Performance Modelling of Partially Replicated In-Memory Transactional Stores

Diego Didona, Paolo Romano
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa

***Abstract*—This paper presents *PROMPT*, a PeRfOrmance Model for Partially replicated in-memory Transactional cloud stores. *PROMPT* combines white box Analytical Modelling and Machine Learning techniques, with the goal of achieving the best of the two methodologies: low training times, high extrapolation power, and portability across heterogeneous cloud infrastructures. We validate *PROMPT* via an extensive experimental study based on a popular open-source transactional in-memory data store (Red Hat's Infinispan), industry-standard benchmarks, and deployments on both public and private cloud infrastructures.**

I. Introduction

The advent of the Cloud Computing paradigm has empowered programmers with the ability to scale out their applications easily to hundreds of nodes, fostering research in the area of highly scalable, elastic distributed data platforms. Modern cloud storage systems, often denoted as NoSQL, explore new trade-offs in the design space of distributed data stores in order to maximize scalability. NoSQL data stores typically adopt less expressive data models than the classic relational one, and opt for simpler, yet more scalable, paradigms, as in key-value stores [1]–[3]. In order to enhance performance, and remove logging to stable storage from the critical path of execution, these systems typically maintain data fully in-memory and rely on replication to ensure fault-tolerance and data durability.

Regarding data replication, in large scale systems the reference approach consists in replicating data on a number of nodes that is typically much lower than the global scale of the platform: this is in contrast with classic full replication mechanisms, which quickly incur prohibitive updates dissemination costs as the scale of the system grows. As for consistency, first generation of NoSQL stores [3] embraced very weak consistency models, like eventual consistency. However, the inherent complexity of building applications on top of weakly consistent systems has been recently recognized by some of the pioneers of eventual consistency [4], and motivated several works providing stronger consistency guarantees [2,5]–[7].

On the other hand, these systems can exhibit non-linear scalability trends [8]–[10] due to factors related to contention on both physical (e.g., CPU and network) and logical resources (transactions' conflicts), which challenge existing performance modelling methodologies for distributed transactional platforms [9,11]. The lack of accurate performance prediction models for such a relevant class of cloud data stores represents a major impairment for the development of automatic QoS-oriented provisioning schemes, which are essential to take maximum advantage of the utility computing paradigm.

This paper tackles this timely and relevant issue by introducing *PROMPT*, a novel performance model for partially replicated transactional key-value stores. *PROMPT* combines Analytical Modelling and Machine Learning techniques, with the goal of achieving the best of the two methodologies: low training times, high extrapolation power, and portability across heterogeneous cloud infrastructures.

PROMPT relies on white box Analytical Modelling (AM) to capture data contention dynamics, and on black box Machine Learning (ML) techniques to infer performance models of the underlying network infrastructure. The use of black box statistical techniques allows the applicability of *PROMPT* also in public cloud scenarios in which very limited or no information is provided on the topology of the network infrastructure over which the data store is deployed (hence inhibiting the development of detailed white box models). The joint usage of an analytical model for data contention, on the other hand, allows to accurately predict the scalability trends of a broad range of transactional workloads, and to support what-if analysis across a large number of workload's parameters. Further, it makes it possible to reduce the dimensionality of the features space of the ML component of *PROMPT*, and, consequently, its training time.

This work makes three main contributions:

- we introduce an analytical model that captures the effects of data locality on throughput and data contention, as well as shifts of data locality due to changes of the platform's scale. The model considers data distribution schemes that are widely employed in modern NoSQL stores, like those based on consistent hashing [12];
- we present the analytical model of the concurrency control and replication schemes of a popular open-source NoSQL in-memory transactional store, i.e., Infinispan by Red Hat [2], which combines highly scalable techniques, such as partial replication schemes and commit-time locking strategies;
- in order to enhance the accuracy of the ML-based network model, we introduce and evaluate the idea of using multiple specialized learners trained on disjoint data sets, each capturing radically different workload/deployment scenarios. This technique is particularly relevant for the case of partially replicated data stores, in which even minor changes of the data replication degree can lead to strong non-linearity in the latency of network-bound operations.

We demonstrate the viability of our proposal via an extensive evaluation study using both a private and public cloud infrastructure, and relying on popular benchmarks generating

This work has been partially supported by the projects specSTM (PTDC/EIA-EIA/122785/2010), Cloud-TM (FP7/257784) and by FCT - Fundação para a Ciência e a Tecnologia through PEst-OE/EEI/LA0021/2013. The authors thank Koji Tanaka for his support with the FutureGrid test-bed.

strongly heterogeneous workloads.

The structure of this paper is the following: in Sec. II we discuss related work; Sec. III overviews the concurrency control and replication schemes considered in this work; in Sec. IV we describe the proposed performance model; Sec. V presents the model validation; Sec. VI concludes the paper.

II. Related Work

The body of literature on transactional systems is quite broad, and a number of analytical and simulative performace models have been proposed, both for centralized and distributed architectures [13]. Nevertheless, these works typically rely on assumptions on the workload or on simplifications that can hinder their accuracy in real environments. These include: uniform data access patterns [14,15] or detailed a-priori knowledge of the distribution of access to data [8,16]–[18]; extreme replication policies, i.e., full [19,20] or no replication [21]; scale-independent probabilities of accessing remote data [22] — which do not match the dynamics induced by popular distribution policies like consistent hashing; constant communication delays [9,19] or detailed knowledge of the underlying network topology/protocols [13,23,24].

Unlike these solutions, the analytical model of *PROMPT* captures the shift in data locality as the size of the system changes and relies on an abstraction that concisely allows to encompass (and characterize at runtime) non-uniform data access patterns. Moreover, it exploits ML techniques to predict response time of network-bound operations, which makes it applicable also in complex cloud environments.

Our work is also related to techniques for resource provisioning in multi-tier systems. Existing solutions in this area [25]–[28] do not take into account contention on data, targeting mainly CPU-bound applications, even though they also consider the database tier. Moreover, they neglect the effect of data distribution/replication, which have a major impact on the performance of partially replicated data stores.

Other approaches [11,29] rely exclusively on ML techniques to predict the performance of distributed data-centric systems. However, ML techniques have typically a reduced extrapolation power with respect to analytical models, thus making their adoption cumbersome to perform what-if analysis [8]. By restricting the scope of black box modelling exclusively to the network dynamics, and offloading to an analytical model the prediction of the effects of data contention/locality, *PROMPT* allows for significantly reducing ML training time, as we will discuss more in detail in Sec. V.

The closest work to the presented approach is Transactional Auto Scaler (TAS) [20], which introduced a performance model for a fully replicated transactional key-value store. Similarly to *PROMPT*, TAS relies on the synergistic use of AM and ML for predicting, respectively, the impact on performance of data contention and distributed transaction synchronization. However, building accurate performance models for partially replicated transactional data stores requires tackling a number of additional challenges concerning both the development of the analytical data contention model and the black box network model. The former needs to take into account the intertwined effects of replication degree and data locality on data contention. Moreover, the concurrency control scheme modeled in [20] is based on an encounter-time locking scheme, whereas in this paper we consider a more scalable commit-time locking scheme [2]. On the other hand, in a partially replicated system, the ML-based network performance model grows in complexity for a twofold reason: i) in addition to the latency of the commit phase, it is also necessary to predict the one associated with the fetching of data maintained on remote nodes, and ii) the performance of the network layer is affected by a larger number of factors (e.g., replication degree, size of objects retrieved remotely, and number of nodes involved in the commit phase), which lead to an increase of the dimensionality of the features space over which the ML needs to be trained.

