Transactional Auto Scaler: Elastic Scaling of Replicated In-Memory Transactional Data Grids

DIEGO DIDONA and PAOLO ROMANO, INESC-ID/Instituto Superior Técnico, Universidade de Lisboa
SEBASTIANO PELUSO, Sapienza, Universitá di Roma/Instituto Superior Técnico, Universidade de Lisboa
FRANCESCO QUAGLIA, Sapienza, Universitá di Roma

In this article, we introduce TAS (Transactional Auto Scaler), a system for automating the elastic scaling of replicated in-memory transactional data grids, such as NoSQL data stores or Distributed Transactional Memories. Applications of TAS range from online self-optimization of in-production applications to the automatic generation of QoS/cost-driven elastic scaling policies, as well as to support for what-if analysis on the scalability of transactional applications.

In this article, we present the key innovation at the core of TAS, namely, a novel performance forecasting methodology that relies on the joint usage of analytical modeling and machine learning. By exploiting these two classically competing approaches in a synergic fashion, TAS achieves the best of the two worlds, namely, high extrapolation power and good accuracy, even when faced with complex workloads deployed over public cloud infrastructures.

We demonstrate the accuracy and feasibility of TAS's performance forecasting methodology via an extensive experimental study based on a fully fledged prototype implementation integrated with a popular open-source in-memory transactional data grid (Red Hat's Infinispan) and industry-standard benchmarks generating a breadth of heterogeneous workloads.

Categories and Subject Descriptors: C.4 [**Modelling Techniques**]; H.2.4 [**Systems**]: Distributed Databases, Transaction Processing

General Terms: Performance

Additional Key Words and Phrases: Transactional data grids, performance forecasting, elastic scaling, queueing theory, machine learning

ACM Reference Format:
Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. 2014. Transactional auto scaler: Elastic scaling of replicated in-memory transactional data grids. ACM Trans. Auton. Adapt. Syst. 9, 2, Article 11 (May 2014), 32 pages.
DOI: http://dx.doi.org.10.1145/2620001

This work has been partially supported by the project Cloud-TM (cofinanced by the European Commission through the contract no. 257784), by the FCT projects ARISTOS (PTDC/EIA/102496/2008) and specSTM (PTDC/EIA-EIA/122785/2010), and by national funds through FCT - Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2013 and by COST Action IC1001 EuroTM.
Authors' addresses: Diego Didona and Paolo Romano, INESC-ID, Rua Alves Redol 9, Lisbon, Portugal; emails: didona@gsd.inesc.id; romano@inesc-id.pt; Sebastiano Peluso and Francesco Quaglia, DIIAG, Sapienza, Via Ariosto 25, 00185, Rome, Italy; emails: peluso@dis.uniroma1.it; quaglia@dis.uniroma1.it.

DOI: http://dx.doi.org.10.1145/2620001

1. INTRODUCTION

The advent of commercial cloud computing platforms has led to the proliferation of a new generation of in-memory, transactional data platforms, often referred to as NoSQL data grids. This new breed of distributed transactional platform (which includes products such as Red Hat's Infinispan [Marchioni and Surtani 2012], Oracle's Coherence [Oracle 2011], and Apache Cassandra [Lakshman and Malik 2010]) is designed from the ground up to meet the elasticity requirements imposed by the pay-as-you-go cost model characterizing cloud infrastructures. These platforms adopt a number of innovative mechanisms precisely aimed at allowing the efficient resizing of the set of nodes over which they are deployed, such as simple data models (e.g., key value, as opposed to the conventional relational model) and efficient mechanisms to achieve data durability (e.g., replication of in-memory data, as opposed to synchronous disk logging).

By allowing nonexpert users to provision a data grid of virtually any size within minutes, cloud computing infrastructures give tremendous power to the average user, while also placing a major burden on the user's shoulders. Removing the classic capacity planning process from the loop means, in fact, shifting the nontrivial responsibility of determining a good-size configuration to the nonexpert user [Shen et al. 2011].

Unfortunately, forecasting the scalability trends of real-life, complex applications deployed on distributed transactional platforms is an extremely challenging task. As the number of nodes in the system grows, in fact, the performance of these platforms definitely exhibits nonlinear behaviors. Such behaviors are imputable to the simultaneous, and often interdependent, effects of contention affecting both physical (computational, memory, network) and logical (conflicting data accesses by concurrent transactions) resources.

These effects are clearly shown in Figures 1 and 2, where we report results obtained by running two transactional benchmarks (Radargun[1] and TPC-C[2]) on a popular replicated in-memory transactional data grid, namely, Red Hat's Infinispan [Marchioni and Surtani 2012] (which will be used later on in the article to assess the accuracy of the proposed solution, and whose architecture will be described in Section 3.1). We deployed Infinispan over a private cluster encompassing a variable number of nodes and ran benchmarks generating heterogeneous workloads for what concerns the number of (read/write) operations executed within each transaction, the percentage of read-only transactions, the number of items in the whole dataset, and the size of the individual objects manipulated by each operation.

As shown in Figure 1, the scalability trends of the three considered workloads (in terms of the maximum throughput) are quite heterogeneous. The TPC-C benchmark scales almost linearly and the plots in Figures 2(a) and 2(b) show that its scalability is limited by a steady increase of contention at both network and data (i.e., lock) levels. This leads to a corresponding increase of network round-trip time (RTT) and transaction abort probability. On the other hand, the two Radargun workloads clearly demonstrate how the effects of high contention on logical and physical resources can lead to nonlinear scalability trends, even though, as in the case of accesses to a small dataset ("RG - Small"), the performance degradation of the network layer (in terms of RTT) is not so relevant.

In this article, we present Transactional Auto Scaler (TAS), a system that introduces a novel performance prediction methodology based on the joint usage of analytical and machine-learning (statistical) models. TAS relies on black-box, machine-learning (ML) methods to forecast fluctuation of the network latencies imputable to variations

[1]https://github.com/radargun/radargun/wiki.
[2]http://www.tpc.org/tpcc.

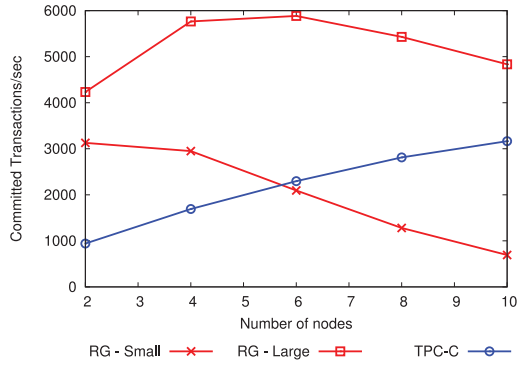
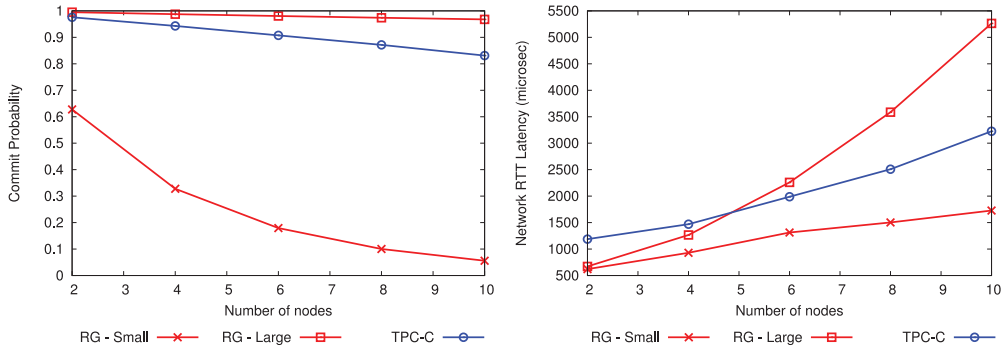


Fig. 1. Performance of different data grid applications.



(a) Transaction Commit Probability (b) Round-Trip Network Latency

Fig. 2. Scalability analysis of different data grid applications.

of the system's scale. The analytical model (AM) employed by TAS serves a twofold purpose: on one hand, it exploits the knowledge on the dynamics of the concurrency control/replication algorithm to forecast the effects of data contention using a white-box approach; on the other hand, it allows capturing the effects of CPU contention on the duration of the various processing activities entailed by the transactions' execution.

Using AM and ML techniques in synergy allows TAS to take the best of these two, typically competing, worlds. On the one hand, the black-box nature of ML spares from the burden of modeling explicitly the dynamics of the network layer. This not only is a time-consuming and error-prone task given the complexity and heterogeneity of existing network architectures but also would constrain the portability of our system (to a specific instance of infrastructure), as well as its practical viability in virtualized cloud environments where users have little or no knowledge of the underlying network topology/infrastructure.

On the other hand, analytical modeling allows one to address two well-known drawbacks of ML, namely, its limited extrapolation power (i.e., the ability to predict scenarios that have not been previously observed) and lengthy training phase [Bishop 2007]. By exploiting *a priori* knowledge on the dynamics of data consistency mechanisms, AMs can achieve good forecasting accuracy even when operating in still unexplored regions of the parameters' space. Further, by narrowing the scope of the problem tackled via ML techniques, AM allows one to reduce the dimensionality of the ML input features' space, leading to a consequent reduction of the training phase duration [Bishop 2007].

The hybrid AM/ML methodology proposed in this article can be adopted with a plethora of alternative replication/concurrency control mechanisms. In this article, we instantiate it, and demonstrate its effectiveness, by developing performance prediction models for two data management protocols supported by the Infinispan data grid: a multimaster replication protocol that relies on Two-Phase Commit (2PC) to detect distributed data conflicts and ensure atomicity and a Primary-Backup (PB) replication protocol that allows the execution of update transactions on a single node and regulates concurrency via an encounter-time locking strategy [Bernstein et al. 1986].

One of the key innovative elements of the analytical performance models presented in this article consists of the methodology introduced to characterize the probability distribution of transactions' access to data items. Existing white-box models of transactional systems [di Sanzo et al. 2008; Ciciani et al. 1990; Tay et al. 1985; Elnikety et al. 2009], in fact, rely on strong approximations on the data accesses distribution (e.g., uniformly distributed accesses on one or more sets of data items of fixed cardinality) that are hardly met in complex, real-life applications. Further, models relying on realistic distribution of the access pattern would require complex and time-consuming workload characterization studies in order to derive the parameters featuring the actual distribution of data accesses. Conversely, in the presented model, we capture the dynamics of the application's data access pattern via a novel abstraction, which we call *Application Contention Factor* (ACF). It exploits queuing theory arguments and a series of lock-related statistics measured in (and dependent on) the current workload/system configuration to derive, in a totally automatic fashion, a probabilistic model of the application's data access pattern that is independent of the current level of parallelism (e.g., number of concurrently active threads/nodes) and utilization of physical resources (e.g., CPU or network).

We demonstrate the viability and high accuracy of the proposed solution via an extensive evaluation study using both a private cluster and public cloud infrastructures (Amazon EC2) and relying on industry standard benchmarks generating a breadth of heterogeneous workloads, which give rise to different contention levels for both logical and physical resources. The results also highlight that the overhead introduced by TAS's monitoring system is negligible, and that the time required to solve the performance forecasting model is on the order of at most a few hundreds of milliseconds on commodity hardware.

The remainder of this article is structured as follows. In Section 2, we discuss related work. The target data grid architecture for the TAS system is described in Section 3. In Section 4, we present the forecasting methodology that we integrated in TAS. In Section 5, we validate the methodology via an extensive experimental study. In Section 6, we provide an assessment of TAS's models resolution time and an analysis of the overhead introduced by the workload and performance monitoring framework. Finally, Section 7 concludes this article.

2. RELATED WORK

The present work is related to the literature on analytical performance modeling of transactional platforms. This includes performance models for traditional database systems and related concurrency control mechanisms (see, e.g., [Tay et al. 1985; Menascé and Nakanishi 1982; Yu et al. 1993]), approaches targeting Software Transactional Memory (STM) architectures (see, e.g., [di Sanzo et al. 2012]), and solutions tailored to distributed/replicated transaction processing systems, such as Ciciani et al. [1990]. Analogously to these works, TAS relies on analytical modeling techniques to predict the performance of concurrency control and replication protocols.

Literature analytical models somehow rely on the knowledge of the data access pattern exhibited by transactions. Particularly, existing white-box models either rely on

strong approximations on the data access probability or require a fine-grain characterization of the data access pattern. The first category includes models that assume data to be uniformly accessed on one or more datasets of fixed cardinality. The models in Yu et al. [1993] and Elnikety et al. [2009], for example, assume that each datum has the same probability of being accessed. In Tay et al. [1985], the b-c model is proposed, in which b% of the transactions uniformly access elements within the c% of the dataset, and (1-b%) of the accesses targets the remaining data. This model has been further extended in Zhang and Hsu [1995] and Thomasian [1998] in order to encompass several transactional classes, each accessing a disjoint portion of the whole dataset.

