Supporting Multiple Data Replication Models in Distributed Transactional Memory

João A. Silva, Tiago M. Vale, Ricardo J. Dias, Hervé Paulino, and João M. Lourenço
NOVA LINCS/CITI – Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
{jaa.silva,t.vale}@campus.fct.unl.pt {rdias,herve.paulino,joao.lourenco}@fct.unl.pt

ABSTRACT

Distributed transactional memory (DTM) presents itself as a highly expressive and programmer friendly model for concurrency control in distributed programming. Current DTM systems make use of both data distribution and replication as a way of providing scalability and fault tolerance, but both techniques have advantages and drawbacks. As such, each one is suitable for different target applications, and deployment environments. In this paper we address the support of different data replication models in DTM. To that end we propose ReDstm, a modular and non-intrusive framework for DTM, that supports multiple data replication models in a general purpose programming language (Java). We show its application in the implementation of distributed software transactional memories with different replication models, and evaluate the framework via a set of well-known benchmarks, analysing the impact of the different replication models on memory usage and transaction throughput.

Categories and Subject Descriptors

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

General Terms

Design, Experimentation, Performance

Keywords

Data Replication; Distributed Transactional Memory; Concurrency Control; Distributed Systems

1. INTRODUCTION

Cloud computing has democratized the access to distributed computing infrastructures, popularizing the use of distributed systems to meet the ever-growing scalability, availability, and low latency requirements of modern Internet services. Many of these distributed software systems require the sharing of data among their (distributed) processes, a fact that increases the systems' overall design complexity and hinders their performance. In this context, distributed transactional memory (DTM) presents itself as a highly expressive and programmer friendly alternative for concurrency control in such software systems. It extends the scope of the software transactional memory (STM) model [18] to distributed environments, combining it with data replication and distribution. The result is a high-level concurrency control mechanism that offers transactional semantics over a distributed shared memory addressing space.

Despite being initially studied in the context of chip-level multiprocessing, the benefits of STM over traditional concurrency control may also be harvested in distributed environments. To that end, recent research has led to the development of multiple DTM frameworks [4, 11, 15, 20]. The majority of these have grown from existing STM frameworks, by adding communication layers, schedulers, mechanisms for replication and contention management, and object lookup. Moreover, these frameworks offer *intrusive* programming models, demanding the rewriting of the application code in order to comply to their specificities. This makes the decision of using a specific framework a rather serious commitment as new applications become tied to the provided APIs, and later porting of existing applications to another framework is a tedious and error prone job.

Naturally, each data replication strategy has its advantages and drawbacks, hence being suitable for different target applications, and deployment environments. This raises another issue: the bulk of DTM frameworks are tailored for a concrete data replication strategy, not allowing the unbiased evaluation of different strategies under the same circumstances. In the control-flow model [15,20] data is immobile and transactions access data through remote procedure calls (RPCs). Contrary, in the data-flow model [15,20] transactions are immobile and data moves through the network to requesting transactions. In what regards data replication, the full replication model [4] replicates all data items in all the system's nodes. This strategy provides the best possible tolerance to data loss but limits the system's total storage capacity and requires coordination between *all* the nodes, which raises scalability issues [17]. Conversely, the partial replication model replicates each data item in only a *subset* of the system's nodes, i.e., data items are partially replicated. This model provides a reasonable tolerance to data loss but requires nodes to perform remote read operations searching for data items that are not locally replicated.

In this paper we present ReDstm, a *modular*, *flexible* and

ICDCN '15, January 04 - 07 2015, Goa, India

http://dx.doi.org/10.1145/2684464.2684481.

non-intrusive framework for DTM. ReDstm's modularity allows for different implementations of the local STM algorithm, of the transaction validation protocol, of the communication system, among others. It also intrinsically provides support for multiple data replication models, ranging from no replication to partial and full replication. Another distinctive feature of ReDstm is its non-intrusive, *annotation-based*, programming model. The choice of a particular replication strategy for a data structure may have a considerable impact of the system's performance, and hence should derive from a careful analysis. Therefore, despite the high-level nature of the programming model, the task of choosing the appropriate replication strategy may not be trivial. Hence, we present guidelines for combining full and partial replication when faced with the choice of different replication strategies for a data structure.

Thus, the contributions of this paper are as follows: (i) we draft a modular Java DTM framework supporting different data replication models; (ii) we propose a highly expressive and non-intrusive programming model for defining the corresponding data replication strategy, along with guidelines for when combining full and partial replication in ReDstm; and (iii) we evaluate the impact of applying different replication strategies to well-know transactional benchmarks running on top of ReDstm.

The remainder of this paper is organized as follows: §2 presents ReDstm; §3 elaborates on how to implement distributed STMs, with different data replication models, on top of ReDstm; §4 proposes guidelines for when combining full and partial replication in ReDstm; §5 presents and analyses experimental results; §6 discusses related work; and §7 concludes this paper.

2. A MODULAR FRAMEWORK FOR DTM

ReDstm builds upon the TribuSTM framework [5] to offer a generic framework for DTM. It extends TribuSTM with support for (i) distributed objects, including data placement and data replication strategies; and (ii) the distributed commit of transactions accessing these objects.

2.1 TribuSTM & Deuce

All STM algorithms associate some information to each managed memory location (*metadata*), whose nature varies depending on the algorithm itself, e.g., locks, timestamps, version lists. Such metadata can be stored either in an external table (*out-place* strategy) or adjacent to each memory location (*in-place* strategy).

The out-place strategy is implemented using a table-like data structure that efficiently maps memory locations to metadata. Storing metadata in such a pre-allocated table avoids the overhead of dynamic memory allocation, but incurs in the overhead of evaluating the mapping function. Additionally, the bounded size of the external table induces a false sharing situation where multiple memory locations share the same table entry and hence the same metadata, resulting in a *many-to-one* relation between memory locations and metadata. This approach suits STM algorithms with weak ties between the metadata and the associated memory locations, e.g., the TL2 algorithm [6] whose metadata are locks. However, it falls short when these ties grow stronger, e.g., the version list associated with each memory location in JVSTM [3]. A direct dependency relationship between the metadata and the associated memory locations requires them to be mapped by a *one-to-one* relation.

