When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication

Sebastiano Peluso[†*], Pedro Ruivo[†], Paolo Romano[†], Francesco Quaglia[*], Luís Rodrigues[†]
[†] INESC-ID/IST, Lisbon, Portugal
[*] Sapienza, University of Rome, Italy

***Abstract*—In this article we introduce GMU, a genuine partial replication protocol for transactional systems, which exploits an innovative, highly scalable, distributed multiversioning scheme. Unlike existing multiversion-based solutions, GMU does not rely on a global logical clock, which represents a contention point and can limit system scalability. Also, GMU never aborts read-only transactions and spares them from distributed validation schemes. This makes GMU particularly efficient in presence of read-intensive workloads, as typical of a wide range of real-world applications.**

GMU guarantees the Extended Update Serializability (EUS) isolation level. This consistency criterion is particularly attractive as it is sufficiently strong to ensure correctness even for very demanding applications (such as TPC-C), but is also weak enough to allow efficient and scalable implementations, such as GMU. Further, unlike several relaxed consistency models proposed in literature, EUS has simple and intuitive semantics, thus being an attractive, scalable consistency model for ordinary programmers.

We integrated the GMU protocol in a popular open source in-memory transactional data grid, namely Infinispan. On the basis of a large scale experimental study performed on heterogeneous experimental platforms and using industry standard benchmarks (namely TPC-C and YCSB), we show that GMU achieves linear scalability and that it introduces negligible overheads (less than 10%), with respect to solutions ensuring non-serializable semantics, in a wide range of workloads.

***Keywords*-Partial Data Replication, Multiversioning, Transactional Systems, Fault Tolerance**

I. Introduction

The advent of grid and cloud computing has led to the proliferation of a new generation of in-memory, transactional data platforms, often referred to as NoSQL data grids, such as Cassandra, BigTable, or Infinispan. In these in-memory transactional platforms, data replication plays a fundamental role for both performance and fault-tolerance purposes, since it allows enhancing the throughput by distributing the load and ensuring data survival despite the occurrence of failures. A common trait characterizing this new generation of distributed data platforms is the adoption of a range of weak consistency models, such as eventual consistency [1], restricted transactional semantics (e.g. single object transactions [2], or static transactions whose accessed dataset is pre-declared [3]), and non-serializable isolation levels [4].

The point is that classical strongly consistent transactional replication protocols, based on eager locking and two phase commit (2PC), have well-known scalability issues, imputable to the rapid pace with which the distributed deadlock rate grows as the number of nodes increases [5]. More recent transactional replication protocols (see, e.g., [6]), rely on total order communication primitives [7] (typically provided by a Group Communication Toolkit, such as [8]) precisely in order to avoid distributed deadlocks. However, most of these protocols replicate data synchronously across all nodes, thus leading total order to become the bottleneck as the scale of the system grows.

Other proposals, e.g. [6], aim at higher scalability by relying on, so called, *genuine* replication protocols. Genuineness maximizes scalability by ensuring that, for any transaction T, only the sites that replicate the data items read or written by T exchange messages to decide the final outcome (commit/abort) of T. Unfortunately, these solutions introduce considerable overhead, as they require read-only transactions (that are largely predominant in typical applications' workloads [9]) to undergo expensive distributed validation phases.

In this paper, we present GMU (Genuine Multiversion Update serializability), the first genuine partial replication protocol guaranteeing that read-only transactions are never aborted or forced to undergo any additional remote validation phase.

The core of GMU is a distributed multiversion concurrency control algorithm, which relies on a novel vector clock [10] based synchronization mechanism to track, in a totally decentralized (and consequently scalable) way, both data and causal dependency relations among transactions.

In terms of formal properties, our protocol (as its name suggests) ensures the so called *Extended Update Serializability* (EUS) consistency criterion, originally introduced in [11] and further investigated in [9] (in the form of the *no-update-conflict-misses* property). EUS provides guarantees analogous to those offered by classic 1-Copy Serializability (1CS) for update transactions, thus ensuring consistent evolution of the system's state. Analogously to 1CS, with EUS read-only transactions are also guaranteed to observe a snapshot equivalent to some serial execution (formally, a linear extension [10]) of the partially ordered history of update transactions. Additionally, EUS extends the guarantee of observing consistent snapshots also to transactions that

have to be eventually aborted. This property can be important when the application executes in non-sandboxed environment (such as Transactional Memory systems [12]) and may behave erroneously upon observing non-serializable histories.

However, unlike 1CS, EUS allows concurrent read-only transactions to observe snapshots generated from different linear extensions of the history of update transactions. As a consequence, the only discrepancies in the serialization orders observable by read-only transactions are imputable to the re-ordering of update transactions that neither conflict (directly or transitively) on data, nor are causally dependent. In other words, the only discrepancies perceivable by end-users are associated with the ordering of logically independent concurrent events, which has typically no impact on the correctness of a wide range of real-world applications [9].

The GMU protocol we propose is, to the best of our knowledge, the first genuine partial replication protocol that exploits EUS semantics in order to implement a scalable distributed multiversioning concurrency control scheme that does not introduce any global synchronization point (such as logically centralized clocks [13]), and does not require expensive remote validation phases to commit read-only transactions [6].

We have integrated the proposed GMU protocol into Infinispan, namely a mainstream open-source transactional in-memory data grid developed by the Red-Hat/JBoss team. Given that the native replication mechanism implemented in Infinispan only supports non-serializable consistency (i.e., Repeatable Read), this data grid represents an ideal baseline to evaluate the additional costs incurred in by GMU to ensure EUS consistency. The study has been based on TPC-C [14], an industry standard benchmark for OLTP systems, and YCSB [15], a recent benchmark for distributed key-value stores. The results show that our protocol achieves linear scalability up to 40 nodes, and provides stronger consistency than Repeatable Read at a negligible cost (less than 10%).