The second category includes models that are able to capture more complex data access patterns. In di Sanzo et al. [2008], the data access pattern is modeled via a generic function in order to be able to model complex, nonuniform data access patterns; the evaluation of the model is carried out exploiting several parameterizations of the zipf function. In di Sanzo et al. [2010], the characterization of the data access pattern allows one to attribute different probabilities of accessing a set of data depending on the phase of execution of the transaction. This information is encoded by means of a matrix A, such that A(i,j) specifies the probability for a transaction to access datum j at operation i. Unfortunately, the usage of analytical models based on a detailed characterization of the data access distribution demands the adoption of complex, time-consuming workload characterization techniques (in order to derive the parameters characterizing the distribution of data accesses). The instrumentation overhead required to trace (possibly online) the accesses of transactions to data and instantiate these data access models can represent a major impairment for the adoption of these techniques in realistic, performance-sensitive applications.

In the proposed analytical model, conversely, we capture the dynamics of the application's data access pattern via a novel abstraction, which we call Application Contention Factor (ACF). With respect to the aforementioned approaches, by relying on high-level lock-related statistics, the computation of ACF requires minimal profiling of the application and nonintrusive instrumentation of the platform. Moreover, the runtime computation of ACF can be totally automated and can be performed in a lightweight fashion. It allows one to obtain a concise probabilistic model of the application's data access pattern that is independent of the current scale of the system and utilization of physical resources (e.g., CPU or network).

Typical analytical models of distributed transactional platforms also introduce strong simplifications in the modeling of the network latencies, which play a relevant role in the synchronization phase among replicas. The network layer is modeled either as a fixed delay [Menascé and Nakanishi 1982; Ren et al. 1996] or via simple queuing models [Nicola and Jarke 2000]. Also in this case, existing solutions are too simplistic to model real deployment scenarios, such as virtualized/cloud environments, where little or no knowledge is available about the actual network topology, or platforms relying on complex software stacks (e.g., Java-based Group Communication Toolkits [Ban 2012]) providing a plethora of interprocess synchronization services (like failure detection, group membership, RPCs).

Conversely, as for the network layer, we rely on black-box ML techniques, which, as we will show in Section 4.2, allow achieving high accuracy both in private clusters and in public cloud deployments.

Our work also has a relationship with techniques for automatic resource provisioning, which, unlike TAS, do not model the effects of data contention on performance. Queuing networks are exploited within different approaches to analytically model the performance of complex systems. In Singh et al. [2010], a k-means data-clustering algorithm is employed to characterize the workload mix of a multitier application. A multiclass analytical model is then instantiated, in which each class models the

different arrival rate and resource demand of a data cluster. Each server of a tier is modeled as an open G/G/1 queue in order to obtain an upper bound on percentiles of response times. In Sharma et al. [2012], a multitier application is modeled as an open tandem network of M/G/1/PS queues and the service time distribution of requests is modeled as a mixture of K shifted exponentials; the number of exponentials and their parameters are obtained online by iteratively running a k-means data-clustering algorithm. In Zhang et al. [2007], a closed queuing model is considered and solved through the MVA algorithm; service demands for requests are obtained at runtime through regression. The work in Urgaonkar et al. [2005] proposes a general model for multitier applications capable of taking into account application idiosyncrasies such as caching effects, multiclass requests, and limits on the amount of concurrency at each tier.

As these techniques do not model the effects of data contention, they cannot straightforwardly be applied to the case of transactional data grids. In fact, as highlighted by the experimental results reported in Figures 1 and 2(a), logical contention between concurrent transactions can have a detrimental and nonnegligible impact on the performance and scalability of this type of systems.

State-of-the-art solutions to the problem of automatic resource provisioning also rely on different machine-learning techniques. In Chen et al. [2006], a K-nearest-neighbors machine-learning approach is exploited for the autonomic provisioning of a database. The training phase is performed offline on the target application generating a specific workload, while varying the load intensity and the number of replicas for the database. At runtime, the machine learner is queried in order to determine the number of database replicas needed to sustain a given load. The proposal in Ghanbari et al. [2007] is based on the exploitation of support vector machines (SVMs). During an offline training phase, SVMs are used to determine and select the features that are more correlated with respect to the output performance indicator (response time). Then, they are exploited as online learners: samples relevant to the selected features are taken at runtime; whenever the value predicted by the machine learner is different from the one actually measured, a new sample, representative of that scenario, is integrated in the SVM's knowledge base. In Dutreilh et al. [2011], a Q-learning-based [Watkins and Dayan 1992] solution is proposed to provision server instances to a generic application. The reward for a given action (i.e., addition or removal of a server) is expressed as the sum of the cost for the acquisition of new instances and a penalty, proportional to the extent of the possible violation of the SLA.

Being based on pure machine learning, the effectiveness of the aforementioned proposals is hampered when the application generates a workload whose results are very dissimilar from those observed during the training phase. By restricting the employment of ML techniques exclusively to predict the network dynamics and offloading to an analytical model the forecasting of the effects of data contention, TAS drastically reduces the scope of the problem tackled via ML techniques. As we will also discuss while presenting the results of our experimental study, this design choice allows enhancing the robustness of the predictive model and reducing the duration of the ML training phase.

Control theory techniques are at the basis of several works in the area of self-tuning of application performance. These solutions rely on different performance models to instantiate the control loop that drives resources allocation. In Xu et al. [2007], fuzzy logic is exploited to determine how much CPU and I/O bandwidth to allocate to a virtualized database system. In Trushkowsky et al. [2011], a feed-forward control loop is employed, which relies on an offline built machine-learning-based model to determine, at runtime and depending on the workload mix, whether an eventually consistent distributed data store is provisioned with sufficient resources to meet the SLA. In Ali-Eldin et al. [2012a, 2012b], a cloud infrastructure as a whole is modeled as a G/G/N

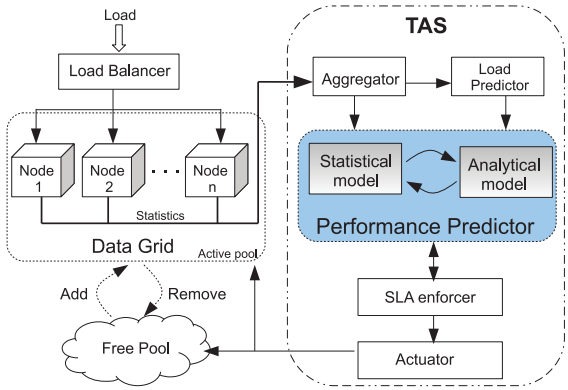


Fig. 3. TAS reference architecture.

queue and a load predictor filter is employed to build hybrid reactive/proactive controllers to orchestrate the provisioning of VMs. Compared to these models, which only focus on physical resource demands of applications, the ability of TAS to forecast both logical and physical contention can accurately capture the system's entire behavior and allows optimized resource allocation over the entire operating space.

3. SYSTEM ARCHITECTURE

The architecture of TAS is depicted in Figure 3. Incoming transactions are dispatched by a front-end load balancer toward the set of nodes composing the data grid. As already mentioned in the Introduction section, we consider two alternative replication protocols, based on 2PC and PB schemes, respectively. Given the different nature of these two protocols (multimaster vs. single master), depending on the considered scheme, the load balancer has to behave differently: routing update, resp. read-only, transactions exclusively toward the primary, resp. backup nodes, in case PB is being used, versus distributing uniformly requests across the platform's nodes if 2PC is in use. On a periodical basis, statistics concerning load and resource utilization across the set of nodes in the data grid are collected via a *distributed monitoring system*, whose implementation has been based on the Lattice Monitoring Framework [London's Global University 2013]. The latter has been extended in order not only to gather statistics relevant to the utilization of resources but also to collect measurements taken at the Infinispan level in order to obtain the current workload characterization [Palmieri et al. 2011]. Collected statistics are convoyed to a so-called *aggregator* module; aggregated statistics are then fed to the *load predictor*, which serves the twofold purpose of forecasting the future workload volume and its characteristics (e.g., read-only vs. update transactions ratio, average number of read/write operations for read-only/update transactions), as well as detecting relevant workload shifts. The current TAS prototype relies on the Kalman filter algorithm [Welch and Bishop 1995] for load forecasting and on the CUSUM [Dai et al. 2011] algorithm for distinguishing, in a robust manner, actual workload shifts from transient statistical fluctuations. Similar techniques have been employed in prior systems for automatic resource provisioning [Chen et al. 2006; Ghanbari et al. 2007] and have been shown to enhance the stability of the auto-scaling process.

Assessing the accuracy of the load forecasting techniques integrated in the TAS system is outside the scope of this article, which is instead focused on the key innovative contribution of TAS, namely, the methodology employed for predicting the performance of transactional applications when varying the platform's scale. More in detail, the *performance predictor* employed by TAS takes in input the workload characteristics

(as provided by the load predictor and/or aggregator) and the platform scale (i.e., number of nodes to be used by the data grid) and generates, in output, predictions on several key performance indicators (KPIs), including average response time, maximum sustainable throughput, and transaction abort probability. As shown in Figure 3, TAS relies on the joint usage of a white-box AM (to forecast the effects of data contention and CPU utilization) and black-box ML techniques (to forecast the effects of contention on the network layer). A detailed description of the proposed performance forecasting methodology will be provided in Section 4.

The component in charge of querying the performance predictor is the *SLA enforcer*, which identifies the optimal platform configuration (in terms of number of nodes) on the basis of user-specified SLA and cost constraints. Our current prototype supports elastic-scaling policies that take into account constraints on cost, average response time, and throughput. However, given that the performance predictor can forecast a number of additional KPIs (such as commit probability or response time of read-only vs. update transactions), our system lends itself to support more complex optimization policies involving constraints on additional performance metrics.

The *actuator* reconfigures the system by adding or removing (virtual) servers from the data grid. In order to maximize portability, TAS relies on δ-cloud,[3] an abstraction layer that exposes a uniform API to automate provisioning of resources from heterogeneous IaaS providers (such as Amazon EC2, OpenNebula, and RackSpace). The addition/removal of new nodes needs, of course, to be coordinated also at the data grid level (not only at the IaaS level). To this end, TAS assumes the availability of APIs to request the join/departure of nodes from the data grid, which is a feature commonly supported by modern NoSQL data stores. Finally, in the context of the Cloud-TM project,[4] TAS has been extended with a set of interfaces, both programmatic and web based, that maximize the ease of use for system administrators. The source code of TAS has been made publicly available on the software repository[5] of the Cloud-TM project.

3.1. Overview of the Reference Data Grid for TAS

As already mentioned, we selected as the target platform for TAS a popular open-source in-memory replicated transactional data grid, namely, Infinispan [Marchioni and Surtani 2012]. Infinispan is developed by JBoss/Red Hat and, at the moment of writing, represents the reference NoSQL data platform and clustering technology for JBoss AS, one of the most popular open-source J2EE application servers. Given that TAS employs a white-box analytical model for capturing the effects of data contention on a system's performance, in the following we provide an overview of the main mechanisms used by Infinispan to ensure transactional consistency.

Infinispan exposes a key-value store data model and maintains data entirely in-memory, relying on replication as its primary mechanism to ensure fault tolerance and data durability. This design choice allows one to remove logging to stable storage from the critical path of execution of transactions, as well as to adopt optimized strategies (e.g., asynchronous writes [Perez-Sorrosal et al. 2011], batching [Baker et al. 2011], data de-duplication [Zhu et al. 2008]) aimed at minimizing the costs associated with the usage of cloud storage systems (typically offered as an additional pay-per-use service [Amazon 2013] by IaaS infrastructures). Therefore, in this article, we do not model interactions with persistent storage systems and focus on capturing the costs associated with transaction management in in-memory replicated data grids, namely, local concurrency control and interreplica synchronization.

[3] δCloud, http://http://deltacloud.apache.org/.
[4] Cloud-TM: http://www.cloudtm.eu.
[5] http://www.github.com/cloudtm.

As other recent NoSQL platforms, Infinispan opts for sacrificing consistency in order to maximize performance. Specifically, it does not ensure serializability [Bernstein et al. 1986], but only guarantees the Repeatable Read ANSI/ISO isolation level [Berenson et al. 1995]. More in detail, Infinispan implements a nonserializable variant of the multiversion concurrency control algorithm, which never blocks or aborts any transaction upon a read operation, and relies on an encounter-time locking strategy to detect write-write conflicts. Write locks are always first acquired locally during the transaction execution phase, which does not entail any interaction with remote nodes. Instead, the commit protocol depends on the selected replication scheme.

In the PB scheme, only one node, namely, the primary, is entitled to serve update transactions; all the other nodes, namely, the backups, are allowed to process exclusively read-only transactions. Thus, the aforementioned local concurrency control algorithm is enough to guarantee the isolation property of the transactions, not requiring any distributed coordination at commit time. Conversely, in order to commit an update transaction, the primary broadcasts to the replicas the modifications locally performed by the transaction; then, in order to enforce global consistency, it waits until all the replicas have modified their state before notifying to the application the successful commitment of the transaction.