III. Model of the Target System

PROMPT targets partially replicated in-memory transactional key-value stores. These systems typically rely on deterministic functions, such as consistent hashing [12], which allow for: i) determining the location of any data item in the system using a local lookup function, avoiding the costs of accessing remote lookup/directory services; ii) spreading data (including hot-spots) evenly across the nodes of the system, achieving good balancing both in terms of load and storage consumption; iii) minimizing the number of data redistributed in the platform upon joins/leaves of nodes.

A crucial parameter of any partially replicated data store is the, so called, *replication degree*, which determines how many nodes of the system replicate each datum. In fact, the tuning of this parameter has a strong impact both on the probability that a transaction accesses a datum not stored by the node where it is executed, as well as on the number of nodes to contact in order to commit update transactions.

Regarding concurrency control and data replication, *PROMPT* considers the protocols employed by Infinispan [2], a popular open-source in-memory transactional key-value store that is widely employed in a number of Red Hat products, including JBoss AS, namely one of the most popular J2EE open-source application servers, and deployed in a large number of application domains (from e-commerce, to telecommunication and finance). Given that *PROMPT* employs a white box analytical model for capturing the effects of data contention, in the following we provide an overview of the main mechanisms used by Infinispan to ensure transactional consistency.

Like other NoSQL platforms [3,30], Infinispan sacrifices consistency in order to enhance scalability, and, despite providing transactional semantics, it ensures a weaker isolation criterion than classic serializability, namely Repeatable Read [31]. Infinispan implements a non-serializable variant of the multi-version concurrency control algorithm, which never blocks or aborts any transaction upon a read operation, allowing read-only transactions to commit locally, without requiring any inter-node synchronization. Upon commit of an update transaction, instead, a variant of the Two Phase Commit protocol (2PC) is executed, in order to ensure that the transaction's updates are atomically applied on all replicas (also called cohorts) that store data modified by the transaction. More in detail, Infinispan associates (by means of a deterministic hash function) a, so called, *primary owner* node with each data item. Upon commit of an update transaction, the transaction

originator first tries to acquire the locks for the written items it is primary owner of. If no conflict arises, then acquisition of the remaining locks is attempted on the other relevant primary owner nodes. If such lock acquisition phase is successful on all nodes, the transaction originator commits locally and broadcasts a commit message, in order to apply the transaction's modifications on the remote nodes; otherwise it broadcasts an abort message and the transaction is rolled back. Note that, although locks are acquired only on the primary owner nodes, all the nodes that store a replica of written data items are contacted at prepare-time. This allows the implementation of schemes aimed at enabling the recoverability of a transaction in case of a failure of primary owners during the prepare phase.

Deadlocks are detected using a simple, user-tunable, timeout based approach. In this paper, we consider the scenario in which the timeout on deadlock detection is set to 0 in order to achieve deadlock freedom.

IV. Performance Model

This section describes the performance prediction methodology employed by *PROMPT*. As already discussed, *PROMPT* uses AM and ML in synergy. Specifically, AM is employed to capture the effect of data and CPU contention, whereas ML is used to forecast response time of network-bound operations.

We present the AM component of *PROMPT* in Sec. IV-A, and the ML one in Sec. IV-B. Finally, in Sec. IV-C, we describe how the AM and ML are coupled, and how the model is solved to obtain different performance metrics.

A. Analytical Model of Data Contention

Our analytical model relies on Average Value Approximation [32] to forecast the probability of transaction commit, the mean transaction duration, and achievable throughput. The aim of the model is to support what-if analysis on the application's performance when changing parameters such as the scale (number of nodes and possibly number of threads) and the replication degree, or shifts of workload characteristics, such as changes of the transactions' data access patterns or of the transactional mix.

The model considers the number of nodes in the system (noted N), the number of threads processing transactions at each node (θ) and the replication degree of data items (r) as input parameters. For the sake of simplicity, the nodes are assumed to be homogeneous in terms of computational power and RAM, and the model distinguishes only two classes of transactions, namely read-only and update transactions. As we focus on in-memory data stores, we assume that the data set maintained at each node fits fully in RAM, and we do not model interactions with persistent storage systems.

We consider an open system in which transactions arrive according to a Poisson process with rate λ_{tx}, evenly distributed across all nodes in the system. We note $\%w$ the percentage of update transactions, which perform, on average, N_g^{UP} read and N_w update operations on distinct data items; read-only transactions, which compose the other $1-\%w$ of the transactional mix, read, on average N_g^{RO} distinct data items.

The model relies on two sets of assumptions. The first concerns the independence of data accesses and re-executions of transactions. More precisely, we assume that each read or written datum is chosen independently from each other; also, once a transaction is restarted upon experiencing an abort, it is indistinguishable from a transaction that joins the system for the first time. The second kind of assumption concerns the mapping of data replicas onto nodes of the system, and the data locality exhibited by transactions. The model relies on the definition of a primary owner function $P_O(N,r)$ that, given the number of nodes in the system and the replication degree, determines what is the probability that a transaction originated on a node ν accesses a datum for which ν is primary owner. We assume that the $r-1$ non-primary replicas of a datum are scattered across the system uniformly at random. Hence, assuming that ν is not the primary replica for a data item x, the probability that ν is a non-primary owner replica of x is $l = \frac{r-1}{N-1}$. Thus, the probability that a transaction originated on node ν accesses a datum stored by ν is:

$$\mathscr{L}(N,r) = P_O(N,r) + (1-P_O(N,r))l$$

To simplify notation, in the following we write simply P_O and $\mathscr{L}$, omitting the arguments N and r. We further assume that whenever a transaction originated on a node ν accesses a datum x for which ν is not primary owner, any other node has the same probability $\frac{1}{N-1}$ of being primary owner of x.

Although the model encompasses the possibility for some data items to be more frequently accessed than others, it assumes that data hot spots are evenly distributed across nodes of the system. Note that this assumption is not unrealistic, as load balancing and data hot-spot avoidance are among the key advantages of the data distribution policies (e.g., consistent hashing [12]) that are typically employed by modern distributed NoSQL stores [1]–[3]. Overall, this assumption, along with the one on the uniformity of the transactional arrival rate, allows us to consider all nodes as evenly loaded.

Finally, we assume that the system is stable and ergodic: this means that all the parameters are defined to be either long-run averages or steady-state quantities; also, this implies that the transactions arrival rate does not exceed the service rate.

Execution times. We describe the analytical model of *PROMPT* in a top-down fashion: we start by showing how response times for read-only and update transactions are computed. It should be noted that the equations in this section assume the knowledge of the execution time of the various transactions' phases and of the expected number of aborts experienced by transactions, which will be derived later.

