Exploiting Locality in Lease-Based Replicated Transactional Memory via Task Migration

Danny Hendler[1], Alex Naiman[1], Sebastiano Peluso[2], Francesco Quaglia[2], Paolo Romano[3] and Adi Suissa[1]

[1] Ben-Gurion University of the Negev, Israel
[2] Sapienza University of Rome, Italy
[3] Instituto Superior Técnico, Universidade de Lisboa/INESC-ID, Portugal

Abstract. We present *Lilac-TM*, the first locality-aware Distributed Software Transactional Memory (DSTM) implementation. Lilac-TM is a fully decentralized lease-based replicated DSTM. It employs a novel self-optimizing lease circulation scheme based on the idea of dynamically determining whether to migrate transactions to the nodes that own the leases required for their validation, or to demand the acquisition of these leases by the node that originated the transaction. Our experimental evaluation establishes that Lilac-TM provides significant performance gains for distributed workloads exhibiting data locality, while typically incurring little or no overhead for non-data local workloads.

1 Introduction

Transactional Memory (TM) has emerged as a promising programming paradigm for concurrent applications, which provides a programmer-friendly alternative to traditional lock-based synchronization. Intense research work on both software and hardware TM approaches [16, 22] and the inclusion of TM support in world-leading multiprocessor hardware and open source compilers [17, 21] extended the traction it had gained in the research community to the mainstream software industry.

Distributed Software TM (DSTM) systems extend the reach of the TM model to distributed applications. An important lesson learnt by the deployment of the first enterprise-class TM-based applications [6, 19] is that in order to permit scalability and meet the reliability requirements of real-world applications, DSTMs must support data replication. As a result, several replication techniques for distributed TM have been proposed, deployed over a set of shared-nothing multi-core systems [1, 2, 11, 20], as typical of cloud computing environments.

A key challenge faced by replicated DSTMs, when compared with more conventional transactional systems (such as relational databases), is the large increase of the communication-to-computation ratio [19]: unlike

classical DBMSs, DSTMs avoid disk-based logging and rely on in-memory data replication to achieve durability and fault-tolerance; further, the nature of the programming interfaces exposed by DSTMs drastically reduces the latencies of accessing data, significantly reducing the duration of typical TM transactions, when compared to typical on-line transaction processing (OLTP). Overall, the reduction of the transaction processing time results in the growth of the relative cost of the distributed (consensus-based [14]) coordination activities required by conventional replication protocols, and in a corresponding increase of their relative overhead.

Model and Background: We consider a classical asynchronous distributed system model [14] consisting of a set of processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate via message passing and can fail according to the fail-stop (crash) model. We assume that a majority of processes is correct and that the system ensures sufficient synchrony for implementing a *View Synchronous Group Communication Service* (GCS) [10].

GCS provides two complementary services: group membership and multicast communication. Informally, the role of the *group membership service* is to provide each participant in a distributed computation with information about which process is active (or reachable) and which is failed (or unreachable). Such information is called a *view* of the group of participants. We assume that the GCS provides a *view-synchronous primary-component group membership service* [5], which maintains a single agreed view of the group at any given time and provides processes with information on whether they belong to the primary component.

We assume that the *multicast communication* layer offers two communication services, which disseminate messages with different reliability and ordering properties: *Optimistic Atomic Broadcast* (OAB) [12] and *Uniform Reliable Broadcast* (URB) [14]. URB is defined by the primitives *UR-broadcast*(m) and *UR-deliver*(m) and guarantee causal order and uniform message delivery. Three primitives define OAB: *OA-broadcast*(m), which is used to broadcast message m; *Opt-deliver*(m), which delivers message m with no ordering or reliability guarantees; *TO-deliver*(m), which delivers message m ensuring uniform and total order guarantees.

The ALC (Asynchronous Lease Certification) protocol [8] is based on the *lease* concept. A *lease* is an ownership token that grants a node temporary privileges on the management of a subset of the replicated data-set. ALC associates leases with data items indirectly through *conflict classes*, each of which may represent a set of data items. This allows flexible control of the granularity of the leases abstraction, trading off

accuracy (i.e., avoidance of aliasing problems) for efficiency (amount of information exchanged among nodes and maintained in-memory) [3].