In the 2PC scheme, during the first phase of the commit protocol (also called the prepare phase), lock acquisition is attempted at all the replicas, in order to detect conflicts with transactions concurrently executing on other nodes, as well as for guaranteeing transaction atomicity. If the lock acquisition phase is successful on all nodes, the transaction originator broadcasts a commit message, in order to apply the transaction's modifications on the remote nodes, and then it commits locally.

In the presence of conflicting, concurrent transactions, however, the lock acquisition phase (taking place either during the local transaction execution or during the prepare phase) may fail due to the occurrence of (possibly distributed) deadlocks. Deadlocks are detected using a simple, user-tunable, timeout-based approach. In this article, we consider the scenario in which the timeout on deadlock detection is set to 0, which is a typical approach for state-of-the-art transactional memories [Dice et al. 2006] to achieve deadlock freedom. We also note that these platforms usually avoid relying on complex (costly) distributed deadlock detection schemes, thus preferring simple aggressive timeout approaches to tackle distributed deadlocks, which represent a major threat to system scalability, as highlighted by the seminal work in Gray et al. [1996] and confirmed by our experimental results.

4. PERFORMANCE PREDICTOR

This section describes the performance prediction methodology employed by TAS. We start in Section 4.1 by introducing methodologies, concepts, and equations that apply to the modeling of both 2PC and PB. In Sections 4.1.2 and 4.1.3, we present the analytical models used to capture data contention among transactions respectively in the PB and the 2PC schemes. In Section 4.1.4, we present the analytical model used to capture contention on the CPU. Next, in Section 4.2, we present the machine-learning-based approach used to forecast network latencies. Finally, we describe how to couple AM and ML approaches in Section 4.3.

4.1. Analytical Model

Our analytical model uses mean-value analysis techniques to forecast the probability of transaction commit, the mean transaction duration, and the maximum system throughput. This allows one to support what-if analysis on parameters such as the degree of parallelism (number of nodes and possibly number of threads) in the system

or shifts of workload characteristics, such as changes of the transactions' data access patterns or of the percentage of read versus write transactions.

The model treats the number of nodes in the system (denoted as ν) and the number of threads processing transactions at each node (denoted as θ) as input parameters. For the sake of simplicity, we will assume these nodes to be homogeneous in terms of computational power and RAM and distinguish only two classes of transactions, namely, read-only and update transactions. A discussion on how to extend the model and relax these assumptions will be provided in Section 4.1.6.

We denote with λ_{Tx} the mean arrival rate of transactions, and with w the percentage of update transactions, which perform, on average, a number N_l of write operations before requesting to commit. Note that, at this abstraction level, any operation that updates the state of the key-value store (e.g., put or remove operations) is modeled as a write operation. We say that a transaction is "local" to a node if it was activated on that node. Otherwise, we say that it is "remote."

Our analytical model is aimed at forecasting lock contention dynamics. The key idea, exploited for both 2PC and PB, is to model data items as M/G/1 queues, such that their average utilization can be used to approximate the lock contention probability (namely, the probability that a transaction attempts to access an item that is currently being locked by another transaction). The focus of the model is on update transactions since, in the considered concurrency control schemes, read-only transactions do not acquire locks, and hence they do not require an explicit data contention model. This also means that read operations are never blocked and can never induce any abort (see Section 3.1).

We denote with $T_{localRO}$, resp. $T_{localWR}$, the average time to execute a read-only, resp. update, transaction, since its beginning till the time in which it requests to commit, assuming that it does not abort earlier due to lock contention (in the case it is a write transaction).

We denote with T_{prep} the mean time for the transaction coordinator to complete the first part of the atomic commitment phase of 2PC, which includes broadcasting the prepare message, acquiring locks at all the replicas, and gathering their replies. For PB, we denote with T_{pc} the mean time that the primary spends waiting for all the replicas to reply. Note that the value of T_{prep} and T_{pc} can vary as the system scale changes, as an effect of the shift of the level of contention on physical resources (network *in primis*, but also CPU and memory). Given that these phenomena are captured in TAS via machine-learning techniques (described in Section 4.2), the analytical model treats these quantities as input parameters.

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

4.1.1. Data Access Pattern Characterization. In order to compute the transaction response time, we need first to obtain the probability that it experiences lock contention, that is, whether it requires a lock currently held by another transaction. As already mentioned, in the PB scheme, only local lock contention is possible, whereas in the 2PC scheme, a transaction can also experience lock contention on remote nodes. For both schemes, however, the modeled concurrency control algorithm entails that any lock contention leads to an abort of the transaction, hence the probability of lock contention, P_{lock}, and of transaction abort, P_a, coincide.

As in other AMs capturing locking phenomena [Yu et al. 1993; di Sanzo et al. 2012], in order to derive the lock contention probability, we model each data item as a server that receives locking requests at an average rate λ_{lock}, and that takes an average time T_H before completing the "service of a lock request" (i.e., freeing the lock). This level of

abstraction allows us to approximate the probability of encountering lock contention upon issuing a write operation on a given data item with the utilization of the corresponding server (namely, the percentage of time the server is busy serving a lock request), which is computable as $U = \lambda_{lock} T_H$ [Kleinrock 1975] (assuming $\lambda_{lock} T_H < 1$).

The key innovative element of our AM is that it does not rely on any *a priori* knowledge about the probability of a write operation to access a specific datum. Existing techniques, in fact, assume uniformly distributed accesses on one [di Sanzo et al. 2012] or more [Tay et al. 1985] sets of data items of cardinality D (where D is assumed to be *a priori* known) and compute the probability of lock contention on any of the data items simply as:

$$P_{lock} = \frac{1}{D} \lambda_{lock} T_H. \tag{1}$$

Unfortunately, the requirement of information on the parameter D and the assumption on the uniformity of the data access patterns strongly limits the employment of these models in complex applications, especially if these exhibit dynamic shifts in the data access distribution. We overcome these limitations by introducing a powerful abstraction that allows the online characterization of the application data access pattern distribution in a lightweight and pragmatical manner. As hinted, we call this abstraction *Application Contention Factor* (ACF), and we define it as:

$$ACF = \frac{P_{lock}}{\lambda_{lock} T_H}. \tag{2}$$

The ACF has two attractive features that make it an ideal candidate to characterize the data access pattern of complex transactional applications:

(1) It is computable online, on the basis of the values of $P_{lock}, \lambda_{lock,}$ and T_H measured in the current platform configuration by exploiting Equation (1). Always by Equation (1), it is possible to see that $\frac{1}{ACF}$ can be alternatively interpreted as the size D of an "equivalent" dataset accessed with uniform probability. Here, equivalent means that, if the application had generated a uniform access pattern over a dataset of size $D = \frac{1}{ACF}$, it would have been incurred the same contention probability experienced during its actual execution (in which it generated arbitrary, nonuniform access patterns).
(2) As we will show in Section 5, even for applications with arbitrary, complex data access patterns (such as in TPC-C, whose access pattern exhibits strong skew), ACF is an invariant with respect to the arrival rate, degree of concurrency in the system (i.e., number of nodes/threads generating transactions), physical hardware infrastructure (e.g., private cluster vs. public cloud platform), and adopted replication scheme.

The ACF abstraction represents the foundation on top of which we built the AMs of the lock contention dynamics, to be discussed shortly. These models allow one to predict the contention probability that would be experienced by an application in the presence of different scenarios of workloads (captured by shifts of λ_{lock} or ACF), as well as of different levels of contention on physical resources (that would lead to changes of the execution time of the various phases of the transaction life cycle, captured by shifts of T_H).

4.1.2. PB Lock Contention Model. In the PB replication protocol, concurrency among update transactions is regulated by the primary node, during the local execution phase, by exploiting the encounter-time locking scheme introduced in Section 3.1. This implies that replicating the outcome of an update transaction cannot give rise to aborts

due to concurrency. As a result, in the model described in the following, the only contention dynamics to consider are the ones relevant to the local execution of an update transaction.

Let us denote with λ_{lock} the lock request rate generated by local transactions on the primary node. This can be computed as:

$$\lambda_{lock} = \lambda_{Tx} \cdot w \cdot \tilde{N_l},$$

where we have denoted with $\tilde{N_l}$ the number of locks successfully acquired on average by an update transaction executing on the primary, independently of whether it aborts or commits.

When an update transaction executes on the primary, it can experience lock contention and therefore abort. By using Equation (1) (see Section 4.1.1), we can compute the probability P_a for an update transaction to abort during the execution phase at the primary as:

$$P_a = P_{lock} = \lambda_{lock} \cdot ACF \cdot T_H.$$

Hence, we can compute the probability that a transaction reaches its commit phase (P_{cp}) as:

$$P_{cp} = (1 - P_a)^{N_l}. \tag{3}$$

We can then compute the mean number of locks successfully acquired by an update transaction, $\tilde{N_l}$, taking into account that it can abort during its execution:

$$\tilde{N_l} = P_{cp} \cdot N_l + \sum_{i=2}^{N_l} P_a \cdot (1 - P_a)^{i-1} \cdot (i - 1).$$

In order to compute these probabilities, we need to obtain the mean holding time for a lock. To this end, let us define as $G(i)$ the sum of the mean lock hold time over i consecutive lock requests, assuming that the average time between two lock requests is uniform and equal to $\frac{T_{localWR}}{N_l}$:

$$G(i) = \sum_{j=1}^{i} \frac{T_{localWR}}{N_l} \cdot j.$$

We can then compute the local lock hold time as the weighted average of two different lock holding times, referring to the case that a transaction aborts (H_a) or successfully completes (H_c):

$$\begin{aligned} T_H &= H_a + H_c \\ H_a &= \sum_{i=2}^{N_l} P_a \cdot (1 - P_a)^{i-1} \cdot \frac{1}{i-1} \cdot G(i-1) \\ H_c &= P_{cp} \cdot \left[T_{cd} + \frac{1}{N_l} \cdot G(N_l) \right], \end{aligned}$$

where we have denoted with T_{cd} the duration of the synchronous broadcast RPC issued by the primary to disseminate the updates toward the backups.

Given that an update transaction can terminate its execution (either aborting or committing) in two different phases, its mean service time, denoted as T_{pb}^{W}, is the

average among these cases:

$$T_{pb}^{W} = T_c + T_a$$
$$T_c = P_{cp} \cdot (T_{localWR} + T_{cd} + T_{comm})$$
$$T_a = \sum_{i=1}^{N_l} \left[T_{roll} + \left(\frac{T_{localWR}}{N_l} \cdot i \right) \right] \cdot P_a \cdot (1 - P_a)^{i-1}.$$

We denoted with T_{roll} the time spent to perform the rollback of a transaction, and T_{comm} the time to finalize it upon commit. The average service time for a transaction, considering both read-only and update transactions, is

$$T_{pb} = w \cdot T_{pb}^{W} + (1 - w) \cdot T^{localRO}. \tag{4}$$

4.1.3. 2PC Lock Contention Model. In the 2PC scheme, update transactions can be generated at any node. Hence, unlike in the PB case, the analytical model needs to take into account also the possibility of conflict arising with *remote* transactions (i.e., transactions that have been originated on a different node and that are executing their commit phase). Further, transactions can conflict among each other (and therefore abort) during both the local execution and the remote validation phase. In the following, given a transaction t, we will denote as ν_t the node on which t was originated.

We start the description of the 2PC model by presenting the equations that approximate the aforementioned conflict probabilities, which are at the core of the model construction. To this end, we introduce λ_{lock}^{l} and λ_{lock}^{r}, which are, respectively, the lock request rate generated by local and remote transactions on a given node and are computed as follows:

$$\lambda_{lock}^{l} = \frac{\lambda_{Tx} \cdot w \cdot \tilde{N}_l}{\nu} \qquad \lambda_{lock}^{r} = \tilde{N}_r \cdot \lambda_{Tx} \cdot w \cdot \frac{\nu - 1}{\nu} \cdot P_p.$$

We have denoted with P_p the probability for a transaction to reach the prepare phase (i.e., not aborting during the local execution), and with $\tilde{N}_l$ (respectively $\tilde{N}_r$) the number of locks successfully acquired on average by local (respectively remote) transactions, independently of whether they abort or commit. We will provide an analytical derivation of these quantities later in this section.

When a transaction executes locally, it can experience lock contention (and therefore abort) both with other local transactions and remote ones. By using Equation (1), we can express the probability of abort during local transaction execution, P_a^l, as:

$$P_a^l = P_{lock}^l = ACF \cdot \left(\lambda_{lock}^l \cdot T_H^l + \lambda_{lock}^r \cdot T_H^r \right) \tag{5}$$

with T_H^l and T_H^r being the lock holding time of, respectively, local and remote transactions. We will show how the model computes these two parameters in short.

