Cloud-TM

Specific Targeted Research Project (STReP)

Contract no. 257784

D3.3: Performance forecasting models.

Date of preparation: 30 May 2012
Start date of project: 1 June 2010
Duration: 36 Months

Contributors

Bruno Ciciani,CINI
Maria Couceiro, INESC-ID
Joao Cachopo, INESC-ID
Diego Didona, INESC-ID
Pierangelo Disanzo, CINI
Stoyan Garbatov, INESC-ID
Roberto Palmieri, CINI
Sebastiano Peluso, INESC-ID
Luis Rodrigues, INESC-ID
Francesco Quaglia, CINI
Paolo Romano, INESC-ID

Table of Contents

1 Introduction

This deliverable describes the results achieved so far by the Cloud-TM Project in the design and validation of performance forecasting models of the key building blocks of the Cloud-TM Data Platform.

At the time of writing, the performance models presented in this deliverable are being integrated in the Adaptation Manager (to be presented in Deliverable D3.4), which will exploit them with the twofold purpose of automatically tuning the various layers of the Data Platform and of controlling the provisioning of resources from IaaS providers.

In order to frame and organize the presentation of the various performance forecasting models that we have developed so far, it is convenient to contextualize them by referring to the diagram in Figure 1. This diagram provides a high level overview of the Cloud-TM platform, and represents an evolved version of the Architectural diagram introduced in Deliverable D2.1.

As depicted in the diagram, the Cloud-TM Data Platform is structured into two main logical layers, the Data Platform Programming API and the Distributed Transactional Memory. In the remainder of this section we provide an overview of the performance models developed for each of these two layers.

1.1 Distributed Transactional Memory

The bottommost layer of the Cloud-TM Data Platform is a Distributed Transactional Memory based on Red Hat's Infinispan data grid. This component, which is the backbone of the Cloud-TM data platform, is in charge of regulating the concurrent data access performed by the distributed nodes composing the Data Platform, ensuring its consistency and high availability.

We developed performance models of the following key subcomponents of the Distributed Transactional Memory:

- **Group Communication System.** The Group Communication System sits at the bottom of the Distributed Transactional Memory's stack, providing a set of convenient abstractions for membership maintenance (e.g. failure suspicion, and handling of processes joining/leaving the platform [32]) and data dissemination (e.g. broadcast/multicast primitives with different reliability and ordering delivery guarantees [39]).

  The performance models developed so far in this area have targeted different message broadcast primitives (ordered vs non-ordered, reliable vs unreliable) and analyzed the effects of message batching, a common optimization technique that trades-off latency for maximizing throughput. The main performance forecasting methodology employed in this area has been machine-learning, although we have explored, for the case of batching, also the usage of analytical models.

  These performance models will be presented in Section 2.

- **Software Transactional Memory.** Each node of the Cloud-TM data platform maintains application's data within a Software Transactional Memory. The STM

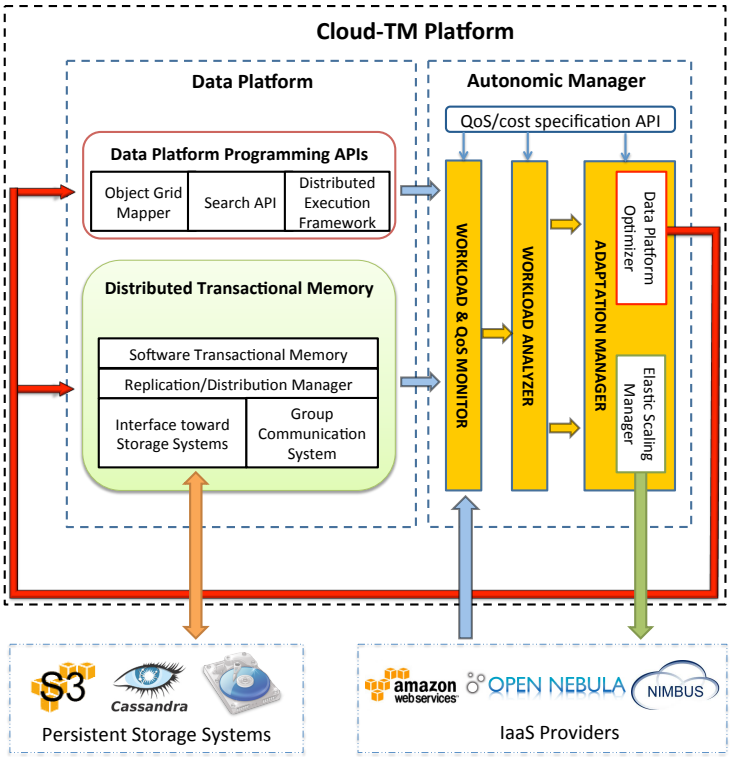


Figure 1: Architecture of the Cloud-TM platform.

is basically an in-memory data store that externalizes a transactional data manipulation API and uses, under the hood, a concurrency control algorithm to ensure transactional consistency. Over the project, Infinispan has been extended to support multiple concurrency control strategies, e.g. encounter or commit time locking [45], and different consistency levels, e.g. read committed/repeatable read/serializable [13]. This allows to configure Infinispan to deliver optimal performance in presence of application level workloads having heterogeneous characteristics in terms of conflict/locality dynamics and consistency requirements.

Our wok in this area has been aimed at defining an analytical framework that allows to simplify the development of performance models of arbitrary STM platforms by decoupling the modelling of the thread execution phases (i.e. executing transactional vs non-transactional code, or blocked in back-off following an abort) from the modelling of the STM-specific concurrency control algorithm. The proposed framework has been used with two concurrency control algorithms representative of different strategies for what concerns lock acquisition timing, namely Commit Time Locking (CTL) and Encounter Time Locking (ETL). CTL has been used in several state of the art STMs [24, 45], and has been included in a very recent release of Infinispan. Encounter Time Locking is the scheme

adopted in classic Two Phase Locking, and represented the default concurrency control strategy of Infinispan at the time of project's start.

These performance models will be presented in Section 3.

- **Replication/Distribution Manager.** This component is responsible for coordinating the transactional data manipulation activities of the distributed nodes of the Data Platform, implementing different data replication/distribution protocols optimized for different operational contexts for what concerns both the workload characteristics and the scale of the platform.

  Our research in this area has led to the development of performance models of different data replication and distribution strategies that have been, in parallel, implemented in Infinispan. These include Two-Phase Commit (2PC) [15], Total Order (TO) [39] and Primary-Backup (PB) [62] based replication strategies. Also in this case, we have explored the usage of both analytical models and machine learning approaches.

  These performance models will be presented in Section 4.

1.2 Data Platform Programming API

The Data Platform Programming API is the top-level layer of the Cloud-TM Data Platform, and represents the main interface exposed to application developers. It allows developers to create and manipulate an object-oriented domain model that is then mapped transparently to the underlying Distributed Transactional Memory. Therefore, all data accesses made by the application are ultimately reflected in the underlying layer.

Yet, because the mapping from objects to the underlying data grid is not fixed in the Cloud-TM platform and may affect the performance of the application significantly, depending on how data is accessed by the application, we need to be aware of those data access patterns to be able to optimize that mapping (either statically or dynamically).

Thus, our efforts in this area have targeted the construction of models that are capable of representing the data access patterns of an object-oriented application, and, more importantly, predicting which data will be accessed in the future at each given context. The models developed so far try to strike a good balance between the accuracy of the models and the overheads imposed by the construction and maintenance of those models at runtime. We have developed three types of models—based on Bayesian Inference, Importance Analysis, and Markov Chains—that model the access patterns at the class and class field's level only. In our evaluation, these models show good accuracy without incurring into excessive overhead.

These data access patterns shall be presented in more detail in Section 5.

1.3 Related publications

Most of the results described in this deliverable have been published in the following international conferences or journals.

1. Diego Didona and Paolo Romano and Sebastiano Peluso and Francesco Quaglia, Transactional Auto Scaler: Elastic scaling of NoSQL transactional data grids,

Proc. 9th International Conference on Autonomic Computing (ICAC), September 2012.

2. Decreasing Memory Footprints for Better Enterprise Java Application Performance. S. Garbatov and J. Cachopo. Proc. 23rd International Conference on Database and Expert Systems Applications, Vienna, Austria, Springer Verlag, September 2012.

3. Data Access Pattern Analysis and Prediction for Object-Oriented Applications. S. Garbatov and J. Cachopo. INFOCOMP Journal of Computer Science, December 2011.

4. Software Cache Eviction Policy based on Stochastic Approach. S. Garbatov and J. Cachopo. Proc. 6th International Conference on Software Engineering Advances (ICSEA 2011), October 2011.

5. Optimal Functionality and Domain Data Clustering based on Latent Dirichlet Allocation. S. Garbatov and J. Cachopo. Proc. 6th International Conference on Software Engineering Advances (ICSEA 2011), October 2011.

6. PolyCert: Polymorphic Self-Optimizing Replication for In-Memory Transactional Grids. M. Couceiro, P. Romano and L. Rodrigues, Proc. ACM/IFIP/USENIX 12th International Middleware Conference (Middleware 2011).

7. Self-tuning Batching in Total Order Broadcast Protocols via Analytical Modelling and Reinforcement Learning. P. Romano and M. Leonetti. ACM Performance Evaluation Review, to appear (also presented as a Poster at IFIP Performance 2011 Symposium).

8. On the Analytical Modeling of Concurrency Control Algorithms for Software Transactional Memories: the Case of Commit-Time-Locking. P. Di Sanzo, B. Ciciani, F. Quaglia, R. Palmieri and Paolo Romano. Elsevier Performance Evaluation Journal, vol.69, no.5, 2012.

9. Analytical Modeling of Commit-Time-Locking Algorithms for Software Transactional Memories. P. Di Sanzo, B. Ciciani, F. Quaglia, R. Palmieri and P. Romano. Proc. 35th International Computer Measurement Group Conference (CMG), Orlando, Florida, Computer Measurement Group, December 2010 (also presented in the 1st Workshop on "Informatica Quantitativa" (InfQ), Pisa, July 2010).

10. A Machine Learning Approach to Performance Prediction of Total Order Broadcast Protocols. M. Couceiro, P. Romano and L. Rodrigues. Proc. 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Budapest, Hungary, IEEE Computer Society Press, September 2010.

The reader interested in gaining further insights on the techniques presented in this deliverable is invited to refer to the above listed publications, as they contain additional information (e.g. additional experimental results or insights on the employed modelling methodology) with respect to that included in this deliverable.

1.4 Relation with other deliverables

The performance models presented in this deliverable are being currently integrated in the Adaptation Manager (which will be presented in Deliverable D3.4), which will exploit them with the twofold purpose of automatically tuning the various layers of the Data Platform and of controlling the provisioning of resources from IaaS providers.

The development of these performance models has had a strong impact also on the design and implementation of the Workload Monitor [113] and Workload Analyzer [114]. Throughout all the project, in fact, these modules have been (and still are being) extended to allow extracting and synthesizing the statistical information required for the instantiation of these performance models.

Finally, this deliverable has a strong relation with deliverables D2.2 [112] and D2.3 (which, at the time of writing, is due in five months), which have led to the development of the prototype of the Cloud-TM Data Platform. The performance models herein presented, in fact, aim at forecasting the dynamics of algorithms that have been, and in some cases are still being, integrated in the Data Platform. Indeed, whenever a stable prototype of a given component of the Cloud-TM platform was already available, we have been able to evaluate directly our performance models on those implementations. For those performance models that targeted functionalities not yet fully implemented in WP2 (e.g. some concurrency control algorithms), the experimental evaluation and validation was performed using publicly available prototypes already implementing those functionalities.

2 Group Communication System

The Group Communication System lies at the bottommost level of the Distributed Transactional Memory, and provides a set of fundamental building blocks for group membership [32] and data dissemination [39].

The performance models developed so far in this area have targeted different message broadcast primitives, specifically plain broadcast vs total-ordered broadcast algorithms. Focus has been placed on modelling the performance of these communication primitives as they represent key building blocks used by the data replication/distribution protocols supported by the Cloud-TM data platform. In fact, these performance models have been used as building blocks of the data replication schemes that will be presented in Section 4.

Additionally, we have explored the issue of mechanisms aimed at forecasting the effects on performance of message batching [54]. This is a simple, yet extremely effective optimization technique that is based on the idea of buffering messages for some time, so to be able to process multiple messages together. This allows amortizing the costs of processing each individual message in the batch, reducing the header overhead per message, the contention on the network and the CPU load [55], and allowing to boost the maximum throughput sustainable by the system.

The remainder of this section is structured as follows. In Section 2.1 we present the performance model for non-ordered broadcasts. Our work on the performance modelling of Total Order Broadcast protocols is presented in Section 2.2. Finally, Section 2.3 presents models capturing the performance of message batching.

2.1 Broadcast-based Synchronous RPC

Broadcast-based Synchronous RPC is a key building block of many distributed algorithms, and plays a central role in several Two Phase Commit (2PC) based data replication protocols.

2PC-based replication protocols are widely employed in distributed transactional plaforms, and, most importantly for the Cloud-TM project, they were the first data replication protocols to have been integrated into Infinispan. In these schemes, the first phase (also called prepare phase) of the 2PC protocol is executed by having the transaction coordinator initiate a broadcast-based synchronous RPC towards the set (or, in partial replication schemes, towards a subset) of replicas in the platform. Upon the delivery of the message, the recipients perform some protocol-dependant data validation activity and reply back to the coordinator specifying their vote on the transaction outcome (commit/abort).

In fact, 2PC-based replication protocols represented the key application that motivated us to investigate techniques capable of forecasting the network latency of broadcast-based synchronous RPCs.

The latency incurred in by a broadcast synchronous RPC is in practice affected by a number of factors related to the level of contention on different physical resources (network *in primis*, but also CPU and memory) that can be triggered both by workload's fluctuations and by the elastic re-sizing of the data platform. Developing white-box models capable of capturing accurately the effects on performance due to contention on

hardware resources can be very complex (or even non-feasible, especially in virtualized cloud infrastructures), given the difficulty to gain access to detailed information on the exact dynamics of such low-level components. This motivated us to pursue the exploration of black-box methodologies, using techniques developed by the machine learning community.

The problem of forecasting the network latency of a broadcast synchronous RPC, that we denote as T_{rpc}, can be seen as a non-linear regression problem, in which one wants to learn the value of a continuous function (that outputs forecasts for T_{rpc}) defined on a multivariate domain that comprises one variable for each input metric to the machine learner (also called feature). Clearly, an essential part of this solution consists in the identification of the key input metrics for the machine learner. Ideally, one would like to use the minimum set of features that are capable of accounting for most of the variations in the output variable. The well-known, curse of dimensionality problem [82], also called Hughes effect, in fact implies that the time for a machine learning technique grows exponentially with the number of dimensions, or, equivalently, that the predictive power reduces significantly as the dimensionality increases.

In the following we list the set of features we selected, motivating our choices.

- Number of nodes squared: we used this features, instead of the plain number of nodes since, with two-phase commit, the number of messages exchanged in the network grows quadratically in the number of nodes.
- Number of threads: this parameter is important to capture hardware resource demand.
- Used memory: this feature has been added since we experimented the system to react in different manners when loaded with data grids of different sizes, not only in terms of local execution time exhibited by the transactions, but also in terms of network latency (note that T_{net} encapsulates also the costs relevant to Infinispan's JAVA based group communication toolkit and not just the plain round trip time). This can be imputed to the garbage collecting operations and locality issues.
- Size of the broadcast message: this parameter is of course highly related to the time needed to pack and transmit messages over the network.
- The average number of RPCs executed 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.

In order to build an initial knowledge base to train the machine learner, we generated heterogeneous workloads by running Infinispan and configuring it to use a 2PC-based replication scheme. We relied on a suite of synthetic benchmarks that generate heterogeneous transactional workloads in terms of mean size of messages, memory footprint at each node and network load (number of transactions that activate 2PC per second). During the initial, off-line, 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 to achieve rather quickly (in about one hour) a satisfactory coverage of the parameters' space, which is the reason why we did not decide to integrate more advanced sampling mechanisms, like adaptive sampling [138].

We experimented with several machine learning techniques tailored for multi-variate regression problems, including various neural networks type and decision-trees. Our experiments revealed that the most accurate forecasts were achievable using Cubist, a decision-tree regressor that approximates non-linear multivariate functions by means of piece-wise linear approximations. Analogously to classic decision tree based classifiers, such as C4.5 and ID3 [117], 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.