Read-only transactions' execution time. As discussed in Sec. III, read-only transactions do not acquire locks, thus never incurring in aborts, and not requiring validation phases. Therefore, their response time, noted R_L^{RO}, is simply determined as the sum of the time spent to i) initialize the transaction (R^{beg}), ii) perform read operations (R^g), iii) execute the business logic of the transaction R_B^{RO}, and iv) commit (R_{com}^{RO}):

$$R_L^{RO} = R^{beg} + R_B^{RO} + N_g^{RO} R^g + R_{com}^{RO}$$

where R^g is computed as the average cost of performing a local read (R_L^g), and a remote one (R_R^g):

$$R^g = \mathscr{L} R_L^g + (1-\mathscr{L})(R_L^g + R_R^g)$$

The cost for a remote get is the sum of a local computation (R_L^g) and of the latency for retrieving the remote datum (R_R^γ).

Update transactions' execution time. Unlike read-only transactions, update transactions can abort while acquiring locks during the final validation phase. The response time of an update transaction is, thus, given by the sum of the response time of a successful execution plus the time spent in previous aborted executions.

We denote as R_L^{UP} the local execution time of an update transaction whose average business logic duration is R_B^{UP} and that executes, on average, N_R read operations and N_w write operations. Noting R^p the response time of a put operation (which is executed locally even if the updated datum is remote), we have:

$$R_L^{UP} = R^{beg} + R_B^{UP} + N_g^{UP} R^g + N_w R^p$$

Further, we note i) R^{wb} the time needed to locally perform the write-back phase of updated data, ii) R^{dec} the time to send the commit or abort decision to the remote nodes, iii) R_L^{prep} the time it takes to perform the local locks acquisition, and iv) R^{prep} the response time to complete the remote prepare phase of the two-phase commit. Note that a transaction coordinator always waits for all the cohorts' replies, thus the latter cost is considered independent from the final outcome (abort/commit) of the transaction. Finally, we note R_L^{roll} the execution time of a rollback executed locally.

Hence, a run of an update transaction that successfully completes its execution (R_C^{UP}) has a response time given by:

$$R_C^{UP} = R_L^{UP} + R_L^{prep} + R^{prep} + R^{wb} + R^{dec}$$

At commit time, a transaction can fail either when acquiring locks for which the node is primary owner or, if the local validation is successful, on a remote node during prepare phase of 2PC. In the first case, the response time is given by:

$$R_{AL}^{UP} = R_L^{UP} + R_L^{prep} + R_L^{roll}$$

while, in the second case, it is:

$$R_{AR}^{UP} = R_L^{UP} + R_L^{prep} + R^{prep} + R_L^{roll} + R^{dec}$$

Note that we consider the time to complete, either successfully or aborting, the sequence of lock acquisitions/releases during the prepare phase as independent from the number of updated data items. This assumption is justifiable considering that transactions immediately abort upon conflict. Also, in a distributed data platform, the time for successfully acquiring a lock (without waiting) is significantly lower than the entire transaction duration, which is typically dominated by the latency of inter-node communication.

Finally, we note N_{AR} the number of expected aborts due to lock conflicts arising during the remote validation phase; for each of the transaction executions that reach the remote validation phase (N_{AR} of which are unsuccessful, plus a final, successful one) we note N_{AL} the number of aborts due to lock contention arising on the local node. Therefore, the total response time of an update transaction (R^{UP}) is computed as:

$$R^{UP} = N_{AR}(R_{AR}^{UP} + N_{AL} R_{AL}^{UP}) + N_{AL} R_{AL}^{UP} + R_C^{UP}$$

Data contention model. In order to obtain the expected number of aborts for an update transactions, we need to compute the lock conflict probability. To this end, like in previous works (e.g., [14,15,20]), each lock is modeled as an independent $M/G/1$ server: a lock is acquired at a rate λ_{lock} and the average service time to complete a request, T_h, is the lock hold time, namely the time since the lock acquisition and until its release. Note that, in general, a transaction can hold more than one lock at a single time and, as we shall see, hold times for locks taken by the same transaction are correlated: such characteristics violate the independence assumption for the aforementioned server. We opt, nevertheless, for this simplifying assumption in order to ease the tractability of the model. By exploiting this assumption, the probability that a transaction incurs contention when trying to acquire a lock is computed as the utilization of the corresponding server. Such metric is defined as the fraction of time that a server is busy serving a request [33] and, in our case, it is computed as $U = \lambda_{lock} T_h$ (assuming $\lambda_{lock} \cdot T_h \leq 1$).

On its turn, λ_{lock} depends on the application's data access pattern: if every datum was equally likely to be accessed in a data-set of cardinality D, then, noting Λ_{lock} the total locks acquisition rate, we would have $\lambda_{lock} = \frac{\Lambda_{lock}}{D}$. However, this uniformity assumption does not hold in general, as data access distributions are often skewed. Unfortunately, a full characterization of an application's data access pattern requires an extensive and costly profiling phase, typically performed offline [8]. In order to address this issue, *PROMPT* relies on the abstraction of the Application Contention Factor (ACF) [20], a metric that allows to succinctly characterize the skew of a data access pattern. The ACF is a scalar value that is inferred from the application behavior, with minimal profiling, and represents the inverse of the cardinality of an equivalent database that, under a uniform data access pattern, would yield to the measured contention probability. Namely, noting P_{lock} the lock contention probability:

$$ACF = \frac{P_{lock}}{\Lambda_{lock} T_h} \tag{1}$$

Previous work [20] has shown that, under the hypotheses described in Sec. IV-A, the ACF can be considered an invariant of the application's workload. Therefore, it can be exploited by *PROMPT* to speculate about the performance of the application when deployed over a different number of nodes, or when changing the replication degree in the platform. Moreover, given that its computation relies only on the profiling of average values, it is very lightweight to compute online, thus allowing the adoption of *PROMPT* also in presence of time varying data access patterns.

Abort Probabilities Computation. When a transaction T tries to acquire a lock on node v, it can conflict both with local and remote transactions. Since a lock can be held by only one transaction at any given time and there is no lock waiting, the events of contending with a local transaction and with a remote one are disjoint. Thus, the lock contention probability (noted P_{lock}) is expressed as the sum of these probabilities (noted resp. $P_{L,lock}$ and $P_{R,lock}$):

$$P_{lock} = P_{L,lock} + P_{R,lock} \tag{2}$$

In the model, we only consider conflicts arising on a node v between T and any other transaction that successfully

completes the lock acquisition phase on ν. In fact, a transaction releases all the locks it holds on a node upon detecting a conflict on that node. Hence, we consider negligible the hold time of the locks owned by an aborting transaction up to the occurrence of the conflict.