The remainder of this paper is structured as follows. In Section II we discuss related work. In Section III we present the model of the target system. Section IV overviews the target consistency model. The GMU protocol is introduced in Section V. The results of the experimental study are illustrated in Section VI. Finally, Section VII concludes the paper.

II. Related Work

Transactional systems' replication is a well investigated topic and a number of proposals exist in literature for dealing with replicated transactional data. Most of these proposals are suited for the case of full replication, where a copy of each data item is retained at each involved site. In this context, solutions have been proposed addressing protocol specification [13], as well as replication architectures that have been based on middleware level approaches [16] and/or on extensions of the inner logic of transactional systems [13]. Comparative studies among these approaches [17] have demonstrated how the solutions based on total order primitives, such as [18], exhibit the potential to improve performance. Also, total order based protocols relying on speculative transaction processing schemes, such as in [19], [20], have been shown to further reduce the impact of distributed synchronization. Consequently, some of these approaches have been demonstrated to be viable even when considering (very) fine-grain transactions, which are typical of workloads of in-memory transactional data layers (e.g., transactional memory systems). On the other hand, compared to all these proposals, in this paper we address performance and scalability of the replicated system from an orthogonal perspective since our focus is on architectures making use of partial data replication.

As for solutions natively oriented to partially replicated transactional systems, literature proposals can be grouped depending on (i) whether they can be considered genuine, and on (ii) the specific consistency guarantees they provide. The proposals in [21], [22] address the case of partial replication, but do not provide genuine approaches since the commitment of a transaction requires interactions with all the sites within the replicated system, and not only with those sites keeping copies of the data read/written by the transaction. Compared to these approaches, our proposal, being genuine, exhibits the potential to increase scalability.

The protocol in [6] provides a genuine solution supporting strict consistency, namely 1CS. However, differently from the present proposal, this protocol imposes that read-only transactions undergo a distributed validation phase. Also, these transactions are potentially subject to rollback/retry. Instead, our proposal only entails a local validation scheme, and never aborts read-only transactions. Overall, compared to the work in [6], our proposal provides a different trade-off between consistency (since it provides EUS semantics) and performance (since it improves the efficiency while handling read-only transactions).

Our proposal has also relations with approaches for the management of distributed transactions in non-replicated systems, especially when considering the type of consistency guarantees that we support. As for this aspect, the works more close to our solution are those proposed in [9], [23], where protocols for distributed transaction processing are presented, which support (E)US semantic. In particular, the proposal in [23] is specifically tailored to exploit this semantic in order not to abort read-only transactions. As hinted before, we still exploit this semantic for the same purposes, but we do this within a partially replicated transactional system, which gives rise to a protocol that addresses orthogonal issues as compared to the one in [23].

As for the reliance on multiversioning, our proposal is also related to the one in [24], where a multiversion concurrency

control mechanism is provided in order to cope with distributed transaction processing in the context of distributed software transactional memories. However, differently from our proposal, this protocol does not cope with (partially) replicated data. Also, it guarantees Snapshot Isolation (SI), while our target is EUS. Hence, our approach guarantees strict serializability at least for update transactions, which is instead not provided by the work in [24].

III. System and Data Model

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. A site that never crashes is correct; otherwise it is faulty.

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 sequence of versions $\langle k, val, ver \rangle$, where k is a key representing d's identifier, val is its value and ver is a scalar, monotonically increasing logical 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. 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 $\bigcup_{j=1 \ldots m} g_j = \Pi$. We denote with $groups(p_i)$ the set of groups to which p_i belongs, and with $proc(g_j)$ the set of processes that replicate data belonging to partition j. 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 [2] based distribution policies in order to: i) minimize data transfer upon joining/leaving of nodes (which, for ease of presentation, we do not model explicitly in this work, although we will briefly discuss how to cope with dynamic groups in Section V-D); ii) ensure the achievement of predetermined replication degrees; iii) avoid distributed lookups to retrieve the identities of the group of processes storing the 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 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 history H over a set of transactions consists of a partial order of events E that reflects the operations (begin, read, write, abort, commit) of those transactions, and a version order $\ll$, that is a total order defined for each data on its committed versions

IV. Consistency Model

Our target consistency criterion is Extended Update Serializability (EUS), which is a stronger variant of Update Serializability (US). In the following we present their formal specifications. US was originally defined by the work in [11], in terms of view serializability, and later re-formulated by Adya [9] in terms of conflict serializability. We report in the following the latter specification of this consistency criterion.

US is specified in terms of topological properties on the, so called, Direct Serialization Graph (DSG). A DSG captures three types of dependency relations:

1) *read dependencies* capture write-read conflicts. T_j directly read-depends on T_i if it reads T_i's updates ($T_i \xrightarrow{wr} T_j$);
2) *write dependencies* capture write-write conflicts. T_j directly write-depends on T_i if it overwrites a data item that T_i has modified ($T_i \xrightarrow{ww} T_j$);
3) *anti-dependencies* capture read-write conflicts. T_j directly anti-depends on T_i if it overwrites a data item that T_i has read ($T_i \overset{rw}{\dashrightarrow} T_j$);

A DSG built over a history H, denoted as DSG(H), has a node for each committed transaction in H and a read/write/anti-dependency edge from transaction T_i to transaction T_j if T_j directly read/write/anti-depends on T_i. A history H guarantees update serializability (also called PL-3U in [9]) if it avoids the following phenomena:

1) *G1a: Aborted Reads.* A history H exhibits phenomenon *G1a* if it contains an aborted transaction T_i and a committed transaction T_j such that T_j has read some data item modified by T_i.
2) *G1b: Intermediate Reads.* A history H exhibits phenomenon *G1b* if it contains a committed transaction T_j that has read a version of data item x written by transaction T_i that was not T_i's final modification of x.
3) *G1c: Circular Information Flow.* A history H exhibits phenomenon *G1c* if DSG(H) contains a directed cycle consisting entirely of dependency edges. Intuitively,

disallowing G1c ensures that if a transaction T_j is affected by the execution of transaction T_i, it does not affect in its turn T_i, i.e., there is a unidirectional flow of information from T_i to T_j.