Note that this machine learning technique requires off-line training, namely it can not update its knowledge base whenever a new sample is available. Nevertheless, periodically whenever sufficient new samples are gathered, one can re-train the machine learner to incorporate in its knowledge base profiling data specific to the recent response of the system to the workload generated by user level applications.

2.1.1 Validation

In this section we report the results of an experimental study aimed at evaluating the accuracy and viability of the above described machine-learning based performance models. Before presenting the results, we describe the workloads and experimental platforms used in our study.

Workloads. We consider two well-known benchmarks, namely TPC-C [140] and Radargun. 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 ones, but have the advantage of being very easily tunable, thus allowing assessing the accuracy of the performance models in a wider range of workload settings.

For TPC-C we consider two different workload scenarios: i) a read dominated workload (containing 90% of read-only transactions) which generates reduced contention on both physical and data resources as the scale of the cluster grows; ii) a workload that includes around 50% of update transactions and generates a high data contention level. Also for Radargun we consider two workloads. Both workloads generate uniform data access patterns, but yield very different data contention probabilities.

Experimental Platforms. We use, as experimental test-beds for this study, both a private cluster and Amazon EC2. The private cluster is composed by 10 servers equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) processors and 8 GB of RAM and

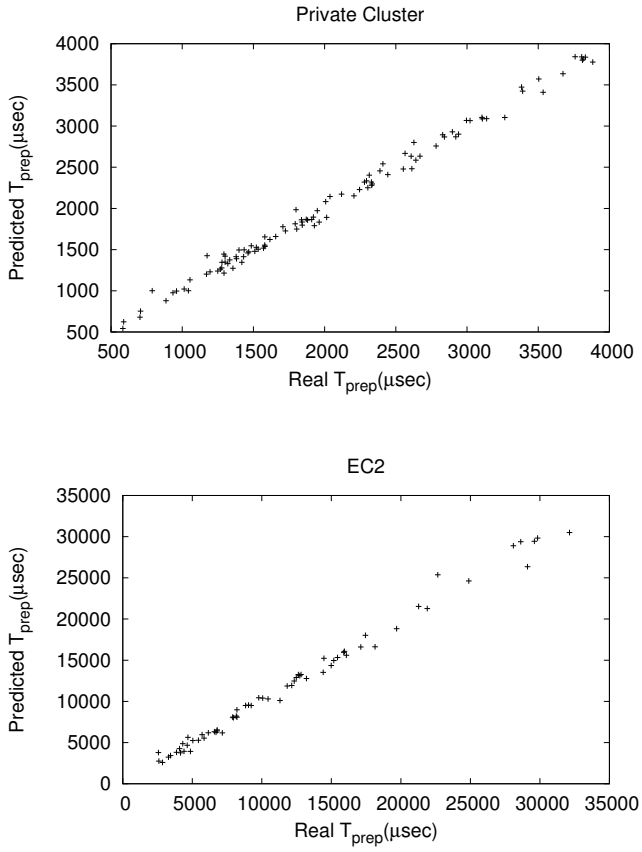


Figure 2: Accuracy of the machine-learning based T_{net} predictions on Private Cluster (left) and on EC2 (right)

interconnected via a private Gigabit Ethernet. For EC2 we used up to 20 Extra Large Instances, which are equipped with 15GB of RAM and 4 virtual cores with 2 EC2 Compute Units each.

Model Accuracy. The scatter-plots in Figure 2 report the results obtained using the machine learning based models built according to the methodology described in Section 2.1, and testing them against the workloads described earlier in this section.

The results highlight that, on both the private cluster and on EC2, the decision-tree based performance forecasting model attains a high prediction accuracy. Specifically, we found the correlation factor to be around 99% in both cases, with an average absolute error equal to 500 micro-seconds for EC2 and around 60 micro-seconds for the private cluster.

Interestingly, the relative error on both platforms appeared to be very similar, de-

spite, on EC2, the maximum value of T_{net} is around 10 times larger than the maximum T_{net} value on the private cluster.

As a final remark, it is noteworthy to highlight that, in all our experiments, the performance attained with or without the monitoring framework necessary to gather the statistics required by the machine learning-based models were indistinguishable. Also, the time required to query a decision tree-based model is on order of less than a millisecond, highlighting the practical viability of the proposed solution.

2.1.2 Integration in the Cloud-TM platform

These performance models, and the corresponding tools for gathering at run-time the statistical information that these models require as input, have been developed, since their conception, targeting Infinispan.

At the moment of writing these models are being integrated in the Cloud-TM autonomic manager, where they will be used as building blocks both by the Data Platform Optimizer (e.g. to determine the optimal replication algorithm given the current workload characterisitcs) and by the Elastic Scaling Manager (to foresee the effects on horizontal scaling on the network latencies incurred when running 2PC-based replication protocols).

2.2 Total Order Broadcast Protocols

Total Order Broadcast (TOB) [39] is a fundamental building block for developing strongly consistent replicated systems. TOB greatly simplifies the development of fault-tolerant systems by ensuring that messages are delivered in the same order despite variable communication delays and replica failures and hiding the issues associated with enforcing replica agreement on the streams of updates to be applied by the applications. Total Order Broadcast is, in fact, at the heart of the classic, general-purpose, active replication scheme [129]. Furthermore, it has also been employed to design efficient protocols for, e.g., database systems [110] and transactional memories [35].

Over the last decades, a wide body of literature has been devoted to the design and evaluation of TOB protocols (extensively surveyed by Defago et al. [39]). However, we are not aware of any work proposing engineering methods and tools capable of providing real-time, fine-grained (i.e. on a per message basis) forecasts of the performance of TOB protocols, when deployed in real systems and subject to complex workloads.

In this following we report the results of the work carried out in the context of the Cloud-TM project on the usage, to the best of our knowledge for the first time in literature, of using machine learning techniques to derive fine-grained (i.e. on a per message basis) performance prediction models of TOB protocols. The ability of our approach to forecast the TOB's latency on a per message basis makes it an extremely useful building block for architecting self-optimizing replication schemes [124]. This is a promising research area, and one of the goals pursued in the Cloud-TM project, which is at current still largely unexplored precisely because of the lack of effective TOB protocol's performance predictors.

We start by presenting a semi-opaque self-monitoring architecture that relies on the tracing of a basic set of protocol independent performance metrics, which can be possibly augmented with protocol specific context information in a modular fashion via the use of standard programmatic interfaces. Our generic (i.e. protocol independent) monitoring tools track the usage of system resources (such as network bandwidth, CPU and memory) across multiple time scales spanning several orders of magnitude. This allows to combine information representative of stationary phenomena (captured by long term averages) as well as transient burstiness (captured by short term averages) which can significantly affect the latency of the ongoing TOB.

We carried out an extensive experimental study using:

- three machine learning methods, namely neural networks [65], support vector regression [131], and regression decision trees [116]
- three highly heterogeneous and demanding (in terms of amount of injected traffic) workloads, consisting of a synthetic traffic generator that allows us to widely span in the workload's parameter space, and two complex applications running on top a distributed software transactional memory platform [35] which generate high contention on the computational and memory resources locally available at each node.
- two different TOB algorithms relying on radically different approaches for establishing agreement on the delivery order (centralized vs distributed) and aiming at optimizing distinct performance metrics (latency vs throughput).

Our results highlight that the set of context information (also called features in the machine learning literature and in the remainder of this document) that maximizes the machine learners accuracy varies significantly when one considers heterogeneous, realistic workloads. We also evaluate to what extent incorporating time series, protocol dependant information and garbage collection metrics can allow enhancing the accuracy of the machine learners.

We then focus on the issue of feature selection, a problem of combinatorial nature which becomes rapidly intractable in scenarios, such as the one evaluated in this study, characterized by a large abundance, and redundancy, of input variables. Our experimental data highlights that, while being certainly more efficient than a exhaustive exploration of the feature space, existing heuristics approaches [63] to the feature selection problem still have prohibitively high execution times. This can represent a major impairment in scenarios demanding frequent re-training of the performance predictors, due to, e.g., workload fluctuations or alterations of the group size caused by failures or dynamic expansions/contractions triggered by spikes of the load pressure. To tackle this issue we propose and evaluate two alternative solutions:

1) An optimized search heuristic, whose search trajectory in the features' power set is drastically restricted with respect to classical, general purpose, greedy search heuristics. This is achieved by exploring exclusively the combinations of features which were found to generally maximize the accuracy of the machine learners. Such a specialization allows reducing the feature selection execution time on average by a factor 10x at a negligible cost in terms of accuracy degradation ($<2\%$) across the whole spectrum of considered workloads.

2) A technique based on the ensemble of a small set of models, each one relying on different (and largely non-overlapping) subsets of features, and whose predictions are combined on the basis of the expected confidence intervals of the individual models in operating in the corresponding region of their feature space. When compared with classical greedy heuristics for feature selection, this ensemble technique allows boosting feature selection by two orders of magnitude, at the cost of an average 10% degradation of the prediction accuracy.

2.2.1 System Overview

The architecture of our system is depicted in Figure 3. Our reference architecture includes a distributed application, such as the Cloud-TM platform (or more specifically the replication manager of Cloud-TM's Distributed Transactional Memory, namely Infinispan) which is supported a group communication service (GCS) (such as JGroups or Appia [95]). The system is augmented with a monitoring layer and a latency predictor, which are the key contributions of our work. Before detailing the description of these components, we will first provide a brief overview of the interdependencies among the system components and discuss some key principles underlying their design.

In our system, each node develops, in an independent and fully distributed fashion, predictive models of the TOB latency as observed by applications residing on that same node. More specifically, the latency predictor component provides the client of the TOB service with a latency estimator. The predictor is able to forecast the time it takes to self-deliver (after being totally ordered) a message of a given size sent by the

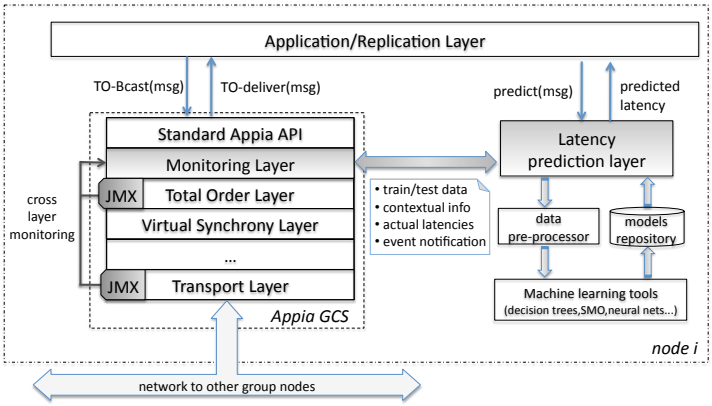


Figure 3: Architectural Overview (Single Node Perspective).

application (the size of the message is passed to the estimator as an input parameter).

The monitoring layer is the component responsible for collecting training data for the machine learners, as well as to provide the latency predictor with information concerning the actual workload characteristics and resource utilization levels. Additionally, it provides feedback to the latency predictor component on the accuracy of its forecasts, as well as notifications on the occurrence of relevant changes in the system configuration that may affect the quality of the currently employed predictive model, for instance, a change in the number of active replicas (as the performance of TOB is typically a function of the number of participants).

Monitoring Layer. Our monitoring layer has been developed for the Appia [95] protocol kernel, that implements a GCS in JAVA. Appia follows an architectural design that allows to compose layered stacks of micro-protocols according to the application needs. The flow of information among the layers of the Appia stack is supported by the exchange of events that are propagated upwards and downwards through the stack. In Appia, this flow of events is regulated by a single, dedicated thread which we will refer to in the following as Event Scheduler (ES) thread.

The monitoring mechanisms are implemented as a layer that can be transparently disabled/enabled at run-time, ensuring that there is no monitoring overhead outside the tracing phase. As depicted in Figure 3, the monitoring layer is placed between the TOB later and the interface with the application. This allows, on one hand, to achieve total transparency for the application and, on the other hand, to straightforwardly trace any event generated by or delivered to the application. Thus, the monitoring later is able to intercept TO broadcast/delivery events and events notifying of changes in the group membership. As noted before, membership changes may have a significant impact on the TOB performance, thus they can be used to trigger the generation of a new performance model.

When the monitoring layer is enabled, it collects information on the following basic set of metrics:

1) *Network related metrics:* moving averages across multiple time scales of i) the number of TO Broadcast/Delivery events, and of ii) the amount of bytes sent/received by the TO layer; additionally it keeps track of the number of TO Broadcast events generated by the application layer and for which it has not been generated the corresponding TO Delivery event yet.

2) *CPU related metrics:* moving averages across multiple time scales of the total CPU utilization, and of the CPU utilization of the Appia's ES thread.

3) *Memory related metrics:* the free memory in the Java Virtual Machine (JVM), as well as two metrics describing the activity of the JVM's Garbage Collector (GC) thread namely, i) the time occurred since the last garbage collection cycle, and ii) the percentage of time elapsed since the last garbage collection cycle with respect to the time between the last two garbage collection cycles. Note that, since there is no standard Java API to directly track the status of the GC thread, to trace the GC activity in a portable manner we extend the *finalize()* method[1] of a dummy object to keep track of the time in which the GC thread is activated (and re-instantiate the dummy object).

In addition to the above context information, the monitoring layer has been designed to support also cross-layer tracing in an elegant and modular fashion. Specifically, at system's bootstrap, and upon any alteration of the Appia stack, the monitoring layer queries the whole set of Appia's layers via the standard Java Management Extensions (JMX) interface to determine whether there are any layers that externalize information related to their internal state that could be exploited by the machine learners to generate more accurate performance models. JMX, which is part of the J2SE specification since version 5, lends itself perfectly to this purpose, defining standard interfaces that allow the developers of a software component to advertise the list of the component's monitorable attributes and to associate with each one of them a human readable textual description. We leverage on this JMX feature to allow Appia layers' developers to specify which attributes, among those monitorable via the JMX interface, should be traced by our monitoring layer as deemed potentially beneficial to enhance the machine learners' accuracy. As we will further discuss in Section 2.2.2, in our experimental analysis we exploit this mechanism to monitor the number of outgoing message queued at the Transport Layer, as well as to track the internal state of a TOB algorithm. We report the whole set of metrics gathered by the monitoring layer in Table 1.

Whenever a TO broadcast event for message m is intercepted by the monitoring layer, the latter takes a snapshot of the current state of the context information. As soon as the monitoring layer intercepts the TO deliver event for message m, it determines the self-delivery latency, logs the associated context information namely the context information at m's sending time along with its self-delivery latency) asynchronously to a memory buffered file and propagates the TO deliver event upwards. The choice of measuring exclusively the self-delivery latencies allows to circumvent the issue of ensuring accurate clock synchronization among the communicating nodes, which would have clearly been a crucial requirement in case we had opted for monitoring the delivery latencies of messages generated by different nodes. Preliminary experiments

[1]The *finalize()* method of an object is invoked by the GC thread when it determines that no more references to that object exist.

Metric	Description
freeMem	Free memory in the Java Virtual Machine
tLGC	The time since the last garbage collection
pLGC	% of time since the last GC cycle w.r.t. the time between the last 2 GC cycles
undelivMsgs	#TO Broadcast msgs and not yet self-delivered
bytesUp_x	#Bytes received over a x msec. time window
bytesDown_x	#Bytes sent over a x msec. time window
TOBUp_x	#TOB deliver events over a x msec. time window
TOBDown_x	#TOB broadcast events over a x msec. time window
totCPU_x	% total CPU utilization over a x msec. time window
esCPU_x	% CPU utilization by ES thread over a x msec. time window
esCPU_x	% CPU utilization by ES thread over a x msec. time window
TCPqueue	Outgoing messages queued at the Transport Layer *(protocol dependant metric traced via JMX interface)*
toTime	Elapsed time since the token was last owned *(protocol dependant metric traced via JMX interface)*

Table 1: List of metrics collected by the Monitoring Layer.

conducted in our cluster have indeed highlighted that the accuracy achievable using conventional clock synchronization schemes, such as NTP, is often inadequate for collecting meaningful measurements of the TO broadcast inter-nodes delivery latency, being the latter frequently around or even under a millisecond.

Latency Predictor. The latency predictor is responsible for forecasting the performance of the TOB layer when broadcasting a message of a given size. In order to build the performance models used to output such forecast, the latency predictor layer triggers the pre-processing of the training data collected by the monitoring layer. In this phase, the training data is prepared (properly filtered and manipulated, e.g., to generate time-series) to allow its successful processing by the chosen machine learning tool. Our system architecture in fact supports the modular integration of alternative machine learning libraries.