With ALC, a transaction is executed based on local data, avoiding any inter-replica synchronization until it enters its commit phase. At this stage, ALC acquires a lease for the transaction's accessed data items, before proceeding to validate the transaction. In case a transaction T is found to have accessed stale data, T is re-executed without releasing acquired leases. This ensures that, during T's re-execution, no other replica can update any of the data items accessed during T's first execution, which guarantees the absence of remote conflicts on the subsequent re-execution of T, provided that the same set of conflict classes accessed during T's first execution is accessed again.

To establish lease ownership, ALC employs the OAB communication service. Disseminating data items of committed transactions and lease-release messages in done using the URB service. The ownership of a lease ensures that no other replica will be allowed to successfully validate any conflicting transaction, making it unnecessary to enforce distributed agreement on the global serialization order of transactions. ALC takes advantage of this by limiting the use of atomic broadcast exclusively for establishing the lease ownership. Subsequently, as long as the lease is owned by the replica, transactions can be locally validated and their updates can be disseminated using URB, which can be implemented in a much more efficient manner than OAB.

Our Contributions: In this paper, we present an innovative, fully decentralized, LocalIty-aware LeAse-based repliCated TM (Lilac-TM). Lilac-TM aims to maximize system throughput via a distributed self-optimizing lease circulation scheme based on the idea of dynamically determining whether to migrate transactions to the nodes that own the leases required for their validation, or to demand the acquisition of these leases by the transaction originating node.

Lilac-TM's flexibility in deciding whether to migrate data or transactions allows it not only to take advantage of the data locality present in many application workloads, but also to further enhance it by turning a node N that frequently accesses a set of data items D into an attractor for transactions that access subsets of D (and that could be committed by N avoiding any lease circulation). This allows Lilac-TM to provide two key benefits: (1) limiting the frequency of lease circulation, and (2) enhancing contention management efficiency. In fact, with Lilac-TM, conflicting concurrent transactions have a significantly higher probability

of being executed on the same node, which prevents them from incurring the high costs of distributed conflicts.

We conduct a comprehensive comparative performance analysis, establishing that Lilac-TM outperforms ALC by a wide margin on workloads possessing data locality, while incurring little or no overhead for non-data local workloads.

2 Lilac-TM

Figure 1 provides an overview of the software architecture of each replica of Lilac-TM, highlighting in gray the modules that were either redesigned or that were not originally present in ALC.

The top layer is a wrapper that intercepts application level calls for transaction demarcation without interfering with application accesses (read/write) to the transactional data items, which are managed directly by the underlying local STM layer. This approach allows transparent extension of the classic STM programming model to a distributed setting.

The prototype of Lilac-TM has been built by extending the ALC implementation shipped in the GenRSTM framework [7]. GenRSTM has been designed to support, in a modular fashion, a range of heterogeneous algorithms across the various layers of the software stack of a replicated STM platform. Lilac-TM inherits this flexibility from GenRSTM. In this work, we use TL2 [13] as the local STM layer.

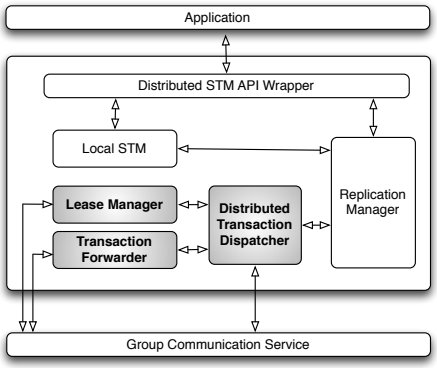


Fig. 1: Middleware architecture of a Lilac-TM replica.

The Replication Manager (RM) is the component in charge of interfacing the local STM layer with its replicas deployed on other system nodes. The RM is responsible of coordinating the commit phase of both remote and local transactions by: (i) intercepting commit-request events generated by local transactions and triggering a distributed coordination phase aimed at determining transactions' global serialization order and detecting the presence of conflicts with concurrently executing remote transactions; and (ii) validating remote transactions and, upon successful validation, committing them by atomically applying their write-sets in the local STM.

At the bottom layer we find a GCS (Appia [18] in our prototype), which, as mentioned in Section 1, provides the view synchronous membership, OAB and URB services.