4) *G-update: Single Anti-Dependency Cycles with Update Transactions.* A history H and transaction T_i show phenomenon *G-update* if a DSG containing all update transactions of H and transaction T_i contains a cycle with 1 or more anti-dependency edges. This property ensures that if T_i depends on T_j, it must not miss the effects of T_j and all update transactions that T_j depends or anti-depends on.

US is weaker than 1CS since read-only transactions are not considered in phenomenon *G-update*. In US, two read-only transactions may observe a serializable state (i.e. obtained via a linear extension of the partial order of events of the update transactions in H, denoted as H^{up}) but unlike 1CS, the serial ordering observed by both transactions could be different (i.e. corresponding to different linear extensions of H^{up}).

Finally, EUS extends US semantic not only to transactions that commit, but to any executing transaction (even if it is later on aborted due to the detection of a non-serializable dependency). This sort of guarantees may be necessary to ensure that the application does not behave in an unexpected manner [25] due to the observation of non-serializable snapshots. If this happens, with US, the transaction will be aborted when it tries to commit. However, before the transaction reaches its commit point, the application program may behave in an unexpected manner, e.g., it may crash, go into an infinite loop, or output unexpected results.

The specification of EUS can be derived from that of US, by: i) including in the DSG used to test property *G1c* (which was built only based on the committed transactions' history) a node for each executing transaction T_i, and the set of dependency/anti-dependency edges associated with T_i's reads; ii) testing whether *G-update* holds for T_i by adding the dependency edges associated with T_i's operations as soon as it executes them, rather than at commit time.

V. THE GMU PROTOCOL

As classical multiversion concurrency control (MVCC) algorithms, GMU stores multiple versions of a same data item, and allows read-only transactions to observe consistent (which for GMU means Update Serializable), but possibly outdated snapshots of the available data. As in typical (centralized [26] or fully replicated [13]) MVCC implementations, in GMU each node arranges the locally stored versions of each data item into a chain tagged with a "version timestamp". This is a scalar, monotonically increasing (integer) clock, which is used to totally order the commit events of transactions that update some locally stored data item. In addition, GMU guarantees that the commit events of transactions that update any data item d belonging to a partition j are totally ordered among all replicas that replicate partition j (namely, g_j).

Algorithm 1 Read/Write operations (node p_i)

```
1:  write(Key k, Value val) from local Transaction T
2:      T.ws ← T.ws ∪ ⟨k, val⟩
3:
4:  read(Key k) from local Transaction T
5:      if k ∈ T.ws then
6:          return T.ws.get(k)
7:      end if
8:      Set rep←proc(k)                      ▷ Processes that replicate k
9:      if p_i ∈ rep then                                  ▷ k is local
10:         [val, VC*, last] ← getV(k, T.VC, T.hasRead)
11:     else                                              ▷ k is remote
12:         send READREQ[k, T.VC, T.hasRead] to rep
13:         receive READREPLY[val, VC*, last]
14:                 from any p_j ∈rep
15:     end if
16:     if ¬ last ∧ T.isNotReadOnly() then
17:         abort T
18:     end if
19:     ∀ r ∈ rep do T.hasRead[r]←true
20:     T.VC ← max(T.VC, VC*)
21:     T.rs ← T.rs ∪ ⟨k, val⟩
22:     return val
23:
24: on receive READREQ[k, T.VC, T.hasRead] from p_j
25:                   ▷ wait for causally dependant transactions
26:     wait until CLog.mostRecentVC[i] ≥ T.VC[i]
27:     [val, VC*, last] ← getV(k, T.VC, T.hasRead)
28:     send [val, VC*, last] to p_j
29:
```

More in general, GMU ensures total order among the commit events of update transactions that exhibit (possibly transitive) data dependencies. This is in fact what guarantees that the history restricted to update transactions generated by GMU is 1CS, as demanded by US.

However, unlike existing distributed/replicated MVCC schemes [21], [22], [27], [13], GMU does not order transaction commit events by relying on a centralized, or fully replicated, global clock. Conversely, GMU relies on a novel, highly scalable, fully distributed synchronization scheme that exploits vector clocks to achieve the twofold objective of:

1) determining which data item versions have to be returned by read operations issued by transactions;
2) ensuring agreement among the nodes replicating the data items updated by a transaction T on the scalar clock to be used when locally applying the write-set of T, as well as on the vector clock to associate with the commit event of T.

Before explaining these two key mechanisms of GMU, we will discuss the main data structures locally maintained at each node p_i, namely $CommitQueue$, $CLog$ and $LastPrepSC$.

$CommitQueue$ is an ordered queue whose entries are

Algorithm 2 Version visibility logic (node p_i)

```
1:  getV(Key k, VC xactVC, boolean[] hasRead)
2:     if ¬hasRead[i] then
3:        MaxVC ← max{vc : vc ∈ CLog ∧
4:                 ∀w (hasRead[w] ⇒ vc[w] ≤ xactVC[w])}
5:     else
6:        MaxVC ← xactVC
7:     end if
8:     int V° ← max{v : k_v ∈ versions(k) ∧ v ≤ MaxVC[i]}
9:     return [k_V°, MaxVC, isMostRecent(k_V°)]
10:
```

tuples $\langle T, VC, status \rangle$ such that T is a transaction, VC is its current vector clock, and status is a value in the domain $\{pending, ready\}$. The entries stored in $CommitQueue$ at node p_i are ordered according to the i-th entry of their vector clocks, and possible ties are broken using deterministic functions (e.g. hashes) on the transaction identifier T.