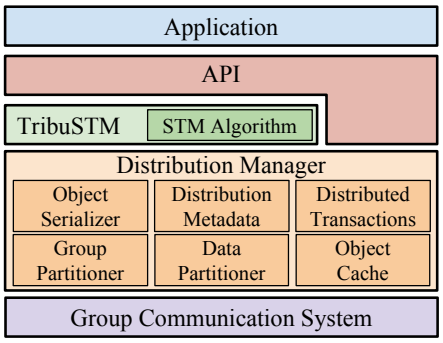


Figure 1: ReDstm's architecture overview.

The in-place strategy provides such *one-to-one* relation, preventing the occurrence of false sharing situations. This strategy is usually implemented by using the *decorator* design pattern [7], where the functionality of an original class is wrapped in a decorator class that contains the required metadata. This allows direct access to the metadata but is very intrusive to the application's code, which must be rewritten to use the decorator classes.

Deuce [10] is a Java STM framework that allows for the implementation of various STM algorithms, standing out for its non-intrusion to the application programmer. However, its applicability is bound to STM algorithms that can use the out-place strategy. TribuSTM [5] extends the Deuce STM framework with support for the in-place metadata placement strategy, *without* making use of the decorator pattern.

2.2 Supporting Distribution and Replication

Supporting DTM on top of TribuSTM raises some challenges that drive the organization of this subsection, namely: (i) how to design a modular DTM framework; (ii) where to store distributed data and metadata and how to access them, and; (iii) how to validate and apply transactions in a distributed context.

2.2.1 Architecture Overview

The architecture of ReDstm comprises several modules, organized in three layers (see Figure 1). From a bottom-up perspective, the group communication system (GCS) provides inter-node communication, the distribution manager (DM) is responsible for all the distribution-related issues, and the TribuSTM layer is in charge of the concurrency control at each node. Being modular, ReDstm allows different implementations of all the modules that will be described.

Directing our attention to the first two layers, the purpose of the DM is twofold: implement a distributed (shared) memory space that installs the desired data replication model (distribution, full or partial replication), and implement protocols to support the execution of distributed transactions. To that end, it is comprised by the following modules:

Object serializer (ObS) Implements the serialization logic for the remote communication of objects;

Distribution metadata (DMd) Each data replication model requires its own distribution metadata, where it stores information on the data's whereabouts. Together with the ObS, they establish the logic for objects' distribution.

Distributed transactions (DT) Regulates the implementation of distributed transaction validation protocols, embedding all the interaction with TribuSTM.

Group partitioner (GP) Divides the system's nodes into well defined groups. This module is particularly relevant in the context of partial replication. A group defines the system's *unit of replication*, i.e., given a group of nodes $\mathcal{G}$ and a distributed object $\mathcal{O}$, if $\exists\ \mathcal{N} \in \mathcal{G} : \mathcal{O}$ is replicated in $\mathcal{N} \Rightarrow \forall \mathcal{N}' \in \mathcal{G}$, $\mathcal{O}$ is replicated in $\mathcal{N}'$.

Data partitioner (DP) It is also a module tailored for partial replication environments. It establishes a mapping function between the objects that comprise the distributed shared memory space and the groups on which they must be replicated on.

Object cache (ObC) Provides a generic caching service for distributed objects. It defines a mapping between distributed metadata and generic objects (see §2.2.5).

The GCS layer encapsulates the communication primitives required to implement the DM (e.g., atomic multicast, reliable unicast/multicast) providing an uniform interface regardless of the concrete communication system being used.

2.2.2 Metadata & Distributed and Shared Memory

ReDstm supports object distribution by combining metadata with the Java serialization support. As highlighted in §2.1, STM algorithms associate metadata to managed memory locations. Similarly, the implementation of different data replication models may also grow from the metadata associated to each managed memory location. For instance, in a full replication scenario one may associate a unique identifier to each memory location, allowing the system to recognize that location in the local replica. In turn, in a purely distributed context, the metadata may contain the information necessary for the execution of a RPC targeted to the *owner* of the corresponding memory location.

Note that we are referring to two different kinds of metadata. STM algorithms associate metadata to each field of an object, in order to manage concurrency. Plus, the support for distributed objects builds from object-level metadata to implement the desired data replication strategy. These are clearly two distinct kinds of metadata, thus, to disambiguate whenever necessary, we shall refer to them as *transactional* and *distribution* metadata, respectively.

Metadata. Regardless of the data replication strategy in place, the framework requires the association of distribution metadata to objects. As such, in order to be ReDstm-compliant, an object must implement the `DistributedObject` interface, with methods `getMetadata` and `setMetadata`. This interface provides the means for the framework to access the object's metadata (a subclass of `DistMetadata`).

Serialization. A ReDstm distributed object is serialized in function of its distribution metadata, so that different implementations of the DM may serialize objects differently, according to their replication model. For this purpose, objects have to implement methods `writeReplace` and `readResolve` (from the `java.io.Serializable` interface).

In order for ReDstm to be both flexible and non-intrusive, instead of requiring the application programmer to explicitly implement the `DistributedObject` interface, the framework features an instrumentation agent that automatically injects such code into the application.

Table 1: Operations provided by the reflexive API.

Operation	Description
onStart($\mathcal{T}$)	Start of transaction $\mathcal{T}$.
onRead($\mathcal{T}$, m)	Read on transactional metadata m by transaction $\mathcal{T}$.
onWrite($\mathcal{T}$, m, v)	Write of value v on transactional metadata m by transaction $\mathcal{T}$.
onCommit($\mathcal{T}$)	Commit request issued from transaction $\mathcal{T}$.
onAbort($\mathcal{T}$)	Abort of transaction $\mathcal{T}$.

Table 2: Operations provided by the actuator API.

Operation	Description
createState($\mathcal{T}$) : $\mathcal{S}$	Returns a representation of transaction's $\mathcal{T}$ state.
recreateTx($\mathcal{S}$) : $\mathcal{T}$	Returns a transaction $\mathcal{T}$ recreated from state $\mathcal{S}$.
validate($\mathcal{T}$) : *bool*	Validates transaction $\mathcal{T}$.
applyWS($\mathcal{T}$)	Applies all updates by transaction $\mathcal{T}$ on the local STM.