The role of the Lease Manager (LM) is to ensure that no two replicas simultaneously disseminate updates for conflicting transactions. To this end, the LM exposes an interface consisting of two methods, GETLEASE() and FINISHEDXACT(), which are used by the RM to acquire/release leases on a set of data items. This component was originally introduced in ALC and has been re-designed in this work to support *fine-grained leases*. As we explain in more detail in Section 2.1, fine-grained leases facilitate the exploitation of locality and consequently reduce lease circulation.

The Transaction Forward (TF) is responsible for managing the forwarding of a transaction to a different node in the system. The transaction forwarding mechanism represents an alternative mechanism to the lease-based certification scheme introduced in ALC. Essentially, both transaction forwarding and lease-based replication strive to achieve the same goal: minimizing the execution rate of expensive Atomic Broadcast-based consensus protocols to determine the outcome of commit requests. ALC's lease mechanism pursues this objective by allowing a node that owns sufficient leases to validate transactions and disseminate their writesets without executing consensus protocols. Still, acquiring a lease remains an expensive operation, as it requires the execution of a consensus protocol.

The transaction forwarding scheme introduced in this work aims at reducing the frequency of lease requests triggered in the system, by migrating the execution of transactions to remote nodes that may process them more efficiently. This is the case, for instance, if some node n owns the set of leases required to certify and commit a transaction T originated on some remote node n'. In this scenario, in fact, n could validate T locally, and simply disseminate its writeset in case of success. Transaction migration may be beneficial also in subtler scenarios in which, even though no node already owns the leases required to certify a transaction T, if T's originating node were to issue a lease request for T, it would revoke leases that are being utilized with high frequency by some other node, say n''. In this case, it is preferable to forward the transaction to n'' and have n'' acquire the lease on behalf of T, as this would reduce the frequency of lease circulation and increase throughput in the long term.

The decision of whether to migrate a transaction's execution to another node or to issue a lease request and process it locally is far from being a trivial one. The transaction scheduling policy should take load balancing considerations into account and ensure that the transaction

migration logic avoids excessively overloading any subset of nodes in the system. In Lilac-TM, the logic for determining how to manage the commit phase of transactions is encapsulated by the Distributed Transaction Dispatching (DTD) module. In this paper, we propose two decision policies based on an efficiently solvable formulation in terms of an Integer Linear Programming optimization problem.

In the following we describe the key contributions of this paper, i.e. the fine-grained lease management scheme, the TF and the DTD.

2.1 Fine-Grained Leases

In ALC, a transaction requires a *single lease object*, associated with its data set in its entirety. A transaction T, attempting to commit on a node, may reuse a lease owned by the node only if T's data set is a subset of the lease's items set. Thus, each transaction is tightly coupled with a single lease ownership record. This approach has two disadvantages: i) upon the delivery of a lease request by a remote node that requires even a single data item from a lease owned by the local node, the lease must be released, causing subsequent transactions accessing other items in that lease to issue new lease requests; ii) if a transaction's data set is a subset of a union of leases owned by the local replica but is not a subset of any of them, a new lease request must be issued. This forces the creation of new lease requests, causing extensive use of *TO-broadcast*.

To exploit data-locality, we introduce a new lease manager module that decouples lease requests from the requesting transaction's data set. Rather than having a transaction acquire a single lease encompassing its entire data set, each transaction acquires a set of fine-grained *Lease Ownership Records* (LORs), one per accessed conflict class.

Implementation details: ALC's *Replication Manager* (RM) was not changed. It interfaces with the LM via the getLease() and finishedXact() methods for acquiring and releasing leases, respectively. As in ALC, Lilac-TM maintains the indirection level between leases and data items through conflict classes. This allows flexible control of the leases abstraction granularity. We abstract away the mapping between a data item and a conflict class through the `getConflictClasses`() primitive, taking a set of data items as input parameter and returning a set of conflict classes.

As in ALC, each replica maintains one main data structure for managing the establishment/release of leases: *CQ* (Conflict-Queues), an array of FIFO queues, one per conflict class. The CQ keeps track of conflict relations among lease requests of different replicas. Each queue contains