The status field has the following semantics. If status is equal to $pending$, it means that T is currently successfully prepared by p_i and is waiting for a final commit/abort decision from the transaction coordinator. In the following we will refer to the VC of a pending transaction as to its $prepare\ VC$. The $ready$ value, instead, means that the transaction has already received (a) the commit decision from the transaction originator and that (b) it has already been assigned a final vector clock. We will refer to such a VC as to the $commit\ VC$. As it will be discussed in the following, a $ready$ transaction T will be committed as soon as T becomes the top standing transaction in the $CommitQueue$.

$CLog$ is a simple list that maintains, for each committed transaction, the tuple $\langle T, VC, updatedKeys \rangle$, such that T is the identifier of a committed transaction, VC is its vector clock and $updatedKeys$ is the set of keys locally stored by process p_i that T has updated during its execution.

$LastPrepSC$, finally, is a simple scalar clock that, as it will be discussed, is used by p_i, during the prepare phase of a transaction, to determine (the i-th entry of) its $prepare\ VC$.

In the following we present the pseudocode formalizing the GMU protocol. For space constraints we have to omit the formal correctness proof (which can be found in our technical report [28]). However, while presenting the protocol, we will provide several insights and arguments on its correctness.

A. Transaction execution phase

GMU stores, in the transactional context of each executing transaction T, the following information.

1) The transaction VC, namely an array of scalar (integer) logical timestamps, having cardinality equal to the number of nodes in the system, and whose i-th entry (with $i \in [0, \ldots, n-1]$) keeps track of the (data and causal) dependencies developed by the transaction during its execution.
2) The transaction read-set (rs in the pseudocode), which stores the set of identifiers of the keys read by T.
3) The transaction write-set (ws in the pseudocode), which stores, as a set of pairs $\langle key, value \rangle$, the identifiers and values of the keys written by the transaction.
4) An array of boolean values, called $hasRead$, which has an entry for each node in the system, and whose j-th entry stores the flag indicating whether T has already issued a read operation on a key stored by p_j.

The pseudocode describing the behavior of a transaction during its execution phase is reported in Algorithm 1 and Algorithm 2.

Write operations are simply handled by storing the identifier and the key value in the transaction write-set.

If a transaction T issues a read operation on key k at a process p_i, it is first checked whether T has already written k. In this case, the value stored in T's write-set is returned. Otherwise, T determines whether the key is local or not. If this is the case, the key's value is retrieved from the local data store via the getV() function. This function (see Algorithm 2) iterates over the versions of k and returns the most recent one having a timestamp lower than the maximum one visible by T. This timestamp is determined in different ways depending on whether it is the first time that T issues a read operation on a key stored by p_i.

If this is the case, the most recent local snapshot that is visible by the transaction is determined by iterating over $CLog$ (which we recall stores the totally ordered list of the update transactions that committed at this node) until it is found the most recent transaction T^* whose VC, denoted as $MaxVC$, ensures that T^*'s commit event has not happened after [10] the set of events associated with the first read operation issued by T on any node (see lines 3-4 of Algorithm 2). Roughly speaking, the first time that T issues a read on a node p_i, GMU serializes T after T^*, establishing an upper bound on the $freshness$ of the snapshots that T can observe during subsequent reads. Specifically, this bound prevents T from observing versions committed, on any node, by transactions that depend/anti-depend (either directly or transitively) on T^*.

Note that $MaxVC$ stores in position i (as we are assuming that the read takes place on node i), the $version\ timestamp$ of the most recent transaction committed on p_i whose updates are visible by T. $MaxVC$ can therefore be used (see line 8 of Algorithm 2) to determine the version of k, denoted as k_{V°, that is visible by the transaction.

The behavior in case it is not the first time that T reads on p_i is similar, with the exception that the $MaxVC$ used to determine version visibility is the one already stored in the transaction's VC, which, as it will become clear shortly, corresponds to the $MaxVC$ returned upon the first read of T on p_i.

Finally, getV() returns the value of the selected version, along with $MaxVC$ and a boolean flag that specifies whether the returned version of k is the most recent currently committed. This is perfectly acceptable for read-only transactions, which can be serialized in the past as typical of MVCC algorithms. However, it does doom update transactions to abort, which is the reason why GMU forces the immediate abort of the transaction (see lines 16-17 of Algorithm 1).

It remains to address the case of remote reads (see lines 12-14 and 24-28 of Algorithm 1). In this case, a read request is sent to all the replicas of k, and it is waited for the first of their replies. Analogously to the local read case, the getV() function is used to determine the version of k visible by transaction T. However, in this case GMU ensures that, before invoking getV(), p_j has finalized the commit of all the update transactions from which T depends, and that have written keys replicated by p_j. It is easy to show that this is a necessary condition to ensure property $G1c$.

Independently of whether the read is local or remote, before returning the value of the requested key k to the application, k is added to T's read-set and the set of processes that replicate key k is flagged as already read by T. Finally, the vector clock of T is updated to reflect the happened before relationship [10] between the commit event of the transaction that wrote the version observed by T's read, and the corresponding read event of T.

B. Transaction commit phase

As already anticipated, one of the key strength points of GMU is that it allows committing read-only transactions without requiring any kind of local or remote validation phase.

The scheme used to commit *update* transactions in GMU is specified by the pseudocode shown in Algorithms 3 and 4. GMU uses a Two Phase Commit (2PC) protocol, involving *all and only* the set rep of nodes that replicate keys read or written by a committing transaction T. Exploiting 2PC, GMU can use standard techniques to ensure transaction atomicity and to verify its compatibility with the history of committed (update) transactions. The latter goal is achieved by acquiring read/write locks on all keys read/written by the transaction, at all nodes in which these keys are stored, and then performing a validation of the transaction's read-set.