For this reason, when computing the contention probability P_{lock} for a lock on node ν, we consider four kinds of transactions: i) local transactions that commit; ii) local transactions that abort during the remote validation phase; iii) remote transactions that commit; iv) remote transactions that successfully complete the locks acquisition phase on a node ν, but abort on another node ν'. Hence:

$$P_{L,lock} \approx P^C_{L,lock} + P^A_{L,lock} = N \cdot ACF(\lambda^C_{L,lock} T^C_{Lh} + \lambda^A_{L,lock} T^A_{Lh})$$

$$P_{R,lock} \approx P^C_{R,lock} + P^A_{R,lock} = N \cdot ACF(\lambda^C_{R,lock} T^C_{Rh} + \lambda^A_{R,lock} T^A_{Rh})$$

where we use the subscript L, resp. R, to denote local, resp. remote, transactions, and added the superscript C, resp. A, to denote committing, resp. aborting, transactions. These last equations are specializations of Eq. 1; however, here, the ACF is multiplied by the total number of nodes in the system. This is because, as we shall show shortly, the locks acquisition rates used in the previous equations refer to the node where the lock is requested. Given that, by hypothesis, each node stores the same amount of data and data hot spots are evenly spread across N nodes, the "per node" equivalent uniform dataset cardinality is $(N \cdot ACF)^{-1}$.

The locks acquisition rate on a node for each of the four classes is computed as the product of the transactions arrival rate λ for that class in the whole system and of the expected number of locks N_{lock} acquired by a transaction of that class on a specific node, i.e.:

$$\lambda^C_{L,lock} = \lambda^{UP}_{Ltx} N^C_{L,lock}, \qquad \lambda^A_{L,lock} = N_{AR} \lambda^{UP}_{Ltx} N^A_{L,lock}$$

$$\lambda^C_{R,lock} = \lambda^{UP}_{Rtx} N^C_{R,lock}, \qquad \lambda^A_{R,lock} = N_{AR} \lambda^{UP}_{Rtx} N^A_{R,lock}$$

where we noted λ^{UP}_L, resp. λ^{UP}_R, the global arrival rate of local, resp. remote, update transactions, which can be computed as:

$$\lambda^{UP}_{Ltx} = \frac{\lambda \%w}{N}, \qquad \lambda^{UP}_{Rtx} = (N-1)\lambda^{UP}_{Ltx}$$

Let us now show how to compute the per class expected number of locks acquired on a node. To this end, however, we shall first introduce the following set of probabilities: i) $P_{LL}(i)$, i.e., the probability that exactly i locks are acquired on the local node of a transaction; ii) $P_A(i)$, i.e., the probability of aborting while trying to acquire i locks on a node; iii) P_{com}, i.e., the probability of successfully committing a transaction; iv) P_{AR}, i.e., the probability of successfully acquiring all the local locks and aborting during the two-phase commit. Exploiting the assumption that data are accessed independently, we can compute these probabilities as follows:

$$P_{LL}(i) = \binom{N_w}{i} P_O^i (1-P_O)^{N_w - i}$$

$$P_a(i) = 1 - (1-P_{lock})^i, \qquad P_{com} = 1 - P_a(N_w)$$

$$P_{AR} = \sum_{i=0}^{N_w} P_{LL}(i)(1-P_a(i))P_a(N_w - i)$$

The last probability has been computed as a weighted sum of the probability of requesting i local locks, acquiring all of them locally, and failing in acquiring at least one of the $N_w - i$ remote locks (with i ranging from 0 to N_w).

From these equations we can obtain other intermediate probabilities that can be used to compute the expected number of acquired locks on a node per transaction type. For a local transaction T originated on node ν, we note i) $P(N_{L,lock} = i|C)$ the probability that T acquires i locks on ν, provided that it commits; ii) $P(N_{R,lock} = i|C)$ the probability that T acquires i locks on a remote node ν', provided that it commits; iii) $P(N_{L,lock} = i|AR)$ the probability that T acquires i locks on ν, provided that it remotely aborts; iv) $P(N_{R,lock} = i|AR)$ the probability that T successfully acquires all the i locks that it requires on a remote node ν', provided that it aborts remotely (i.e., on some other remote node). By using Bayes' theorem, we compute these probabilities as follows:

$$P(N_{L,lock} = i|C) = P_{LL}(i), \quad P(N_{R,lock} = i|C) = \sum_{j=0}^{N_w - i} P^\dagger_L(j,i,N_w)$$

$$P(N_{L,lock} = i|AR) = \frac{P_{LL}(i)(1-P_a(i))P_a(N_w - i)}{P_{AR}}$$

$$P(N_{R,lock} = i|AR) = \frac{\sum_{j=0}^{N_w - i - 1} P^\dagger_L(j,i,N_w) P^\dagger_A(j,i,N_w)}{P_{AR}} \qquad (3)$$

In the last equations, we denoted as $P^\dagger_L(i,j,k)$ the probability that T acquires i locks locally, j locks on a node ν' and $k-i-j$ locks on other remote nodes; as $P^\dagger_A(i,j,k)$ the probability of successfully acquiring the i locally requested locks and the j locks requested on a remote node ν', while failing to acquire the remaining $k-i-j$ locks on other remote nodes. Exploiting the hypothesis of independent accesses to data, we can compute the former probability as a multinomial distribution, i.e.:

$$P^\dagger_L(i,j,k) = \frac{k!}{i!j!(k-i-j)!} P_O^i \left(\frac{1-P_O}{N-1}\right)^j \left(\frac{(1-P_O)(N-2)}{N-1}\right)^{k-i-j}$$

and the latter as:

$$P^\dagger_A(i,j,k) = (1-P_a(i+j))P_a(k-i-j)$$

From these sets of probabilities we can finally compute the expected number of locks acquired on a node by the different transactions kinds, namely:

$$N^C_{L,lock} = \sum_{i=1}^{N_w} iP(N_{L,lock} = i|C), \quad N^A_{L,lock} = \sum_{i=1}^{N_w} iP(N_{L,lock} = i|AR)$$

$$N^C_{R,lock} = \sum_{i=1}^{N_w} iP(N_{R,lock} = i|C), \quad N^A_{R,lock} = \sum_{i=1}^{N_w - 1} iP(N_{R,lock} = i|AR)$$

We can now compute the transactions' lock hold times. Local transactions hold locks during the whole prepare phase and release them before sending the final commit/rollback message. As already mentioned, the transaction coordinator waits for all the cohorts' replies, regardless of the outcome of the distributed commit phase. Remote transactions, on the other hand, hold locks for the time necessary to send back to the coordinator their vote and to receive the final commit/rollback message. In the model, in order to simplify

the analysis, we consider this last latency comparable to the latency experienced by the coordinator to complete the prepare phase. Moreover, we consider negligible the impact that the lock acquisition/release phase has on hold time (as compared to distributed the commit latency). Thus:

$$T^C_{L,H} = T^A_{L,H} = T^C_{R,H} = T^A_{R,H} = R^{prep}$$

Finally, we are able to obtain the expected number of aborts experienced locally and remotely by a transaction, noted, respectively, N_{AL} and N_{AR}, which we introduced in Sec. IV-A. By leveraging on the hypothesis that a retrying transaction is indistinguishable from a transaction that enters the system for the first time, we have $N_{AL} = \frac{P_{AL}}{1-P_{AL}}$ and $N_{AR} = \frac{P_{AR}}{1-P_{AR}}$, with $P_{AL} = \sum_{i=1}^{N_w} P_{LL}(i)P_a(i)$.