Algorithm 1: Lease Manager at process p_i

```
1  FIFOQueue<LOR>
   CQ[NumConflictClasses]={⊥, ..., ⊥}
2  Set<LOR> GetLease(Set DataSet)
3      ConflictClass[] CC =
       getConflictClasses(DataSet)
4      if (∃(Set<LOR>)S⊆CQ s.t.
       ∀cc∈CC(∃lor∈S : (lor.cc=cc ∧
       lor.proc=p_i ∧ ¬lor.blocked))) then
5          foreach lor∈S do
6              lor.activeXacts++
7      else
8          Set<LOR> S =
           createLorsForConflictClasses(CC)
9          LeaseRequest req = new
           LeaseRequest(p_i,S)
10         OA-broadcast([LeaseRequest,req])
11     wait until isEnabled(S)
12     return S
13 void FinishedXact(Set<LOR> S)
14     Set<LOR> lorsToFree
15     foreach lor∈S do
16         lor.activeXacts−−
17         if (lor.blocked ∧ lor.activeXacts=0)
           then lorsToFree=lorsToFree ∪ lor
18     if (lorsToFree ≠ ∅) then
       UR-broadcast([LeaseFreed,lorsToFree])
19 upon Opt-deliver([LeaseRequest, req])
   from p_k do
20     freeLocalLeases(req.cc)
21 upon TO-deliver([LeaseRequest, req])
   from p_k do
22     Set<LOR> S =
       createLorsForConflictClasses(req.cc)
23     foreach lor∈S do
       CQ[lor.cc].enqueue(lor)
24 upon UR-deliver([LeaseFreed, Set<LOR>
   S]) from p_k do
25     foreach lor∈S do
       CQ[lor.cc].dequeue(lor)
26 void freeLocalLeases(ConflictClass[]
   CC)
27     Set<LOR> lorsToFree
28     foreach cc ∈ CC do
29         if ∃lor in CQ[cc] s.t. lor.proc=p_i
           then
30             lor.blocked=true
31             if (CQ[lor.cc].isFirst(lor) ∧
               lor.activeXacts=0) then
32                 lorsToFree=lorsToFree ∪
                   lor
33     if (lorsToFree ≠ ∅) then
       UR-broadcast([LeaseFreed,lorsToFree])
34 boolean isEnabled(Set<LOR> S)
35     return ∀lor∈S : CQ[lor.cc].isFirst(lor)
36
```

LORs, each storing the following data: (i) **proc**: the address of the requesting replica; (ii) **cc**: the conflict class this LOR is associated with; (iii) **activeXacts**: a counter keeping track of the number of active local transactions associated with this LOR, initialized to 1 when the LOR is created; and (iv) **blocked** : a flag indicating whether new local transactions can be associated with this LOR - this flag is initialized to false when the LOR is created (in the `createLorsForConflictClasses` primitive), and set to true as soon as a remote lease request is received.

Algorithm 1 presents the pseudo-code of Lilac-TM's LM. The method getLease() is invoked by the RM once a transaction reaches its commit phase. The LM then attempts to acquire leases for all items in the committing transaction's data set. It first determines, using the `getConflictClasses`() method, the set of conflict classes associated with the transaction's data set (line **3**). It then checks (in line **4**) whether CQ contains a set S of LORs such that i) the LORs were issued by p_i, and

ii) additional transactions of p_i may still be associated with these LORs (this is the case for each LOR owned by the current node that is not blocked). If the conditions of line **4** are satisfied, the current transaction can be associated with all LORs in S (lines **5**–**6**). Otherwise, a new lease request, containing the set of LORs, is created and is disseminated using OAB (lines **7**–**9**). In either case, p_i waits in line **11** until S is enabled, that is, until all the LORs in S reach the front of their corresponding FIFO queues (see the isEnabled() method). Finally, the method returns S and the RM may proceed validating the transaction.

When a transaction terminates, the RM invokes the finishedXact() method. This method receives a set of LORs and decrements the number of active transactions within each record (line **16**). Every blocked LOR that is not used by local transactions is then released by sending a single message via the *UR-broadcast* primitive (lines **17**–**18**).

Upon an *Opt-deliver* event of a remote lease request *req*, p_i invokes the freeLocalLeases() method, which blocks all LORs owned by p_i that are part of *req* by setting their *blocked* field (line **30**). Then, all LORs that are blocked and are no longer in use by local transactions are released by sending a single *UR-broadcast* message (lines **31**–**33**). Other LORs required by *req* that have local transactions associated with them (if any) will be freed when the local transactions terminate. Blocking LORs is required to ensure the fairness of the lease circulation scheme. In order to prevent a remote process p_j from waiting indefinitely for process p_i to relinquish a lease, p_i is prevented from associating new transactions with existing LORs as soon as a conflicting lease request from p_j is *Opt-delivered* at p_i.