The models output by the machine learners are stored in a model repository, where they are associated with metadata that captures the context in which these models were built (such as the number of machines participating in the TOB group and the TOB algorithm employed while generating the training data). Furthermore, the models are ranked (e.g. for feature selection purposes) and made available to the latency predictor layer for generating performance predictions. Note that the performance models output by the machine learners take as input features not only the size of the message to be broadcast, but also a (possibly quite large) set of system metrics. These are obtained by querying at run-time the monitoring layer. The latter makes also available information on the actual self-delivery latencies of recently broadcast messages, which can be used by the latency predictor to assess the accuracy of its predictions and possibly trigger the construction of a new model.

2.2.2 Validation

At current date, we have integrated in our system two machine learning tools, namely Rulequest's Cubist© [115] and Weka [53], enabling the user to choose which one to

use. We now briefly overview these two tools.

Cubist© is a decision tree based regression commercial tool developed by Quinlan, the author of C4.5 [117] and ID3, two popular decision tree based classifiers. Analogously to these algorithms, Cubist© builds decision trees choosing the branching attribute such that the resulting split maximizes the normalized information gain (namely the difference in entropy). Unlike C4.5 and ID3, which contain an element in a finite discrete domain (i.e. the predicted class) as leafs of the decision tree, Cubist© places a multivariate linear model at each leaf. An appealing characteristic of Cubist© is that the decision tree can be reformulated as set of human-readable rules, where each rule identifies a region in the feature space. Also, each rule contains a multivariate linear model in the "then" clause and is associated with the expected average error in the prediction. Since Cubist© generates a piecewise regression (each multivariate linear model being applicable under certain rules), it can be more powerful than a simple multivariate linear model as it allows variables to be weighted differently as conditions change. When two rules overlap, the predicted values by using the models associated with each model are averaged with a weight that depends on the degree of confidence in the prediction generated by the two rules.

Weka is an open-source framework providing a common interface to a large number of machine learning algorithms. In this work we will evaluate two major regression techniques, namely, Neural Networks [65] and Support Vector Machines [131]. These methods are well-known and have been extensively described in the machine learning literature, so, due to space constraints, we will only briefly overview them. The neural network algorithm implemented in the Weka framework trains a multi-layered network using the classic back-propagation algorithm [125] to determine the weights that minimize the local error at each perceptron. We used the default configuration in Weka, which generates a number of hidden layers equals to half the number of input features. Concerning the Support Vector Machine technique, we also rely on the default configuration of the Weka's SMOreg package, which uses a polynomial kernel whose parameters are learnt using the algorithm in [131].

Workload Description. For our experimental study we will consider the following three workloads:

Synth: this is a synthetic benchmark which injects traffic at each node following a regular and homogeneous pattern. On each node we run a single application level thread which TO broadcasts, in intervals lasting 30 seconds each, messages of growing size, namely {100, 200, 500, 1K, 2K, 5K, 10K, 20K, 50K, 100K, 200K, 500K} bytes, at an increasing sending rate, namely {1, 2, 10, 20, 50, 100, 125, 166, 333, 500, 1000} messages per second. For each configuration (message size / sending rate), we collect at most 90 messages, which will then be the input of the machine learning tool.

RBTree: this workload is generated by D^2STM [35], a distributed software transactional memory platform, running the Red Black Tree benchmark. D^2STM uses a TOB-based distributed certification scheme; TOB is used propagate the readset and writeset of local transactions, and ensure that all replicas validate transactions in the same common order. The Red Black Tree [69] is a well-know benchmark for the evaluation of software transactional memories, in which a red black tree data structure is concurrently updated (by inserting and/or removing items) by several threads.

The benchmark was ported to run on the D^2STM platform and configured to generate transactions entailing a variable number of operations. Note that this has the effect of further increasing the heterogeneity of the generated workload: as the number of operations issued by each transaction varies over time, the frequency of generation of TO broadcasts, and the size of the TO broadcast messages (which encode the transactions' readset and writeset) also vary accordingly. Unlike the Synth benchmark, in RBTree each replica hosts a variable number of threads performing computational intensive tasks before issuing a TO broadcast. This scenario is therefore characterized by a much higher contention among (GCS and application) threads on the local resources (CPU in primis). We will see that this has a significant impact on the predictability of the GCS performance. The data set used to train and test the machine learner consists of the messages exchanged in a 15 minute run, which is the time needed to collect the training data across the whole range of generated workload.

STMBench7: this workload is also generated by D^2STM, running STMBench7 [60], a complex benchmark which manipulates an object-graph with millions of objects, featuring a number of operations with different levels of complexity. This benchmark includes both very short and very long-running transactions; the latter traverse hundreds of thousands of objects and generate extremely large readsets and writesets. As a consequence, the workload for the TO service entails both very short (on the order of few hundreds of bytes) and very large messages (on the order of several megabytes). Also, as transactions need to store in memory their readset and writesets, long-running transactions end up stressing significantly the memory system of the local JVM, triggering frequent garbage collection cycles. Like in the RBTree benchmark, this benchmark allows running multiple concurrent application level threads in each node. Each run of this benchmark lasts around 30 minutes in order to ensure that, independently of the number of machines and threads, the collected entries in the data set is always approximately 12.000.

Evaluated TOB Algorithms. In our experiments, we consider two classic, well-known TOB algorithms [39]. The first one is a sequencer-based algorithm in which a single node, called the sequencer, determines the order according to which all nodes have to TO deliver messages. The second one is a token-based algorithm which ensures agreement on a common TO delivery order by circulating among the nodes a token that grants the right to broadcast messages.

The choice of these two algorithms was aimed at maximizing the diversity, with the ultimate purpose of widening the representativeness of our testbed. The above TO algorithms, in fact, rely on extremely different approaches for establishing agreement on the delivery order (centralized vs distributed), aim at optimizing different performance metrics (latency vs throughput), and have complementary pros and cons [39].

Analysis of the Results. In the following we present the results of our experimental study. We will initially focus on the analysis of the results obtained using Cubist© and only subsequently move to compare the performance of the Neural Network and SMO methods. All the results reported in the following were obtained using a tested of nodes equipped with an Intel QuadCore Q6600 at 2.40GHz with 8 GB of RAM running Linux 2.6.27.7 and interconnected via a private Gigabit Ethernet.

The accuracy of the machine learners is measured using the following metrics. Relative Average Error (RAE), which compares the performance of the predictor with that of a naive predictor that simply outputs the average value of the training data. When comparing two different models M_1, M_2 we will rely on the Normalized Additional Mean Absolute Error (NAE), defined as $NAE = \frac{MAE_{M_1} - MAE_{M_2}}{Lat_{avg}}$, namely the difference between the Mean Absolute Error (MAE) of model M_1 (Lat_1) and the MAE of model M_2 (Lat_2) normalized by the average value of the delivery latency in the test set data (Lat_{avg}). The NAE is a scale-free metric that we deem as being more informative than a simple comparison between the MAEs of M_1 and M_2. In fact, small differences between the MAEs are irrelevant in case the delivery latencies are, on average, large and, vice versa, large differences between the MAEs become relevant only if, on average, delivery latencies are large.

What features to use? A crucial challenge that has to be faced for accurately predicting the performance of any complex system via machine learning techniques is to carefully identify the set of metrics/context information to be used as input variables for the model construction [63].

One of the first problems that we had to address while building our system was related to the difficulty to identify an optimal time window for computing the moving averages concerning the percentage of utilization of CPU and network resources. Our experimental results accentuated the fact that the choice of the wrong time window could significantly affect the machine learners' accuracy. This phenomenon is clearly highlighted by the first two rows in Table 2, which compares the accuracy of the predictions when using specific features when the Token algorithm is being used to disseminate messages in a group of 4 machines. The fact that the Synth workload is stable over relatively long periods of time explains the 25% decrease in accuracy (measured through the NAE) when using a time window of 6 msec rather than 50 msec. On the other hand, an opposite result is obtained when considering the RBTree workload, where the accuracy decreases by 32% when using 500 msec, rather than 10 msec, time windows. This can be explained considering that shorter time windows are more sensitive to transient burstiness phenomena; this can be beneficial in presence of highly variable workloads, such as RBtree, but disadvantageous in the case of more stable workloads, such as for Synth. These results have led us to the choice of computing the moving averages across multiple time windows, ranging from 2 up to 500 msecs, and of relying on feature selection scheme to filter out the ones that turned out to be uninformative or misleading for the machine learner.

Another interested finding highlighted in Table 2 is related to the relevance of the GC related metrics. The third row of the table reports a degradation of the model's accuracy of 29% for the Synth workload when the metric that indicates the time elapsed since the last garbage collection cycle (tLGC in Table 1) is not used (and otherwise using the same set of features).

These experiments have also highlighted the usefulness of incorporating time series information in the set of features used by the machine learners. To this end we pre-process the training data generated by the monitoring layer in order to include, in the set of features provided to the machine learner, the latencies of the last k TO broad-

Worst vs Best Set of Feat.	Benchmark	NAE
6 vs 50 time window	Synth (m4)	25%
500 vs 10 time window	RBTree (m2t3)	32%
without vs with GC	Synth (m4)	29%
without vs with time serie	RBTree (m2t3)	81%
without vs with token	RBTree (m2t1)	61%

Table 2: Models' Accuracy Increase (NAE) with Selected Features.

casts self-delivered by that node. As shown by the 4th row of table 2, when time series information is not employed, the Cubist's© predictions accuracy increases by 81% in the Synth workload scenario. This is due to the fact that, especially in the less fluctuating workloads, there is often a significant correlation among the delivery latencies of recently broadcast messages.

Finally, the last row in Table 2 reports a 61% decrease in the accuracy for the RBTree workload when the machine learner is not provided with information concerning the elapsed time since the token was owned by the node for the last time (toTime in Table 1). In the token-based algorithm, in fact, the delivery latency is strongly affected by the time elapsed before the token is owned by the sending node, and the latter is closely correlated with toTime. Overall, these results confirm the relevance of our semi-opaque monitoring approach, which provides the TOB layers' developers with standard interfaces to instruct the monitoring layer to trace protocol-dependant state information.

Based on the above analysis, we identified a total number of 53 relevant features (43 of which being directly traced by our monitoring layer, and 10 additional ones used to build the time serie on the delivery latency). When faced with such an abundance (and redundancy) of available metrics, it is easy to fall prey of the, so called, curse of dimensionality [97]. As the number of dimensions in the feature space (i.e. the degrees of freedom of the performance model to be built by the machine learner) increases, in fact, the amount of training data required to ensure an equivalently dense sampling coverage of the feature space grows exponentially. This makes the machine learners much more exposed to the risk of overfitting [46], a phenomenon in which the machine learner infers erroneous dependencies among random features of the training data with no causal relation to the target function, with the result of increasing their accuracy in fitting known data (hindsight) while actually degrading the accuracy in predicting new data (foresight).

Note that the feature selection problem is of combinatorial nature, as identifying optimal solutions would entail exhaustive searching the powerset of the feature set. This motivated the design of a number of alternative heuristic approaches that enhance efficiency at the cost of not achieving optimality. In the machine learning literature, greedy algorithms are probably the most used for implementing feature selection. There are two variants of this approach: forward selection (FS) and backward elimination (BE). In FS, features are progressively added to build larger models, whereas in BE one starts with the set of all features and progressively eliminates the least promising ones. At each iteration, one feature is added/removed and cross-validation is used to identify the

	Sequencer							
	2 Machines				4 Machines			
	1Thread		3Threads		1Thread		3Threads	
	COR	RAE	COR	RAE	COR	RAE	COR	RAE
Synth	1.00	0.01	-	-	1.00	0.20	-	-
RBTree	0.44	0.42	0.63	0.39	0.69	0.37	0.77	0.42
STMB7	0.44	0.37	0.64	0.35	0.67	0.35	0.54	0.36

Sequencer

	Token							
	2 Machines				4 Machines			
	1Thread		3Threads		1Thread		3Threads	
	COR	RAE	COR	RAE	COR	RAE	COR	RAE
Synth	1.00	0.01	-	-	1.00	0.01	-	-
RBTree	0.80	0.31	0.95	0.12	0.57	0.49	0.84	0.30
STMB7	0.65	0.38	0.71	0.36	0.49	0.51	0.74	0.42

Token

Table 3: Correlation Coefficient and Relative Absolute Error of Cubist© using Forward Selection.

	FS	**BE**	**OSH**	**ENS**
# Models Built	401	484	72	7
Time (sec)	250	579	53	2.1

Table 4: Average Execution Time of Feature Selection Algorithms.

best performing subsets of features, i.e. a 60% of the available data is used to build a model and the remaining 40% is used as test data to evaluate the model's accuracy, together with a two-fold cross-validation. Both approaches stop when adding/removing one more feature to/from the remaining set of features no longer improves accuracy.

We report in Table 3 the Relative Absolute Error and correlation coefficient achieved by using the FS heuristic across all the considered workloads (we omit report results for BE as they are extremely close to those achieved by FS). The plots are relative to scenarios where the number of machines varies between 2 and 4, and the number of threads in the RBtree and STMBench7 benchmarks vary from 1 to 3. Results show that the prediction accuracy clearly depends on the complexity of the considered workload. When considering the Synth workload, the accuracy and correlation of the predictor output are extremely high, even if this workload is highly heterogeneous and encompasses phases where nodes generate both very low and high network traffic. This happens because, in each phase, the fluctuations of the delivery latencies are rather limited. The reasons for the observed stability in each phase are twofold. First, nodes are very lightly loaded, not running any computational or memory intensive tasks. Second, nodes send messages at the same rate, which makes the performance of the GCS in each phase rather stable.

The other two considered workloads are, on the other hand, definitely more challenging. First, the applications lack a well defined traffic pattern and second, nodes execute computational intensive applications. Together, these factors induce a strong variance in the self-delivery latencies. As a result, even though the set of features se-

	Sequencer							
	2 Machines				4 Machines			
	1Thread		3Threads		1Thread		3Threads	
	OSH	ENS	OSH	ENS	OSH	ENS	OSH	ENS
Synth	0.2%	0.7%	-	-	2.5%	14.2%	-	-
RBTree	0.2%	3.4%	1.8%	20%	0.2%	14.2%	5.3%	14.1%
STMB7	0.4%	0.7%	2.1%	19.1%	0.3%	3.9%	3.1%	18.3%

Sequencer

	Token							
	2 Machines				4 Machines			
	1Thread		3Threads		1Thread		3Threads	
	OSH	ENS	OSH	ENS	OSH	ENS	OSH	ENS
Synth	7.4%	7.5%	-	-	10.7%	10.6%	-	-
RBTree	3.8%	4.1%	1.6%	2.5%	7.2%	11.7%	5.4%	8.3%
STMB7	6.4%	15.3%	11.1%	17.6%	2.6%	3.6%	20.4%	22.9%

Token

Table 5: Normalized Additional Mean Absolute (NAE) Error using OSH and Ensemble.

lected by the FS algorithm varies significantly for each workload, the performance of the predictors is similar across the two workloads, with an average correlation factor of 66% and an average RAE of 37%.

The results collected with these two workloads show an interesting trend, namely, the correlation generally grows, on average, from 59% to 73% when moving from scenarios with one thread to scenarios with three threads. This correlation is mainly due to the fact that in this context the undelivMsgs feature (which captures the number of threads that are blocked waiting for the self-delivery of a message, see Table 1) becomes extremely useful in this context. Indirectly, this metric provides a measure of congestion in the system. On the other hand, this feature is only meaningful when there are at least two application level threads, as it is constantly equal to 0 in case there is a single application level thread. This explains the increase in the model's correlation as the number of threads increase.

Boosting Feature Selection. Unfortunately, despite being significantly more efficient than exhaustive searches, the FS and BE heuristics still demand the construction of hundreds of models (see the first two columns from left of Table 4) and require execution times on the order of the hundreds of seconds even on a (currently) very fast machine equipped with two quad-core 2.33 Ghz Intel Xeon processors, 4 GB of RAM and running Linux 2.6.27[2]. These costs are clearly prohibitive in scenarios where models may have to be re-built rapidly to adapt to workload fluctuations.