Remote nodes involved in 2PC. We now derive the average number of remote nodes contacted during the 2PC (noted NR), given that a transaction does not abort while acquiring local locks. We compute NR as the product of the number of remote nodes, $N-1$, and the probability (noted $P(R \geq 1|\neg LA)$) that a remote node replicates at least one datum written by the transaction, given that the transaction does not abort locally prior to starting 2PC. We obtain this probability by marginalizing over the distribution of the number of acquired local locks given that the transaction does not abort locally:

$$P(R \geq 1|\neg LA) = \sum_{i=0}^{N_w} P(R \geq 1 \wedge N_{L,lock} = i|\neg LA)$$

For the law of total probability, this can be rewritten as

$$P(R \geq 1|\neg LA) = \sum_{i=0}^{N_w} P(R \geq 1|\neg LA \wedge N_{L,lock} = i)P(N_{L,lock} = i|\neg LA)$$

We note $P(N_{L,lock} = i|\neg LA)$ the probability that a transaction has requested i local locks given that it has reached the distributed prepare phase, i.e.:

$$P(N_{L,lock} = i|\neg LA) = \frac{P_{LL}(i)(1-P_a(i))}{P_{AL}}$$

Note that $P(R \geq 1|\neg LA \wedge N_{L,lock} = i) = P(R \geq 1|N_{L,lock} = i)$ because the fact that a node replicates at least one datum is independent from the outcome of the transaction and only depends on the number i of local locks.

We now obtain the probability that a remote node replicates a datum written by the local node. In the case the local node is primary owner for a datum, this probability coincides with l. If the local node is not primary owner of the datum, there are two cases: the local node replicates the datum, which yields a probability l^2, or does not replicate it, which yields a probability $\frac{(1-l)r}{N-1}$. We note $P(R|\neg P_O)$ the sum of these two probabilities, i.e., $P(R|\neg P_O) = l^2 + \frac{(1-l)r}{N-1}$. Overall, when the local node writes i items, for which it is primary owner, and $N_w - i$ remote ones, the probability that a remote node replicates at least one of such items is

$$P(R|N_{L,lock} = i) = 1 - (1-l)^i(1-P(R|\neg P_O))^{N_w - i}$$

Thus, the expected number of remote nodes being contacted upon a prepare phase, given the prepare phase is reached, is:

$$NR = (N-1)\sum_{i=0}^{N_w} P(R|N_{L,lock} = i)P(N_{L,lock} = i|\neg LA)$$

CPU contention model. Like in previous models [15,20], we model the CPU of the nodes of the platform as a $M/M/K$ multi-class queue with FCFS discipline, where K is the number of cores per CPU. The CPU serves five classes of jobs: read-only (L^{RO}) and local update (L^{UP}) transactions, requests for serving remote gets (R^{RG}) and remote update transactions that commit (RC^{UP}) or abort (RA^{UP}). Denoting as D_i, resp. λ_i, the service demand, resp. the arrival rate, of jobs belonging to class i, one can compute the CPU utilization, ρ, as:

$$\rho = \frac{\lambda^{RO}_L D^{RO}_L + \lambda^{UP}_L D^{UP}_L + \lambda^{UP}_{RA} D^{UP}_{RA} + \lambda^{UP}_{RC} D^{UP}_{RC} + \lambda^{RG}_R D^{RG}_R}{K}$$

Then, defining

$$\alpha = \frac{K\rho^K}{K!(1-\rho)}, \quad \beta = \sum_{i=1}^{K-1} \frac{K\rho^i}{i!}, \quad \gamma = 1 + \frac{\alpha}{K(\alpha+\beta)(1-\rho)}$$

we have that the CPU response time R_i corresponding to a job with demand D_i, without taking into account the latency of network-bound operations, is given by $R_i = \gamma D_i$.

Let us now derive the CPU demands and arrival rates for the classes. We note D^g_L the CPU demand of a local read, D^p the CPU demand of a put operation, and D^γ_R the CPU demand to retrieve a remote datum. Hence, for read-only transactions, we have:

$$D^{RO}_L = D^{beg} + D^{RO}_B + \sum_{i=1}^{N_r} [\mathscr{L} D^g_L + (1-\mathscr{L})(D^g_L + D^\gamma_R)] + D^{RO}_{com}$$

$$\lambda^{RO} = \frac{(1-\%w)\lambda_{tx}}{N}$$

The CPU demand of a local update transaction is given by the service time of a successful run (D^{UP}_{LC}) plus the CPU time spent in locally or remotely aborted runs (noted, respectively, D^{UP}_{LLA} and D^{UP}_{LRA}). Defining D^{UP}_{LL} the CPU demand of the local execution of an update transaction, we have

$$D^{UP} = D^{UP}_{LC} + N_{RA}(D^{UP}_{LRA} + N_{LA} D^{UP}_{LLA}) + N_{LA} D^{UP}_{LLA} + D^{UP}_{com}$$

We compute the contributes in the former equation as follows:

$$D^{UP}_{LL} = D^{beg} + D^{UP}_B + N_w D^p + \sum_{i=1}^{N_R} [\mathscr{L} D^g_L + (1-\mathscr{L})(D^g_L + D^\gamma_R)]$$

$$D^{UP}_{LC} = D^{UP}_{LL} + D^{prep}_L + D^{prep} + D^{wb} + D^{dec}$$

$$D^{UP}_{LLA} = D^{UP}_{LL} + D^{prep}_L + D^{rol}_L$$

$$D^{UP}_{LRA} = D^{UP}_{LL} + D^{prep}_L + D^{prep} + D^{rol}_L + D^{dec}$$

where we adopted the following notation for CPU demands: D^{prep}_L for the local validation phase; D^{prep} for executing the distributed prepare phase; D^{wb} for the write-back; D^{dec} for broadcasting the decision about a transaction's outcome; D^{rol}_L for performing a rollback locally. Likewise, the CPU demand for remote transactions is computed as:

$$D^{UP}_R = D^{prep}_R + D^{com}_R + N_{AR}(D^{prep}_R + D^{rol}_R)$$

where D^{prep}_R, D^{com}_R and D^{rol}_R are, respectively, the CPU demands

of remote prepare, commit and rollback operations. The arrival rate of local update transactions, λ_L^{UP}, is equal to λ_{Ltx}^{UP}, which has already been obtained in the previous section. The arrival rate of remote transactions being validated on a node v, instead, is obtained as the product of the global arrival rate of remote transactions and the probability that v is primary owner for at least one lock. As before, we have to compute this probability for the case of remote committing and aborting transactions. The former is computed as:

$$P(N_{R,lock} \geq 1|C) = \sum_{i=1}^{N_w} P(N_{R,lock} = i|C)$$

In order to obtain the latter one (noted $P(\tilde{N}_{R,lock} \geq 1|AR)$), we first obtain the probability that a remotely aborting transaction requests i locks on a remote node:

$$P(\tilde{N}_{R,lock} = i|AR) = \frac{\sum_{j=0}^{N_w - i} P_L^{\dagger}(j,i,N_w) P_A^{\dagger}(j,0,N_w)}{P_{AR}}$$

Note that this probability differs from Eq. 3, as here we do not require that a transaction successfully acquires all the locks it requests on a node. Then, similarly to the committing transaction's case, we have:

$$P(\tilde{N}_{R,lock}^R \geq 1|AR) = \sum_{i=1}^{N_w} P(\tilde{N}_{R,lock} = i|AR)$$