Upon a *TO-deliver* of a lease request *req* (line **21**), p_i creates the corresponding set of LORs, and enqueues these records in their conflict class queues. The logic associated with a *UR-deliver* event (line **24**) removes each LOR specified in the message from its conflict class queue.

2.2 Transaction Forwarder

The TF is the module in charge of managing the process of migrating transactions between nodes. If at commit time the set S of conflict classes accessed by a transaction T is not already owned by its origin node, say n, the DTD may decide to avoid requesting leases for T, and forward its execution to a different node n'. In this case node n' becomes responsible for finalizing the commit phase of the transaction. This includes establishing leases on S on behalf of transaction T, which can be achieved avoiding any distributed coordination, in case n' already owns all the leases re-

quired by T'. Else, if some of the leases requested by T' are not owned by n', n' has to issue lease requests on behalf of T via the OAB service.

Next we can use a remote validation optimization and let n' perform T's final validation upon arrival (without re-executing T) in order to detect whether T has conflicts with concurrently committed transactions.[4] In case of successful validation, T can be simply committed, as in ALC, by disseminating a Commit message via the *UR-Broadcast*. Additionally, in Lilac-TM, this has the effect of unblocking the thread that requested the commit of T on node n. On the other hand, if T fails its final validation, it is re-executed on node n' until it can be successfully committed, or it fails for a pre-determined number of attempts. In this latter case, the origin node is notified of the abort of T, and the user application is notified via an explicit exception type. Note that, in order to commit the transaction associated with the re-execution of T, which we denote as T', n' must own the set of conflict classes accessed by T'. This may not be necessarily true, as T' and T may access different sets of conflict classes. In this case, Lilac-TM prevents a transaction from being forwarded an arbitrary number of times, by forcing n' to issue a lease request and acquire ownership of the leases requested by T'.

It must be noted that, in order to support the transaction forwarding process, the programming model exposed by Lilac-TM has to undergo some minor adaptations compared, e.g., with the one typically provided by non-replicated TM systems. Specifically, Lilac-TM requires that the transactional code is replicated and encapsulated by an interface that allows to seamlessly re-execute transactions originating at different nodes.

2.3 Distributed Transaction Dispatching

The DTD module allows encapsulating arbitrary policies to determine whether to process the commit of a transaction locally, by issuing lease requests if required, or to migrate its execution to a remote node. In the following we refer to this problem as the *transaction migration problem*. This problem can be formulated as an Integer Linear Programming (ILP) problem as follows:

(1) $\mathbf{min} \sum_{i \in \Pi} N_i \cdot C(i, S)$
subject to: (2) $\sum_{i \in \Pi} N_i = 1$, (3) $CPU_i \cdot N_i < maxCPU$

[4] In order to use this remote validation optimization, the TF module must be augmented with a TM-specific validation procedure and append the appropriate metadata to forwarding messages. TM-specific adaptation and overhead can be avoided by simply always re-executing the forwarded transaction once it is migrated to n'.

The above problem formulation aims at determining an assignment of the binary vector N (whose entries are all equal to 0 except for one, whose index specifies the selected node) minimizing a generic cost function $C(i, S)$ that expresses the cost for node i to be selected for managing the commit phase of a transaction accessing the conflict classes in the set S. The optimization problem specifies two constraints. Constraint (2) expresses the fact that a transaction can be certified by exactly a single node in Π. Constraint (3) is used to avoid load imbalance between nodes. It states that a node i should be considered eligible for re-scheduling only if its CPU utilization (CPU_i) is below a maximum threshold ($maxCPU$).

We now derive two different policies for satisfying the above ILP formulation, which are designed to minimize the long-term and the short-term impact of the decision on how to handle a transaction. We start by defining the cost function $LC(i, S)$, which models the *long-term cost* of selecting node i as the node that will execute the transaction as the sum of the frequency of accesses to the conflict classes in S by every other node $j \neq i \in \Pi$:

$$LC(i, S) = \sum_{x \in S} \sum_{j \in \Pi \vee j \neq i} \mathcal{F}(j, x)$$

where $\mathcal{F}(j, x)$ is defined as the per time-unit number of transactions originated on node j that have object x in their dataset.