To circumvent the limitations above, we propose and evaluate the following two techniques:

OSH: An Optimized Search Heuristic (OSH) that evaluates only combinations

[2]Interestingly, the average performance of BE is significantly worse (by a factor 2.3) with respect to that of FS even though the latter builds, on average, only 20% less models than than the former. This is due to the fact that the models built by BE have, on average, a larger number of features with respect to those explored by FS, and that the time taken by Cubist© to build a model is strongly affected by the number of features it uses.

of features that were pre-selected based on a preliminary exhaustive experimentation across the whole spectrum of workloads with classical statistical tools, such as Primary Component Analysis [109], and cross-validation testing. This preliminary phase allowed us to identify and discard the combinations of features whose usage either provided negligible increases, or even deterioration of the prediction accuracy.
OSH explores a total of 72 different models built by using as input features, a common set of attributes (namely, msg_size, TCPqueue, and undelivMsgS) and the combinations obtained by picking exactly one item from the following sets:
T={latency of the last TO broadcast, latencies of the last 5 TO broadcasts, latencies of the last 10 TO broadcasts}; M={no memory information, freeMem, freeMem and tLGC, freeMem and pLGC}; R={moving_avgs$_x$}, where moving_avgs$_x$ denotes following metrics {bytesUP$_x$, bytesDOWN$_x$, TOBUp$_x$, TOBDown$_x$, totCPU$_x$, esCPU$_x$} computed over the same time window of duration x msecs, where $x \in$ {2,6,10,50,100,500}.

Ensemble: An ensemble of independent models built over largely non-overlapping sets of features and whose predictions are reconciliated on the basis of their estimated confidence interval. The intuition underlying this approach is that models built using diverse set of features have the potentiality to capture distinct phenomena affecting the delivery latency of TOB algorithms with different degrees of accuracy. In addition, by focusing each model on a smaller subset of attributes, they are less prone to suffer of overfitting problems. Further, by selecting the prediction generated by the model with the highest degree of confidence in the current region of the feature space, our ensemble technique may enhance the accuracy of each independent model.
Our ensemble technique generates 7 models, where each model uses the same common set of attributes as in OSH (msg_size, TCPqueue, and undelivMsgS), but differs from the other ones as it uses either i) a time-series containing the last k (where we set $k = 10$) TOB latencies, or ii) moving_avgs$_x$ computed as before. In preliminary experiments we have evaluated several alternative methods for conciliating the predictions provided by the various models. We only report results for the best performing strategy, which is based on the simple approach of selecting the prediction associated with the smallest confidence interval.

As expected, by evaluating a much smaller number of models, OSH and Ensemble achieve striking performance gains, reducing feature selection time up to two orders of magnitude (see Table 4).

On the other hand, table 5 reports data quantifying to what extent the quality of the predictions deteriorate when using OSH and Ensemble with respect to the case in which FS is used. The accuracy of OSH is extremely close to that of FS, being its average NAE around 2.2%. Concerning Ensemble, the average NAE increase is larger, namely around 10%. We argue that, in practical settings, this (limited) degradation of the prediction accuracy is largely compensated by the significant performance gains it achieves. On the other hand, in these experiments, we relied on Cubist's© estimates of the confidence intervals, whose details are unfortunately not publicly available. An interesting open research question is whether the accuracy of the Ensemble technique could be enhanced by leveraging on alternative techniques, e.g. [74], for the computa-

# Machines	# Threads	Overhead (%)
2	1	5.71
2	3	5.21
4	1	2.63
4	3	2.23

Table 6: Average overhead due to Monitoring Layer

tion of the predictions' confidence intervals.

2.2.3 Additional Performance Considerations

In Table 6 we report the average overhead (measured in terms of TOB throughput reduction) due to the tracing activities carried out by the Monitoring Layer with a different number of machines/threads. The numbers show that the overhead is in practice very limited, being always less than 5%, and decreasing to 2% in the case of four machines. This can be explained by considering that, as the number of nodes in the system increases, the TO delivery latency also accordingly grows. In a closed model, such as the one characterizing both the RBTree and the STMBench7 benchmarks, this leads to a reduction of the frequency of TO broadcast issued by each node and, consequently, of the number of messages traced by the Monitoring Layer in a given time window.

In Figure 4 we analyze to what extent the size of the training data set affects the model's prediction accuracy and building time. The plots report the data associated with the model that uses the features selected by the FS technique when running the STMBench7 and Synth benchmarks, progressively reducing the training data set with respect to the size reported in Section 2.2.2. Note that the plotted data is obtained by averaging across all the configurations (machines, threads, TOB protocol) described in Section 2.2.2.

The plots highlight an expectable, but relevant trade-off: the model building time can be significantly reduced by using smaller training data sets, at the cost of an increased prediction error. Specifically, our experimental data shows that we could have achieved a very similar prediction accuracy (1%, resp. 10%, higher RAE for the Synth, resp. STMBench7), by using 50% smaller data sets, boosting the model building time by a larger than 2x factor. The selection of the training data set size, therefore, represents a key tuning knob that, when used in combination with the previously described Ensemble technique, can further reduce the overhead required to derive the TOB performance prediction model. Nevertheless, as also confirmed by our experimental data, the optimal choice of the training data set size is highly workload dependent and is a time consuming process which is typically performed offline. Extending our system to automatize this process is an important research direction that will be pursued in our future work.

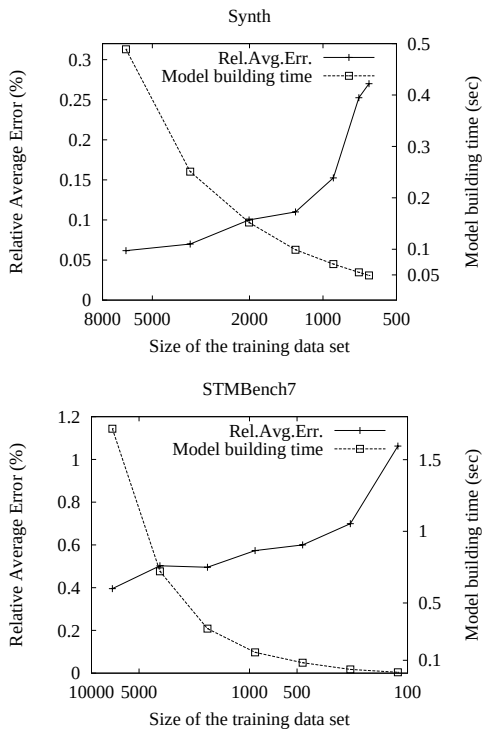


Figure 4: Evaluating model's building time and accuracy while varying the training data set's size.

2.2.4 Alternative Machine Learners

Finally, we compare the performance and accuracy of the models obtained by using Cubist© with those generated by two other machine learning algorithms available in Weka, namely a Multilayer Neural Nework (Neural) and a Support Vector Machine regression (SMO) method. The reported results are obtained providing the same set of features as input to all machine learners, namely those selected by running the FS scheme with Cubist©.

Figure 5 reports the NAE between Neural and Cubist©, and SMO and Cubist©, averaged across all the three considered workload. By the plot, we see that Cubist© significantly outperforms both Neural and SMO across almost every workload with the exception of the scenarios where sequencer-based protocol is evaluated using 2 machines. In these cases, the Weka's machine learners in fact achieve a lower Minimum Absolute Error with respect to Cubist©, determining an inversion of the trend highlighted by the plot. It is interesting to note, however, that in these scenarios the correlation of Neural and SMO (not shown in the plots) is significantly worse (around the half) than that of Cubist©. These differences can be motivated by considering that different machine learning approaches are known to optimize distinct metrics [97] and by referring to the well-known "No free lunch theorem" [146], which states that the performance of no single machine learner can be optimal across all possible scenarios.

As a final remark, we compared the training time of the various machine learning tools when using data set containing approximately 7.200 training cases. We trained each model 100 times in order to ensure the statistical representativeness of the collected data. The results are strongly in favour of Cubist©, which on average takes 0.64 seconds to build a model, whereas the average time for completing the training phase for Neural and SMO was, respectively, 243 and 575 seconds. Albeit the performance difference is quite striking, it is not completely surprising considering that Cubist© is a commercial, and highly optimized performance tool (written entirely in C and parallelized to take advantage of multi-core CPUs), whereas Weka is an open-source framework designed to simplify development and testing of novel machine learning methods rather than fine-tuned for performance purposes.

2.2.5 Related Work

Existing performance evaluation and modelling studies of TOB [38, 47] (and related agreement problems, consensus in primis [34]) have been aimed at providing a steady state estimate of the average performance of several TOB protocols in presence of simple synthetic workloads. Typically, the purpose is to identify the most favourable settings for each of the considered algorithmic alternatives. Also, due to the inherent complexity of TOB protocols, the only analytical models of TOB we are aware of [34, 47] make rather stringent assumptions on the workload, e.g. symmetric Poissonian traffic sources generating messages at the same rate. In these works, the system model is also simplified, using synthetic (constant or exponential) communication latency distributions, that neglect important factors such as the impact of the message size on the observed latency. To the best of our knowledge, our work represents the first attempt to leverage on machine learning methods for assessing the performance of

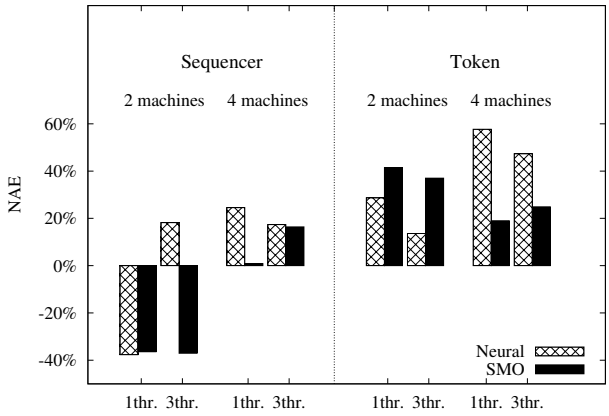


Figure 5: NAE of Neural and SMO with respect to Cubist©.

TOB protocols. Unlike existing analytical/simulation models, we leverage on statistical methods to automatically build fine-grained TOB performance models capable of forecasting in real-time the delivery latency perceived by user level applications on a per message basis.

Our work is clearly related to the machine learning literature addressing performance prediction of computer systems. These include works aiming at forecasting the throughput of TCP flows [96] and Pub-Sub systems [56], solutions aimed at automatizing the allocation of resources in cloud-computing infrastructures [149], and at generating software aging models to be used in the context of rejuvenation frameworks [4].

The idea of ensembling different machine learning models to enhance the performance of single predictors has been widely investigated in general contexts [97], as well as applied to predict performance failures of complex systems [153]. The latter work ensembles models based on the same set of features but representative of different phases of the life cycle of applications, and dynamically selects the model which better matches the current load scenario by ranking them based on the Brier score. Conversely, our ensemble technique ensemble models built using largely disjoint sets of features capturing different aspects of the application workload with different degrees of accuracy.

Finally, our work is related with the body of literature addressing the issue of identifying the most informative attributes to be used by machine learners. In addition to the already mentioned greedy search techniques, such as Backward Elimination or Forward Selection [63], we can mention also techniques, e.g. [150], aimed at clustering highly correlated attributes for a preliminary screening of redundant metrics, or at identifying the set of attributes accounting for the greater variability in the output variable, such as Primary Component Analysis [109] and Projection Pursuit [141]. Unfortunately, these methods do not provide a direct indication of the actual accuracy achievable by the machine learner and rely on a set of input parameters (e.g. the frac-

tion of the total variability of the output variable should be accounted when evaluating the feature set) whose optimal settings may be not trivial to determine.

2.2.6 Integration in the Cloud-TM platform

The work on machine learning-based performance models has started in a very early phase of the project (in fact, this work has been published around 6 months after the project's starting date in [36]). As at this time, the reference platform for the Cloud-TM project was yet to be agreed upon, we opted for using one of the state-of-the-art Java-based open-source Group Communication Systems, namely the Appia GCS [95], which had been designed and implemented, a few years before, by some of the INESC-ID team members.

The choice of adopting Infinispan as reference platform for the Cloud-TM's DTM, however, implies also an implicit choice for the reference GCS of the project. Infinispan, in fact, is tightly integrated with JGroups, an other open-source project by Red Hat. Indeed the Cloud-TM project contributed directly to the development of a new Total Order multicast algorithm, which has been integrated in JGroups during the second year of our project.

Note that, given that machine-learning models operate on a black-box basis, porting our methodology and tool to a different platform is pretty straightforward. In fact, it is simply required to feed the statistics on resource utilization/performance that we were originally gathering from Appia from JGroups. We are confident that this re-basing will be relatively straightforward given that JGroups already exposes abundant statistics via JMX. At the moment of writing, the work to migrate our machine learners to operate on top of JGroups has already started, and we plan to have it completed by the time we will deliver D3.4, namely the prototype of the Autonomic Manager.

2.3 Message Batching in Total Order Broadcast Protocols

Message batching [54] (a.k.a. message packing [8] or message aggregation [28]) is a simple, yet very effective optimization technique, that is based on the idea of buffering messages for some time, so to be able to process multiple messages together. This allows amortizing the costs of processing each individual message in the batch, reducing the header overhead per message, the contention on the network and the CPU load [55]. Indeed, several works have highlighted the striking impact of batching in boosting system throughput. On the other hand, at low load, waiting for additional messages to form a batch induces unnecessary stalls that can hamper the performance of delay sensitive applications, e.g. interactive, real-time applications.

The problem is exacerbated in the presence of dynamic, fluctuating workloads. In this (in practice quite common) case, the optimal batching factor, or equivalently batch waiting time, actually varies over time, making any static configuration policy clearly suboptimal.

Batching is a particularly important technique for a very popular class [71, 84] of Total Order Broadcast (TOB) protocols, namely Sequencer-based TOB algorithms (STOB). This class of algorithms relies on a single node, called the sequencer, to impose a total order on the stream of messages broadcast by the group of processes. STOB algorithms have their key strength point in that they are optimal in terms of the number of communication steps necessary to establish the total order [85]. On the down side, their main limitation is that their maximum throughput is upper bounded by the capacity of the sequencer to generate sequencing messages. Interestingly, the sequencer capacity can be greatly increased by using an adequate batching level.

At current date, however, the problem of how to self-tune the batching level in STOB protocols is largely unexplored. The only solutions we are aware of, in fact, are far from being fully satisfactory as they depend on the accurate, manual, tuning of different kinds of system parameters [54]. Hence, rather than solving the problem of tuning the batching level, they actually replace it with the problem of manually configuring some different system parameter.

During the Cloud-TM project we have attempted to fill this important by gap by building performance predictors of STOB algorithms using, in a synergic fashion, analytical modeling and Reinforcement Learning (RL) techniques. The joint use of these two techniques allows to take the best of the two worlds.

By exploiting the knowledge of a queuing-theory based mathematical model, we can drastically abate the training time required by standalone RL techniques. This has a fundamental impact not only on the time required to achieve optimal performance, but also, and perhaps more importantly, on the stability of the system at high loads. In these scenarios, the lack of initial knowledge on the system's performance would force solutions based on plain RL to explore, with equal probability, the whole range of possible batching configurations. Unfortunately, at medium-high throughput, the usage of excessively small batching configurations has the effect of overloading the sequencer, and destabilizing the whole group communication system, even for short periods of time.

In addition, by complementing the analytical model with a RL mechanism, we can rely on a computationally efficient, analytical model. The unavoidable prediction errors

of the model can be corrected over time, by accumulating feedback from the operation of the system. We cast the learning problem in the context of *regret minimization* for *multi-armed bandit problems* [6]. This is a fundamental problem of the RL area, in which an agent is faced with a *bandit* (gambling machine) with multiple levers, each one associated with an unknown stochastic return. We break the optimization task into several sub-problems by discretizing the possible loads (incoming messages per second) into a number of ranges. The decision of choosing a batching level is faced separately for each load range. Specifically, given a load range, we see each batching level as a lever of a *bandit* (gambling machine). We leverage recent results on bandit problems to face the traditional exploration-exploitation trade-off, in a robust and efficient manner. Such a trade-off is determined by the necessary balance between using the best batching level determined at any given time, and the need to try other ones to assess its optimality.

2.3.1 The Analytical Performance Model

The design of the analytical performance model used in our system has been driven by two key requirements: i) high computational efficiency, thus allowing to compute its solution also in real-time; ii) possibility to identify its parameters via an automatized and extremely fast procedure, thus making it easily employable in practical settings also by non-specialists.