Hence, we compute the arrival rates of remote transactions to a node as:

$$\lambda_{RC}^{UP} = \lambda_{Rtx}^{UP} P(N_{R,locks}^R \geq 1|C), \quad \lambda_{RA}^{UP} = \lambda_{Rtx}^{UP} N_{AR} P(N_{R,locks}^R \geq 1|AR)$$

Finally, we compute the average arrival rate or remote requests, each having a CPU demand equal to D_R^{RG}:

$$\lambda_G^R = \frac{1 - \mathscr{L}}{N-1} \left(\lambda_L^{RO} N_r + \lambda_{Rtx}^{UP} N_R (N_{AR} + N_{AL}(N_{AR}+1)) \right)$$

B. Machine Learning-based Model of Network Latency

Developing a white box network communication model capable of accurately predicting the response time of network-bound operations in complex applications, deployed over virtualized cloud infrastructures is a very challenging task. First, in this type of infrastructures, little or no knowledge is available about the underlying network topology, hardware infrastructure and virtualization software overhead: this affects the possibility of measuring resource demands accurately, and makes the analytical derivation of response times cumbersome [34]. Moreover, complex applications' software stack typically lies on top of group communication toolkits that provide several inter-process synchronization services (like failure detection, group membership, remote procedure calls) the configuration and the internal design of this layer also affect performance in a way that is hard to predict [35].

For these reasons, *PROMPT* relies on ML to predict the latency of network-bound operations, i.e., of the remote get (R_R^{γ}), prepare (R^{prep}), and final decision phases (R^{dec}). As we shall see in Sec. V, after experimenting with several ML tools we opted for employing Cubist[1], a Decision Tree (DT) regressor that approximates multivariate functions by means of piece-wise linear approximations.

In order to build an initial knowledge base to train the machine learner, *PROMPT* relies on a suite of synthetic benchmarks that generate heterogeneous transactional workloads in terms of mean size of messages, CPU utilization and network load. The set of input features that we provide as input to Cubist characterizes the workload from the point of view of network utilization, as dynamics relevant to data contention are captured by the white box model that we described in the previous section. Specifically, for each of the three predicted network latencies we build an independent ML-based model, based on the following features: number of nodes in the system, average number of nodes contacted during the 2PC, average size of the messages exchanged in remote interactions (i.e., prepare and remote gets), the rate at which these interactions occur, CPU utilization, number of active threads on each node. Moreover, given that Cubist only exploits linear approximations in the leaves of the decision tree, we widen the set of input features by providing also metrics that are obtained as product of these basic features and that we know from analytical network communication models to be highly correlated with the response time of network-bound operations (e.g., the throughput, in byte per second, of sent and received messages) [13,21].

The training set can be built by gathering measurements of the above metrics while deploying the benchmark using different values for the platform scale and the data replication degree. However, at query time, some of the features are not known, as they depend on the (sought after) performance of the application in the target configuration (e.g., prepares per second). We address this problem by exploiting the availability of a complementary white box model of the system's performance to obtain an estimate of the value of such features in the target configuration. Only thanks to this coupling the ML is able to deliver high accuracy in its prediction: in fact, the AM is able to provide in input to the ML features that are highly correlated with the output and that may be impossible to obtain from direct measurements.

C. Model Resolution

From the analysis carried out in previous sections, it is clear that there are some interdependencies among the CPU, network and data contention models: the CPU and network response times are influenced by the lock contention probability, which depends on the lock hold times, which, on their turn, depend on network and CPU response times. This cyclic dependency is solved through a fixed point recursion on the value of P_{lock}. On the first iteration, it is set to 0; the value in input at iteration i is computed by applying Eq. 2 to metrics obtained at iteration $i-1$; this process ends when the relative difference between values computed on two consecutive iterations falls under a threshold (1% in our model) and typically converges in a few iterations. It is out of the scope of this paper to demonstrate the convergence of this iterative solution method, which has been adopted to solve several previous performance models of concurrency control protocols (see, e.g, [15,19,36]); as in previous studies, we have empirically observed that it always converges in a few iterations, provided that the input assignment defines a stable system.

[1] https://www.rulequest.com/cubist-info.html

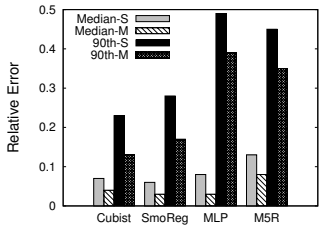
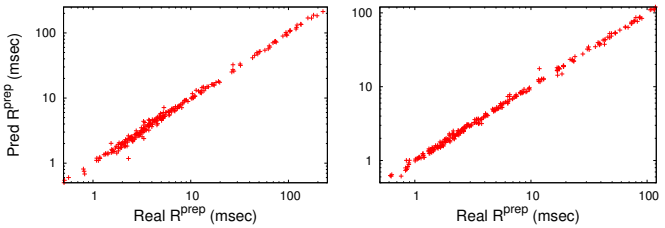


Fig. 1: Single and multi-model validation for different MLs



(a) Private Cloud (b) FutureGrid

Fig. 2: R^{prep} predictions on different virtualized infrastructures

The AM and the ML models are similarly intertwined. The AM takes as input parameters the response time of network-bound operations. These are output by the ML, which in turn requires the estimate of several features by the AM itself. This dependency is broken in the following way. At the beginning of each step of the aforementioned fixed point recursive scheme, the AM computes the values needed by the ML, as a function of the input abort probability. Then, taking such values as features, the ML outputs the predictions for network-bound operations' response time, which are finally exploited by the AM to compute the transactions' response time.

D. Predicted KPIs

We have shown so far how the *PROMPT*' performance models are able to predict transactions' response time, abort probability and latency of network-bound operations. We now show how the model can be also exploited to obtain an approximation for the closed-system throughput, i.e., the throughput delivered by the application when deployed over a set of N nodes having each θ active threads that process transactions with zero think time.

This metric can be computed by exploiting Little's law [37] in an iterative fashion: at each iteration, the closed-system throughput is obtained starting from the average transactions' response time [15,20]. This process typically completes in a few iterative steps, except for high contention scenarios, which may yield to convergence problems. This is a typical issue that arises when adopting such a recursive resolution algorithm for analytical models of transactional systems [15]. To cope with such an issue, *PROMPT* implements a fallback solving algorithm that spans, at a given granularity (10 tx/sec in our settings), all possible arrival rate values within a configurable interval. This algorithm returns the solution which minimizes the error between the input arrival rate an the output closed system throughput. This guarantees convergence to the desired accuracy (1% in our case) in a bounded number of steps.

V. Evaluation

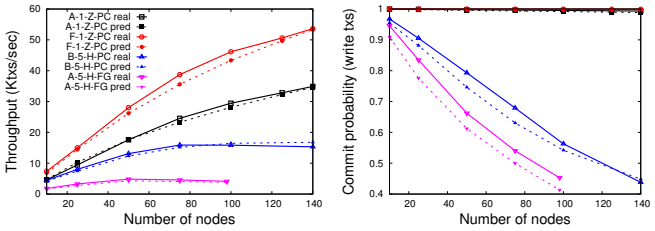
In this section we report the results of an experimental study aimed at evaluating the accuracy of *PROMPT*. Before presenting the results, we describe the workloads and the experimental platforms that we used in the study.