The key innovative feature of GMU's commit algorithm, however, consists in the scheme employed to establish agreement among the nodes involved in the execution of a transaction T, on the commit vector clock to assign to T. To this end, GMU blends into the 2PC messaging pattern a distributed consensus scheme that resembles the one used by Skeen's total order multicast algorithm [7].

When node p_i receives a prepare message for transaction T (and after its successful validation), it sends back with the VOTE message the proposal of a new vector clock for T. This proposal is built by incrementing $LastPrepSC$ and constructing a vector clock equal to the one of the most recent locally committed transaction, except for its i-th entry, which is set to the new value of $LastPrepSC$. The prepare phase is concluded by storing the $prepare\ VC$ in the local $CommitQueue$.

The 2PC coordinator gathers the proposed prepare VCs, and performs two operations in order to derive the $commit\ VC$ for the transaction. First, it merges the $prepare\ VC$s with the current VC of the transaction using the max operator, which outputs a VC having, for each of its entries i, the maximum of the i-th entry of the VC passed as input. This allows to keep track in the $commit\ VC$ of the causal dependencies developed both by T during its execution as well as by the most recently committed transactions at all the nodes contacted by T. Next, the coordinator determines the common value to attribute to the entries of the $commit\ VC$ related to the nodes whose keys have been updated by the transaction (i.e. the nodes $p_j \in proc(T.ws)$). This is achieved by picking the maximum value among all the entries in the $prepare\ VC$ of each node involved in the commit.

At this point the coordinator sends back a commit notification to all the nodes in rep. This triggers the update of the entry associated with transaction T in $CommitQueue$, whose VC is replaced with the $commit\ VC$ and whose status is set to ready.

In order to finalize the commit of T, however, it is waited until its entry has become the first in $CommitQueue$ (recall that the $CommitQueue$ at process p_i is ordered based on the i-th entry of the VCs that it contains). This scheme guarantees that all the nodes updated by a transaction T will assign the same $commit\ VC$ to T. It also ensures that if a node p_i commits a transaction with a local scalar timestamp (i.e., having as i-th entry in its VC the value) equal to v, then the local scalar timestamps of the transactions that subsequently commit at p_i will be larger than v.

The two aforementioned properties guarantee that all the replicas in Replicas($T.ws$) commit T using the same $commit\ VC$ and in the same total order.

Finally, the merging of the causal histories encoded by the transaction's VCs and by the $prepare\ VC$s (see line 19 of Algorithm 3) guarantees that the total order of the commit events is propagated across chains of, possibly transitively dependant transactions. This represents one of key mechanisms leveraged on by GMU in order to ensure 1CS of the history of update transactions.

C. Garbage Collection

GMU integrates an efficient distributed garbage collection protocol that relies on background dissemination (either via gossip [29] and/or via piggybacking) of the VC of the oldest active transaction known as active at each node. This lightweight rumor-mongering mechanism allows each

Algorithm 3 Commit phase (node p_i)

```
1:  validate(Key k, VC xactVC) on node p_i
2:      return max{v : k_v ∈ versions(k)} ≤ xactVC[i]
3:
4:  boolean commit(Transaction T)
5:      VC commitVC ← T.VC
6:      int xactVN                    ▷ Timestamp for data updated by T
7:      if T.ws = ∅ then              ▷ Read-only transactions commit locally
8:          return true
9:      end if
10:     Set rep ←proc({T.rs ∪ T.ws})   ▷ rep may include p_i
11:     send PREPARE[T] to all p_j ∈ rep
12:     for all p_j ∈ rep do
13:         wait VOTE[vote_j, VC_j] from p_j
14:         if vote_j=NO then
15:             send ABORT[T] to all p_j ∈ rep
16:             return false
17:         else
18:                             ▷ Ensure tracking of causal dependencies
19:             commitVC ←max(commitVC, VC_j)
20:         end if
21:     end for
22:     xactVN ←max{commitVC[w] : p_w ∈ Π}
23:     for all p_j ∈ rep s.t. p_j ∈ proc(T.ws) do
24:         commitVC[j]← xactVN
25:     end for
26:     send COMMIT[T, commitVC] to all p_j ∈ rep
27:     return true
28:
```

Algorithm 4 Prepare/Commit/Abort messages (node p_i)

```
1:  on receive PREPARE[T] from p_j
2:      for all k ∈ {T.rs ∪ T.ws} do
3:          if k ∈ T.rs ∧ k is local then
4:              Acquire read lock on k
5:              if (failed lock on k ∨ ¬validate(k, T.VC) then
6:                  send VOTE[NO,-]
7:                  release any lock held by T
8:                  return
9:              end if
10:         end if
11:         if k ∈ T.ws ∧ k is local then
12:             Acquire write lock on k
13:             if failed write lock on k then
14:                 send VOTE[NO,-]
15:                 release any lock held by T
16:                 return
17:             end if
18:         end if
19:     end for
20:     VC prepVC ← CLog.mostRecentVC
21:     prepVC[i]←incrementAndGet(LastPrepSC)
22:     CommitQueue.put(⟨T, prepVC, pending⟩)
23:     send VOTE[YES, prepVC] to p_j
24:
25: on receive COMMIT[T, commitVC] from p_j
26:     LastPrepSC ←max(LastPrepSC,commitVC[i])
27:     CommitQueue.update(⟨T, commitVC, ready⟩)
28:
29: on first ⟨T, vc, s⟩ in CommitQueue has s = ready
30:     ∀k ∈ T.ws s.t. isLocal(k) do apply(k,vc[i])
31:     CLog.add(vc)
32:     CommitQueue.remove(T)
33:     release any lock held by T
34:
35: on receive ABORT[T] from p_j
36:     CommitQueue.remove(T)
37:     release any lock held by T
38:
```

node to determine a conservative estimative of the scalar timestamp, say t^*, of the oldest locally committed transaction whose versions are still visible by any currently active transaction. This means that it is possible to garbage collect every version having timestamp less than t^* (provided that a fresher version has already been committed) without risking to remove versions that may be later requested by some transaction.