These two requirements led us to opt for a relatively simple model that captures the impact of batching on self-delivery latency focusing on two main aspects: the additional delay introduced by forcing the sequencer to wait for the completion of a batch of messages, and the alleviation of the load pressure on the CPU of the sequencer due to the generation of a reduced number of sequencing messages. We choose to explicitly model the possible effects of contention on the network, as typical applications that use TOB services in a LAN [35, 111] generate messages large at most a few KBs. In these settings, the sequencer's CPU remains the bottleneck even at very high batching values.

We model the CPU of the sequencer as a $M/M/1$ queue [79] for which each job corresponds to a batch of messages of size b. We denote with $\lambda(b, m)$ the arrival rate of a batch of b messages given that the TO-Bcast rate is equal to m, and with $\mu(b, m)$ the average rate at which a batch of size b is sequenced given that the TO-Bcast rate is m. We can then express, by using well known queuing theory results, the STOB self-delivery latency as the response time of the queue:

$$T(b, m) = \frac{1}{\mu(b, m) - \lambda(b, m)} \tag{1}$$

subject to the stability constraint: $0 \leq \lambda(b, m) < \mu(b, m)$. We take into account the effects of batching by expressing $\lambda(b, m)$ and $\mu(b, m)$ in Equation 1 as follows:

$$\lambda(b, m) = \frac{m}{b} \tag{2}$$

where we denoted with m the average message arrival rate to the sequencer, and accounted for the fact that, when using a batching value b, the sequencer processes

batches at a rate inversely proportional to b.

$$\mu(b,m) = \frac{1}{T_{1st} + \frac{(b-1)}{2m} + T_{add}(b-1)} \tag{3}$$

which expresses $\mu(b)$ as the inverse of the sum of three components:

- i) the CPU time, denoted as T_{1st}, required to generate a sequencing message for a single incoming message, or, equivalently, for a batch containing a single message;
- ii) the time required to receive the $b-1$ remaining messages of the batch;
- iii) the CPU time required to process the $b-1$ remaining messages of the batch. This typically corresponds to unmarshalling each incoming message and buffering it in memory, but it does not include the actual sending of the sequencing message (which is accounted for by T_{1st} in our model). Note that, in practice, the CPU time required to include an additional message into an existing batch (T_{add}) is much lower than the one associated with sending the sequencing message for a message (T_{1st}). In the following we will therefore assume that $T_{add} < T_{1st}$.

By merging Equations (1-3) we can derive the average self-delivery latency as a function of the batching level and of the average message arrival rate:

$$T(b,m) = \frac{1}{\frac{1}{T_{1st}+\frac{(b-1)}{2m}+T_{add}(b-1)} - \frac{m}{b}} \tag{4}$$

subject to the constraint:

$$\frac{1}{T_{1st} + \frac{(b-1)}{2m} + T_{add}(b-1)} - \frac{m}{b} > 0 \tag{5}$$

By letting b tend to infinity in the above constraint, we can easily determine an upper bound on the maximum throughput, m^* sustainable by the sequencer:

$$m^* = lim_{b\to\infty} \frac{b+1}{2T_{1st} + 2T_{add}(b-1)} = \frac{1}{2T_{add}} \tag{6}$$

Finally, by imposing

$$\begin{cases} \frac{\partial T(b,m)}{\partial m} = 0 \\ \frac{\partial^2 T(b,m)}{\partial^2 m} > 0 \end{cases}$$

we can derive the optimal batching value as a function of the message arrival rate m, as reported in Equation 7, where we used the shorthand $\sigma = \frac{1}{T_{1st}}$. The optimal batching value is predicted via a piece-wise function. For low values of the arrival rate, as expectable, the model predicts that batching harms performance, rather than improving it. As the load increases, as long as it remains lower than the maximum sustainable

throughput, the optimal batching value grows non-linearly and has a vertical asymptote at $m = m^*$.

$$b^*(m) = \begin{cases} 1, \quad \text{if } m < \frac{T_{add}\sigma^2}{2} + \frac{1}{2}\sqrt{\frac{4\sigma^2+2T_{add}^2\sigma^4}{2}} \\ \\ \frac{2m-\sigma-2mT_{add}\sigma}{\sigma-2mT_{add}\sigma+\sqrt{\frac{2(\sigma+2m(T_{add}\sigma-1))^2}{(2\sigma T_{add}-1)^2(1+2\sigma T_{add})\sigma^2}}}, \\ \quad \text{if } \frac{T_{add}\sigma^2}{2} + \frac{1}{2}\sqrt{\frac{4\sigma^2+2T_{add}^2\sigma^4}{2}} < m < m^* \end{cases} \tag{7}$$

Determining the Model's Parameters. In order to solve our analytical model, it is first necessary to determine the value of the parameters T_{1st} and T_{add}. Even though the semantics of T_{1st} and T_{add} might not appear immediately manifest, it is in practice possible to estimate their values via the following, very quick, training phase. To compute T_{1st} it suffices to observes that, for $b = 1$, Equation 4 reduces to:

$$T(1, m) = \frac{1}{\frac{1}{T_{1st}} - m}, \text{ where } m < \frac{1}{T_{1st}}$$

In other words, T_{1st} is simply the inverse of the maximum throughput sustainable by the system when $b = 1$, and its value can therefore be easily computed (in an approximated manner of course) by setting the batching level to 1 and injecting traffic at an increasing rate, until saturating the GCS.

In order to derive the value of the T_{add} parameter we exploit Equation 6. To this end, we set b to the maximum value that we intend to use in our self-tuning system, which we denote with b_{max} and identify the maximum throughput sustainable by the system, denoted as $m^*_{b_{max}}$. In all our experiments we found that the throughput's gains achievable by increasing the batching value over 128 become negligible, thus we used $b_{max} = 128$. Using Equation 6, we can then set $T_{add} = \frac{1}{2m^*_{b_{max}}}$.

The techniques for finding the values of the model's parameters described above have been implemented as simple application level benchmarking tool developed in JAVA which, in all of our experiments, required between 20 and 30 seconds to complete its execution.

2.3.2 Validation of the Analytical Model

In order to determine the accuracy of the presented analytical performance model, we performed an extensive experimental study aimed to manually identify the optimum batching value as a function of the message arrival rate. We used, as our experimental testbed, a cluster of 10 machines equipped with two Intel Quad-Core XEON at 2.0 GHz, 8 GB of RAM, running Linux 2.6.32-26-server and interconnected via a private Gigabit Ethernet. In this experiment, we let the batching level b vary in the set {1,2,3,4,6,8,16,32,64,128} and, for each batching level, we injected 512 bytes messages at an arrival rate ranging from 1 msg/sec up to saturating the GCS. In all of the

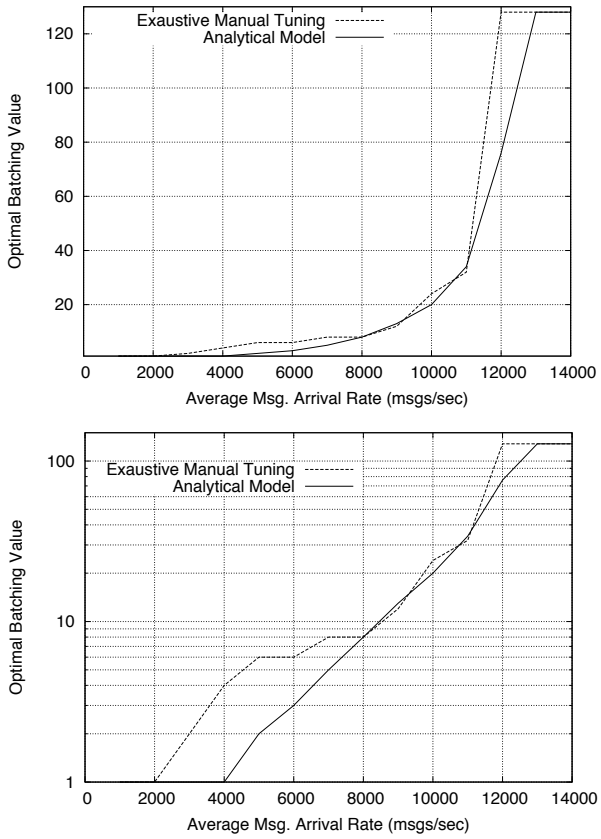


Figure 6: Validating the accuracy of the analytical model.

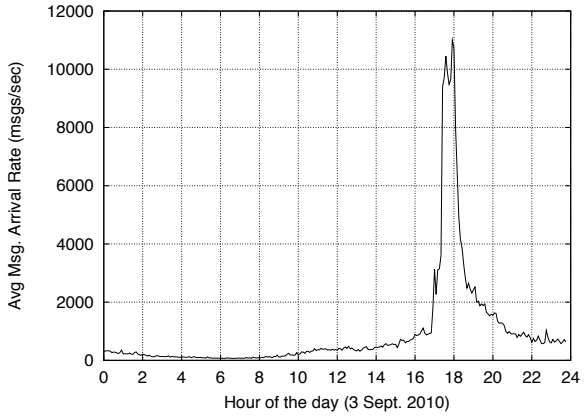


Figure 7: Traffic at the FenixEDU system (Sept. 3 2010)

experiments, independently of the batching level used, it was verified that the bottleneck has always resulted to be the sequencer CPU, and the available network bandwidth was always far from being saturated.

In Figure 6 we compare the optimal batching value predicted by the analytical model with the one manually found via exhaustive exploration of the parameters' space. The two plots report the same data, but the one on the bottom uses a log scale on the y-axis. By looking at the top plot, we observe that globally the model captures quite closely the dynamics of the real system. The logarithmic scale plot, on the other hand, allows to better visualize that the analytical model tends to underestimate the optimal settings of the batching value at medium loads (at approx. 3000-8000 msgs./sec).The fact that the proposed model fails under some circumstances comes indeed with no surprise, as real systems are characterized by some degree of uncertainty that no model can escape, and the proposed model relies on assumptions whose correctness can only be hypothesized.

In order to assess the actual impact of the model's error on system's performance we inject traffic using the traces collected by a real system, namely FenixEDU, the Web application that is responsible for the management of the whole campus of one of the main universities in Portugal, the Instituto Superior Técnico of Lisbon. The traces we used report the number of messages in input to the cluster hosting the FenixEDU system during September 3, 2010. In this day, at 18:00, the enrolment of students for the following semester started, and the FenixEDU system was subject to a spike of load lasting several hours, see Figure 7.

We injected traffic according to the traces from 16:00 to 22:00, and used the analytical model's prediction to dynamically adapt, with a frequency of 1Hz, the batching level. Figure 8 shows in the bottom plot the message arrival rate, and in the top plot the self-delivery latency at the sequencer node (in log scale). It is possible to note that the self-delivery latency is subject to two spikes: one during the ramp-up front of the traffic surge, and one during the ramp-down front, and both occurring as the average message arrival rate is of around 5000-6000 messages per second. As already dis-

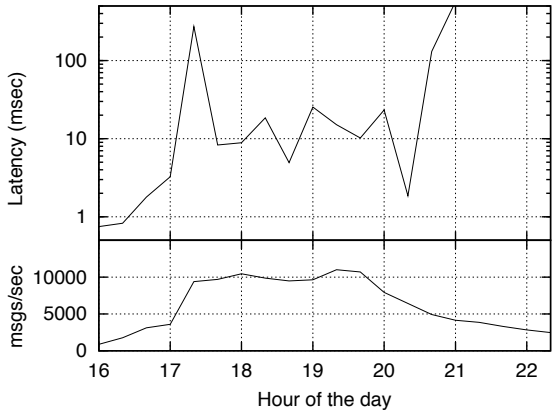


Figure 8: Self-tuning based on analytical model

cussed, at this load pressure, in fact, the analytical model underestimates the optimal batching level, which led the system to thrashing. Interestingly, since the ramp-up front is faster than the ramp-down, the system is able to recover from the thrashing caused by the incorrect choice of the batching level during the ramp-up. As soon as the ramp-up is completed, at high load, the analytical model predicts correctly the optimal level of batching, avoiding the GCS from crashing. Conversely, since the ramp-down lasts for a much longer time period (several hours vs around 30 minutes), the prolonged permanence in a state in which the batching level is erroneously tuned (namely configured with an excessively low value) causes the GCS to eventually collapse.

Concerning the computation efficiency of the analytical model, we implemented it in Java pre-computing every expression of Equation 7 that is independent of the message arrival rate m. We measured the time required to solve the model by passing as input parameter m the whole set of integer values in the range [1,14000], and repeated the process 100 times. On a machine equipped with an Intel Core 2 Duo at 2.53GHz running Mac OS X 10.6.6, the average time to determine the optimal batching value was in the order of 20 nanoseconds.

As a final note, it is noteworthy to highlight that, in preliminary experiments, we have also experimented with the usage of analytical models more complex than the above described one. These include models in which the (average) time to sequence a batch of size b at load m (essentially the denominator of Equation 3) is expressed as a generic polynomial of b. This kind of approaches could in theory allow for capturing more complex non-linear dynamics. However, they require a complex and time consuming phase of identification of the model parameters via non-linear fitting techniques [9]. Also, in our experiments, their accuracy resulted only marginally better than the one achievable by the model presented in Section 2.3.1. These considerations have ultimately led us to opt for adopting a simpler, and consequently easier to instantiate and solve, analytical model, whose errors are dynamically compensated via the usage of a RL technique, as we discuss in the following.

2.3.3 Combining a RL Approach

In order to compensate the errors of the analytical model, we investigated the possibility of using a RL technique aimed to dynamically update the initial knowledge provided by the model based on the feedback gathered by observing the consequences of the self-tuning choices. To this end, we cast the problem of deciding the optimal batching level given the current system load to a classical RL problem, namely the multi-armed bandit [122]. In this problem, a gambling agent is faced with a bandit (a slot machine) with k arms, each associated with an unknown reward distribution. The gambler iteratively plays one arm per round and observes the associated reward, adapting its strategy in order to maximize the average reward. Formally, each arm i of the bandit, for $0 \leq i \leq k$, is associated with a sequence of random variables $X_{i,n}$ representing the reward of the arm i, where n is the number of times the lever has been used. The goal of the agent is to learn which arm i maximizes the criterion: $\mu_i = \sum_{n=1}^{\infty} \frac{1}{n} X_{i,n}$, that is, achieves maximum average reward. To this purpose, the learning algorithm needs to try different arms in order to estimate their average reward. On the other hand, each suboptimal choice of an arm i costs, on average, $\mu^* - \mu_i$, where μ^* is the average obtained by the optimal lever. Several algorithms have been studied that minimize the *regret* r, defined as $r = \mu^* n - \sum_{i=1}^{K} \mu_i E[T_i(n)]$, where $T_i(n)$ is the number of times arm i has been chosen. In our system we leverage on a recent result of Auer et al. [6], who introduced an algorithm, UCB, that achieves a logarithmic bound on the number of suboptimal trials not only in the limit, but also for any finite sequence. Building on the idea of *confidence bounds* this algorithms create an overestimation of the reward of each possible decision, and lowers it as more samples are drawn. Implementing the principle of *optimism in the face of uncertainty* the algorithm picks the option with the highest current bound.

In particular, assuming that rewards are limited in [0,1], each arm is associated with a value:

$$\overline{\mu_i} = \overline{x_i} + \sqrt{\frac{\log n}{n_i} min\{1/4, V_i(n_i)\}} \tag{8}$$

where $\overline{x_i}$ is the current estimated reward for arm i, n is the number of the current trial, n_i is the number of times the level i has been tried, and:

$$V_i(s) = |\frac{1}{s} \sum_{\tau=1}^{s} X_{i,\tau}^2 - \overline{x}_i^2| + \sqrt{\frac{2 \log n}{s}}$$

The right-hand part of the sum in Eq. 8 is an *upper confidence bound* that decreases as more information on each option is acquired. By choosing, at any time, the option with maximum $\overline{\mu_i}$, the algorithm searches for the option with the highest reward, while minimizing the *regret* along the way.