Experimental Platforms. The experimental test-bed for our study consists of a private and a public cloud infrastructure. The Virtual Machines (VMs) deployed over both clouds are equipped with 1 Virtual CPU (VCPU) and 2GBs of RAM; each VM runs a Fedora 17 Linux distribution with 3.3.4-5.fc17.x86_64 kernel. The private cloud consists of 140 VMs deployed over a cluster composed by 18 physical servers equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) processors and 32 GB of RAM and interconnected via a private Gigabit Ethernet. The employed virtualization software is Openstack Folsom. The public cloud consists of 100 VMs, deployed over the FutureGrid India infrastructure [38,39], which exploits the Openstack Havana virtualization software. The validation is performed using Infinispan 5.2.

Workloads. To validate the proposed model, we rely on YCSB [40], the *de facto* standard benchmark for key-value stores. In order to stress both the white box and black box models of *PROMPT*, we encapsulate the logic of the YCSB workloads into transactions, and focus our study on YSCB workloads that contain mixes of read-only and update transactions. Specifically, we consider, as baselines, YCSB workloads A, B and F: workload A has a mix of 50/50 reads and writes; workload B contains a 95/5 reads/update mix; in workload F records are first read and then modified within a transaction. In order to evaluate the accuracy of *PROMPT* in predicting a wider set of transactional workloads, we included, in our evaluation, variants of the aforementioned workloads, in which we vary the number of performed operations.

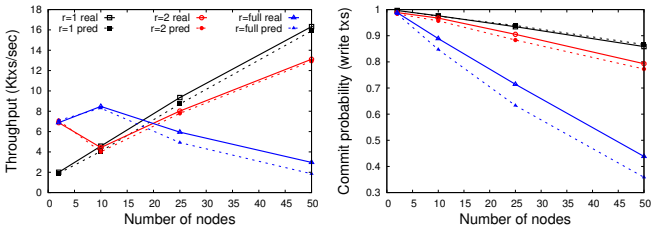
Finally, we consider two data access patterns: zipfian and hot-spot. In the first one, the popularity of data items follows a zipfian distribution (with YCSB's *zipfian constant* set to 0.7); in the second one, 99% of the data requests are issued against the 1% of the whole data set. We will refer to a workload using the notation N-D-P-I, where: N refers to the original workload's YCSB notation [40]; D is the number of distinct data items that are read by a read-only transaction; for update transactions, it is the number of distinct data items that are written (for the F workload, which exhibits a read-modify-write pattern, data are both read and written); P encodes the data access pattern (Z stands for zipfian, H for hot-spot); finally, I specifies the cloud infrastructure over which the benchmark has been run (PC stands for private cloud, FG for FutureGrid). A workload generator is deployed on each node and consists of one thread that injects requests against the collocated Infinispan instance, in closed loop.

For all the experiments, the data platform is populated with 500000 keys and data are scattered across nodes according to Infinispan's default consistent hash function. This choice results in setting $P_O = \frac{1}{N}$ in the model, with N being the number of nodes in the system.



(a) Throughput (b) Commit probability (write txs)

Fig. 3: *PROMPT*'s accuracy for different workloads (r = 2)



(a) Throughput (b) Commit probability (write txs)

Fig. 4: *PROMPT*'s accuracy while varying r (B-5-H-PC)

ML choice and validation. Previous solutions relying on black box modeling of the network interactions in distributed data platforms only cope with the case of full data replication [20,35]. Extending these solutions to encompass also partial replication was not straightforward. In fact, the replication degree affects other input features for the ML model in a strongly non-linear fashion. For instance, for small values of the replication degree, small shifts of this parameter typically lead to large changes of the arrival rate of the remote gets. Conversely, when close to full replication, the remote gets' arrival rate drops slowly to 0.

To avoid deriving a single ML-based model over a single strongly non-linear dataset, we partition the training set so as to isolate samples in the parameters' space corresponding to the case of "extreme" replication policies, i.e., full replication and data partitioning (corresponding to replication degree N and 1). This solution is related in spirit to the well known technique of boosting (and to *leveraging algorithms* in general) [41], which consists into iteratively combining several *weak* learners to obtain a *stronger* learner. In our solution, the weak learners are trained on disjoint training sets, in order to increase their predictive power in a reduced portion of the parameters' space, in which the relations between input and output are subject to more linear, and hence more easily deductible, dynamics. In order to assess the impact of the proposed approach with a wide set of machine learners, we tested it with Cubist and with several other ML techniques included in the Weka [42] suite, namely MultiLayerPerceptron (based on Neural Networks), SmoReg (based on Support Vector Machines) and M5Rules (a DT regressor like Cubist) [41].

The results of our tests are shown in Fig. 1, which reports the accuracy in terms of median and 90-th percentile of the prediction for the R^{prep} feature for the private cloud deployment, when using the single model (S) or the "multi-model" obtained by training the learners on disjoint data-sets (M). We do not present the results relevant to impact of this technique on the accuracy of the models for other features (e.g., R_R^{γ}) and for the FutureGrid infrastructure, as they are similar to the presented ones. In Fig. 2, we show instead the scatter plots contrasting the predicted vs the actual values of R^{prep} feature, on our private cloud and on FutureGrid, obtained using 10 folds cross-validation.

Global validation. We now show the accuracy achievable by *PROMPT* when exploiting jointly its analytical and ML-based models. Specifically, we will assess the accuracy of *PROMPT* in predicting the closed-system throughput and the abort rate of the aforementioned YCSB workloads while varying the number of nodes in the system, the replication degree and the underlying cloud infrastructure.

We show, in Fig. 3, the accuracy of *PROMPT* in predicting the performance achieved with workloads characterized by very different scalability trends (with a replication degree fixed to 2). The model is able to predict that workloads A/F-1-Z-PC are network bound, and that their abort probability is negligible. Analogously, *PROMPT* is able to predict that data contention is what limits the scalability of workloads B-5-H-PC and A-5-H-FG, leading the system to thrash as the degree of concurrency in the system grows.

Next, we evaluate how *PROMPT* is able to predict the performance of an application when changing replication degree and scale. To this end we show in Fig. 4 the throughput and commit probability for workload F-5-H-PC while varying the scale and the replication degree. It is possible to see that a higher replication degree is better for small deployments (up to 15 nodes): in fact, though it yields, on average, a higher number of nodes to be contacted at prepare time, it reduces (or eliminates, in the case of full replication) the generation rate of remote get operations, whose cost is dominant at small scales for this workload. As the size of the system grows, however, the latency of the distributed commit phase at high replication degree becomes the dominant cost, and yields higher lock hold times that result into higher data contention; hence, a lower replication degree yields better performance. As shown in Fig. 4, *PROMPT* is able to quantitatively capture the effect of the shift of replication degree and scale on performance, suggesting the viability of our proposal not only for resource provisioning, but also for the self-tuning of distributed transactional in-memory platforms.