The instrumentation is automatically applied to every loaded class, transforming each class' implementation into one that is ReDstm-compliant (via Java bytecode rewriting). Let $\mathcal{C}^{\mathcal{I}} = \{f_1, \ldots, f_n, m_1, \ldots, m_k\}$ denote a class $\mathcal{C}$ that implements the set of interfaces $\mathcal{I}$ and where f_i and m_j, with $i \leq n$ and $j \leq k$, represent the fields and methods of the class, respectively. The transformations applied upon $\mathcal{C}$ aim at turning it into a subtype of the `DistributedObject` interface, to insert a new field f^{dm} of type `DistMetadata`, and to add methods `writeReplace` and `readResolve`. Namely:

$$\mathcal{C}^{I \cup \{\texttt{DistributedObject}\}} = \left\{ \begin{array}{c} f^{dm}, \\ f_1, \ldots, f_n,\ m_1, \ldots, m_k, \\ \texttt{getMetadata},\ \texttt{setMetadata}, \\ \texttt{writeReplace},\ \texttt{readResolve} \end{array} \right\}$$

2.2.3 Distributed Transactions

To allow the flexible integration of various realizations of the DM layer with TribuSTM, the DM and TribuSTM interact through two distinct and well defined interfaces. Furthermore, in order to support distributed transactions, the DM is aware of some of the events triggered by the application on TribuSTM, such as transactional accesses and commit requests, and may also query or modify the state of the local STM, e.g., to apply updates made by a remote transaction.

With the first interface, the *reflexive* API (see Table 1), the DM is able to react to certain events from TribuSTM. It registers callbacks for transactional-related events such as transaction start (onStart), transactional accesses (onRead, onWrite), and commit and abort (onCommit, onAbort).

The second interface, the *actuator* API (see Table 2), enables the DM to inspect the state of the local STM and act upon it. It can acquire an opaque representation of a transaction's state (createState) which can then be used to recreate the transaction (recreateTx). It can also explicitly trigger the validation (validate) and apply the updates (applyWs) of a transaction.

The reaction of the DM to an event may trigger synchronous request-reply communication with remote nodes. Accordingly, the execution of the DM blocks until one of the callback methods (startProcessed, readProcessed, writeProcessed, commitProcessed, and abortProcessed) notifies the reception of the corresponding response.

Table 3: Operations provided by the communication API.

Operation	Description
ABcast(m)	Atomically broadcasts message m.
RBcast(m)	Reliably broadcasts message m.
AMcast(m, g)	Atomically multicasts message m to group g.
RUcast(m, dst)	Reliably unicasts message m to node dst.
RMcast(m, g)	Reliably multicasts message m to group g.
self(s) : *bool*	Returns true if s is the local node, false otherwise.

2.2.4 Communication System

Different implementations of the DM can have specific requirements for the GCS. For instance, a certification scheme running on a fully replicated environment requires a GCS with support for an atomic broadcast primitive. However, in a purely distributed context we can devise scenarios where only point-to-point communication is necessary.

Table 3 shows the communication primitives offered by the GCS to the DM. To be notified of incoming messages, the DM subscribes to the deliveries using the *observer* pattern [7] (the on*Deliver(m, src) operations notify of the delivery of message m from sender src).

2.2.5 Cache Support

With the exception of full replication, read operations over objects in the distributed shared memory space may imply remote communication. This raises performance problems that may even cancel out other benefits. To reduce time variance in the access to distributed objects, ReDstm provides support for a generic caching service. This service maps keys (distribution metadata) to generic objects and offers a set of generic operations (the basic map operations, e.g., contains, get, insert, remove) over the data container. This approach promotes flexibility, allowing the development of cache mechanisms tailored for specific replication protocols, given that only these specific implementations may efficiently answer the questions: what to cache and for how long should the cached objects be valid.

2.3 Programming Model

Pursuing our non-intrusiveness goal, the programming model exported by ReDstm is annotation-based, so no code rewriting is required.

Transactions. As legacy from TribuSTM, the programmer only has to add a `@Atomic` annotation to the methods that must execute as transactions. Then, in a transparent way to the programmer, ReDstm rewrites the application's bytecode, intercepting and conferring control to the framework at the beginning and end of transactions, and at memory accesses performed in a transactional context.

Partial Data Replication. In a partial replication setting, it is useful to be able to express what is to be fully and partially replicated. So, we grant to the programmer the power to express what data should be partially replicated.

In the Java virtual machine (JVM), the heap can be seen as a directed graph, where each node represents an object that, in turn, may have references to other objects. Figure 2 depicts an example of such graph, on which edges represent references, leaf nodes represent values of primitive data types, and the remainder nodes represent (composite) objects. Nodes A and H can be seen has two references pointing to the same memory location, i.e., aliasing.

In our approach the heap graph is replicated in all nodes, i.e., fully replicated, by default. Exceptions must be *explicitly* conveyed by the means of a `@Partial` annotation to be applied to class' fields. The annotation expresses that everything *downstream* of such field, in the graph, is to be partially replicated. Listings 1 and 2 illustrate the application of the annotation to the simple case of a linked list.

3. ON THE IMPLEMENTATION OF DISTRIBUTED STMs IN ReDstm

In this section we reason on how to use the mechanisms provided by ReDstm to implement distributed STMs with different data replication models.

3.1 A Distributed STM

Consider that distribution metadata is comprised by a unique identifier id, and the address of the node which owns the object associated with the metadata, $owner$. Let $id(\mathcal{O})$ and $owner(\mathcal{O})$ denote the id and $owner$ of a distributed object $\mathcal{O}$, respectively. If the distributed object $\mathcal{O}$ is created at node $\mathcal{N}$, $owner(\mathcal{O})$ is set to $\mathcal{N}$. Let $host(\mathcal{T})$ denote the node where transaction $\mathcal{T}$ executes, and $id(\mathcal{T})$ the unique identifier of $\mathcal{T}$. Let $proxy_{\mathcal{N}}(id)$ denote the proxy transaction of $\mathcal{T}$ on node $\mathcal{N}$ such that $id(\mathcal{T}) = id$, and consider, for simplicity, that every node contains a proxy transaction for every transaction currently executing in the system.

When a transaction $\mathcal{T}$ issues a read access on transactional metadata m (onRead at Table 1), if $owner(m) = host(\mathcal{T})$ the read access is a regular local access. Otherwise, $host(\mathcal{T})$ sends a message $[id(\mathcal{T}), id(m)]$, henceforth rd, to $owner(m)$, and waits response. When $owner(m)$ receives rd, it resolves $id(m)$ to m, issues a local read on m on behalf of $proxy_{owner(m)}(id(\mathcal{T}))$, and responds to $host(\mathcal{T})$ with message $[id(\mathcal{T}), v, a]$ where v is the value read and a is true if there has been a conflict. When $host(\mathcal{T})$ delivers the response message, the execution of $\mathcal{T}$ is resumed. Write accesses behave similarly.