When a transaction t is being remotely validated on a node ν', it can experience contention with any transaction that was originated on ν' (either still locally running or already in its validation phase) and with any transaction coming from the other $\nu - 2$ nodes in the system (ν_t is not included, since transactions that are local to a node can conflict one with another only on that node). Thus, we compute the probability of such a contention to arise as:

$$P_a^r = ACF \cdot \left(\lambda_{lock}^l \cdot T_H^l + \lambda_{lock}^r \cdot \frac{(\nu - 2)}{(\nu - 1)} \cdot T_H^r \right).$$

With the last two probabilities, the model can compute the probability that a transaction reaches the prepare phase (P_p) and gets successfully validated on the other nodes in the system (P_{coher}).

P_p can be straightforwardly obtained from Equation (5):

$$P_p = \left(1 - P_a^l\right)^{N_l}.$$

Conversely, in order to compute P_{coher}, the model must take into account that events relevant to remote conflicts, which may occur for the same transaction on different nodes in parallel, are not statistically independent. In fact, if a transaction t aborts due to conflict with transaction t' on a node that is different from $\nu_{t'}$, then it is very likely that the same conflict arises also on other nodes. In other words, the model considers the remote abort probability as a function of the number of transactions executing on remote nodes, independently of the number of nodes they are being validated on.

Thanks to this approximation, we introduce the probability for a transaction to abort upon issuing a lock request on a remote node ν' because of a conflict with another transaction local to ν':

$$P_a^{\dagger} = \left(\lambda_{lock}^l \cdot ACF \cdot T_H^l\right).$$

Hence, P_{coher} is computed as follows:

$$P_{coher} = \left(1 - P_a^{\dagger}\right)^{N_l \cdot (\nu - 1)}.$$

The commit probability of a transaction is $P_c = P_p \cdot P_{coher}$.

With the probabilities computed so far, it is possible to compute the number of locks successfully acquired by local and remote transactions, which we introduced at the beginning of this section:

$$\tilde{N_l} = P_p \cdot N_l + \sum_{i=1}^{N_l} P_a^l \cdot \left(1 - P_a^l\right)^{i-1} \cdot (i-1)$$

$$\tilde{N_r} = \left(1 - P_a^r\right)^{N_l} \cdot N_l + \sum_{i=1}^{N_l} P_a^r \cdot \left(1 - P_a^r\right)^{i-1} \cdot (i-1).$$

From these, it is possible to obtain the mean holding time for a lock. Using the same notation as in Section 4.1.2 to denote the cumulative hold time over i consecutive lock requests, we compute the local lock hold time as the weighted average of three different lock holding times, referring to the case that a transaction aborts locally (H_l^{la}) or remotely (H_l^{ra}) or successfully completes (H_l^c):

$$T_H^l = H_l^{la} + H_l^{ra} + H_l^c$$

$$H_l^{la} = \sum_{i=2}^{N_l} P_a^l \cdot \left(1 - P_a^l\right)^{i-1} \cdot \frac{G(i-1)}{i-1}$$

$$H_l^{ra} = P_p \cdot (1 - P_{Coher}) \cdot \left[T_{prep} + \frac{G(N_l)}{N_l}\right]$$

$$H_l^c = P_p \cdot P_{Coher} \cdot \left[T_{prep} + \frac{G(N_l)}{N_l}\right].$$

We now compute the remote lock hold time, T_h^r. Here, we neglect the lock holding times for transactions that abort while acquiring a lock on a remote node, as in this case locks are acquired consecutively (without executing any business logic between two lock requests). On the other hand, if a remote transaction succeeds in acquiring all

its locks, then it holds them until it receives either a commit or an abort message from the coordinator. Therefore, we compute T_h^r as:

$$T_h^r = \left(1 - P_a^r\right)^{N_l} \cdot \left[T_{prep} + \left(1 - P_a^\dagger\right)^{N_l \cdot (\nu - 2)} \cdot T_{comm}\right], \tag{6}$$

where $(1 - P_a^\dagger)^{N_l}$ approximates the probability for a remote transaction t executing its prepare phase at node n to successfully acquire all the locks it requests on n, and $(1 - P_a^r)^{N_l \cdot (\nu - 2)}$ approximates the probability for t to successfully acquire its remote locks on the remaining $\nu - 2$ nodes.

Given that an update transaction can terminate its execution (either aborting or committing) in three different phases, its mean service time, denoted as T_{2pc}^W, is the average among these cases:

$$\begin{aligned} T_{2pc}^W &= T_c + T_a^l + T_a^r \\ T_c &= P_c \cdot (T_{localWR} + T_{prep} + T_{comm}) \\ T_a^l &= \sum_{i=1}^{N_l} \left[T_{roll} + \left(\frac{T_{localWR}}{N_l} \cdot i\right)\right] \cdot P_a^l \cdot \left(1 - P_a^l\right)^{i-1} \\ T_a^r &= P_p \cdot (1 - P_{coher}) \cdot (T_{localWR} + T_{prep} + T_{roll}). \end{aligned} \tag{7}$$

Considering read-only transactions, the average service time of a transaction, denoted as T_{2pc}, is:

$$T_{2pc} = w \cdot T_{2pc}^W + (1 - w) \cdot T^{localRO}. \tag{8}$$

4.1.4. CPU Model. We model the CPU of the nodes of the platform as an $M/M/K$ multiclass queue with FCFS discipline [Kleinrock 1975], where K is the number of cores per CPU. The CPU can potentially serve three classes of transactions: read-only (RO), local update (WR^l), and remote update (WR^r) transactions. We denote with ρ the utilization of the CPU, with D_i the service demand for a transaction of class i and with λ_i the arrival rate for transactions of class i. The CPU's utilization can be approximated as follows:

$$\rho = \frac{1}{K} \cdot \left(\lambda_{RO} \cdot D_{RO} + \lambda_{WR}^l \cdot D_{WR}^l + \lambda_{WR}^r \cdot D_{WR}^r\right).$$

Finally, defining:

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

we have that for any CPU response time T_i corresponding to a demand D_i [Kleinrock 1975]:

$$T_i = D_i \cdot \gamma.$$

In the following, we specialize these equations for the PB and the 2PC schemes, using a notation that resembles the one adopted in previous sections.

In the PB scheme we have for the master:

$$\lambda_{WR}^l = \lambda_{Tx} \cdot w$$

and for the slaves:

$$\lambda_{RO} = \frac{\lambda_{Tx} \cdot (1 - w)}{\nu - 1}; \qquad \lambda_{WR}^r = \lambda_{Tx} \cdot w \cdot P_{cp}.$$

The arrival rate for the other classes is equal to zero. CPU demand on the primary is:

$$D^l_{WR} = P_{cp} \cdot D_{localWR} + \sum_{i=1}^{N_l} \left[D_{roll} + \left(\frac{D_{localWR}}{N_l} \cdot i \right) \right] \cdot P_a \cdot \left(1 - P_a\right)^{i-1},$$

whereas for the slaves it is:

$$D_{RO} = D_{localRO}; \qquad D^r_{WR} = D_{remoteWR},$$

where we denoted with $D_{remoteWR}$ the CPU demand to serve a remote transaction.

In the 2PC scheme, each CPU processes all the three classes of transactions, whose arrival rates are

$$\lambda^l_{WR} = \frac{\lambda_{Tx} \cdot w}{\nu}; \qquad \lambda^r_{WR} = \lambda_{Tx} \cdot w \cdot \frac{\nu - 1}{\nu} \cdot P_p; \qquad \lambda_{RO} = \frac{\lambda_{Tx} \cdot (1 - w)}{\nu}.$$

Regarding the CPU demand of local transactions, we have, similarly to Equation (7):

$$\begin{aligned}
D^l_{WR} &= D^l_c + D^l_a l + D^l_a r \\
D^l_c &= P_c \cdot (D_{localWR} + D_{comm}) \\
D^{ll}_a &= \sum_{i=1}^{N_l} \left[D_{roll} + \left(\frac{D_{localWR}}{N_l} \cdot i \right) \right] \cdot P^l_a \cdot \left(1 - P^l_a\right)^{i-1} \\
D^{lr}_a &= P_p \cdot (1 - P_{coher}) \cdot (D_{localWR} + D_{roll}).
\end{aligned}$$

Following a reasoning similar to the one at the basis of Equation (6), we have that CPU demands of remote transactions are:

$$\begin{aligned}
D^r_{WR} &= D^r_c + D^r_a l + D^r_a r \\
D^r_c &= \left(1 - P^r_a\right)^{N_l} \cdot \left[\left(1 - P^\dagger_a\right)^{N_l \cdot (\nu - 2)} \right] \cdot (D_{comm} + D_{remoteWR}) \\
D^{rl}_a &= \sum_{i=1}^{N_l} \left[D_{roll} + \left(\frac{D_{remoteWR}}{N_l} \cdot i \right) \right] \cdot P^l_a \cdot \left(1 - P^l_a\right)^{i-1} \\
D^{rr}_a &= \left(1 - P^r_a\right)^{N_l} \cdot \left[1 - \left(1 - P^\dagger_a\right)^{N_l \cdot (\nu - 2)} \right] \cdot (D_{roll} + D_{remoteWR}).
\end{aligned}$$

4.1.5. AMs Resolution and Predicted KPIs. As in previous analytical approaches capturing transactional data contention [Yu et al. 1993; di Sanzo et al. 2010], ours also exhibits mutual dependency between the model of the concurrency control scheme and the CPU model. Moreover, in the data contention model, dependencies also arise between the abort probabilities and other parameters, such as the mean lock hold time. Prior art copes with this issue by using an iterative scheme in which abort probabilities are first initialized to zero. Next, CPU response times are computed and, hence, the lock contention model's parameters are computed. On the basis of their values, a new set of abort probabilities is obtained and used in the next iteration; the process continues till the relative difference between the abort probabilities at two subsequent iterations becomes smaller than a given threshold.

It is known [Yu et al. 1993] that this iterative solution technique can suffer from convergence problems at high contention rates; the problem gets exacerbated in the 2PC model, since it encompasses two different, independent abort probabilities. We tackle this issue by adopting a binary search in the bidimensional space $[0, 1] \times [0, 1]$

associated with the abort probabilities (local and remote), which is guaranteed to converge at a desired precision $\epsilon \in (0, 1]$ after a number of steps $n \leq 1 + \lceil -log_2 \epsilon \rceil$. This analysis was confirmed by our evaluation study, reported in Section 5, for which we set $\epsilon = 0.01$ and observed convergence in at most seven iterations.

For the PB model, the binary search is performed on a monodimensional domain representing the local abort probability, since remote aborts cannot arise.

Once the commit probability and average response time of a transaction are obtained, the model can be exploited to compute additional KPIs typically employed in the SLA definition, such as system throughput or percentiles on response times.

Closed-system throughput. The throughput achieved when considering a closed system with ν nodes, with θ threads per node, can be computed by exploiting Little's law [Little 1961] in an iterative fashion. At the first step of the iteration, a low value of transactions' arrival rate is provided as input to the model. Next, the open model is solved to compute the transaction's response time. This value is used as input for the Little's law to compute an updated throughput value, which is used as the input arrival rate for the next iteration. The iterations continue till the relevant difference between the arrival rate in input to the model and the computed closed-system throughput falls below a given threshold. This process typically completes in a few iterative steps, except for high-contention scenarios where it may suffer from convergence problems. This is a typical issue that arises when adopting such a recursive resolution algorithm for analytical models of transactional systems [Yu et al. 1993]. To cope with such an issue, TAS implements a fallback solving algorithm that spans, at a given granularity, all possible arrival rate values within a configurable interval. This algorithm returns the solution that minimizes the error between the input arrival rate and the output closed-system throughput. This guarantees convergence to the desired accuracy in a bounded number of steps.

For the 2PC model, the closed-system throughput is computed as:

$$\lambda^{2pc} = \frac{\nu\theta}{w \cdot T_{2pc}}.$$

In the PB case, the closed-system throughput, denoted as λ^{pb}, is computed by taking into account that the bottleneck can either be the primary, which processes update transactions at rate:

$$\lambda^{primary} = \lambda^{pb} \cdot w = \theta / T^{W}_{pb},$$

or the backups, which globally can process read-only transactions at rate:

$$\lambda^{backups} = \lambda^{pb} \cdot (1 - w) = \frac{\theta(\nu - 1)}{T^{localRO} \cdot (1 - w)}.$$

These two quantities are obtained separately and then the closed-system throughput is computed as follows:

$$\lambda^{pb} = \min \left\{ \frac{\theta}{w \cdot T^{W}_{pb}}, \frac{(\nu - 1) \cdot \theta}{T^{localRO} \cdot (1 - w)} \right\},$$

which is obtained by imposing that the stream of arriving requests includes a mix containing w% of update transactions.

