive experimental study to analyze the efficiency and effectiveness of the proposed caching mechanism, using large scale deployments on both private and public cloud infrastructures and adopting diverse benchmarks. The experimental results clearly show that different cache invalidation strategies can exhibit very different performances depending on the environment and the workload, motivating further research in the area of adaptive caching schemes. The results show striking speedups of up to 14 times in read dominated workloads, and a reduction of the network bandwidth by up to one order of magnitude.

References

[1] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," PhD Thesis, Massachusetts Institute of Technology, 1999.

[2] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, Jun. 1981.

[3] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy, "Adaptive Push-Pull: Disseminating Dynamic Web Data". IEEE Trans. Comput., vol. 51, no. 6, pp. 652–668, Jun. 2002.

[4] N. Carvalho, P. Romano, and L. Rodrigues, "Scert: Speculative certification in replicated software transactional memories". In proc. of the *4th Annual International Conference on Systems and Storage*, 2011.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data". *ACM Transactions on Computer Systems*, vol. 26, no. 2, Jun. 2008.

[6] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D^{2}STM: Dependable distributed software transactional memory". In proc. of the *15th IEEE Pacific Rim International Symposium on Dependable Computing*, 2009.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database". In proc. of the *10th USENIX conference on Operating Systems Design and Implementation*, 2012.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store". In proc. of the *21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[9] S. Elnikety, S. Dropsho, and W. Zwaenepoel, "Tashkent+: memory-aware load balancing and update filtering in replicated databases". In proc. of the *2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[10] G. Fox, G. von Laszewski, J. Diaz, K. Keahey, J. Fortes, R. Figueiredo, S. Smallen, W. Smith, and A. Grimshaw, "FutureGrid - a reconfigurable testbed for Cloud, HPC and Grid Computing", *Contemporary High Performance Computing: From Petascale toward Exascale*, April, 2013. Editor J. Vetter.

[11] M. J. Franklin, M. J. Carey, and M. Livny, "Transactional client-server cache consistency: alternatives and performance". *ACM Transactions on Database Systems*, vol. 22, no. 3, pp. 315–363, Sep. 1997.

[12] M. Herlihy and J. E. B. Moss. "Transactional memory: architectural support for lock-free data structures". In Proceedings of the 20th annual international symposium on computer architecture (ISCA '93). ACM, New York, NY, USA, 289-300.

[13] A. Lakshman, and P. Malik, "Cassandra: a decentralized structured storage system". *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[14] G. von Laszewski, G. C. Fox, F. Wang, A. J. Younge, A. Kulshrestha, G. G. Pike, W. Smith, J. Voeckler, R. J. Figueiredo, J. Fortes, et al., "Design of the FutureGrid Experiment Management Framework", GCE2010 at SC10, New Orleans, IEEE, 11/2010.

[15] F. Marchioni and M. Surtani, "Infinispan Data Grid Platform". *Packt Publishing*, Aug. 2012.

[16] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford Transactional Applications for Multi-Processing". In proc. of the *IEEE International Symposium on Workload Characterization*, 2008.

[17] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues, "AutoPlacer: scalable self-tuning data placement in distributed key-value stores". In proc. of the *10th International Conference on Autonomic Computing*, 2013.

[18] R. Palmieri, F. Quaglia, and P. Romano, "AGGRO: Boosting STM Replication via Aggressively Pptimistic Transaction Processing". In proc. of the *9th IEEE International Symposium on Network Computing and Applications*, 2010.

[19] F. Pedone, R. Guerraoui, and A. Schiper, "The Database State Machine Approach". *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 71–98, Jul. 2003.

[20] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication". In proc. of the *IEEE 32nd International Conference on Distributed Computing Systems*, 2012.

[21] S. Peluso, P. Romano, and F. Quaglia, "SCORe: A Scalable One-Copy Serializable Partial Replication Protocol". In proc. of the *ACM/IFIP/USENIX 13th International Middleware Conference*, 2012.

[22] H. Pimentel, "Enhancing locality via caching in the GMU protocol". *MSc Thesis. Instituto Superior Técnico, Universidade Técnica de Lisboa*, Jul. 2013.

[23] F. Raab, "Tpc-c - the standard benchmark for online transaction processing (oltp)". *The Benchmark Handbook*, Morgan Kaufmann, 1993.

[24] N. Schiper, P. Sutra, and F. Pedone, "P-Store: Genuine Partial Replication in Wide Area Networks". In proc. of the *29th IEEE Symposium on Reliable Distributed Systems*, 2010.

[25] D. Serrano, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme, "Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation". In proc. of the *13th Pacific Rim International Symposium on Dependable Computing*, 2007.

[26] D. Serrano, M. Patiño Martínez, R. Jiménez-Peris, and B. Kemme, "An autonomic approach for replication of internet-based services". In proc. of the *IEEE Symposium on Reliable Distributed Systems*, 2008.

[27] R. Van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed systems monitoring, management, and data mining". *ACM Transactions on Computer Systems*, vol. 21, no. 2, pp. 164–206, May 2003.

[28] R. Guerraoui and A. Schiper, "Genuine atomic multicast in asynchronous distributed systems". *Theoretical Computer Science*, vol. 254, no. 1-2, pp. 297–316, Mar. 2011.