Note that since the commit log stores the references to the write-sets of local committed transactions, it can be used as an index to efficiently retrieve any obsolete version committed by transactions older than t^*.

D. Failure Handling and Dynamic Process Groups

With respect to conventional 2PC based transactional systems, GMU does not introduce additional sources of complexity for what concerns the handling of failures and of dynamic process groups. If, as it is typically the case in practice [8], the underlying group communication toolkit provides virtual synchrony guarantees, it is straightforward, upon a view change notification, to associate the initial vector clock to be used during the next view v with the nodes that transit in v.

Finally, as GMU relies on 2PC, which is well known to be blocking upon failure of the coordinator, additional, orthogonal solutions need to be taken in order to deal with these scenarios. Possible solutions include schemes based on the synchronous replication of the coordinator state across a (typically small) set of processes, such as in [30], or, pragmatic solutions that admit heuristic exceptions, trading-off consistency for performance.

VI. Experimental Evaluation

We have integrated GMU in Infinispan[1], a mainstream open source in-memory distributed data platform. Analogously to many other contemporary distributed cache platforms, Infinispan externalizes a simple key-value store interface. In order to maximize scalability, Infinispan relies on weak consistency models, and on a lightweight consistent hashing scheme [31] that allows partitioning data efficiently while ensuring good load balancing and minimum reshuffling of keys in presence of joins/departures of nodes from

[1] An open-source implementation of GMU is available at this public repository: https://github.com/cloudtm

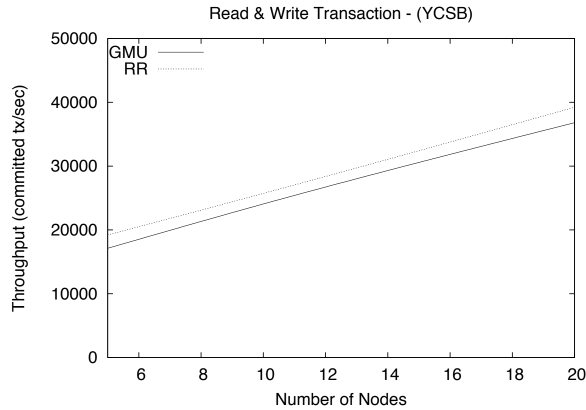


Figure 1: YCSB Benchmark (Cloud-TM).

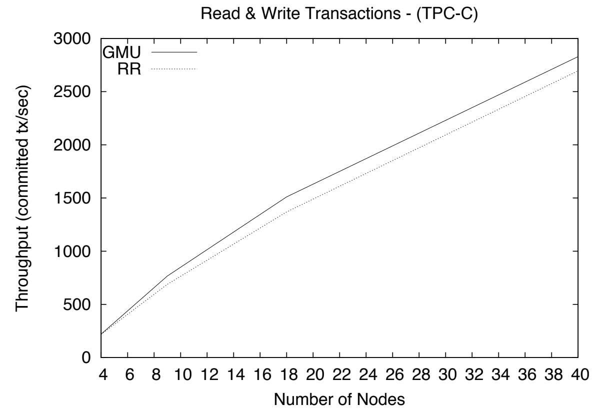


Figure 2: TPC-C Benchmark (FutureGrid).

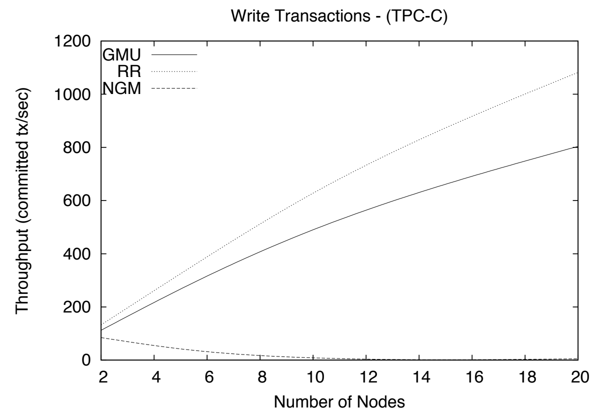
the platform. Further, Infinispan supports partial replication, allowing to store each key across a fixed, user-tunable number of replicas, thus achieving fault-tolerance without hampering scalability.

For what concerns consistency, the stronger consistency level ensured by Infinispan is Repeatable Read [4] (RR), an isolation level which ensures that no intermediate or aborted values are ever observed, and that no two consecutive reads within the same transaction can return different values. RR is significantly weaker than (E)US, as it allows the commit of (both read-only and update) transactions that observe non-serializable schedules [9].

Infinispan relies on an encounter based two phase locking scheme, which is applied only to write operations and that does not synchronize reads. Repeatability of read operations is instead guaranteed by storing the data items observed by read operations, and returning them upon subsequent reads. For what concerns the replication protocol of Infinispan, it relies on a classic 2PC-based distributed locking algorithm [5].

Designed to achieve high scalability and support weak consistency models, Infinispan represents an ideal baseline to evaluate the costs incurred in by GMU to provide stronger consistency guarantees. We implemented also a non-genuine multiversion-based replication scheme, which, analogously to the one in [13], relies on a fully replicated, logically centralized, global scalar clock, used to totally order committing update transactions. We refer to this protocol as NGM (Non-Genuine Multiversioning) in the following.