In order to derive the the *short-term policy*, we first define the function $SC(i, S)$, which expresses the immediate costs induced at the GCS level by different choices of where to execute a transaction:

$$SC(i, S) = \begin{cases} c_{URB} \text{ if } i = O \wedge \forall x \in S : \mathcal{L}(i, x) = 1 \\ c_{AB} + 2c_{URB} \text{ if } i = O \wedge \exists x \in S : \mathcal{L}(i, x) = 0 \\ c_{p2p} + c_{AB} + 2c_{URB} \text{ if } i \neq O \wedge \exists x \in S : \mathcal{L}(i, x) = 0 \\ c_{p2p} + c_{URB} \text{ if } i \neq O \wedge \forall x \in S : \mathcal{L}(i, x) = 1 \end{cases}$$

where we denote by O the node that originated the transaction, and by c_{URB}, c_{AB} and c_{p2p} the costs of performing a URB, an AB, and a point-to-point communication, respectively. The above equations express the cost of the following scenarios (from top to bottom): i) the originating node already owns all the leases required by it; ii) the originating node does not own all the necessary leases and issues a lease request; iii) the originating node forwards the transaction to a node that does not does not own all the necessary leases; iv) the transaction is forwarded to a node that owns the leases for all required conflict classes. The DTD can be configured to use the long-term or the short term policy simply by setting the generic cost function $C(i, S)$ in (1) to, respectively, $LC(i, S)$ or $SC(i, S)$.

It is easily seen that the ILP of Equation 1 can be solved in $O(|\Pi|)$ time regardless of whether the long-term or the short-term policy is used. The statistics required for the computation of the long-term policy are computed by gathering the access frequencies of nodes to conflict classes. This information is piggybacked on the messages exchanged to commit transactions/request leases. A similar mechanism is used for exchanging information on the CPU utilization of each node. For the short term policy, we quantify the cost of the P2P, URB and OAB protocols in terms of their communication-steps latencies (which equal 1, 2, and 3, resp.).

3 Experimental Evaluation

In this section, we compare the performance of Lilac-TM with that of the baseline ALC protocol. Performance is evaluated using two benchmarks: a variant of the *Bank* benchmark [15] and the *TPC-C* benchmark [23]. We compare the following algorithms: ALC (using the implementation evaluated in [7]), FGL (ALC using the fine-grained leases mechanism), MG-ALC (ALC extended with the transaction migration mechanism), and two variants of Lilac-TM (transaction migration on top of ALC using fine-grained leases), using the short-term (Lilac-TM-ST) and the long-term (Lilac-TM-LT) policies, respectively. The source code of ALC, Lilac-TM and the benchmarks used in this study is publicly available [4].

All benchmarks were executed running 2 threads per node, and using a cluster of 4 replicas, each comprising an Intel Xeon E5506 CPU at 2.13 GHz and 32 GB of RAM, running Linux and interconnected via a private Gigabit Ethernet.

Bank. The *Bank* benchmark [9, 15] is a well-known transactional benchmark that emulates a bank system comprising a number of accounts.

We extended this benchmark with various types of read-write and read-only transactions, for generating more realistic transactional workloads. A *read-write transaction* performs transfers between randomly selected pairs of accounts. A *read-only transaction* reads the balance of a set of randomly-selected client accounts. Workloads consist of 50% read-write transactions and 50% read-only transactions of varying lengths.

We introduce data locality in the benchmark as follows. Accounts are split into *partitions* such that each partition is logically associated with a distinct replica and partitions are evenly distributed between replicas. A transaction originated on replica r accesses accounts of a single (randomly selected) partition associated with r with probability P, and ac-

counts from another (randomly selected) remote (associated with another replica) partition with probability $1 - P$. Larger values of P generate workloads characterized by higher data-locality and smaller inter-replica contention. Hence, the optimal migration policy is to forward a transaction T to the replica with which the partition accessed by T is associated. We therefore implement and evaluate a third variant of Lilac-TM (called Lilac-TM-OPT) using this optimal policy.[5]

Figure 2(a) shows the throughput (committed transactions per second) of the algorithms we evaluate on workloads generated by the bank application with P varying between 0% to 100%.