When $\mathcal{T}$ requests to commit (onCommit at Table 1), $host(\mathcal{T})$ sends a message to all nodes to trigger the validation of $proxy_{\mathcal{N}}(id(\mathcal{T}))$ on each node (validate at Table 2). All nodes respond with the result of the validation. Once $host(\mathcal{T})$ has collected all validation results, if none of the validations failed, a message is sent to all nodes instructing the commit of each $proxy_{\mathcal{N}}(id(\mathcal{T}))$. If at least one validation was unsuccessful, all nodes are instructed to discard their proxies and $\mathcal{T}$ aborts.

This is a simple approach, thus there are various optimizations to consider, such as creating transaction proxies on demand only on strictly necessary nodes and caching to reduce network communication.

3.2 A Fully Replicated STM

In a fully replicated environment, in which all replicas maintain a local copy of all the objects in the system, each object must be identifiable in the global context of the system. In a centralized system, an object can be uniquely identified by its memory address, but this does not apply in a distributed context, where an object's metadata must contain a globally unique identifier.

Consider that a fully replicated object $\mathcal{O}$ can be in two possible states: private or published. In the first case, the serialization of a private object $\mathcal{O}$ consists in generating its respective *oid* (metadata) and effectively serializing $\mathcal{O}$. In

```
class Node<T> {
  Node next;
  int id;
  @Partial
  T value;

  ...
}
```

Listing 1: Class `Node`.

```
class List<T> {
  Node<T> head;

  public List() {
    this.head = ...
  }
  ...
}
```

Listing 2: Class `List`.

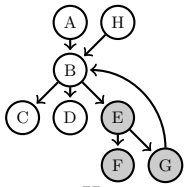


Figure 2: Heap graph.

Algorithm 1 Transactional memory (TM) component.

```
1: committed ← false                                   ▷ Commit result

2: procedure BEGIN(T)
3:     ...
4:     ONSTART(T)

5: function COMMIT(T)
6:     ONCOMMIT(T)
7:     wait until T is processed
8:     return committed

9: procedure COMMITPROCESSED(T, r)
10:    committed ← r
11:    T was processed
```

Algorithm 2 Non-voting certification protocol.

```
1: localTxs ← ∅                          ▷ Map of transaction ids to transactions

2: procedure ONCOMMIT(T)
3:     localTxs ← localTxs[T.id ↦ T]
4:     S ← CREATESTATE(T)
5:     ABCAST(S)

6: procedure ONABDELIVER(S, src)
7:     if SELF(src) then
8:         T ← localTxs(S.id)
9:         localTxs ← localTxs \ T
10:    else
11:        T ← RECREATETX(S)
12:    valid ← VALIDATE(T)
13:    if valid then
14:        APPLYWS(T)
15:    COMMITPROCESSED(T, valid)
```

the second case, it has already been assigned an identifier *oid*, hence we just need to assign *oid* as its representative, serializing *oid* instead of $\mathcal{O}$. In this case, the de-serialization of *oid* returns the object $\mathcal{O}'$ such that $id(\mathcal{O}') = oid$, i.e., returns the object corresponding to the local replica with identifier *oid*. This means that an already published object is never sent through the network, only its identifier is, resulting in a potentially large decrease in the size of the exchanged messages.

Certification-based protocols [1, 8] are very interesting in the context of a fully replicated STM because they do not require synchronization between replicas during transaction execution (only at commit time). In Algorithms 1 and 2 we can see the pseudo-code of the non-voting certification protocol implemented using the interfaces provided by REDSTM. The protocol works as follows: an application thread executes a transaction locally, and only when it finishes it asks for the confirmation of the transaction to the TM component (COMMIT in Algorithm 1), which triggers the certification protocol (ONCOMMIT in Algorithm 2). At this point, the thread waits for the decision of the certification protocol (Line 7 in Algorithm 1), which will validate and either confirm the transaction or abort it. The atomic broadcast performed by the certification protocol is received by all replicas. The replica that broadcasted the message processes the corresponding local transaction, while the other replicas recreate the transaction using state $\mathcal{S}$ received in the message. All replicas proceed to the validation of the transaction and subsequent application in case of confirmation. The process concludes with the invocation of COMMITPROCESSED (Algorithm 1), which causes the application's thread to proceed its execution.

3.3 A Partially Replicated STM

In the context of full replication, a unique system-wide identifier is sufficient to unequivocally represent a distributed object. However, when addressing partial replication, not all objects are replicated in all the system's nodes, and thus read operations may require inter-node communication. Consequently, the system must be able to classify objects as local or remote, and must hold all the necessary information to request the given object from another node that replicates it. To that end, an object's metadata must complement the aforementioned unique identifier with the identifiers of the *groups* of nodes that replicate the associated object.

Being that objects have to be sent through the network, they also have to be previously serialized, hence Algorithm 3 presents the pseudo-code for object serialization in partial replication scenarios. The serialization process is delegated to the Object serializer (ObS), by methods WRITEREPLACE and READRESOLVE to methods OBJECTREPLACE and OBJECTRESOLVE, respectively. As in full replication, a partially replicated object $\mathcal{O}$ can be in two possible states: published or private. The process is also similar to the one described in §3.2: public objects nominate their identifier to be serialized in their place, while the serialization of private objects requires the generation of the respective metadata and the subsequent serialization of $\mathcal{O}$. Additionally, partial replication requires the serialization algorithm to be aware if it was triggered in the context of a remote read operation (ISREADCONTEXT, Line 3 in Algorithm 3). This information is required because, when in the context of a remote read operation, the requesting node does not hold a local copy of $\mathcal{O}$, and thus the original object must always be transferred.