Response time percentiles. In order to compute response time percentiles, it is possible to model each data grid node as a G/G/θ queuing system (i.e., a queue with θ servers subjected to arbitrary service and arrival rate distributions). One can then exploit the Köllerström's [Bhat et al. 1979] approximation for the waiting time distribution of

G/G/θ queues in the heavy-traffic case, namely, when the queue utilization $\rho \simeq 1$. This result states that the approximate distribution of the waiting time, ω, of a G/G/θ queue in heavy traffic is exponential and is given by:

$$P(\omega \leq t) \simeq 1 - e^{-\frac{2(1/\lambda - T_s)}{\sigma_u^2 + \frac{\sigma_b^2}{\theta^2}} t},$$

where σ_u is the interarrival time variance, σ_b^2 is the service time variance (both measurable at run-time, as done in other systems for automated resource provisioning, such as [Singh et al. 2010]), λ is the request arrival rate, and T_s is the average service time. The previous formula can hence be used to compute the maximum arrival rate λ such that the response time is less than a given threshold y with probability k. For instance, if the SLA requires the 95th percentile response time to be less than y seconds, the maximum sustainable arrival rate can be computed as:

$$\lambda < \nu \left\{ \frac{1}{T_s} + \frac{1-\rho}{\sigma_u^2 + \frac{\sigma_b^2}{\theta^2}} \frac{6}{y - T_s} \right\},$$

in which T_s can be specialized according to the employed distributed commit algorithm by exploiting Equations (8) and (4).

4.1.6. Extensions of the AM. The presented model lends itself to be extended in several directions. In the following, we briefly overview some of the most interesting possible extensions.

Retry-on-abort logic. Some applications may require that, in case of abort, a transaction is re-executed until it can successfully commit. In Infinispan, an exception is thrown upon the abort of a transaction and it is up to the business logic of the application to decide whether to retry the transaction or not. TAS can be easily extended in order to forecast the performance of applications implementing the *retry-on-abort* logic. Note that, thanks to the modular approach exploited in TAS, such an extension only targets the employed analytical models, as the network layer is not affected by the application's logic. The resolutive scheme of the model remains unchanged and only the analytical expression for some parameters has to be changed in order to match the behavior of the *retry-on-abort* logic. In this section, we are going to show only the equations of the model that differ from the ones already discussed. The main assumptions at the basis of the new equations is that the transaction abort probability, P_{abort}, is not affected by the number of retries it has performed. This allows us to model the number of aborts a transaction incurs as a geometrically distributed random variable with parameter P_{abort}.

In the PB scheme, a transaction can only abort on the master node during the local execution phase. Hence, reusing the notation of Section 4.1.2, the number of retries for a transaction is:

$$R_a = \frac{1 - P_{cp}}{P_{cp}}.$$

Thus, the actual arrival rate of update transactions becomes:

$$\lambda_{WR}^{l,PB} = \lambda_{Tx} \cdot w \cdot (1 + R_a),$$

and the lock acquisition rate is:

$$\lambda_{lock} = \lambda_{WR}^{l,PB} \cdot \tilde{N_l}.$$

Hence, it is possible to compute P_a, N_l, and T_H, exploiting the same equations shown in Section 4.1.2. The duration of an update transaction is the sum of the time spent in

aborted runs (denoted as T_a), and the final committed execution time:

$$T_{pb}^{W} = R_a \cdot T_a + T_{localWR} + T_{cd} + T_{comm},$$

where

$$T_a = T_{roll} + \sum_{i=1}^{N_l} \left(\frac{T_{localWR}}{N_l} \cdot i \right) \cdot \frac{P_a \cdot (1 - P_a)^{i-1}}{1 - P_{cp}}.$$

In the 2PC scheme, a transaction can abort either locally or remotely. Every time a transaction is aborted remotely, it starts from scratch, thus being potentially subject to new local aborts. The number of local aborts experienced by a transaction any time it starts its local execution (i.e., upon beginning or after a remote abort), namely, R_a^l, is computed as follows:

$$R_a^l = \frac{1 - P_p}{P_p}.$$

Similarly, the number of expected remote aborts, R_a^r, is computed as:

$$R_a^r = \frac{1 - P_{coher}}{P_{coher}}.$$

Local and remote transactions' arrival rates become, respectively,

$$\lambda_{WR}^{l,2PC} = \frac{\lambda_{Tx} \cdot w}{\nu} \cdot \left(1 + R_a^r\right) \cdot R_a^l \qquad \lambda_{WR}^{r,2PC} = \frac{\lambda_{Tx} \cdot w \cdot (\nu - 1)}{\nu} \cdot \left(1 + R_a^r\right),$$

and the lock acquisition rates are

$$\lambda_{lock}^{l} = \lambda_{WR}^{l,2PC} \cdot \tilde{N}_l \qquad \lambda_{lock}^{r} = \lambda_{WR}^{r,2PC} \cdot \tilde{N}_r.$$

As for the PB case, the equations relevant to lock contention probabilities, holding times, and average number of locks acquired by transactions remain unchanged. The model can, thus, obtain the execution times for locally aborted transactions (denoted as T_a^l) and remotely aborted ones (denoted as T_a^r). For the former, we have:

$$T_a^l = T_{roll} + \sum_{i=1}^{N_l} \left(\frac{T_{localWR}}{N_l} \cdot i \right) \cdot \frac{P_a^l \cdot \left(1 - P_a^l\right)^{i-1}}{1 - P_p},$$

and for the latter:

$$T_a^r = T_{roll} + T_{localWR} + T_{prep}.$$

Finally, the average execution time of an update transaction, including the time spent in reruns, is:

$$T_{2PC}^{W} = \left(R_a^r + 1\right) \cdot \left(R_a^l \cdot T_a^l\right) + \left(R_a^r \cdot T_a^r\right) + T_{localWR} + T_{prep} + T_{comm}.$$

Also, CPU utilization is affected by the *retry-on-abort* logic, since transactions do not leave the system upon aborting. The extension to the CPU model to capture the effects of *retry-on-abort* requires adapting the equations relevant to transactions' CPU demands. Since this adaptation is analogous to the one performed for the non-*retry-on-abort* case in Section 4.1.4, for the sake of brevity, we will not show these equations.

Mix-aware modeling. Extending our approach to account for multiple transaction classes having different characteristics (for instance, in terms of data access pattern or duration of local execution) would require two main steps:

(1) Extracting a characterization of the different transactional classes, including per-class information on ACF, abort probability, mean number of locks requested per transaction, and local execution time, and of the ratio of each class in the mix. Identification of different transactional classes can be performed in a transparent way using classic data-clustering techniques, such as the one used, for example, in Ghanbari et al. [2007].
(2) Specializing the analytical model to forecast the contention probability (and depending statistics, such as throughput) per transaction class. This result can be achieved in a relatively simple way by employing a methodology, similar to the one proposed in Yu et al. [1993], in which the transaction conflict probability is computed taking into account the data access patterns (in our case captured by the ACF) of each single transaction class.

Heterogeneous platforms. As in prior approaches for automated resource provisioning for cloud [Singh et al. 2010; Shen et al. 2011], heterogeneous platforms can be handled by using simple multiplication factors between servers depending on their hardware characteristics. For example, Amazon EC2 offers various instances (small, medium, large, etc.), each equipped with different hardware resources. Via a preliminary benchmarking study (using synthetic workloads, as in Shen et al. [2011], or, when possible, directly using the target application, as in Singh et al. [2010]), it is easy to determine scaling factors relating the performance achieved when deploying the application on different types of instances. For example, a medium instance performs 1.5 times better than a small instance, or a large instance provides two times the throughput of a small instance. These scaling factors can then be applied to forecast the values of the parameters associated with the duration of local transaction execution, namely, $T_{localRO}$ and $T_{localWR}$, when deploying the application on a different instance type.

4.2. Machine-Learning-Based Modeling

TAS relies on black-box, machine-learning-based modeling techniques to forecast the impact on performance due to shifts of the level of contention on the network layer depending on the workload's fluctuations or on the resizing of the data grid. Developing white-box models capable of capturing accurately the effects on performance due to contention at the network level can in fact be very complex (or even unfeasible, especially in virtualized cloud infrastructures), given the difficulty to gain access to detailed information on the exact dynamics of network-level components.

In TAS, we exploit the availability of a complementary white-box model of a system's performance to formulate the machine-learning-based forecasting problem in a way that differs significantly from traditional, pure black-box approaches. Conventional machine-learning-based techniques (e.g., Shen et al. [2011]) try to forecast some performance metric p_2 in an unknown system configuration c_2, given the performance level p_1 and the demand of resources d_1 in the current configuration c_1. In TAS, instead, the analytical model can provide the machine learner with valuable estimates of the demand of resources d_2 in the target configuration c_2. Specifically, we use the analytical model to forecast what will be, in the target configuration c_2, the rate of transactions that will initiate a commit phase as well as the percentage of CPU time consumed by the threads in charge of processing transactions.

As already mentioned, contention on the network layer can have a direct impact on the latency of a key phase of the transaction's execution, namely, the duration of the commit phase, denoted as T_{prep} for 2PC and as T_{cd} for PB. We are here faced with a nonlinear regression problem, in which we want to learn the value of continuous functions defined on multivariate domains. Given the nature of the problem, we used

the Cubist machine learner [Quinlan 2012], which is a decision-tree regressor that approximates nonlinear multivariate functions by means of piece-wise linear approximations. Analogously to classic decision-tree-based classifiers, such as C4.5 and ID3 [Quinlan 1993], Cubist builds decision trees choosing the branching attribute such that the resulting split maximizes the normalized information gain. However, unlike C4.5 and ID3, which contain elements in a finite discrete domain (i.e., the predicted class) as leaves of the decision tree, Cubist places a multivariate linear model at each leaf.

In order to build an initial knowledge base to train the machine learner, TAS relies on a suite of synthetic benchmarks that generate heterogeneous transactional workloads in terms of mean size of messages, memory footprint at each node, CPU utilization, and network load (number of transactions that activate the commit phase per second).

It is well known that the selection of the features to be used by machine-learning toolkits plays a role of paramount importance, as it has a dramatic impact on the quality of the resulting prediction models. When performing such a choice, we took two aspects into consideration: first, the set of parameters has to be big enough to guarantee good accuracy, but at the same time it has to not be too large in order to keep low the time taken by the machine learner to create its model and invoke it; since TAS periodically updates its knowledge base with samples taken from the runtime environment, this aspect is very important.

Second, the set of attributes has to be highly correlated to the parameters the machine learner is going to predict, namely, T_{prep} and T_{cd}. In the following, we list the set of features we selected and we motivate their choice.

—Number of nodes (squared): more in detail, we use the plain number of nodes for PB and the squared number of nodes for 2PC. This is motivated by the rationale that, when using 2PC, the number of messages exchanged in the network grows quadratically with the number of nodes.
—Used memory: we have noticed in our experiments that the memory footprint of applications can affect significantly the performance of the network layer (JGroups 2.12 [Ban 2012] in our testbed platform). We argue that this can be due to the growing costs of garbage collection and to the reduced locality when the JVM (JVM 1.6.0_26) needs to manage large heaps.
—CPU utilization: this parameter is needed because the times predicted by the machine learner also include a portion related to CPU processing.
—Number of threads per node: this parameter helps capturing the load at the level of group communication systems (as hinted, JGroups in our reference platform).
—Size of a “Prepare” message: this parameter is of course highly related to the time needed to pack and transmit messages over the network.
—The number of transactions that enter the prepare phase per second: this feature is directly correlated with the number of messages sent, per second, by any node of the system and it is very important since it provides a good measure of the network utilization.

  Of course, this value is known exactly during the training phase, as we can directly measure it. On the other hand, this is unknown at the time in which the machine learner is queried, as it is strictly correlated to the throughput of the system in the target configuration, which is what we ultimately want to forecast. This led us to design the iterative coupling scheme described in Section 4.1, based on which we use the analytical model at query time to provide an estimate of this value.

During the initial, offline, training phase, the benchmark suite injects workload while varying the size of the cluster and the number of threads concurrently processing local

transactions at each node. In our experiments, we found that using a simple uniform sampling strategy allowed us to achieve rather quickly (in about 1 hour) a satisfactory coverage of the parameters' space, which is the reason why we decided not to integrate more advanced sampling mechanisms, such as adaptive sampling [Thompson 2002].

Once deployed on a data grid, the statistical gathering system of TAS periodically collects new samples of the workload and performance of the system. This allows one to support periodic retraining of the machine learner and to incorporate in its knowledge base profiling data specific to the target-user-level applications.

4.3. AM and ML Coupling

By the previous discussion, it is clear that the AM and the ML are tightly intertwined: the AM relies on the predictions of the ML to obtain the values of T_{prep} (or T_{cd}, depending on the employed replication protocol) as input; the ML, on the other hand, uses as one of the input features of its model the transaction throughput forecast by the AM, which represents an estimate of the level of resource contention in the target configuration. This cyclic dependency is broken in the following way. At the beginning of each step of the iterative model resolution algorithm, the AM computes the values needed by the ML, as a function of the transactions' arrival rate, abort probabilities, and workload characterization. Then, taking such values as input, the ML outputs a prediction for T_{prep} (or T_{cd}), which is finally exploited by the AM to compute the transactions' response time.