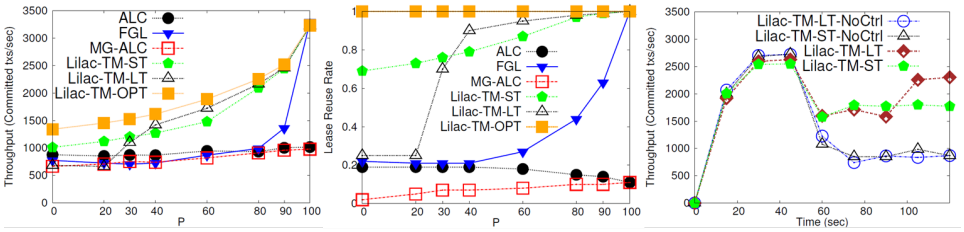
Comparing ALC and FGL, Figure 2(a) shows that, while ALC's throughput remains almost constant for all locality levels, FGL's performance dramatically increases when locality rises above 80%. This is explained by Figure 2(b), that shows the *Lease Reuse Rate*, defined as the ratio between the number of read-write transactions which are piggybacked on existing leases and the total number of read-write transactions.[6] A higher lease reuse rate results in fewer lease requests, which reduces in turn the communication overhead and the latency caused by waiting for leases. FGL's lease reuse rate approaches one for high locality levels, which enables FGL and FGL-based migration policies to achieve up to 3.2 times higher throughput as compared with ALC and MG-ALC.

When locality is lower than 80%, the FGL approach yields throughput that is comparable to ALC. Under highly-contended low-locality workloads, FGL's throughput is even approximately 10%-20% lower than that of ALC. This is because these workloads produce a growing demand for leases from all nodes. FGL releases the leases in fine-grained chunks, which results in a higher load on URB-communication as compared with ALC.

The adverse impact of low-locality workloads on transaction migration policies, however, is much lower. Migrating transactions to replicas where leases might already be present (or will benefit from acquiring it), increases the lease reuse rate, which increases throughput in turn. Indeed, as shown by Figure 2(a), Lilac-TM achieves speed-up of between 40%-100% even for low-locality workloads (0%-60%) in comparison with ALC. For high-locality workloads, both FGL and Lilac-TM converge to similar performance, outperforming ALC by a factor of 3.2.

Comparing the performance of ALC and MG-ALC shows that using transaction migration on top of ALC does not improve the lease reuse rate as compared with ALC. This is because migration only helps when used on

[5] Our MG-ALC implementation also uses this optimal migration policy.
[6] Read-only transactions never request leases.



(a) Throughput (b) Lease reuse rate (c) Overload

Fig. 2: Bank Benchmark

top of the fine-grained leases mechanism. The slightly lower throughput of MG-ALC vs ALC is due to the overhead of the TF mechanism.

Next we evaluate the ability of Lilac-TM to cope with load imbalance. To this end, we set the benchmark to access with 20% probability a single partition, p, from all the nodes, except for the single node, say n, associated with p, which accesses only p. In these settings, with all the considered policies, n tends to attract all the transactions that access p. At second 40 of the test, we overload node n by injecting external, CPU-intensive jobs. The plots in Fig. 2(c) compare the throughput achieved by Lilac-TM with and without the mechanism for overload control (implementing Inequality (3)), and with both the long-term and the short-term policies. The data highlights the effectiveness of the proposed overload control mechanism, which significantly increases system throughput. In fact, the schemes that exploit statistics on CPU utilization (Lilac-TM-ST and Lilac-TM-LT) react in a timely manner to the overload of n by avoiding further migrating their transactions towards it, and consequently achieve a throughput that is about twice that of uninformed policies (Lilac-TM-ST-NoCtrl and Lilac-TM-LT-NoCtrl).

TPC-C. We also ported the TPC-C benchmark and evaluated Lilac-TM using it. The TPC-C benchmark is representative of OLTP workloads and is useful to assess the benefits of our proposal even in the context of complex workloads that simulate real world applications. It includes a wider variety of transactions that simulate a whole-sale supplying *items* from a set of *warehouses* to *customers* within sales *districts*. We ported two of the five transactional profiles offered by TPC-C, namely the *Payment* and the *New Order* transactional profiles, that exhibit high conflict rate scenarios and long running transactional workloads, respectively. For this benchmark we inject transactions to the system by emulating a load balancer operating according to a geographically-based policy that forwards requests on the basis of the requests' geographic origin: in particular

requests sent from a certain geographic region are dispatched to the node that is responsible for the warehouses associated with the users of that region. To generate more realistic scenarios we also assume that the load balancer can do mistakes by imposing that with probability 0.2 a request sent from a certain region is issued by users associated with warehouses that do not belong to that region.