The de-serialization process also grows from the fully replicated scenario. The difference lies in the need to check if the local node should replicate the incoming object $\mathcal{O}$, i.e., if it belongs to one of the groups identified in $\mathcal{O}$'s metadata. If not, $\mathcal{O}$ is a replica of the original object and is simply returned, otherwise there are still two scenarios to cover: the local node *holds* or *not* a local replica of $\mathcal{O}$. In the first case, $\mathcal{O}$ comprises only the original object's metadata, which must be used to retrieve the local replica. In the

Algorithm 3 Object Serializer component.

```
1: memory ← ∅                                              ▷ Local memory

2: function objectReplace(O)
3:     if isReadContext(O) then
4:         return O
5:     oid ← getMetadata(O)
6:     if ∃oid then
7:         return oid
8:     else                                  ▷ Object O is not published
9:         oid ← generate fresh metadata
10:        setMetadata(O, oid)
11:        group ← getGroup(oid)
12:        if isLocal(group) then
13:            memory ← memory[oid ↦ O]
14:        return O

15: function objectResolve(O)
16:    oid ← getMetadata(O)
17:    group ← getGroup(oid)
18:    if isLocal(group) then
19:        obj ← memory(oid)
20:        if ∃obj then
21:            return obj
22:        else
23:            memory ← memory[oid ↦ O]
24:            return O
25:    else
26:        return O
```

second case, $\mathcal{O}$ is a replica of the original object and must be stored in the local memory before being returned.

Unlike with full replication, in partial replication is not easy to support transaction confirmation using certification-based protocols, because nodes do not hold enough information to validate all transactions by themselves. The solution is to use a voting algorithm, so that all nodes may reach the same decision about which transactions to commit and abort. In fact, all the proposed partial replication protocols are based in the two phase commit protocol (2PC) [13, 14, 16, 17].

Some of the distribution manager (DM) modules are specific to partial replication, namely the group partitioner (GP) and the data partitioner (DP). In a partially replicated scenario, nodes are partitioned into *groups* which in turn replicate a subset of the system's data. This division is carried out by the concrete implementations of the GP. With knowledge of the existing nodes in the system and of the desired objects' replication factor, the GP returns a partitioning according to the implemented strategy. Some strategies can be as simple as dividing nodes in a round-robin fashion, or as complex as trying to minimize the latency between the nodes in a group. The decision of in which groups an object is replicated is managed by the DP. Implementations can range from randomly distributing objects by groups to more complex strategies, trying to achieve load balancing or taking into account previous access patterns.

3.4 Implementation Considerations

Using ReDstm, we implemented a fully and a partially replicated STMs. In order to prove the framework's flexibility and modularity we developed a few different implementations of its components.

For the fully replicated STM we implemented two STM algorithms, namely TL2 [6] and MVSTM [5]. We also implemented two different protocols for the DT module, namely non-voting certification [1] and voting certification [8].

For the partially replicated STM we implemented a STM algorithm as described in [13] and for the DT we implemented the SCORe protocol [13]. Still in the context of partially replicated STM and just for the sake of simplicity, in the evaluation of our framework in §5 we adopted a simple scheme where each data item is replicated by a *single* group and groups are comprised by *disjoint* sets of nodes.

4. COMBINING FULL AND PARTIAL REPLICATION IN ReDstm

Supporting partial replication in a DTM framework for a general purpose programming language (GPPL) may have as inspiration the previous application of partial replication to databases, but programming languages are much more expressive than database query languages. As such, addressing partial replication in the context of GPPLs raises some challenges, namely (1) what data should be partially replicated; (2) how to express the replication strategy in a GPPL; and (3) how to partially replicate object graphs.

Challenge (2) was addressed by the programming model in §2.3, and challenge (3) was addressed in §2.2.2 using specific distribution metadata. This section will thus focus on challenge (1) by reasoning on how to balance full and partial replication in the context of DTM for a GPPL.

Fully replicated databases replicate every table and their contents in every node in the system. But, as we look at partially replicated databases, these still replicate *every* table in *every* node, while is their content that is partially replicated. In sum, partial replication is, to some extent, combined with full replication: data structuring information if fully replicated while hard data is partially replicated.

We defend that this premise can be used when applying partial replication to DTM in order to reduce the communication overhead of remote read operations. Figure 3 depicts a practical example of a partially replicated linked list, using three possible options. Dashed rectangles represent different groups, i.e., the data inside a rectangle is partially replicated, and circles represent the data stored in each list node. Everything outside the dashed rectangles is fully replicated.

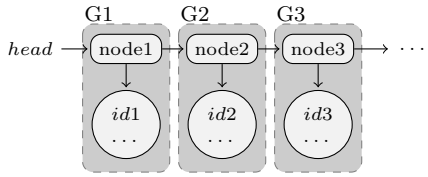
In the case of Figure 3a, the entire nodes are partially replicated. By choosing this option, when searching the list for a specific node, the nodes' traversal would require a possibly remote read operation for each traversed node.

Contrary, by adopting the option depicted in Figure 3b, i.e., partially replicating just the hard data stored in each node, the amount of remote read operations required for traversing the list would be limited by the data we really want to inspect.

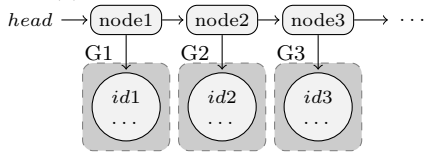
Another interesting option is to partially replicate the hard data stored in each node and take the keys associated with the data from the hard data to the nodes themselves, as in Figure 3c. This option is notably interesting when used in indexed data structure (e.g., dictionaries): a search in such a data structure traverses it looking for a specific key and only then inspects the hard data associated with that key. Hence, by adopting this approach, we can greatly limit the amount of remote read operations.

4.1 Guidelines

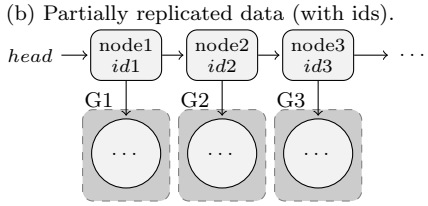
Looking at the options depicted in Figure 3, combining full and partial replication can bring both advantages and disadvantages. By partially replicating data structures, transactions that modify them will only require confirmation by a



(a) Partially replicated nodes.



(b) Partially replicated data (with ids).



(c) Partially replicated data (w/o ids).

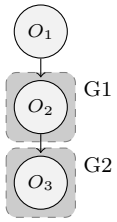
Figure 3: Different partial replication options.

subset of the system's nodes, but will most likely entail multiple remote read operations when traversing the structures. In turn, when fully replicating the data structures, all transactions that modify them will require confirmation by *all* the system's nodes, but the task of traversing the structures can always be performed locally. Taking into account all these trade-offs, we answer to challenge (1) by fully replicating the (data) structures and by partially replicating only the hard data they store. Hence, we trade performance in structure modifying transactions (because they have to be confirmed by all the system's nodes) by performance due to less remote read operations when traversing those structures.