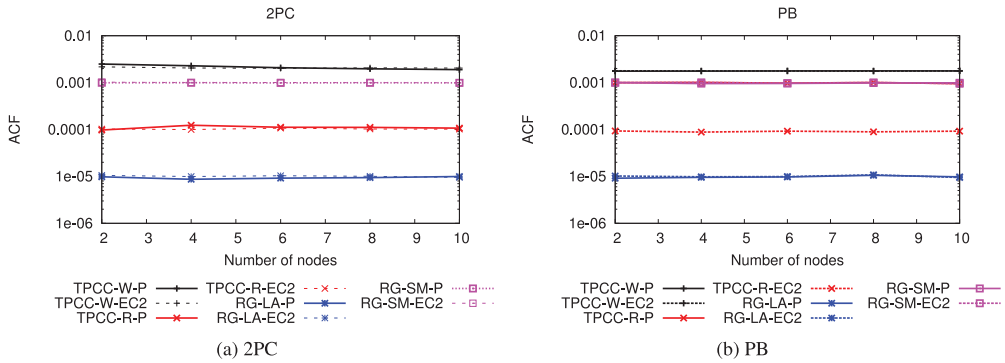
5. VALIDATION

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

Workloads. We consider two well-known benchmarks, namely, TPC-C and Radargun, which have been already mentioned in Section 1. The former is a standard benchmark for OLTP systems, which portrays the activities of a wholesale supplier and generates mixes of read-only and update transactions with strongly skewed access patterns and heterogeneous durations. Radargun, instead, is a benchmarking framework specifically designed to test the performance of distributed, transactional key-value stores. The workloads generated by Radargun are simpler and less diverse than TPC-C's workloads but have the advantage of being very easily tunable, thus allowing assessing the accuracy of TAS in a wider range of workload settings.

For TPC-C, we consider two different workload scenarios. The first, which we denote as TPCC-R, is a read-dominated workload (entailing 90% read-only transactions), which generates reduced contention on both infrastructure resources and data independently of the system's scale. The second (TPCC-W) includes around 50% of update transactions and generates a high level of data contention.

As for Radargun, we also consider two workloads, denoted as RG-LA and RG-SM. Both workloads generate uniform data access patterns, but RG-LA performs, in each transaction, five put operations over a set of 100K data items, yielding very low contention. RG-SM, instead, updates in each transaction five data items selected over a set of cardinality 1K, thus generating high contention probability. In order to assess the accuracy of TAS with workloads characterized by significantly diverse scalability trends, we also vary the percentage of write transactions (hereafter noted "w"), setting it to 5%, 10%, and 50%. We decided to use also the Radargun workloads in our evaluation study because its data access pattern (uniform accesses over a



(a) 2PC (b) PB

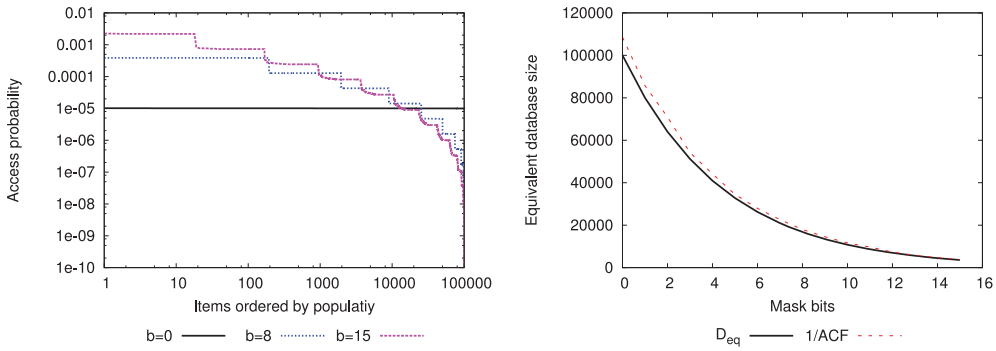
Fig. 4. ACF using heterogeneous benchmarks and platforms.

keyset of fixed size) allows us to validate the correctness and semantics of the ACF abstraction.

Experimental platforms. We use, as experimental testbeds for this study, both a private cluster and two public cloud infrastructures, namely, Amazon EC2 and FutureGrid.[6] The private cluster is composed of 10 servers equipped with two 2.13GHz quad-core Intel(R) Xeon(R) processors and 8GB of RAM and interconnected via a private Gigabit Ethernet. For EC2, we used up to 20 extra large instances, which are equipped with 15GB of RAM and four virtual cores with two EC2 compute units each. Finally, the testbed platform on FutureGrid is composed of a pool of 100 virtual machines (VMs), each equipped with one virtual core and 2GB of RAM and interconnected via an Infiniband network.

AM validation. We start by validating the semantics of the ACF, which we use to succinctly characterize the application's data access patterns' characteristics from a contention-oriented perspective. In Figure 4, we report the ACFs obtained when running the TPC-C and Radargun workloads on EC2 and on the private cluster (note that we tag the curves obtained on the private cluster with the suffix "-P"). The plot on the left reports the ACF values obtained using the 2PC replication scheme, whereas the one on the right reports the ACF computed using the PB scheme. The plots confirm our findings, namely, that once fixed an application workload, the ACF represents an invariant that is not affected by (i) the scale and nature (private vs. public) of the platform on which the transactional data grid is deployed and (ii) the data consistency mechanism employed by the transactional data grid (2PC or PB). Next, we validate the alternative interpretation of ACF that we provided in Section 4.1.1. Specifically, we observed that ACF can be seen as the inverse of the size (D_{eq}) of a uniformly accessed dataset that would generate an equivalent conflict probability when subject to the same average transaction arrival rate. To this end, we configured the Radargun benchmark to execute transactions composed of a single write operation on a datum selected among a set of 100,000 items on the basis of a nonuniform distribution, namely, the data access distribution generated by the NuRand function of the TPC-C benchmark. The choice of NuRand is due to the fact that it can be easily tuned by means of an input parameter (i.e., a bit mask of 15 bits): the higher the number $b \in [1, 15]$ of bits set to 1, the higher the skew in the data accesses generated by NuRand. Moreover, thanks to its relatively simple analytical definition, it also allows us to compute analytically the size of an equivalent uniformly accessed dataset. Employing the NuRand, in fact, implies that

[6]FutureGrid, http://www.futuregrid.org.



(a) Data popularity

(b) Equivalent dataset size and comparison with $\frac{1}{ACF}$

Fig. 5. NuRand's data access pattern varying the bit mask.

transactions are split in $b+1$ classes, corresponding to $b+1$ disjoint and uniformly accessed subdatasets. In Figure 5(a), we show the impact of the mask on the data access pattern by plotting the access probability of every datum in the dataset after having sorted them by decreasing popularity. The picture shows both the skew growing with the size of the mask and the aforementioned partition of the dataset.

We varied b between 0 (uniformly accessed data) and 15 (the maximum skewness achievable by NuRand in our experiment), and for each value of b, we computed D_{eq}. First, for each class i, we derived the probability of a transaction to belong to that class (p_i) and the size of the subdataset accessed by that class of transactions (d_i). The computation of d_i and p_i follows straightforwardly from the definition of NuRand.[7] Then, we obtained D_{eq} as follows:

$$P(lock\ contention) = \sum_{i=0}^{b} P(xact\ is\ of\ class\ i\ \wedge\ lock\ is\ taken\ by\ another\ xact\ of\ class\ i)$$

$$\frac{\lambda_{lock} \cdot T_H}{D_{eq}} = \sum_{i=0}^{b} p_i \cdot \frac{p_i \lambda_{lock} \cdot T_H^i}{d_i}.$$

Since the transactions are uniform in terms of execution time (i.e., $T_H^i = T_H, \forall i \in [0..b]$):

$$\frac{1}{D_{eq}} = \sum_{i=0}^{b} p_i \cdot \frac{p_i}{d_i}. \tag{9}$$

Finally, we compare the values obtained for D_{eq} from Equation (9) with the values computed as the inverse of the ACF that we measured during each run. As shown in Figure 5(b), we can see that the two values are very close, thus backing our claim. It is noteworthy to highlight that computing D_{eq} analytically requires the *a priori* knowledge about b, p_i, and d_i; conversely, such information is not needed to compute the ACF.

Next, we validate the accuracy of the AMs proposed for the 2PC and PB replication schemes. In order to focus exclusively on the evaluation of the AM component of the

[7]Refer to the TPC-C specification [TPC Council 2011] for a detailed description of NuRand and to the paper by Leutenegger and Dias [1993] for a comprehensive analysis of its properties.

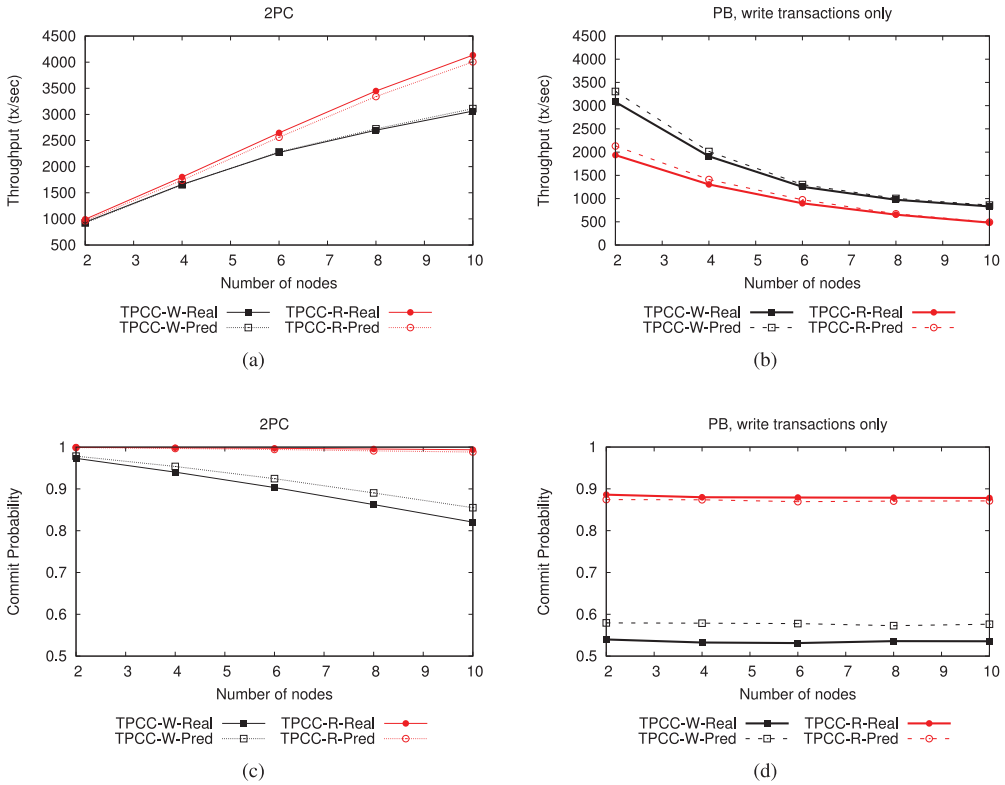
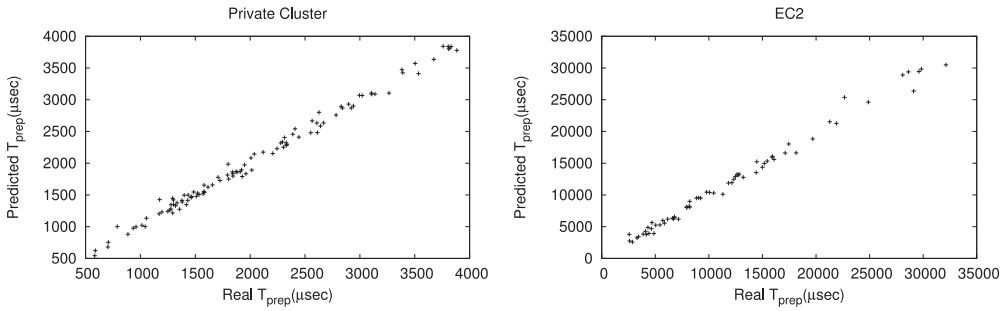


Fig. 6. Validation of AM using the TPC-C benchmark.

TAS methodology, in this experiment we do not use the predictions provided by the ML component (which will be validated in the following). Conversely, we provide the AMs with the actual values of the network latency during the commit phase (namely, T_{prep} and T_{cd}) in the "target" scenario, that is, in the scenario whose performance we forecast via the AMs.

The plots in Figure 6 contrast the predictions of the AMs for 2PC (left-side plots) and PB (right-side plots) with the actual performances measured by deploying Infinispan on a platform composed by 2 up to 10 nodes and running the TPC-C workload. The top plots show the transaction throughput (committed transactions per second), whereas the bottom plots show the transaction commit probability. Note that, for what concerns the PB scheme, we focus our analysis exclusively on the write transactions executed on the primary node, given that forecasting the performance of the read-only transactions while varying the number backups is trivial. In fact, since read-only transactions are processed entirely locally, in a lock-free way and without the possibility of ever aborting, the read-only transactions' throughput simply scales linearly with the number of backups in the system. The plots demonstrate the high accuracy of the AMs, which capture faithfully the performance dynamics of the system and achieve a prediction error constantly below 5%.