Next, we compare our proposal with approaches relying purely on ML. We select as competitors the same ML tools that we experimented with when building the black box network model. We define a training and a test set: the former consists of the same knowledge base that we use to build the network model of *PROMPT*; the latter is composed by the considered YCSB workloads. We compare the accuracy of *PROMPT* over all the workloads in the test set with the accuracy of the ML techniques when progressively trained with additional, randomly selected, portions of the training set. Specifically we iteratively select at random the 20%, 40%, 60% and 80% of the test set, we remove the relevant samples from it, and we add them to the training set of the MLs (but not of *PROMPT*). The MLs are, hence, trained with the updated training set and they are evaluated over the remaining test set.

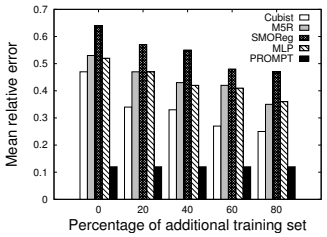


Fig. 5: *PROMPT* vs pure ML approaches

Fig. 5 reports the result of this evaluation, where each bar is the average of ten runs. As expectable, the mean relative error of the MLs decreases as more samples are added to the training set. However, the accuracy of *PROMPT* with the original training set is still twice as high as the one achieved by the best performing learner trained with the 80% of additional training set (12% vs 25%). This highlights that the hybrid modelling technique employed in *PROMPT* can produce reliable predictions requiring a smaller training set (and hence a lower training time) than pure ML-based approaches, even outperforming them in terms of accuracy.

VI. Conclusion

In this paper, we have proposed *PROMPT*, a performance forecasting model for partially replicated in-memory transactional stores. By relying on the joint usage of Analytical Modelling and Machine Learning, *PROMPT* achieves high accuracy also with applications exhibiting diverse scalability trends, as well as portability across heterogeneous virtualized infrastructures. We assessed *PROMPT*'s accuracy through an extensive experimental evaluation based on the YCSB benchmark, and using both private and public Cloud infrastructures.

References

[1] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rew.*, vol. 44, 2010.

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

[3] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

[4] J. Baker *et al.*, "Megastore: Providing scalable, highly available storage for interactive services," in *Proc. of CIDR*, 2011.

[5] S. Peluso *et al.*, "When scalability meets consistency: Genuine multi-version update-serializable partial data replication," in *Proc. of ICDCS*, 2012.

[6] S. Hirve *et al.*, "Hipertm: High performance, fault-tolerant transactional memory." in *Proc. of ICDCN*, 2014.

[7] J. C. Corbett *et al.*, "Spanner: Google's globally-distributed database," in *Proc. of OSDI*, 2012.

[8] B. Mozafari *et al.*, "Performance and resource modeling in highly-concurrent oltp workloads," in *Proc. of SIGMOD*, 2013.

[9] S. Elnikety *et al.*, "Predicting replicated database scalability from standalone database profiling," in *Proc. of Eurosys*, 2009.

[10] D. Didona *et al.*, "Identifying the optimal level of parallelism in transactional memory applications," *Springer Computing*, 2013.

[11] S. Ghanbari *et al.*, "Adaptive learning of metric correlations for temperature-aware database provisioning," in *Proc. of ICAC*, 2007.

[12] D. Karger *et al.*, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *Proc. of STOC*, 1997.

[13] M. Nicola and M. Jarke, "Performance modeling of distributed and replicated databases," *IEEE TKDE*, 2000.

[14] P. di Sanzo *et al.*, "A performance model of multi-version concurrency control," in *Proc. of MASCOTS*, 2008.

[15] P. S. Yu *et al.*, "On the analytical modeling of database concurrency control," *J. ACM*, vol. 40, 1993.

[16] A. Thomasian, "A more realistic locking model and its analysis," *Information Systems*, vol. 21, no. 5, pp. 409 – 430, 1996.

[17] Y. C. Tay *et al.*, "Locking performance in centralized databases," *ACM TODS*, vol. 10, 1985.

[18] ——, "A mean value performance model for locking in databases: The no-waiting case," *J. ACM*, vol. 32, no. 3, pp. 618–651, Jul. 1985.

[19] D. A. Menascé and T. Nakanishi, "Performance evaluation of a two-phase commit based protocol for ddbs," in *Proc. of PODS*, 1982.

[20] D. Didona *et al.*, "Transactional auto scaler: Elastic scaling of replicated in-memory transactional data grids," *ACM TAAS*, to appear, 2014.

[21] A. Raghuram *et al.*, "Approximation for the mean value performance of locking algorithms for distributed database systems: A partitioned database," *Ann. Oper. Res.*, vol. 36, no. 1-4, pp. 299–346, May 1992.

[22] B. Ciciani *et al.*, "Analysis of replication in distributed database systems," *IEEE TKDE*, vol. 2, no. 2, 1990.

[23] E. Coffman *et al.*, "Optimization of the number of copies in a distributed data base," *IEEE TOSE*, vol. 7, no. 1, 1981.

[24] A. Singhal *et al.*, "An analysis of the effect of network load and topology on the performance of a concurrency control algorithm in distributed database systems," in *Proc. of CNS*, 1984.

[25] B. Urgaonkar *et al.*, "Agile dynamic provisioning of multi-tier internet applications," *ACM TAAS*, vol. 3, no. 1, pp. 1:1–1:39, Mar. 2008.

[26] M. Bennani and D. Menascé, "Resource allocation for autonomic data centers using analytic performance models," in *Proc. of ICAC*, 2005.

[27] D. A. Menascé *et al.*, "Preserving qos of e-commerce sites through self-tuning: A performance model approach," in *Proc. of EC*, 2001.

[28] D. A. Menascé and M. N. Bennani, "Autonomic virtualized environments," in *Proc. of ICAS*, 2006.

[29] B. Trushkowsky *et al.*, "The scads director: scaling a distributed storage system under stringent performance requirements," in *Proc. of FAST*, 2011.

[30] Basho, "Riak," http://redis.io/, 2014.

[31] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, 1981.

[32] Y. C. Tay, *Analytical Performance Modeling for Computer Systems, Second Edition*. Morgan & Claypool Publishers, 2013.

[33] L. Kleinrock, *Queueing Systems*. Wiley Interscience, 1975, vol. I: Theory.

[34] J. Whiteaker *et al.*, "Explaining packet delays under virtualization," *SIGCOMM Comp. Comm. Rev.*, vol. 41, no. 1, pp. 38–44, Jan. 2011.

[35] M. Couceiro *et al.*, "A machine learning approach to performance prediction of total order broadcast protocols," in *SASO*, 2010.

[36] P. di Sanzo *et al.*, "Analytical modeling of lock-based concurrency control with arbitrary transaction data access patterns," in *Proc. of ICPE*, 2010.

[37] J. D. C. Little, "A proof for the queuing formula: L= λ w," *Operations Research*, vol. 9, no. 3, 1961.

[38] G. C. Fox *et al.*, *FutureGrid: A Reconfigurable Testbed for Cloud, HPC and Grid Computing*. Chapman & Hall, 2012.

[39] G. von Laszewski *et al.*, "Design of the futuregrid experiment management framework," in *Proc. of GCE*, 2010.

[40] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with ycsb," in *Proc. of SOCC*, 2010.

[41] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, 2007.

[42] M. Hall *et al.*, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.