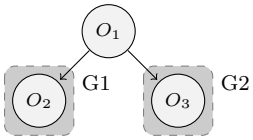
In a generic way, small and/or frequently accessed data should be fully replicated, and big and/or occasionally accessed data should be partially replicated. Turning this into guidelines for when combining full and partial replication, we have that:

- **structure** should be **fully** replicated;
- **search information**, e.g., keys, should be **fully** replicated; and
- **hard data** should be **partially** replicated.

The `@Partial` annotation presented in §2.3 was devised with these guidelines in mind, so it is tailored for partially replicating just the leafs (or small sub-graphs close to the leafs) of the heap graph. This approach provides a flexible programming model that empowers the programmer to adjust the used replication strategy to the characteristics of each application, while requiring the programmer to reason about the impact of the replication model in the application's performance. However, when adapting our benchmarking applications, we verified that the burden of applying the `@Partial` annotation is confined to the internal implementation of the data structures. For instance, the linked list example in Listings 1 and 2 applies the replication option from Figure 3c, where the list structure is fully replicated and the value object of each node is partially replicated.



(a) Cascading type.



(b) Tree-like type.

Figure 4: Limitation of our implementation of the programming model.

When addressing composite data structures that build from pre-existent ones, further reasoning may be required. For instance, consider a hash map with collision lists that builds up from the list in Listings 1 and 2. We can easily think of two possibilities: (1) to partially replicate the entire collision list of each map entry; or (2) to partially replicate only the nodes of the collision lists. In option (1) our guidelines are applied at the hash map level, since the structure and keys of the map are fully replicated, and the collision list of each map entry is partially replicated. In turn, in option (2) the guidelines are applied at the list level, because the hash map's implementation is left unaltered, and the `@Partial` annotation is only applied in the list's implementation, resulting in fully replicated collision lists with partially replicated nodes.

4.2 Limitations

As is, our implementation of the programming model has some limitations, namely: (1) cycles including both fully and partially replicated objects; and (2) partially replicated objects referring other partially replicated objects in a different replication group.

Regarding limitation (1), to ensure the semantics of the `@Partial` annotation, edges that flow upstream from partially to fully replicated objects are not allowed. In the example in Figure 2, if node E was annotated with `@Partial`, the edge between G and B could not exist, since by annotating node E the programmer was declaring that all the objects downstream, represented by the shaded nodes, should be partially replicated. An edge from node G to node B would intuitively mean that node B should be partially replicated, but since it is upstream from node E it also means that it should be fully replicated, creating an ambiguity.

Limitation (2)–partially replicated objects referring other partially replicated objects in a different replication group–results from an implementation decision, since we use weak references in the map that associates distribution metadata to distributed objects (e.g., *memory* in Algorithm 3). Because we use weak references, when an object is no longer referenced by others in the application, the garbage collector is free to erase that object from memory, thus precluding the case in Figure 4a. Since objects O_2 and O_3 are in different groups, O_3 will not be referenced by O_2 leaving object O_3 at the mercy of the garbage collector. On the other hand, the case in Figure 4b is possible since both objects O_2 and O_3 are referenced by a fully replicated object.

During the adaptation of the benchmarking applications these limitations where overcome with relative ease. In fact, by following the recommended guidelines, the management of the `@Partial` annotation is made at the data structure

level.

5. EXPERIMENTAL EVALUATION

The evaluation of the ReDstm framework addressed three main questions: (1) what is the impact in memory consumption if we switch from a full replicated to a partially replicated system?; (2) how does the replication factor influences the throughput of both fully and partially replicated DTM applications?; and (3) how does the ratio of structure data vs. hard data influences the throughput in the presence of full and partial replication?

5.1 Experimental Setup

Experimental Test-bed. All the experiments were conducted in a heterogeneous cluster with 8 nodes. Five nodes have 2 × Quad-Core AMD Opteron 2376 2.3 GHz and 16 GB of RAM. The remainder have 1 × Quad-Core Intel Xeon X3450 2.66 GHz (with Hyper-Threading) and 8 GB of RAM. All machines run *Linux 2.6.26-2* (Debian 5.0.10) and are interconnected via private Gigabit Ethernet. The installed Java platform is OpenJDK 6 (IcedTea 1.8.10).

To allow a comparison between the two implemented replicated STMs, we used *two configurations* of ReDstm: **C1**, a full replication configuration that uses MVSTM as the TM layer and the DT implements a non-voting certification; and **C2**, a partial replication configuration that uses SCORe and its multi-version STM algorithm. Additionally, we have set to *two* the replication factor of each data item [13] (if nothing said on the contrary). With regard to the underlying GCS we used JGroups [2] version 3.4.1.

Benchmarks. The Red-Black Tree (RBT) benchmark [10] has three transactions: insertions, which add an element to the tree (if not present); deletions, which remove an element from the tree (if present); and searches, which search the tree for a specific element. Insertions and deletions are *write* transactions. This benchmark is characterized by very small and fast transactions that perform little work and exhibit low contention.

The Vacation benchmark is part of the STAMP suite [12]. It emulates a travel reservation system implemented as a set of binary trees keeping track of customers and their reservations for various travel items. There are three transactions: reservations, cancellations and updates. We added a new *read-only* transaction that consults reservations.

The TPC-W benchmark [19] models an online book store. Servers handle user requests such as browsing, adding products to a shopping cart, or placing an order. The browsing workload consists of 95% of operations related with browsing and 5% related with purchases.

In all benchmarks, we followed the guidelines described in §4.1, hence the `@Partial` annotation was confined to the data structures: only the values stored in the nodes of the binary trees and hash maps used by the benchmarks were partially replicated.

5.2 Results

Memory Consumption. The out-of-the-box RBT benchmark maps integers to integers, i.e., both keys and values are integers. In this case, ReDstm using configuration C2 consumes a larger, but negligible, amount of memory when compared to configuration C1. Here, the problem is twofold: partially replicating an integer does not reduces memory usage as the memory for the integer is still reserved (this goes for all primitive data types in Java); and the distribution metadata for configuration C2 stores more information than the metadata used by configuration C1.