In order to apply this technique, we discretized the parameters space, defined by the cartesian product $b \times m$, as follows. We considered $k = 8$ different batching levels, denoted as $b_1, .., b_k$ and such that $b_i = 2^i$ for $0 \leq i < 8$. We split the message arrival rate into $l = 15$ intervals, denoted as m_l, having endpoints in $L = \{0, 10, 100\} \cup \{n * 1000 | 1 \leq n \leq 10\} \cup \{12000, 14000, 16000\}$ expressed in messages per second. Each message arrival rate interval m_l is associated with an instance of the bandit problem

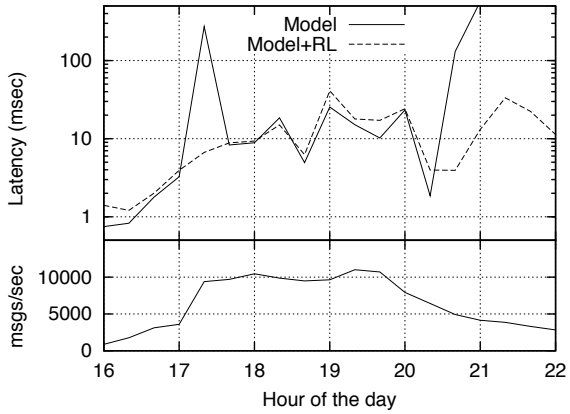


Figure 9: Self-tuning combining RL and analytical model

with k arms, where each arm is associated with a different batching level. Since in UCB rewards are bound in the [0,1] interval, given an observed self delivery latency t, we use the following function to defining its reward $R(t)$:

$$R(t) = \frac{maxLatency - min\{maxLatency, t\}}{maxLatency}$$

where $maxLatency$ is a parameter defining the maximum self-delivery observable by the sequencer that we set to 100 msec (a threshold above which our GCS started thrashing severely).

As already mentioned, the original UCB technique does not rely on the availability of initial knowledge on the arms' reward distribution. In the application domain considered in our study, however, the blind initial exploratory phase undertaken by UCB has severe consequences on system's stability. At high loads (i.e. at more 8000 messages per second in our cluster), in fact, the GCS starts thrashing after a few seconds if the batching level is not adequately tuned. This makes a plain UCB-based self-tuning technique extremely unstable, and, de facto, unusable in practice.

We tackle this issue by initializing the statistics of every arm of each UCB instance with the self-delivery latency predicted by Equation 4 of the analytical model. Figure 9 reports the performance achieved by the combined usage of the UCB-based RL technique and the analytical model, when considering the same trace-driven workload already used in Section 2.3.2. In the plot we also report the performance achieved by the self-tuning mechanism relying solely on the analytical model. These experimental data allow us to make several interesting considerations.

At high loads, the initial knowledge provided by the analytical model allows the RL method to avoid exploring inadequately low batching values which would rapidly lead the GCS to thrashing: for message rate values larger than 10000 msgs/sec, in fact, the initial rewards for batching values lower or equal than 16 are all identically null. At medium loads, where the analytical model incurs in the biggest errors, destabilizing the system, the RL method is able to rapidly update its initial, incorrect knowledge,

ensuring predictable performance and globally enhancing the robustness of the self-tuning mechanism. Finally, at extremely high and low load values, where the analytical model always guesses perfectly the optimal batching value, the exploratory behavior of RL leads to a slight deterioration of performance. This is an unavoidable cost that has to be incurred by any RL based system. However, since the UCB technique ensures that the regret increases at most logarithmically, the performance deterioration imputable to suboptimal exploratory behaviors is expected to become negligible over time.

2.3.4 Related Work

Packing small messages into larger ones to maximize performance is a well known optimization that is commonly employed in several domains [28, 54, 55]. TCP Nagle's algorithm [100] represents a noteworthy, widely deployed example of such a technique.

The effects of batching on the performance of TOB protocols was first studied empirically in [55] and later mathematically in [28]. To the best of our knowledge, the work in [54] is the only one to have investigated the issue of designing self-tuning mechanisms for TOB protocols. Unfortunately, the techniques proposed in this work require the explicit setting of additional parameters, e.g. the duration of timers used to wait for messages to be batched, thus failing to fully automatize the tuning of the batching mechanism.

Our work is also related to performance evaluation and modelling studies of TOB [38, 47] (and related agreement problems, consensus in primis [34, 126]). Instead of deriving a full performance model of the (S)TOB algorithm, the analytical model presented in the following is restricted to capturing exclusively the effects of batching on the CPU utilization of the sequencer node, being designed to serve a different, and more specific purpose.

Machine learning techniques have already been used to predict the performance of computer systems in several contexts. These include solutions aiming at forecasting the throughput of TCP flows [96], Pub-Sub systems [56], and Atomic Broadcast protocols [37], at automatically classifying traffic based on semi-supervised learning techniques [49], at automatizing the allocation of resources in cloud-computing infrastructures [149], and at generating software aging models [4].

2.3.5 Integration in the Cloud-TM platform

As for the performance models described in Section 2.2, and for the same timing reasons that were illustrated in Section 2.2.6 this modelling work has targeted the Appia platform. We are currently working to re-base these models on top of the JGroups protocol stack and plan to include them in the prototype of the Autonomic Manager (Deliverable D3.4).

3 Software Transactional Memory

The Cloud-TM project makes large of use in-memory transactional technologies, namely Software Transactional Memory (STM), specifically supported by the Infinispan data layer. As a consequence, one starting point in order to build performance prediction tools able to capture dynamics related to an in-memory, distributed transactional data platform, is the determination of performance prediction methods/approaches suited for capturing relevant dynamics at the level of each individual transactional node within the distributed platform. A key issue along this research path is related to the need for predicting the performance effects of the core component within any STM layer, namely the (local) concurrency control protocol, which is in charge of regulating the access to data by the concurrent threads that are active within the local instance of the transactional process.

We note that, compared to traditional transactional systems, such as database systems, STMs are based on (and require) innovative design/development approaches, where the optimization focus is shifted on aspects that historically had less importance. Among them we can mention hardware-cache aware design (see, e.g, [51]) as well as tailoring of the design to the specific instruction set offered by the target computing architecture (see, e.g., [147]). At the same time, concurrency control schemes commonly adopted in database environments are not likely to fit all the requirements of fine grained, volatile memory operations typical of STM-based applications [50]. As an example, if blindly ported to STM environments, database oriented concurrency control schemes based on explicit wait (sleep) phases, actuated on top of operating system supported mutex and/or semaphores, would induce excessive overhead and non-negligible thread (re-)schedule delay. These costs were instead affordable in databases by amortizing them across (subsequent) stable storage interactions for both read operations and log writes.

According to the above considerations, the wide set of database oriented performance analysis results (see, e.g., [10,41,73,87,137,151]) cannot be (fully) representative of the actual performance levels provided by STM systems. Hence, a major issue to address when dealing with innovative concurrency control algorithms specifically tailored to STM environments (see, e.g., [24, 45, 69]) is the definition of innovative analytical models able to reliably capture their actual dynamics.

At the time of writing the present deliverable, we have addressed the above issue by defining an analytical modeling approach able to capture the effect of STM oriented concurrency control protocols on the performance (e.g. the throughput) achievable by a stand-alone STM. Such models would be helpful since, e.g., it would be extremely important to assess how well a given STM concurrency control algorithm scales vs the degree of parallelism, namely the number of CPU-cores available to concurrent threads operating within an instance of the transactional process.

Actually, what we present within this deliverable is a two-layered analytical modeling methodology where a *thread-level* model predicts the system performance as a function of the degree of concurrency within the system (e.g., the number of worker threads in charge of executing transactional memory operations, and the probability that they are executing transactional vs non-transactional code portions), independently of the specific scheme adopted for regulating memory accesses by concurrent conflict-

ing transactions. The latter aspect is instead demanded to a so called *transaction-level* model, which can be specialized in a way to determine commit/abort probabilities on the basis of the specific choices determining the actual synchronization (concurrency control) scheme among threads executing conflicting transactional code portions.

Next we present the transaction-level model for two concurrency control algorithms:

- a Commit Time Locking (CTL) concurrency control algorithm ensuring opacity [59], a consistency criterion stronger than serializability that is particularly desirable in non-sandboxed environments like a STM. CTL algorithms have been adopted by several popular STM systems [24,45], and a variant ensuring repeatable read consistency has been integrated in a very recent release of Infinispan (version 5.1).

- an Encounter Time Locking (ETL) concurrency control algorithm ensuring repeatable read semantics. ETL algorithms include the classic Two Phase Locking scheme [15] and are particularly relevant for the Cloud-TM project given that it represented the (only) concurrency control for Infinispan at the moment in which the project started.

Finally, we validate the proposed methodology via en extensive simulation study. We focus our validation on the case of CTL algorithm, as the ETL model will be extensively validated in Section 4, where it will be used as building block of Two Phase Commit-based and Primary Backup replication strategies.

3.1 Analytical Model Basics

As typical of STM applications/benchmarks [5,25,61] we consider a fixed number k of threads, each of which executes on a distinct CPU-core. Threads alternate the execution of transactional and non-transactional code blocks. A non-transactional code block is formed by a sequence of machine instructions which we denote as $ntcb$. Each transaction starts with a $begin$ operation, then executes a number of transactional operations (namely, either $read$ or $write$ operations) on shared data items and finally ends by issuing a $commit$ operation. Overall, after the $begin$ operation and after each transactional operation, the thread executes a code block, denoted as tcb, during which it does not perform transactional read/write operations on shared data items, thus exclusively operating within its private workspace, e.g., by accessing its own stack.

We denote with t_{begin}, t_{read}, t_{write} and t_{commit}, the expected time required by a thread to execute, respectively, $begin$, $read$, $write$ and $commit$ operations. Note that, in practice, these durations are affected by both the speed of the underlying hardware platform and the internals of the underlying STM layer. Compared to existing approaches (see, e.g., [67]), the choice of capturing the above costs through ad-hoc parameters enhances the flexibility of our model, thus allowing it to be employed for what-if analysis aimed at forecasting the performance for diverse scenarios and/or workloads. As an example, the model can be used to assess the performance of STM-based applications when deployed on different hardware platforms (which might give rise to different machine instruction patterns) or when changing the internals of the

underlying STM layer (e.g. via the exploration of trade-offs between alternative implementation strategies).

We denote as t_{tcb} and t_{ntcb}, respectively, the expected duration of tcb and $ntcb$. Whenever a transaction is aborted, an $abort$ operation is executed, whose handling has an expected duration t_{abort}. After experiencing an abort, a transaction is temporarily held in a back-off state for a time interval whose average value is denoted as $t_{backoff}$, at the end of which it gets restarted. Conforming to common implementations/settings [5, 61], the thread taking care of the execution of the transaction gets temporarily suspended, and resumes right upon the end of the back-off period.

3.2 Modeling Approach Overview

As discussed above, we logically structure our model in two distinct parts, each one capturing complementary aspects of the execution dynamics of STM-based applications. The first part of the model, which we name thread-level model, is presented in Section 3.3. It allows determining how the various threads in the system alternate among the following three phases:

(i) execution of a non-transactional code block,

(ii) execution of an STM transaction,

(iii) blocked in back-off.

By allowing the determination of the probability distribution of the number of threads in each of these three phases, this layer of the model can be used to output standard performance metrics such as throughput and execution time. This part of the model is de-facto oblivious of the specific algorithm used by the STM to regulate concurrency, over which it abstracts via two key input parameters: (a) the average transaction execution time (independently of its final outcome) and (b) the commit probability, given a number $i \in [1, k]$ of threads concurrently executing transactions. Instead, these parameters are computed by what we refer to as transaction-level model, one instance of which, tailored to CTL, is presented in Section 3.4. The later modeling component is focused on capturing proper dynamics associated with the specific conflict detection and resolution schemes adopted by the STM layer, assuming a constant, albeit parametric, number of threads simultaneously executing transactions.

By decoupling the modeling of the dynamics associated with thread alternation among the various phases from the modeling of the concurrency control algorithm, our two-layered modeling methodology provides the below reported benefits:

1. It simplifies the modeling stage of the concurrency control algorithm, delegated to the transaction-level model, since this model does not require to explicitly consider dynamic variations of the number of threads concurrently executing transactional code blocks. The model only requires to provide performance predictions under the assumption that exactly i threads are concurrently executing transactions. Then, it will be the responsibility of the thread-level model to exploit the independent performance forecasts associated with different values of i in order to generate the final performance predictions.

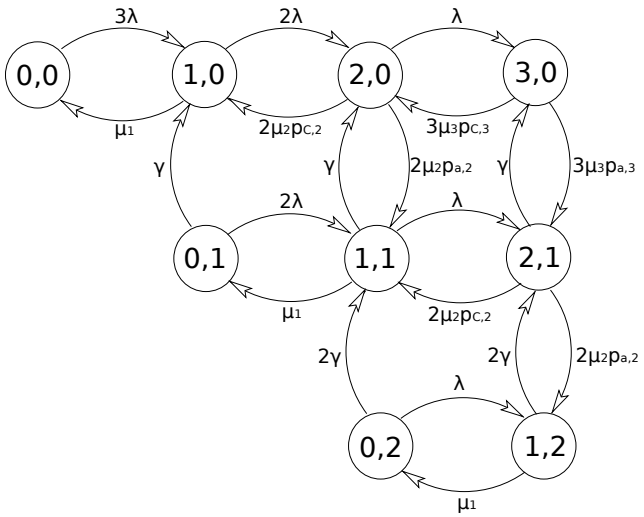


Figure 10: State transition diagram of the CTMC for $k = 3$.

2. It allows seamless replacement of the model of the STM concurrency control scheme presented in this deliverable, namely the CTL model [45] (see Section 3.4), with alternative ones either relying on different modeling approaches and/or targeting different concurrency control algorithms.

3.3 Thread-level Model

We model the alternation of the various phases for the execution of the different threads (inside a transaction, executing a non-transactional code block or blocked in back-off after an abort) via a Continuous Time Markov Chain (CTMC) [80]. Each state of the CTMC is marked and identified by a couple of integers (i, j) representing, respectively, the number of threads running transactions and the number of threads in back-off. Since the total number of threads in the system is equal to k, the only admissible states in the CTMC are those for which the corresponding (i, j) pair respects the constraint $i + j \leq k$.

For each state $(i, \cdot)$, with $i > 0$, the model takes as input parameters (i) the rate μ_i according to which transactions are run within the system (independently of whether the transaction run gets aborted or committed), and (ii) the probability $p_{c,i}$ for a transaction to successfully commit, in case of i threads simultaneously executing transactions. These need to be provided by the transaction-level model in charge of capturing the effects of the specific concurrency control scheme. In the following we will denote with $p_{a,i} = 1 - p_{c,i}$ the probability for a transaction to experience an abort, when considering that i threads are concurrently executing transactions. Also, we will denote with $\lambda = \frac{1}{t_{ntcb}}$, the rate according to which a thread executes a non-transactional code block (in between two transactions).

We note that modeling the system via a CTMC maps onto assuming that $ntcb$, the interarrival times of transactions to the commit phase, and the back-off interval have an exponentially distributed duration. Possible extensions of the model to cope with cases where the values of λ and μ_i represent the mean of generic distributions will be discussed in Section 3.7.

We can now list the rules defining the transition rates from any two states of the CTMC:

- for $i + j < k$, the transition rate from state (i, j) to state $(i + 1, j)$, associated with the activation of a new transaction run after the completion of the execution of a non-transactional code block, is equal to $\lambda \cdot (k - i - j)$;
- for $i > 0$, the transition rate from state (i, j) to state $(i - 1, j)$, associated with transaction commit events and the subsequent activation of a non-transactional code block, is equal to $i \cdot \mu_i \cdot p_{c,i}$;
- for $i > 0$, the transition rate from state (i, j) to state $(i-1, j+1)$, associated with transaction abort events and the start of the back-off period, is equal to $i \cdot \mu_i \cdot p_{a,i}$;
- for $j > 0$, the transition rate from state (i, j) to state $(i + 1, j - 1)$, associated with the termination of back-off periods and transaction restart, is equal to $j \cdot \gamma$, where $\gamma = \frac{1}{t_{backoff}}$.

We exclude state $(0, k)$ as a possible one since, (i) the CTMC characterizing our model does not express state transitions where multiple transactions get simultaneously aborted due to (mutual) conflicts, and (ii) adopting whichever literature STM concurrency control algorithm, if a single thread is currently executing a transactional code block, then the corresponding transaction cannot be aborted. It is easy to show that the set of states of the CTMC, denoted as S, has cardinality equal to $\frac{(k+1)\cdot(k+2)}{2} - 1$. Note also that, if $i+j < k$, it follows that $k-(i+j)$ threads are executing non-transactional code blocks. An example of the CTMC for the case of three threads (namely $k = 3$) is depicted in Figure 10.