For our experimental study we use two well-known benchmarks, TPC-C [14] and YCSB [15]. The workload generated by TPC-C is representative of OLTP environments and characterized by complex and heterogeneous transactions, with very skewed access patterns and high conflict probability. YCSB (Yahoo! Cloud Serving Benchmark) [15] is a framework specifically aimed at benchmarking NoSQL key-value data grids and cloud stores. The transactional profile



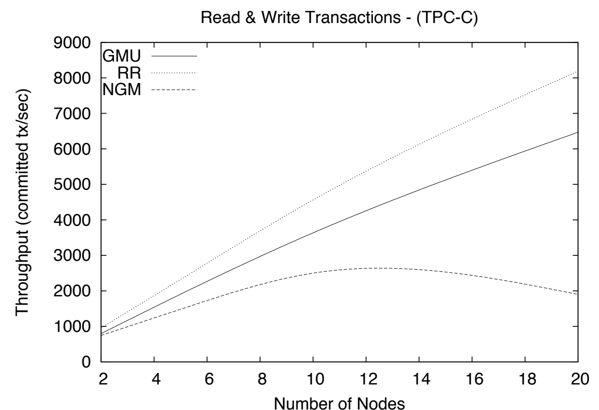


Figure 3: TPC-C Benchmark (Cloud-TM)

.

of this benchmark is quite different from the one of TPC-C, with simpler, shorter transactions that rarely conflict.

The results presented in the following were obtained using two experimental platforms. The first one, FutureGrid [2], is a public distributed test-bed for parallel and cloud computing. This platform allows us to evaluate GMU in environments

[2]www.futuregrid.org

representative of public cloud infrastructures, which are typically characterized by more competitive resource sharing, ample usage of virtualization technology, and relatively less powerful nodes. In the FutureGrid platform we performed experiments using up to 40 virtual machines, equipped with 7GB RAM, two 2.93GHz cores Intel Xeon CPU X5570, running CentOS 5.5 x86_64. All the VMs were deployed in the same physical data-center and interconnected via Gigabit Ethernet. In all experiments performed on FutureGrid we used a single thread per node to inject transactions (in closed loop), which guaranteed a high utilization of the machine's resources without overloading.

The second experimental platform, referred to as Cloud-TM, is a dedicated cluster of 20 homogeneous nodes, where each machine is equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 16 GB of RAM, running *Linux 2.6.32-33-server* and interconnected via a private Gigabit Ethernet. This platform is representative of small/medium private clouds or data-centers environments, with dedicated servers and a fairly large amount of available (computational and memory) resources per node. In order to maintain a similar ratio between threads and available cores with respect to the experiments performed in FutureGrid, in all experiments performed on Cloud-TM we used four threads per node to inject transactions (in closed loop).

Let us start by analyzing the results obtained by running YCSB using the Cloud-TM platform. We used Workload A [15] of the benchmark, which is an update intensive workload (comprising 50% of update transactions) simulating a session store that records recent client actions. Figure 1 reports the maximum throughput (committed transactions per second) achievable by GMU and by Infinispan's partial replication protocol that ensures Repeatable Read (referred to as RR, in the following). The plot shows that the average reduction in throughput for GMU oscillates around 8%, and that it scales linearly at the same rate as RR, providing an evidence of the efficiency and scalability of the proposed solution.

Next we report, in Figure 2, the results achieved using, on FutureGrid, the TPC-C benchmark configured with a read-dominated profile, composed at the 90% by read-only (Order-Status profiles) transactions and, for the remaining 10%, by update transactions (Payment and New-Order profiles) in equal parts. The plot confirms the efficiency and scalability of GMU. Surprisingly, in this scenario, despite providing consistency guarantees, GMU even outperforms RR by up to 10%. Our profiling study has highlighted that these gains are imputable to the fact that GMU, unlike RR, avoids the overhead of storing previously read values to guarantee consistency. Read-only transactions in TPC-C, in fact, tend to perform a large number of operations, forcing RR to perform a large number of cloning operations to store read versions in the transactional context.

Finally, in Figure 3, we report the results achieved by running TPC-C on the Cloud-TM platform. With 4 threads injecting transactions per node, the degree of concurrency is significantly higher than in the former experiment, leading to significant conflicts both at the logical (data) and at the physical (computing/network resources) level. The plots in this case report also the performance of the above described non-genuine multiversion partial replication protocol (NGM). Our experimental data clearly demonstrate the detrimental effect on system scalability due to the high logical contention on the fully replicated global clock, which leads to the drastic decay of the throughput (in particular of write transactions, see plot on top of Figure 3).

By comparing the performance of GMU with that of RR it emerges that the increase in the logical/physical contention level characterizing this configuration has a stronger impact on GMU. In fact, even though GMU shows an almost linear scalability trend, our data reveal that it suffers of a higher abort rate than RR: for 20 nodes, the abort rate is on the order of the 15% for GMU, whereas it is around 8% for RR. This data shows that, in high contention scenarios, strong consistency semantics do pay a performance toll, which, in this specific configuration, corresponds to a throughput reduction ranging from 10% (at 4 nodes) up to 20% (at 20 nodes). On the other hand, we argue that this is an unavoidable cost to pay in the context of applications whose correctness can be endangered by adopting non-serializable isolation levels. Note that TPC-C does belong to this class of applications, as in fact several of its transactional profiles might generate data corruptions in presence of concurrency anomalies such as those possible using Repeatable Read (or even Snapshot Isolation).

VII. Conclusions

In this article we presented GMU (Genuine Multiversion Update serializability), an innovative partial replication protocol for transactional systems. The core of GMU is a distributed multiversion concurrency control scheme, which relies on a novel vector clock based synchronization algorithm to track, in a totally decentralized (and consequently scalable) way, both data and causal dependency relations. In order to maximize scalability, GMU adopts a genuine partial replication mechanism that ensures that transactions only contact replicas storing the data that they accessed. Further, GMU never aborts read-only transactions and spares them from expensive distributed validation schemes. This makes GMU particularly efficient in presence of read-intensive workloads, as typical of a wide range of real-world applications.