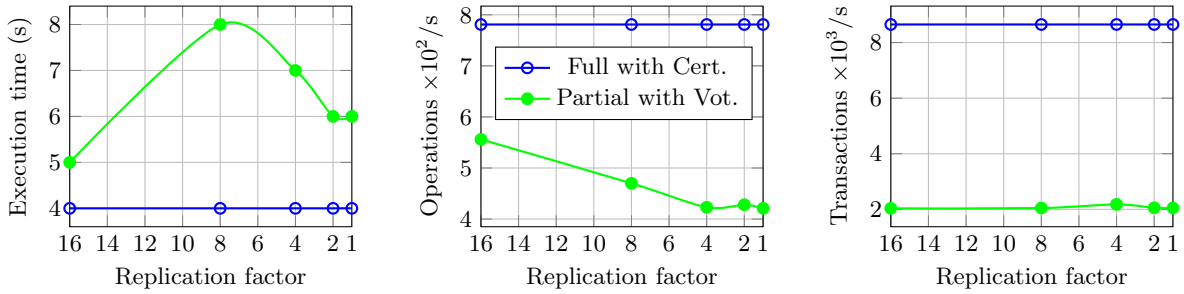
To verify our intuition, we modified the benchmark in order to map integers to jpeg images, each with 3 MB in size. In this case, as expected, by decreasing the data's replication factor, the memory consumed by each node decreases as well, reaching around 55% less memory at replication factor 2.

Partial & Full Data Replication. After running the three benchmarks, namely Adapted Vacation and RBT, both with 10% writes, and TPC-W with the browsing mix, Figure 5 shows that the performance of configuration C2 is very low regarding its full replication counterpart. This results are explained by the simple reason that, in all these benchmarks, the majority of the write transactions modify the data structures (around 90% in Adapted Vacation, 85% in TPC-W and 100% in RBT), and those structures are *fully replicated*. Thus, write transactions require a distributed confirmation involving *all* the system's nodes, which becomes very expensive in configuration C2 since it uses 2PC, while configuration C1 uses a single atomic broadcast. These results reveal a known but rather important aspect of configuration C2: SCORe was not designed for environments mixing full and partial replication.

In order to see if the protocol used by configuration C2 can take advantage of its nature, we modified the RBT benchmark in order to have control over the amount of write transactions that modify partially replicated data, i.e., hard data accessed by transactions. Thus, in this version of the benchmark, named *Adapted RBT*, we have transactions that modify the tree's structure while others modify the values stored in the tree's nodes. This way, configuration C2 is able to leverage its protocol and some transactions will only require confirmation of a *subset* of the system's nodes. Figure 6 depicts the results of running this benchmark with 10%, 50% and 80% of write transactions. With small amounts of hard data accessed, configuration C2 performs poorly, but as we increase that amount it starts to match configuration C1 and is even able to surpass it. With 10% of writes, configuration C2 starts to outperform configuration C1 at 80% of hard data accessed, and with 50% and 80% of writes, it outperforms configuration C1 starting just at 50% of hard data accessed. We present configuration C2 mimicking full replication (Full with Voting) just as a baseline.

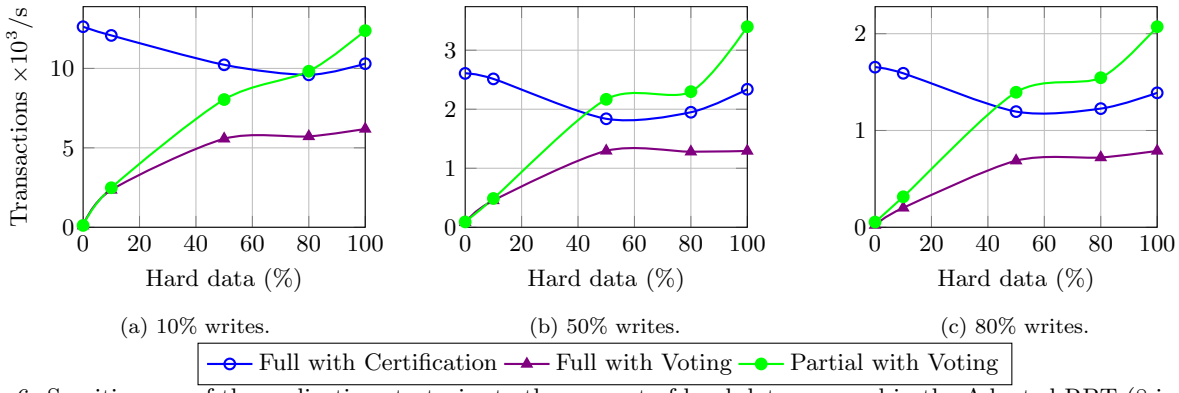
Configuration C1 is affected by the variation in the amount of hard data accessed due to particularities of the benchmark. Both write transactions of RBT only perform work in certain cases: insertions only add an element if it *is not already present*, and deletions only remove an element if it *is present*. Since elements are randomly chosen, write transactions may become read-only (which do not need distributed confirmation), e.g., if an element already exists when doing an insertion. Contrary, in our adaptation *all* write transactions choose elements that *definitely* exist, thus they always perform work and need a distributed confirmation.

Figures 7 and 8 show the latency when performing remote read operations and when committing transactions, respectively. When executing a remote read, configuration C2 experiences increased latency when the replication factor decreases due to the fact that less nodes replicate each object, which leads to more remote accesses. In Figure 8, we can see that configuration C1 is not influenced by the amount of hard data accessed. But configuration C2 starts to expe-



(a) Adapted Vacation. (b) TPC-W. (c) RBT.

Figure 5: Influence of the replication factor in the throughput of the Adapted Vacation and RBT benchmarks with 10% writes, and of TPC-W with the browsing mix (16 instances).



(a) 10% writes. (b) 50% writes. (c) 80% writes.

Figure 6: Sensitiveness of the replication strategies to the amount of hard data accessed in the Adapted RBT (8 instances).

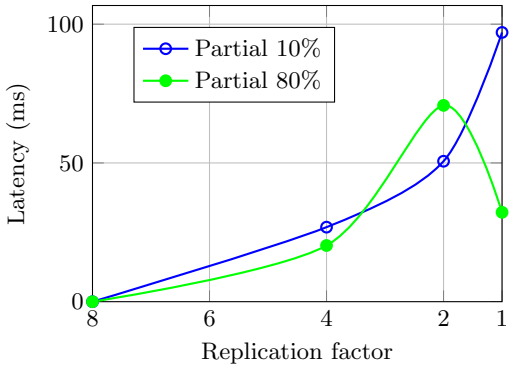


Figure 7: Remote read latency on Adapted RBT with 10% writes and 10% and 80% of hard data accessed (8 instances).