As typically expected in any real system, assuming for any state where $i \in [1, k]$ that $\mu_i > 0$, $p_{c,i} \neq 0$ and $p_{c,i} \neq 1$ (the cases of $p_{c,i} = 0$ or $p_{c,i} = 1$ express, respectively, a pathological scenario with no possibility of transaction progress and a trivial scenario entailing no data contention), the CTMC is irreducible, and is formed by an ergodic set of states. Thus the stationary probability vector $\mathbf{v}$ is unique and satisfies the typical equation

$$\mathbf{v} \cdot Q = \mathbf{0} \tag{9}$$

where Q is the infinitesimal generator matrix of the CTMC [102]. Assuming that the system is in steady-state, and that we are provided with μ_i and $p_{c,i}$ values ($\forall i \in [1, k]$), we can compute the probability to be in each state $(i, j) \in S$ by solving equation (9). We can then evaluate the system throughput τ as the sum of the transaction commit rates in the different states, weighted according to the probability for the system to be in each state (i, j)

$$\tau = \sum_{(i,j)\in S'} v_{i,j} \cdot i \cdot \mu_i \cdot p_{c,i} \tag{10}$$

(where S' is the subset of S containing any state where $i > 0$). The overall transaction commit and abort probabilities, denoted as p_c and p_a, can be accordingly evaluated, using the below expressions

$$p_c = \frac{\sum_{(i,j) \in S'} v_{i,j} \cdot p_{c,i}}{\sum_{(i,j) \in S'} v_{i,j}} \tag{11}$$

and

$$p_a = (1 - p_c) \tag{12}$$

3.4 Transaction-level model: the CTL Case

In this section we introduce an analytical model of Commit-Time-Locking (CTL) concurrency control. We will start by overviewing such a target version of the CTL algorithm, and then we will move to the presentation of its analytical model.

3.4.1 Algorithm Overview

Unlike, e.g., strict 2PL [15], CTL schemes do not acquire locks upon accessing data items. Instead, lock acquisition is delayed to commit time, and only involves written data items (write-locks). This choice enhances concurrency with respect to conventional lock-based schemes by, e.g., avoiding to block transactions issuing a write operation on a data item that has already been read/written by a concurrent transaction.

Given the absence of read-locks, consistency is ensured via a validation mechanism used to notify transaction T, which speculatively read a data item x, about the fact that x was overwritten by some concurrent transaction T' preceding T in the commit order. To this end, a versioning scheme is employed which associates a timestamp value with each data item, referred to as Write-Version-Clock (WVC). The generation of WVC values relies on a unique Global-Version-Clock (GVC), which is read by any transaction upon startup, and is atomically increased upon transaction commit. The updated value is used as the new WVC value for all the data items written by the committing transaction. Manipulation of the GVC typically relies on a Compare-and-Swap (CaS) operation directly exploiting atomic sequences of machine instructions (e.g., via the LOCK prefix in IA-32 compliant processors). In other words, each transaction updates the GVC as an acyclic, one shot operation, which does not require software spin-locking for accessing the corresponding critical section. Hence, any delay in the access to the GVC is only related to the underlying firmware protocol used to support the atomicity of the machine instruction pattern implementing the CaS.

When validating a transaction against a read data item x, two actions are performed:

1) it is checked whether there is a write-lock being held on x (which implies that another transaction has written x and is currently within its commit phase);

2) it is checked whether the current timestamp associated with x is greater than the timestamp read by the transaction upon starting up (which indicates that some concurrent transaction has overwritten x and has already been committed).

If one of the previous checks fails, the transaction gets aborted. This validation scheme is used upon read operations and, as we shall discuss below, also at commit time. Accordingly, the *opacity* property [59] is guaranteed, which ensures that the snapshot observed by any transaction (including transactions that are eventually aborted), is equivalent to the one that would have been observed according to some serial execution history. As discussed in [59], this property is crucial since for several categories of STM-based applications, transactions observing an inconsistent snapshot may be trapped within infinite loops, or may even cause the application program to crash (e.g., due to an invalid memory reference).

As far as write operations are concerned, in CTL they are buffered within a private workspace until the commit phase. When a transaction attempts to commit, it first acquires the write-locks for all the data items within its write-set. If any of these lock acquisitions fails (due to lock holding by some other transaction), the transaction is aborted. Otherwise, the transaction increments the GVC and tries to validate all the data items it has within its read-set (according to the aforementioned validation procedure). If the validation fails for at least one item within the read set, the transaction gets aborted. If no abort occurs, the data within the write-set are copied back to their original memory locations, updating their WVCs with the value of the GVC. All the acquired locks are released at the end of the commit phase, or upon the abort.

By the above description, we have that a read operation on a data item that was previously written by the transaction gives rise to an access to the transaction private workspace. Thus it is not subject to the previously depicted read validation mechanism. In other words, the validation mechanism is used for read operations associated with any data item that has not already been accessed in write mode by the transaction.

3.4.2 Analytical Model

In order to simplify presentation, we present the model in an incremental fashion. We start by presenting a model relying on the following set of assumptions:

- the accesses (both in read or write mode) to data items in the transactional shared memory are uniformly distributed;
- all the transactions encompass the same amount n of transactional accesses;
- the sequence of read operations issued on shared data items form a Poisson process.

A discussion on how to relax the above assumptions will then be provided in Section 3.4, Section 3.6 and Section 3.7.

As previously discussed, the transaction-level model computes the transaction run rate $\mu_i = 1/r_{t,i}$ (where $r_{t,i}$ is the average transaction execution time) and the transaction commit probability $p_{c,i}$ under the assumption that there are i threads simultaneously processing transactions, with $1 \leq i \leq k$. We analyze the case $i = 1$ and $i \neq i$ separately.

If $i = 1$, a single thread is currently executing transactional code, thus no data conflict can arise. This also means that the currently executed transaction can not be

aborted and it follows that $p_{c,1} = 1$. Therefore, for the average transaction execution time we have that

$$r_{t,1} = t_{begin} + n \cdot t_{op} + (n+1)t_{tcb} + t_{commit} \tag{13}$$

where t_{op}, namely the average time to execute an access operation on a shared data item, is equal to

$$t_{op} = t_{read}(1 - p_{write}) + t_{write} \cdot p_{write} \tag{14}$$

where we denote with n the number of transactional operations on shared data items within a transaction, with p_{write} the probability that the access is in write mode, and with $(1 - P_{write})$ the probability that the access is in read mode.

As already discussed in Section 3.4.1, if the transaction accesses a data item x in write mode, producing a new version, any subsequent read on x by the same transaction will return the previously written version, retrieving it from the transaction private workspace. Analogous considerations apply for subsequent writes over the same data item x, which will simply update the copy of x buffered within the private workspace. Hence read/write operations issued on previously updated data items are simply not taken into account by the parameter n. On the other hand the cost of the corresponding accesses within the private workspace is encapsulated in t_{tcb}.

By the previous notation, we have that

$$n_{write} = n \cdot p_{write} \tag{15}$$

is the average number of shared data items accessed by the transaction in write mode, and

$$n_{read} = n \cdot (1 - p_{write}) \tag{16}$$

is the average number of read operations occurring on distinct shared data items that were not already accessed by the transaction in write mode.

For $i \neq 1$ we proceed as follows. Once fixed i, we use a procedure that iteratively recalculates the values of $p_{c,i}$ and $r_{t,i}$. Upon starting the iterative procedure, the initial values can be selected as $p_{c,i} = p_{c,i-1}$ and $r_{t,i} = r_{t,i-1}$ for commodity. The output values by an iteration step are used as the input values for the next step. We conclude the iterative procedure as soon as the corresponding input and output values for $p_{c,i}$ and $r_{c,i}$ differ by at most an ϵ. In all the configurations that we have experimented, using ϵ=1%, the procedure has always converged in at most fifteen iterations.

In each iteration step the following set of parameters, captured by our model, are re-evaluated:

- $p^o_{a,l}$, namely the probability for a transaction to abort while executing its lth operation due to validation fail (recall that a transaction can abort while executing an operation only if the operation is a read);

- p_{alc}, namely the probability for a transaction to abort at commit time due to lock contention experienced in the commit-time lock acquisition phase;

- p_{avf}, namely the probability for a transaction to abort at commit time due to validation failure of its read-set.

In order to model these parameters, we consider that the expected system state *seen* by any of the i active transactions is determined by the activities associated with the other $i-1$ transactions currently within the system. Thus we use the following approach.

When a transaction successfully commits, an average number n_{write} of write-locks are first acquired, and then released after read-set validation and write-back phases. Actually, the duration of the lock acquisition and release phases are typically negligible with respect to the duration of validation and write-back phases (recall that, during lock acquisition, transactions do never block, even if they experience contention). Hence, for simplicity, we assume lock acquisition and release to be instantaneous and to occur, respectively, at the beginning and at the end of the commit phase. Also, if a transaction is aborted, no real rollback operation is required for undoing the effects of the corresponding write operations since transaction write-sets are reflected to memory only in case of succesful commit attempts. Thus, to simplify, we ignore the cost of aborts when we evaluate the average lock holding time, by assuming that if a transaction successfully completes the lock acquisition phase, it holds the locks for an average time equal to t_{commit}.

Let us now compute the probability for a transaction to abort while executing a read operation on a shared data item x, given that it finds the corresponding write-lock currently busy. For this case to be possible, there must exist another transaction that has written x, that is currently in its commit phase and that has successfully acquired the write-locks for all the data items in its write-set. Given that we are assuming uniformly distributed accesses to distinct data items within a transaction, it follows that the probability for a committing transaction to have a specific data item within its write-set is n_{write}/d. Exploiting the aforementioned assumption of Poissonianity of the arrival process of read operations, we can rely on the PASTA property (Poisson Arrivals See Time Averages) [145] to compute the probability to incur in a raised write-lock during a read operation as

$$p_{lock} = l_r \cdot t_{commit} \cdot \frac{n_{write}}{d} \tag{17}$$

where l_r is the rate according to which the remaining $i-1$ transactions in the system successfully execute the write-lock acquisition phase. This rate can be evaluated as follow

$$l_r = \frac{1}{r_{t,i}} \cdot (p_{c,i} + p_{avf}) \cdot (i-1) \tag{18}$$

where p_{avf} is the probability for a transaction to abort during the read-set validation phase. Such a transaction contributes anyway to the lock-acquisition rate since read set validation occurs after write-locks are acquired at commit time over any written data item. We will evaluate p_{avf} later in this subsection.

Now we determine the probability $p^o_{a,l}$ for a transaction T to abort while executing the l-th operation. The rate u_r at which a data item is updated by transactions is equal to

$$u_r = c_r \cdot \frac{n_{write}}{d} \tag{19}$$

where c_r expresses the rate at which the other $i-1$ transactions successfully commit, and can be evaluated as

$$c_r = \frac{1}{r_{t,i}} \cdot p_{c,i} \cdot (i-1) \tag{20}$$

Upon the l-th operation by transaction T, the average time $t_{b,l}$ elapsed since T started its execution can be expressed as $t_{begin} + t_{tcb} \cdot l + t_{op} \cdot (l-1)$. As we are assuming that the arrival of transactions to the commit phase forms a Poisson process, the probability $p^o_{u,l}$ for a read (executed as the l-th operation of T) to access a shared data item that has been updated by some successfully committing transaction after T started can be expressed as

$$p^o_{u,l} = 1 - e^{-u_r \cdot t_{b,l}} \tag{21}$$

In the above expression, in order to avoid overcomplicate the model, we decided not to capture the case of repeated transactional read operations on the same data item. In this case, in fact, the invalidation window for a data item x would no longer correspond to the time elapsed since the beginning of the transaction (namely $t_{b,l}$), but would be equal to the (average) time elapsed since the last occurrence of a read on x. Clearly, the error introduced by this modeling choice depends on the actual frequency of occurrence of repeated read operations on the same data item during the same transaction. On the other hand, the model captures faithfully the effects of a frequent optimization technique (possibly implemented at the compiler level), which allows sparing subsequent read operations issued within the same transaction on the same data item from the cost of validation. To this end, it is sufficient to copy the values read from the shared transactional memory to thread local variables, and to redirect subsequent reads on these data items (within the same transaction) towards the thread local variables. Note that, since with this optimization subsequent read operations on a data item do not target the shared transactional memory, they do not even need to be accounted for while computing the value of the parameter n.

We can now evaluate the probability for a transaction to abort during the execution of its first operation (i.e., when l=1), namely $p^o_{a,1}$ as

$$p^o_{a,1} = (1 - p_{write}) \cdot (p_{lock} + (1 - p_{lock}) \cdot p^o_{u,1}) \tag{22}$$

Since the abort of a transaction T during its l-th operation (where $2 \leq l \leq n$) implies that T did not abort during its previous $l-1$ operations, it follows that

$$p^o_{a,l} = p^o_{na,l} \cdot (1 - p_{write}) \cdot (p_{lock} + (1 - p_{lock}) \cdot p^o_{u,l}) \tag{23}$$

where $p^o_{na,l}$ is the probability of not aborting until the completion of the $(l-1)^{th}$ operation. For this last probability we have

$$p^o_{na,1} = 1 \tag{24}$$

and

$$p^o_{na,l} = (1 - p^o_{a,l-1}) \cdot p^o_{na,l-1} \tag{25}$$

In equations (22)-(23) we have assumed that the event of finding a write-lock raised on the shared data item by a concurrent transaction currently attempting to commit,

and the event that the same data item was previously updated by a different (already committed) concurrent transaction are independent. Overall, independence is related to that these events belong to commit time activities across distinct transactions.

The probability p_{alc} for a transaction T to abort at commit time due to lock contention while acquiring the write-locks can be evaluated as follow. T can experience contention while requesting the lock on a data item x only if, at the time in which T starts its commit phase, some other transaction that has written x has successfully completed its lock acquisition phase, and is still executing the commit procedure. Considering that T aborts only if *at least one* of the data items in its write-set is locked, then, as in [151], we approximate this last probability, namely p_{wlc}, with an upper bound value, that is

$$p_{wlc} = 1 - (1 - p_{lock})^{n_{write}} \tag{26}$$

Thus we have

$$p_{alc} = p^{o}_{na,n+1} \cdot p_{wlc} \tag{27}$$

where we recall that $p^{o}_{na,n+1}$ is the probability for a transaction not to be aborted until the completion of its nth operation, that is until it enters its commit phase. Consequently, the probability p^{la}_{na} for a transaction not to be aborted during its execution and to succeed in its commit-time lock acquisition phase is equal to

$$p^{la}_{na} = p^{o}_{na,n+1} \cdot (1 - p_{wlc}) \tag{28}$$

Let us now show how we can evaluate p_{avf}, namely the probability for a transaction T to abort at commit time due to validation failure for its read-set. The validation fails if at least one data item x belonging to the read-set of T has the corresponding write-lock raised by another transaction, or if a new version of x has been committed after the validation executed by T during its last read operation on x. We denote with $p^{r}_{u,l}$ the probability that the shared data item accessed in read mode at the lth operation by T has been updated after the last validation (occurred upon the corresponding last read operation on x). We calculate this probability as follows

$$p^{r}_{u,l} = 1 - e^{-u_r \cdot t_{v,l}} \tag{29}$$

where $t_{v,l}$ is the elapsed time since the original validation, that is

$$t_{v,l} = (t_{tcb} + t_{op}) \cdot (n - l + 1) + t_{commit} \tag{30}$$

Analogously to what we did in equation (23), we evaluate the abort probability due to failure in the validation of the data item associated with the lth transactional access of T as follows

$$p^{r}_{a,l} = p^{r}_{na,l} \cdot (1 - p_{write}) \cdot (p_{lock} + (1 - p_{lock}) \cdot p^{r}_{u,l}) \tag{31}$$

where $p^{r}_{na,1} = 1$ and, for $l > 1$, $p^{r}_{na,l} = (1 - p^{r}_{a,l-1}) \cdot p^{r}_{na,l-1}$. Then, we can express p_{avf} as

$$p_{avf} = p^{la}_{na} \cdot p_{rvf} \tag{32}$$

where

$$p_{rvf} = \sum_{l=1}^{n} p_{a,l}^{r} \tag{33}$$

Finally, successful commit probability for the case of i active threads can be evaluated as

$$p_{c,i} = p_{na}^{la}(1 - p_{rvf}) \tag{34}$$

The average execution time of a transaction $r_{t,i}$ can now be computed as the sum of the average time for a transaction to reach a different execution phase, weighted with the probability for the transaction to abort exactly during that phase. Let us denote with

- $t_{a,l}$ the average duration of a transaction that aborts during its l-th operation, that is:

$$t_{a,l} = t_{begin} + l \cdot (t_{tcb} + t_{op}) + t_{abort} \tag{35}$$

- $t_1 = t_b + t_{tcb} + t_{abort}$ the average duration of a transaction that aborts during its commit phase due to contention while acquiring locks for the data items in its write-set, where

$$t_b = t_{begin} + n \cdot (t_{tcb} + t_{op}) \tag{36}$$

- $t_2 = t_b + t_{tcb} + t_{commit} + t_{abort}$ the average duration of a transaction that aborts during its commit phase due to failure in validating its read-set;

- $t_3 = t_b + t_{tcb} + t_{commit}$ the average duration of a transaction that successfully commits.