We evaluated GMU by integrating it into Inifinispan, a mainstream in-memory key-value store, and performing a large scale experimental study based on heterogeneous experimental platforms and industry standard benchmarks (namely TPC-C and YCSB). Our results show that GMU achieves linear scalability and that it introduces negligible

overheads (less than 10%) with respect to solutions ensuring non-serializable semantics in a wide range of workloads.

Supported by these experimental results, we argue that GMU hits a sweet spot in the trade-off between consistency and performance. In fact, the consistency semantics guaranteed by GMU, namely Extended Update Serializability (EUS), is sufficiently strong to ensure the correctness even of complex OLTP workloads (such as TPC-C), but also weak enough to allow for efficient and scalable implementations. Further, unlike several relaxed consistency models proposed in literature, EUS has simple and intuitive semantics, which represents an essential requirement for adoption in mainstream, complex applications.

ACKNOWLEDGMENTS

This work has been partially supported by the project "Cloud-TM" (co-financed by the European Commission through the contract no. 257784), the FCT project ARISTOS (PTDC/EIA- EIA/102496/2008) and by FCT (INESC-ID multiannual funding) through the PIDDAC Program Funds.

REFERENCES

[1] 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 Symposium on Operating Systems Principles*, ser. SOSP '07. ACM, 2007, pp. 205–220.

[2] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Operating Systems Review*, vol. 44, pp. 35–40, 2010.

[3] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," *SIGOPS Operating Systems Review*, vol. 41, pp. 159–174, 2007.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Proc. of International Conference on Management of Data*, ser. SIGMOD '95. ACM, 1995, pp. 1–10.

[5] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. of International Conference on Management of Data*, ser. SIGMOD '96. ACM, 1996, pp. 173–182.

[6] N. Schiper, P. Sutra, and F. Pedone, "P-Store: Genuine Partial Replication in Wide Area Networks," in *Proc. of IEEE Symposium on Reliable Distributed Systems*, 2010, pp. 214–224.

[7] X. Defago, A. Schiper, and P. Urban, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, 2004.

[8] H. Miranda, A. Pinto, and L. Rodrigues, "Appia: A Flexible Protocol Kernel Supporting Multiple Coordinated Channels," in *Proc. of International Conference on Distributed Computing Systems*, ser. ICDCS '01. IEEE Computer Society, 2001, pp. 707–710.

[9] A. Adya, "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions," PhD Thesis, Massachusetts Institute of Technology, Tech. Rep., 1999.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communication of the ACM*, vol. 21, pp. 558–565, 1978.

[11] R. Hansdah and L. Patnaik, "Update serializability in locking," in *Proc. of International Conference on Database Theory*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1986, vol. 243, pp. 171–185.

[12] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha, "Unlocking Concurrency," *ACM Queue*, vol. 4, no. 10, pp. 24–33, 2007.

[13] B. Kemme and G. Alonso, "Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication," in *Proc. of International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 2000, pp. 134–143.

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

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. of ACM symposium on Cloud computing*, ser. SoCC '10. ACM, 2010, pp. 143–154.

[16] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris, "Middleware based Data Replication providing Snapshot Isolation," in *Proc. of International Conference on Management of Data*, ser. SIGMOD '05. ACM, 2005, pp. 419–430.

[17] M. Wiesmann and A. Schiper, "Comparison of Database Replication Techniques Based on Total Order Broadcast," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 4, pp. 551–566, 2005.

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

[19] R. Palmieri, F. Quaglia, and P. Romano, "OSARE: Opportunistic Speculation in Actively REplicated Transactional Systems," in *Proc. of IEEE Symposium on Reliable Distributed Systems*, ser. SRDS '11. IEEE Computer Society, 2011, pp. 59–64.

[20] N. Carvalho, P. Romano, and L. Rodrigues, "SCert: Speculative certification in replicated software transactional memories," in *Proc. of International Conference on Systems and Storage*, ser. SYSTOR '11. ACM, 2011, pp. 10:1–10:13.

[21] 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 Pacific Rim International Symposium on Dependable Computing*, ser. PRDC '07. IEEE Computer Society, 2007, pp. 290–297.

[22] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. G. de Mendívil, and F. D. Muñoz Escoí, "SIPRe: a partial database replication protocol with SI replicas," in *Proc. of ACM Symposium on Applied Computing*, ser. SAC '08. ACM, 2008, pp. 2181–2185.

[23] A. Chan and R. Gray, "Implementing Distributed Read-Only Transactions," *IEEE Transactions on Software Engineering*, vol. 11, no. 2, pp. 205–212, 1985.

[24] A. Bieniusa and T. Fuhrmann, "Consistency in hindsight: A fully decentralized STM algorithm," in *Proc. of IEEE International Symposium on Parallel and Distributed Processing*, ser. IPDPS '10, 2010, pp. 1–12.

[25] R. Guerraoui and M. Kapalka, "On the Correctness of Transactional Memory," in *Proc. of ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '08. ACM, 2008, pp. 175–184.

[26] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[27] S. Wu and B. Kemme, "Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation," in *Porc. of International Conference on Data Engineering*, ser. ICDE '05. IEEE Computer Society, 2005, pp. 422–433.

[28] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication," INESC-ID, Tech. Rep. 47, 2011.

[29] 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. 3, 2003.

[30] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Transactions on Database Systems*, vol. 31, pp. 133–160, 2006.

[31] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *Proc. of ACM Symposium on Theory of Computing*. ACM, 1997, pp. 654–663.