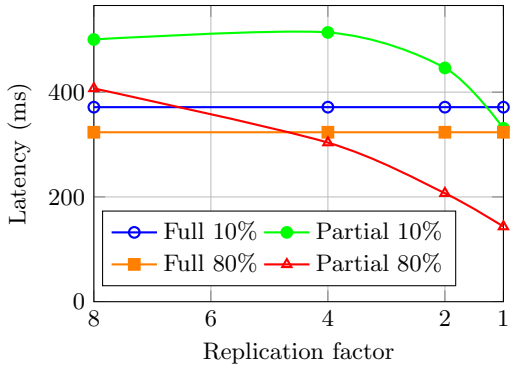


Figure 8: Distributed commit latency on Adapted RBT with 10% writes and 10% and 80% of hard data accessed (8 instances).

rience lower latencies when the replication factor decreases, since transactions have to be confirmed by less nodes, making commit operations faster.

6. RELATED WORK

Transactional Memory. With the recent thrust in DTM research, a handful of different frameworks have been proposed with the goal of facilitating the development, testing, and evaluation of the multiple proposed memory consistency protocols. All of the best known DTM frameworks present a great level of intrusion, requiring a significant part of the application code to be rewritten to comply with the specificities of each framework (as shown in Table 4). DiSTM's [11] objects have to implement special interfaces and are created using transactional factory methods. Transactions have to be executed by a special thread class. D^{2}STM [4] is based on JVSTM [3], hence objects have to be wrapped in boxes and have to be accessed by special `get` and `set` methods. Transactions are methods annotated with a `@Atomic` annotation. HyFlow [15] has many pluggable components. Objects have to implement a special interface and are tagged with a unique identifier that serves as its reference in the system. These identifiers are exchanged by their corresponding objects using a `Locator` instance. Transactions are defined as `@Atomic`-annotated methods. HyFlow2 [20] is written in Scala and its programming model is very similar to one of HyFlow, but instead it uses Scala constructs. In turn, ReD-

Table 4: Comparison between DTM frameworks.

Framework	Replication Model	Intrusiveness	Modularity
DiSTM	Master/Slave	High	Low
D^{2}STM	Full	High	High
Hyflow/Hyflow2	Control/Data-flow	Moderate	High
ReDstm	Distribution, Partial, Full	Low	High

stm manages to be a modular framework, while presenting a much lower level of intrusion to the application code (see Table 4), requiring only the use of two annotations.

Concerning data replication, DiSTM has a master/slave approach, D^{2}STM supports full replication, and both HyFlow and HyFlow2 support data-flow and control-flow approaches. HyFlow-related research also dwells into the partial replication field [9], however the focus is on the scheduling of memory transactions, rather than on the support of different replication strategies in DTM. As a result, ReDstm is more flexible, allowing the implementation of a wider range of data replication strategies.

Databases. A great part of the work presented on this paper was built upon the cumulative knowledge of the databases research field. Solutions natively oriented to partially replicated transactional systems may be divided depending on whether they can be considered genuine, i.e., when the transaction's confirmation only involves the nodes keeping copies of the data accessed by the transaction, and on the specific consistency guarantees they provide. Serrano et al. [17] provide a non-genuine protocol, where the confirmation of a transaction requires interaction with all the nodes in the system. Compared to this approach, genuine schemes have shown to achieve significantly higher scalability [14].

Concerning the provided consistency guarantees, in [17], the authors identify 1-copy-serializability (1CS) as a limitation when designing replicated solutions, and propose a protocol offering 1-copy-snapshot-isolation, whereby replicas run under snapshot isolation. In turn, with P-Store [16], Schiper et al. go back to 1CS proposing the first genuine partial replication protocol offering this kind of consistency guarantee. But this protocol still imposes that read-only transactions undergo a distributed confirmation phase.

7. CONCLUSIONS

In this paper we presented ReDstm, a modular and non-intrusive Java DTM framework that provides support for a wide range of data replication models, from full to partial data replication, in a general purpose programming language. Its modularity allows an easy integration and implementation of several of its components, and its non-intrusiveness enables the adaptation of pre-existent applications to ReDstm with minimal effort.

We used ReDstm to study how different replication models can be integrated in the implementation of distributed STMs. From our experiences we devised some guidelines for when combining full and partial replication, and show that ReDstm is flexible enough to accommodate data structures with different parts under different replication strategies (with gains in scalability, in some cases).

We evaluated the impact of different replication models in the execution of three known benchmarks, being able to conclude that (i) partial replication contributes to the system's scalability by reducing the amount of data stored at each node; (ii) the impact of the replication model in transaction throughput is sensible to the amount of transactions that manipulate structuring and hard data; and (iii) the benefits of partial replication are directly proportional to the amount of transactions that manipulate only hard data.

Acknowledgements

This work was partially funded by FCT-MEC, in the context of the research project PTDC/EIA-EIA/113613/2009, the strategic project PEst-OE/EEI/UI0527/2014, and research grant SFRH/BD/84497/2012.

8. REFERENCES

[1] D. Agrawal et al. Exploiting atomic broadcast in replicated databases. In *Euro-Par*, 1997.

[2] Bela Ban. JGroups - A Toolkit for Reliable Multicast Communication. `http://www.jgroups.org`, 2013.

[3] J. Cachopo et al. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 2006.

[4] M. Couceiro et al. D2STM: Dependable distributed software transactional memory. In *PRDC*, 2009.

[5] R. J. Dias et al. Efficient support for in-place metadata in Java software transactional memory. *Concurrency and Computation: Practice and Experience*, 2013.

[6] D. Dice et al. Transactional locking II. In *DISC*, 2006.

[7] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. 2004.

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

[9] J. Kim et al. Scheduling transactions in replicated distributed software transactional memory. In *CCGRID*, 2013.

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

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

[12] C. C. Minh et al. Stamp: Stanford transactional applications for multiprocessing. In *IISWC*, 2008.

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

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

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

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

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

[18] N. Shavit et al. Software transactional memory. In *PODC*, 1995.

[19] Transaction Processing Performance Counsil. TPC Benchmark W. `http://www.tpc.org/tpcw`, 2013.

[20] A. Turcu et al. Hyflow2: A high performance distributed transactional memory framework in Scala. In *PPPJ*, 2013.