ML validation. Let us now assess the accuracy of the machine learners built using the synthetic benchmarking suite described in Section 4.2. We focus here on the accuracy exhibited in predicting the value of T_{prep}; we omit the results for T_{cd} as they show very



(a) (b)

Fig. 7. Accuracy of machine-learning-based T_{prep} predictions on private cluster (left) and on EC2 (right).

similar trends. For this specific validation study, we considered both our private cluster and Amazon as the testbeds. However, further validation data related to deployments of the data platform on top of up to 100 VMs on FutureGrid will be proposed later in this same section.

In order to evaluate the accuracy of the machine-learning model in isolation (i.e., decoupling it from the analytical model), in this experiment we provide the machine learner with the correct value of its input features. The scatter plots in Figure 7 report the results of 10-fold cross-validation for T_{prep}, highlighting that, on both the private cluster and EC2, the ML attains a high prediction accuracy. Specifically, the correlation factor was around 99% in both cases, with an average absolute error equal to 500 microseconds for EC2 and around 60 microseconds for the private cluster. Note that, in practice, the relative error is similar on both the platforms, since, on EC2, the maximum value of T_{prep} is around 10 times larger than the maximum T_{prep} value on the private cluster.

AM/ML validation. Let us now evaluate the accuracy of the final performance predictions provided by TAS when jointly using the AM and the ML, focusing again on the 2PC replication protocol. We use as KPIs the maximum throughput and commit probability. We will first present the results related to TPC-C deployed over our private cluster and EC2; then we will show results related to a large-scale deployment of the Radargun benchmark over the FutureGrid public cloud. The rationale behind using Radargun for the large-scale tests is that, given the relatively small amount of RAM each VM was equipped with on FutureGrid, we were unable to obtain a scalable deployment of our porting of the TPC-C benchmark up to 100 nodes. In fact, the TPC-C application exhibits a very skewed data access pattern toward a set of entries in the dataset whose cardinality is proportional to the memory footprint of the application, thus requiring several gigabytes of RAM in order to avoid excessive contention even at small scales. Moreover, being easily tunable, Radargun gave us the possibility to assess the accuracy of TAS on a set of significantly diverse workloads that exhibit different scalability trends.

We report in Figure 8 the forecasts for the TPC-C workloads using 2PC. The experimental data demonstrate the ability of TAS to predict with high accuracy not only the maximum transaction throughput but also important intermediate statistics such as the commit probability. More in detail, TAS achieves a remarkable average relative error (defined as $\frac{|real-pred|}{real}$) on the predicted throughput of 2%, with a maximum of 3.5%.

The plots in Figure 9 confirm the accuracy of TAS in predicting throughput, abort probability, and distributed commit latencies of applications deployed over a

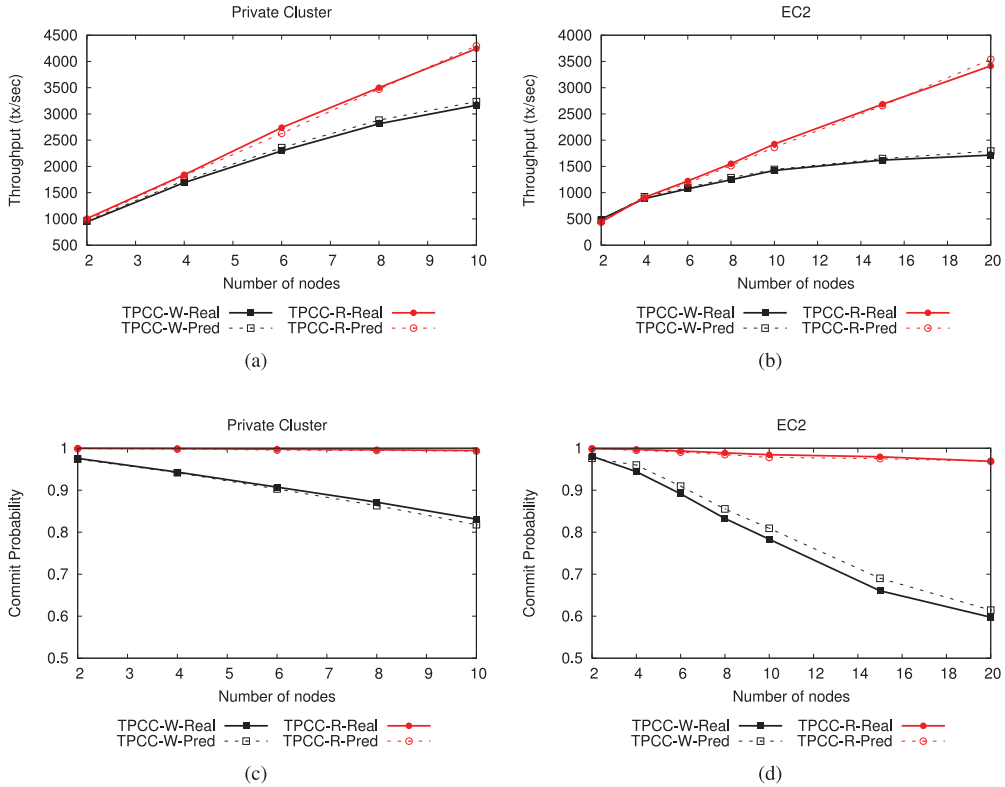


Fig. 8. Validation of the hybrid AM/ML using the TPC-C benchmark.

large-scale data platform. The average absolute relative error exhibited by TAS for the aforementioned KPIs is, respectively, 6%, 0.7%, and 13%. Figure 10 also reports the cumulative density functions (CDFs) of the absolute relative error for these KPIs, showing that the 90th percentile for throughput is less than 10%, the 80th percentile for abort probability is less than 1%, and the 85th percentile for the distributed commit latency is less than 20%.

TAS also turns out to be highly effective in predicting whether the primary scalability constrainer for an application lies in the physical layer or in the level of data contention that it generates. This capability of the model is demonstrated, in particular, in Figures 9(b), 9(d), and 9(f), which report TAS's predictions for the write-intensive Radargun workloads (w = 50). The plots show that the model is able to predict that, while being characterized by similar commit latencies, the RG-SM workload scales half as much as the RG-LA workload (particularly, up to only 40 nodes vs. 80), due to its much higher level of data contention.

Comparison with a pure ML approach. We conclude the validation study by comparing the accuracy of TAS with that of a pure ML-based solution, namely, the approach at the basis of several recent works in the area of elastic scaling [Chen et al. 2006; Ghanbari et al. 2007]. To this end, we trained Cubist on the TPCC-R workload, varying the number of nodes from 2 to 20 and the incoming load from 100 requests per second until reaching the maximum throughput. The input features for the ML included CPU, memory and network utilization, the percentage of update transactions and the mean number of locks they request, the transaction arrival rate, and the number of nodes

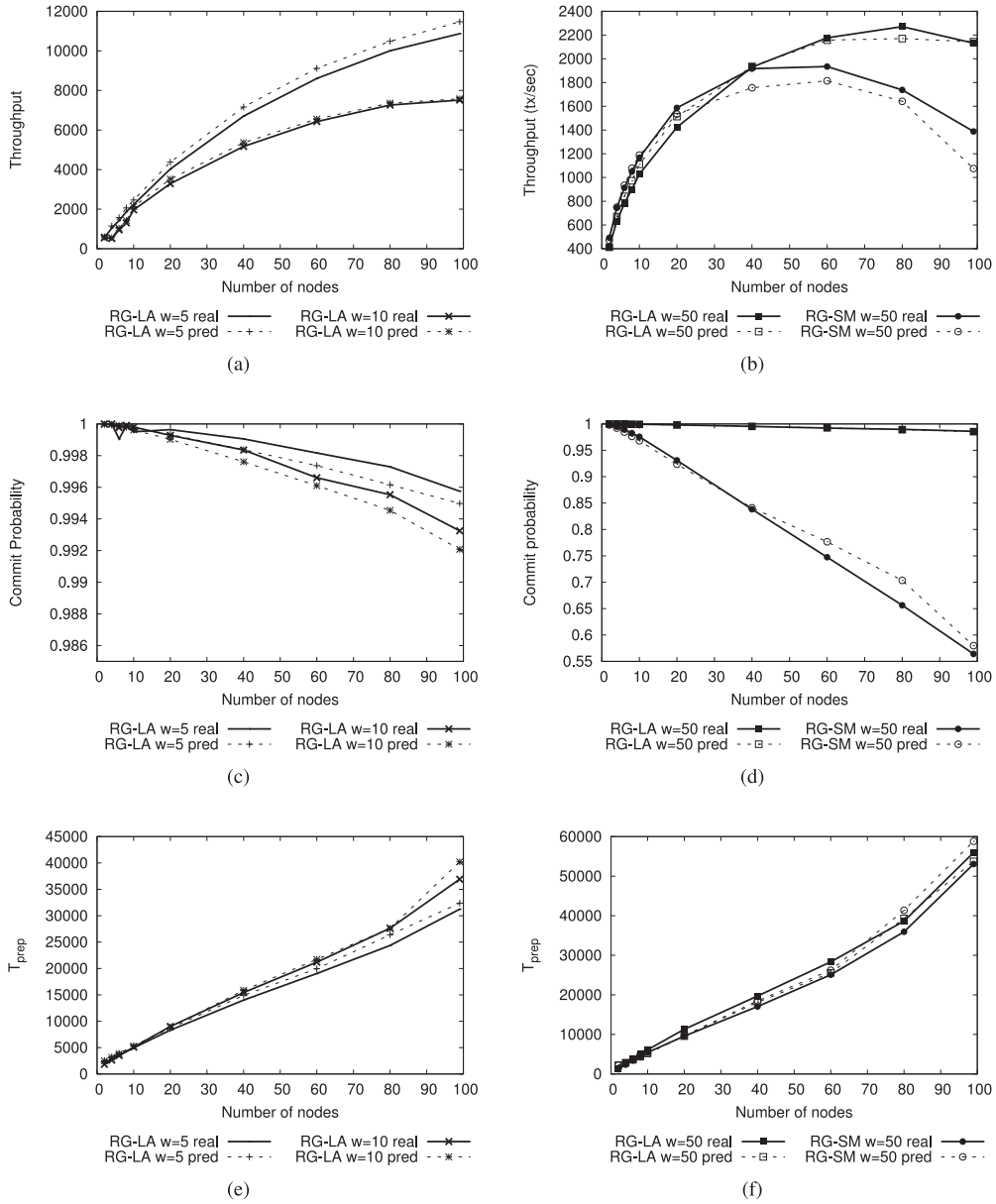


Fig. 9. Validation of the hybrid AM/ML using Radargun.

and active threads per node. As in the previous evaluation study, we use the maximum throughput as the output. These experiments were performed using Amazon EC2.

As the test dataset, we use TPCC-W, which, we recall, generates a significantly higher data contention level with respect to TPCC-R. Further, unlike TPCC-R, TPCC-W exhibits a nonlinear scalability trend. As expected [Chen et al. 2006], in these conditions, the pure ML-based approach manifests its limits in terms of reduced accuracy when working in extrapolation. In fact, the plots in Figure 11 clearly highlight that the pure ML-based solution tends to mimic the linear scalability trend that was observed during

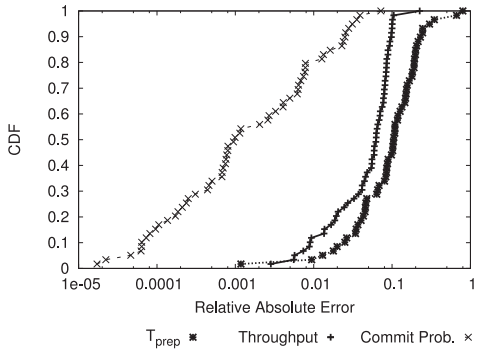


Fig. 10. Absolute relative error CDF of TAS predictions.

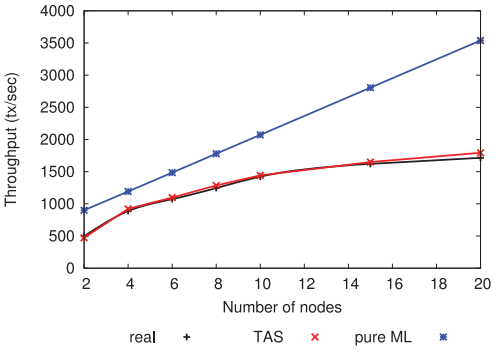


Fig. 11. Comparing TAS with a pure ML approach.

its training phase. As a consequence, it blunders when faced with workloads (like the TPCC-W) that (i) have previously unobserved input characteristics and (ii) exhibit significantly different performance trends. This problem might be tackled to some extent by increasing the coverage of the training phase. However, achieving a good accuracy across a wide range of workloads may require a prohibitive increase of the ML training time. In fact, data contention dynamics in a (distributed) transactional system are influenced by a wide range of parameters [di Sanzo et al. 2010; Bernstein et al. 1986], and it is well known that the training time of ML techniques grows exponentially with the number of input features (the so-called curse of dimensionality problem [Bishop 2007]).