In Figure 3 we present the throughput obtained by running a workload with 95% Payment transactions and 5% New Order transactions; moreover in this case we show the throughput varying over time in order to better assess the convergence of the reschedule policies. We first notice that even in this complex scenario FGL performs better than ALC due to better exploitation of the application, and a higher leases reuse rate. In addition, using the migration mechanism, driven by either the short term (ST) or the long term (LT) policy, over FGL, achieves speedups of between 1.2 and 1.5 when compared to ALC. However, unlike the Bank Benchmark, in this case the ST policy achieves only minor gains compared to the LT policy, due to TPC-C's transactional profiles that generate more complex access patterns. In fact, even when the data set is partitioned by identifying each partition as a *warehouse* and all the objects associated with that *warehouse*, TPC-C's transactions may access more than one partition. This reduces the probability that the ST policy can actually trigger a reschedule for a transaction on a node that already owns all the leases necessary to validate/commit that transaction. On the other hand the LT policy can exploit application locality thus noticeably reducing lease requests circulation, i.e. the number of lease requests issued per second.

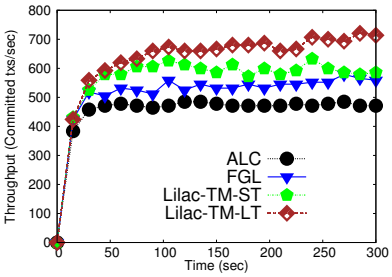


Fig. 3: TPCC

4 Conclusions

In this paper we introduced Lilac-TM, a fully decentralized, LocalIty-aware LeAse-based repliCated TM (Lilac-TM). Lilac-TM exploits a novel, self-optimizing lease circulation scheme that provides two key benefits: (1) limiting the frequency of lease circulation, and (2) enhancing the contention management efficiency, by increasing the probability that conflicting transactions are executed on the same node.

By means of an experimental evaluation based on both synthetic and realistic benchmarks we have shown that Lilac-TM can yield significant

speed-ups, reaching peak gains of up to 3.2 times with respect to state of the art lease-based replication protocol.

References

1. M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07*, pages 159–174.
2. E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., March 2008. Version 1.0.
3. C. Amza, A. Cox, K. Rajamani, and W. Zwaenepoel. Tradeoffs between false sharing and aggregation in software distributed shared memory. In *PPoPP '97*.
4. Aristos Project. `http://aristos.gsd.inesc-id.pt`, 2013.
5. A. Bartoli and O. Babaoglu. Selecting a "primary partition" in partitionable asynchronous distributed systems. In *SRDS '97*.
6. J. Cachopo. *Development of Rich Domain Models with Atomic Actions*. PhD thesis, Technical University of Lisbon, 2007.
7. N. Carvalho, P. Romano, and L. Rodrigues. A generic framework for replicated software transactional memories. In *NCA '11*, pages 271–274.
8. N. Carvalho, P. Romano, and L. Rodrigues. Asynchronous lease-based replication of software transactional memory. In *Middleware*, pages 376–396, 2010.
9. N. Carvalho, P. Romano, and L. Rodrigues. Scert: Speculative certification in replicated software transactional memories. In *SYSTOR*, page 10, 2011.
10. G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.
11. M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D^{2}STM: Dependable Distributed Software Transactional Memory. In *Proc. of PRDC*, 2009.
12. X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.
13. D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *DISC '06*, pages 194–208.
14. R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.
15. M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06*, pages 253–262.
16. M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93*, ISCA '93, pages 289–300.
17. Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-018. March 2009.
18. H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. of ICDCS*, Apr.
19. P. Romano, N. Carvalho, and L. Rodrigues. Towards distributed software transactional memory systems. In *LADIS '2008*.
20. M. M. Saad and B. Ravindran. Transactional forwarding: Supporting highly-concurrent stm in asynchronous distributed systems. In *SBAC-PAD*, pages 219–226. IEEE, 2012.
21. M. Schindewolf, A. Cohen, W. Karl, A. Marongiu, and L. Benini. Towards transactional memory support for GCC. In *GROW'2009*.
22. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
23. TPC Council. TPC-C Benchmark, Revision 5.11. Feb. 2010.