The AM employed by TAS, on the other hand, can exploit the *a priori* knowledge on the dynamics of data consistency mechanisms to achieve higher accuracy when working in extrapolation. Further, it allows one to narrow the scope of (and hence simplify) the problem tackled via ML techniques, reducing the dimensionality of the ML input features' space and, consequently, the duration of the training phase.

6. MODEL CONVERGENCE AND MONITORING OVERHEAD

In this section, we investigate the overhead introduced by the TAS's runtime monitoring framework and the performance of TAS's model solver. We start by evaluating the efficiency of the algorithm used to solve the performance prediction model presented in Section 4.1.5. To this end, we present experimental data obtained querying the model to get predictions for the FutureGrid deployment of the Radargun benchmark, whose results have been reported in Section 5. In all the experiments, the solver was set up as follows: the convergence thresholds on abort probabilities and on closed-system throughput computation were set to 0.01; the minimum/maximum arrival rates were set, respectively, to 10/25K tx/sec, and we used a granularity of 10 tx/sec for the fallback exhaustive search algorithm. We observed that both the average resolution time and its 90th percentile are about 250 msec on a machine equipped with a 2.7GHz Intel Core i7, 8GB of 1,333MHz DDR3 RAM, and running Mac OS X 10.7.5. The heavy tail in the CDF, shown in Figure 12, is due to the resolution times of the model for scenarios characterized by thrashing caused by excessive contention on data and high network latencies: solving the model for such scenarios, in fact, requires the employment of the exhaustive search algorithm introduced in Section 4.1.5.

Finally, we assess the overhead introduced by (i) the distributed monitoring framework used to aggregate statistics gathered from all the nodes in the system and (ii) the additional probes that were integrated in Infinispan to collect detailed statistics

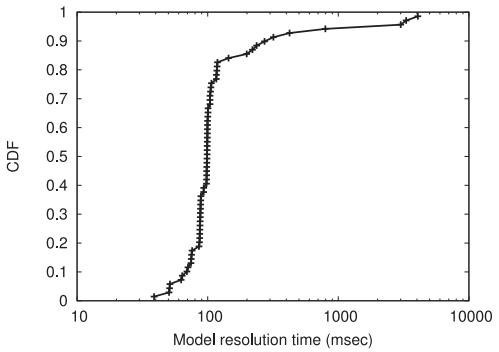


Fig. 12. TAS model resolution time.

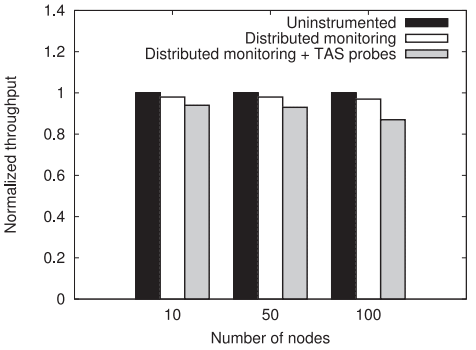


Fig. 13. Monitoring overhead.

on the transactional workload and on the usage of CPU/memory/network resources, which serve as input for the instantiation of the TAS model.

Figure 13 shows the throughput when activating only the distributed monitoring framework and also when activating the probes for detailed statistics collection, normalized to the throughput achieved using a noninstrumented version. In order to evaluate the impact of these mechanisms in platforms of heterogeneous scales, we consider deployments encompassing 10, 50, and 100 nodes.

The plots show that the overhead of the distributed monitoring framework remains unaltered (about 2%) as the size of the data platform varies; conversely, the one introduced by the probes grows with the number of nodes running the application, achieving a maximum of 12% for 100 nodes. Overall, the results show that the monitoring overheads remain largely acceptable even in very large-scale systems. However, it should be noted that these overheads represent, indeed, an upper bound on the actual performance loss that would be introduced in case the probing system avoided tracing *every* transaction and implemented a more efficient/optimized sampling strategy that collected statistics at a lower frequency (e.g., tracing only a percentage of the transactions [Trushkowsky et al. 2011]).

7. CONCLUSIONS

In this article, we introduced TAS (Transactional Auto Scaler), a system designed to accurately predict the performance achievable by applications executing on top of transactional in-memory data grids, in face of changes of the scale of the system. Applications of TAS range from online self-optimization of in-production applications to the automatic generation of QoS/cost-driven elastic scaling policies and support for what-if analysis on the scalability of transactional applications.

TAS relies on a novel hybrid forecasting methodology that jointly utilizes analytical modeling and machine-learning techniques according to a divide-and-conquer approach: analytical models, based on queuing theory, are used to capture the dynamics of concurrency control/replication protocol, as well as to predict the effects on contention due to CPU utilization; black-box statistical techniques are instead used to predict performance of the network level.

We demonstrated the viability and high accuracy of the proposed solution via an extensive validation study based on synthetic and industry standard benchmarks deployed both on a private cluster and on two public cloud infrastructures (Amazon EC2 and FutureGrid).

REFERENCES

Ahmed Ali-Eldin, Maria Kihl, Johan Tordsson, and Erik Elmroth. 2012a. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proc. of the Workshop on Scientific Cloud Computing Date (ScienceCloud'12)*.

Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. 2012b. An adaptive hybrid elasticity controller for cloud infrastructures. In *Proc. of the Network Operations and Management Symposium (NOMS'12)*.

Amazon. 2013. Amazon S3. Available at http://aws.amazon.com/s3/.

Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of the Conference on Innovative Data System Research (CIDR'11)*.

Bela Ban. 2012. JGroups—A Toolkit for Reliable Multicast Communication. Available at http://www.jgroups.org.

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Proc. of the ACM SIGMOD International Conference on Management of Data*.

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1986. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman.

U. Narayan Bhat, Mohamed Shalaby, and Martin J. Fischer. 1979. Approximation techniques in the solution of queueing problems. *Naval Research Logistics Quarterly* 26, 2, 311–326.

Christopher M. Bishop. 2007. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer.

Jin Chen, Gokul Soundararajan, and Cristiana Amza. 2006. Autonomic provisioning of backend databases in dynamic content web servers. In *Proc. of the International Conference on Autonomic Computing (ICAC)*.

Bruno Ciciani, Daniel M. Dias, and Philip S. Yu. 1990. Analysis of replication in distributed database systems. *IEEE Transactions on Knowledge and Data Engineering* 2, 2 (1990), 247–261.

Yi Dai, Yunzhao Luo, Zhonghua Li, and Zhaojun Wang. 2011. A new adaptive CUSUM control chart for detecting the multivariate process mean. *Quality and Reliability Engineering International* 27, 7 (2011), 877–824.

Pierangelo di Sanzo, Bruno Ciciani, Roberto Palmieri, Francesco Quaglia, and Paolo Romano. 2012. On the analytical modeling of concurrency control algorithms for Software Transactional Memories: The case of Commit-Time-Locking. *Performance Evaluation* 69, 5 (2012), 187–205.

Pierangelo di Sanzo, Bruno Ciciani, Francesco Quaglia, and Paolo Romano. 2008. A performance model of multi-version concurrency control. In *Proc. of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'08)*.

Pierangelo di Sanzo, Roberto Palmieri, Bruno Ciciani, Francesco Quaglia, and Paolo Romano. 2010. Analytical modeling of lock-based concurrency control with arbitrary transaction data access patterns. In *Proc. of WOSP/SIPEW International Conference on Performance Engineering (ICPE'10)*.

Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *Proc. of the International Symposium on Distributed Computing (DISC'06)*.

Xavier Dutreilh, Sergey Kirgizov, Olga Melekhova, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. 2011. Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow. In *Proc. of the International Conference on Autonomic and Autonomous Systems (ICAS'11)*.

Sameh Elnikety, Steven Dropsho, Emmanuel Cecchet, and Willy Zwaenepoel. 2009. Predicting replicated database scalability from standalone database profiling. In *Proc. of the European Conference on Computer systems (EuroSys'09)*.

Saeed Ghanbari, Gokul Soundararajan, Jin Chen, and Cristiana Amza. 2007. Adaptive learning of metric correlations for temperature-aware database provisioning. In *Proc. of the International Conference on Autonomic Computing (ICAC'07)*.

Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. The dangers of replication and a solution. In *Proc. of the ACM SIGMOD International Conference on Management of Data*.

Leonard Kleinrock. 1975. *Queueing Systems*. Vol. I: Theory. Wiley Interscience.

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *SIGOPS Operating System Review*, 44, 2 (2010), 35–41.

Scott T. Leutenegger and Daniel Dias. 1993. A Modeling study of the TPC-C benchmark. In *SIGMOD Record* 22, 2, 22–31.

John Dutton Conant Little. 1961. A proof for the queuing formula: $L = \lambda W$. *Operations Research* 9, 3 (1961), 383–387.

London's Global University. 2013. Lattice Monitoring Framework. Available at http://clayfour.ee.ucl.ac.uk/lattice/.

Francesco Marchioni and Manik Surtani. 2012. *Infinispan Data Grid Platform*. Packt Publishing.

Daniel A. Menascé and Tatuo Nakanishi. 1982. Performance evaluation of a two-phase commit based protocol for DDBs. In *Proc. of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*.

Matthias Nicola and Matthias Jarke. 2000. Performance modeling of distributed and replicated databases. *IEEE Transaction on Knowledge and Data Engineering* 12, 4 (2000), 645–672.

Oracle. 2011. Oracle Coherence. Available at http://www.oracle.com/technetwork/middleware/coherence/overview/index.html.

Roberto Palmieri, Pierangelo di Sanzo, Francesco Quaglia, Paolo Romano, Sebastiano Peluso, and Diego Didona. 2011. Integrated monitoring of infrastructures and applications in cloud environments. In *Proc. of the 2011 international conference on Parallel Processing (Euro-Par'11)*.

Francisco Perez-Sorrosal, Marta Patiño Martinez, Ricardo Jimenez-Peris, and Bettina Kemme. 2011. Elastic SI-Cache: Consistent and scalable caching in multi-tier architectures. *VLDB Journal* 20, 6 (2011), 841–865.

John Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

John Ross Quinlan. 2012. Rulequest Cubist. Available at http://www.rulequest.com/cubist-info.html.

Jing Fei Ren, Yutaka Tokahashi, and Toshiharu Hasegawa. 1996. Analysis of impact of network delay on multiversion conservative timestamp algorithms in DDBS. *Performance Evaluation* 26, 1 (1996), 21–50.

Upendra Sharma, Prashant Shenoy, and Donald F. Towsley. 2012. Provisioning multi-tier cloud applications using statistical bounds on sojourn time. In *Proc. of the International Conference on Autonomic Computing (ICAC'12)*.

Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic resource scaling for multi-tenant cloud systems. In *Proc. of the ACM Symposium on Cloud Computing (SOCC'11)*.

Rahul Singh, Upendra Sharma, Emmanuel Cecchet, and Prashant Shenoy. 2010. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proc. of the International conference on Autonomic computing (ICAC'10)*.

Yong Chiang Tay, Nathan Goodman, and Rajan Suri. 1985. Locking performance in centralized databases. *ACM Transactions on Database Systems* 10, 4 (1985), 415–462.

Alexander Thomasian. 1998. Concurrency control: Methods, performance, and analysis. *ACM Computing Surveys* 30, 1 (1998), 70–119.

Steven K. Thompson. 2002. *Sampling* (3rd ed.). Wiley Desktop Editions.

TPC Council. 2011. TPC-C Benchmark. Available at http://www.tpc.org/tpcc.

Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. 2011. The SCADS director: Scaling a distributed storage system under stringent performance requirements. In *Proc. of the Conference on File and Storage Technologies (FAST'11)*.

Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. 2005. An analytical model for multi-tier internet services and its applications. *SIGMETRICS Performance Evaluation Review* 33, 1 (June 2005) 291–302.

Chris Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning* 8, 3 (1992), 279–292.

Greg Welch and Gary Bishop. 1995. *An Introduction to the Kalman Filter*. Technical Report 95-041. Department of Computer Science, University of North Carolina at Chapel Hill.

Jing Xu, Ming Zhao, José A. B. Fortes, Robert Carpenter, and Mazin S. Yousif. 2007. On the use of fuzzy modeling in virtualized data center management. In *Proc. of the International Conference on Autonomic Computing (ICAC'07)*.

Philip S. Yu, Daniel M. Dias, and Stephen S. Lavenberg. 1993. On the analytical modeling of database concurrency control. *Journal of the ACM (JACM)* 40, 4 (1993), 841–872.

Bin Zhang and Meichun Hsu. 1995. Modeling performance impact of hot spots. In *Performance of Concurrency Control Mechanisms in Centralized Database Systems.*, Vijay Kumar (Ed.), Prentice-Hall, 148–165.

Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. 2007. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proc. of the International Conference on Autonomic Computing (ICAC)*.

Benjamin Zhu, Kai Li, and Hugo Patterson. 2008. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. of the Conference on File and Storage Technologies (FAST)*.

Received May 2013; revised October 2013; accepted March 2014