Overall, the average transaction execution time can be expressed as

$$r_{t,i} = \sum_{l=1}^{n} (p_{a,l}^{o} \cdot t_{a,l}) + p_{alc} \cdot t_1 + p_{avf} \cdot t_2 + p_c \cdot t_3 \tag{37}$$

Now let us evaluate the time t_{GVC} spent by any committing transaction while updating the GVC. We consider this time logically included in t_{commit}, thus t_{commit} is the sum of two terms, namely t'_{commit} and t_{GVC}, where t'_{commit} is the time to execute all the other operations, distinct from GVC manipulation, during the commit phase. As explained in Section 3.4.1, the atomic operations required for the update of GVC typically rely on firmware level protocols offered by modern SMP and/or multi-core machines. Assuming fairness by these protocols vs different CPU/core requests, we model the delay for performing an atomic increment of the GVC, denoted as t_{GVC}, by means of an M/D/1 queue [80] with service rate $\mu = \frac{1}{t_{GVC}^{inc}}$ (where t_{GVC}^{inc} expresses latency for the updating machine instructions, once the firmware has granted access to the corresponding critical section) and arrival rate $\beta = l_r$ (note that the increment of the GVC is performed by any transaction that successfully acquired all the requested locks). According to this modeling approach, t_{GVC} corresponds to the residence time within the M/D/1 queue, namely

$$t_{GVC} = (1 + \frac{\rho}{2 \cdot (1 - \rho)}) \cdot t_{GVC}^{inc}, \tag{38}$$

where $\rho = \frac{\beta}{\mu}$.

3.5 Coping with Multiple Transaction Classes

In this section we extend the analytical model by considering the case of q different transactional classes, associated with different transaction profiles. A transaction of class m, with $m \in [1, q]$ executes n^m operations and each operation is a write operation with probability p^m_{write}. Hence $n^m \cdot (1 - p^m_{write})$ expresses the total amount of distinct transactional read accesses. A thread runs a transaction of class m with probability P^m. Also, t^m_{commit} and t^m_{abort} are the expected commit time and abort time for a transaction of class m, respectively.

3.5.1 Multi-class Thread-level Model

For q transactional classes, the state of the CTMC can be identified by $2q$ integers $(i_1, .., i_q, j_1, .., j_q)$ where i_m and j_m (with $m \in [1, q]$) represent the number of threads running transactions of class m and the number of threads in backoff due to an abort of a transaction of class m, respectively. Note that $i_1 + .. + i_q + j_1 + .. + j_q \leq k$ for each state of the CTMC.

For any state $(i_1, .., i_q, j_1, .., j_q)$, the average transaction execution rate and the transaction commit probability for a transaction of class m depend on the mix of active transactions in that state. Thus we denote them as $\mu^m_{i_1,..,i_q}$ and $p^m_{c,i_1,..,i_q}$, respectively. Also, the abort probability for a transaction of class m while residing in state $(i_1, .., i_q, j_1, .., j_q)$ is denoted as $p_{a,i_1,..,i_q} = 1 - p_{c,i_1,..,i_q}$.

The rate according to which a thread executes a new transaction of class m is $\lambda_m = P^m / t_{ntcb}$. The rules defining the transition rates from any two states of the CTMC are the following:

- for $i_1 + ... + i_q + j_1 + ... + j_q < k$, the transition rate from state $(i_1, ..., i_m, ..., i_q, j_1, ..., j_q)$ to state $(i_1, ..., i_m + 1, ..., i_q, j_1, ..., j_q)$, associated with the activation of a run of a transaction of class m is equal to $\lambda_m(k - i_1 - ... - i_q - j_1 - ... - j_q)$;

- for $i_m > 0$, the transition rate from state $(i_1, ..., i_m, ..., i_q, j_1, ..., j_q)$ to state $(i_1, ..., i_m - 1, ..., i_q, j_1, .., j_q)$, associated with a successful commit event of a transaction of class m is equal to $i_m \mu^m_{i_1,...,i_q} p^m_{c,i_1,...,i_q}$

- for $i_1 + ... + i_m + ... + i_q \geq 2$ and $i_m \geq 1$, the transition rate from state $(i_1, ..., i_m, ..., i_q, j_1, ..., j_m, .., j_q)$ to state $(i_1, ..., i_m - 1, .., i_q, j_1, .., i_m + 1, ..., j_q)$, associated with an abort event of a transaction of class m is equal to $i_m \mu^m_{i_1,...,i_q} p^m_{a,i_1,...,i_q}$

- for $j_m > 1$, the transition rate from state $(i_1, ..., i_m, ..., i_q, j_1, ..., j_m, ..., j_q)$ to state $(i_1, ..., i_m + 1, ..., i_q, j_1, ..., j_m - 1, ..., j_q)$, associated with the termination of a back-off period of an aborted transaction of class m is equal to $\gamma \cdot j_m$.

The cardinality of the set S of states of the CTMC can be evaluated by considering that at given time a thread can be in one of $2 \cdot q + 1$ different states (namely, q states corresponding to the execution of a transaction of class m, q states corresponding to the

back-off period after an abort event of a transaction of class m, and 1 state corresponding to the execution of a non-transactional code block). As a consequence, the number of states of the CTMC is the k-combination with repetition of $2 \cdot q + 1$ elements, that is the binomial coefficient $\binom{(2 \cdot q + 1) + k - 1}{k}$, where we have to exclude the states where all the threads are in back-off.

By relying on the same Poissonianity assumptions made in Section 3.3, and by equation (9) applied to the multi-class CTMC, we can evaluate the stationary probability vector v. Hence, the execution rate τ_m of transactions of class m can be expressed as

$$\tau_m = \sum_{(s') \in S'} v_{s'} \cdot i_m \cdot \mu^m_{s''} \cdot p^m_{c,s''} \tag{39}$$

where we used s' in place of $i_1, .., i_q, j_1, .., j_q$ and s'' in place of $i_1, .., i_q$, and where S' is the subset of S containing any state where $i_m > 0$. The overall system throughput is

$$\tau = \sum_{m=1}^{q} \tau_m \tag{40}$$

The commit probability for a transaction of class m is

$$p^m_c = \frac{\sum_{(s') \in S'} v_{s'} \cdot p^m_{c,s}}{\sum_{(s') \in S'} v_{s'}} \tag{41}$$

3.5.2 Multi-class Tread-level Model for CTL

Fixed a configuration of active transactions $i_1, .., i_q$, the thread-level model is in charge of evaluating for each transactional class m the transaction run rate $r^m_{t,i_1,..,i_q}$ and the transaction commit probability $p^m_{c,i_1,..,i_q}$. As for the single-class models, if there is just one active transaction, that is $i_m = 1$ and $i_w = 0$ for each $w \neq m$, the average transaction execution time of the transaction of class m is

$$r^m_{t,i_1,..,i_q} = t_{begin} + n^m \cdot t_{op} + (n^m + 1) t_{tcb} + t^m_{commit} \tag{42}$$

where t^m_{op}, namely the average time to execute an access operation on a shared data item for a transaction of class m, is equal to

$$t^m_{op} = t_{read}(1 - p^m_{write}) + t_{write} \cdot p^m_{write} \tag{43}$$

When the number of active transactions is greater that one we use the same iterative approach as in Section 3.4.2, by stopping the iterations when two consecutive values of the commit probability for transactions of each class m (if $i_m \geq 1$) differ by at most an ϵ. Also, in what follows we use the same assumptions and considerations as in Section 3.4.2.

When a transaction of class m is active, its concurrent transactions are:

- i_x active transactions of each other class x such that $x \neq m$ and $i_x \geq 1$;

- $i_m - 1$ active transactions of the same class m, if $i_m \geq 2$.

At the start of each iterative step we evaluate the following parameters. The lock rate associated with transactions of each class x, expressed as

$$l_r^x = \frac{1}{r_{t,i_1,..,i_q}^x} \cdot (p_{c,i_1,..,i_q}^x + p_{avf}^x) \tag{44}$$

where p_{avf}^x is the probability for a transaction of class x to abort during the read-set validation phase. The probability for a transaction of class m to find a write-lock raised while issuing a read operation, which is expressed as

$$p_{lock}^m = \sum_{x=1,x\neq m}^{q} l_r^x \cdot i_x \cdot t_{commit} \frac{n^x \cdot p_{write}^x}{d} + l_r^m \cdot (i_m - 1) \cdot t_{commit} \frac{n^m \cdot p_{write}^m}{d}. \tag{45}$$

The commit rate associated with transactions of class x, which is expressed as

$$c_r^x = \frac{1}{r_{t,i_1,..,i_q}^x} \cdot p_{c,i_1,..,i_q}^x \tag{46}$$

Finally, the update rate by concurrent transactions of a transaction of class m, which is expressed as

$$u_r^m = \sum_{x=1,x\neq m}^{q} c_r^x \cdot i_x \frac{n^x \cdot p_{write}^x}{d} + c_r^m \cdot (i_m - 1) \frac{n^m \cdot p_{write}^m}{d}. \tag{47}$$

After solving the previous equations, we evaluate in each iterative step all the parameters we list below. The probability $p_{u,l}^{o,m}$ for a read operation, executed as the l-th operation of a transaction T of class m, to access a data item that has been updated by some successfully committing transaction after T started, which can be expressed as

$$p_{u,l}^{o,m} = 1 - e^{-u_r^m \cdot t_{b,l}^m} \tag{48}$$

where $t_{b,l}^m$ is the average elapsed time since the validation performed on the data item upon the read access by the transaction of class m, which can be evaluated the same way as the single-class case.

The probability to abort while executing the 1-st operation on a shared data item for a transaction of class m, expressed as

$$p_{a,1}^{o,m} = (1 - p_{write}^m) \cdot (p_{lock}^m + (1 - p_{lock}^m) \cdot p_{u,l}^{o,m}), \tag{49}$$

and the probability to abort while executing the l-th operation with $l \geq 2$ for a transaction of class m, expressed as

$$p_{a,l}^{o,m} = p_{na,l}^{o,m} \cdot (1 - p_{write}^m) \cdot (p_{lock}^m + (1 - p_{lock}^m) \cdot p_{u,l}^{o,m}) \tag{50}$$

where $p_{na,l}^{o,m}$ is the probability of not aborting until the completion of the $(l-1)^{th}$ operation, for which we have

$$p_{na,1}^{o,m} = 1 \tag{51}$$

and

$$p_{na,l}^{o,m} = (1 - p_{a,l-1}^{o,m}) \cdot p_{na,l-1}^{o,m} \tag{52}$$

The contention probability during write-lock acquisition phase for a transaction of class m can be then approximated as

$$p_{wlc}^{m} = 1 - (1 - p_{lock}^{m})^{n^m \cdot p_{write}} \tag{53}$$

Hence the probability for a transaction of class m to abort at commit time due to write-lock contention is

$$p_{alc}^{m} = p_{na,n^m+1}^{o,m} \cdot p_{wlc}^{m} \tag{54}$$

The probability for a transaction of class m not to be aborted during its execution and to succeed in its commit-time lock acquisition phase is

$$p_{na}^{la,m} = p_{na,n^m+1}^{o,m} \cdot (1 - p_{wlc}^{m}) \tag{55}$$

The probability that a data item in the read-set of a transaction belonging to class m, which is accessed at the l-th transactional operation, has been updated when T executes the read-set validation can be expressed as

$$p_{u,l}^{r,m} = 1 - e^{-u_r^m \cdot t_{v,l}^m} \tag{56}$$

where $t_{v,l}^m$ is the elapsed time since the original validation, which can again be computed teh same way as for the single-class case.

Thus the abort probability due to failure in the validation of the lth data item within the read-set can be evaluated as follows

$$p_{a,l}^{r,m} = p_{na,l}^{r,m} \cdot (1 - p_{write}^{m}) \cdot (p_{lock}^{m} + (1 - p_{lock}^{m}) \cdot p_{u,l}^{r,m}) \tag{57}$$

where $p_{na,1}^{r,m} = 1$ and, for $l > 1$, $p_{na,l}^{r,m} = (1 - p_{a,l-1}^{r,m}) \cdot p_{na,l-1}^{r,m}$. Hence, the probability for a transaction of class m to abort during the read-set validation phase can be expressed as

$$p_{avf}^{m} = p_{na}^{la,m} \cdot p_{rvf}^{m} \tag{58}$$

where

$$p_{rvf}^{m} = \sum_{l=1}^{n^m} p_{a,l}^{r,m} \tag{59}$$

Finally we can evaluate the probability of successful commit when residing within state $(i_1, .., i_q)$ as

$$p_{c,i_1,..,i_q}^{m} = p_{na}^{la,m}(1 - p_{rvf}^{m}) \tag{60}$$

For brevity we do not detail the equations for the evaluation of average transaction execution time and t_{GVC} because they can be simply derived by using the same approach we have show at the end of Section 3.4.2. In fact, the evaluation of the average transaction execution time for a transaction of class m can be done by using the already provided set of equations, by substituting the parameter values that depend on the specific transactional class with the ones we calculated in this section. Regarding

the evaluation of t_{GVC}, by using the approach discussed in Section 3.4.2, we have just to evaluate λ as the sum of the lock rate due to all the active transactions across the different classes, namely

$$\lambda = \sum_{m=1}^{q} l_r^m \cdot i_m \tag{61}$$

3.6 Hints on Model Extension for Non-uniform Data Access

By relying on the approach in [151], which has been proposed for the case of concurrency control algorithms in database systems, our model could be extended to cope with non-uniform data accesses. We provide hints on how the extension could be realized in this section.

The proposed approach considers the whole set of d shared data items as grouped in s disjoint subsets, possibly exhibiting different cardinalities. The set of n operations executed by a transaction are grouped in s different subsets, possibly exhibiting different cardinalities, where each operation accesses a data item belonging to a different data subset. The accesses executed on each subset of data items by a transaction are uniformly distributed over the subset.

Different subsets of data items exhibit different access frequencies. As a consequence, the probability to find a lock raised on a data item and the data item update rate are different for each specific subset of data items. To evaluate them for a given subset we can use the same equations (17) and (19) by considering, in place of n, only the subset of operations executed by the transactions on that specific subset of data items. Consequently, the subsequent equations, expressing the abort probability for a transaction, can be determined by considering the probability of finding the lock raised and the data item update rate as differentiated for each subset, and then weighting the corresponding effects by the fraction of operations executed on the specific subset.

3.7 On Removing Exponential Assumptions

In the model presented so far we have exploited the assumption of exponential distribution of several random variables. In this section we discuss how our modelling approach could be extended to relax these assumptions.

As for the thread-level model in Section 3.3, the reliance on a CTMC representation maps onto exponential assumptions for the frequency with which i) transactions complete their execution (either due to a commit or to an abort event), ii) transactions exit from their back-off period following an abort event, and iii) the execution of a non-transactional code block is completed. We also recall also that the output of the thread-level model is represented by the stationary probability vector v associated with the CTMC.

If one wanted to account in the model for generic, but known, distributions of the above transition rates, it would be simply sufficient to replace the CTMC with an equivalent Semi-Markov process [139]. At this point one should rely on well-known solution techniques [139] for the stationary probability vector of Semi-Markov processes.

For what concerns the transaction-level model, we exploited the assumption on the exponential distribution of the transactions' arrival rate to the commit phase to compute $p^o_{u,l}$ in equation (21) and $p^r_{u,l}$ in equation (29). Further, we exploited the assumption that the execution rate of read operations on shared data items forms a Poisson process to derive the expression of p_{lock} in equation (17).

As for equations (21) and (29), they could be extended to account for arbitrary distributions of the transaction arrivals to the commit phase. We could in fact write them as

$$p^o_{u,l} = \Phi(t_{b,l}, u_r) \tag{62}$$

$$p^r_{u,l} = \Phi(t_{v,l}, u_r) \tag{63}$$

where $\Phi(t, \eta), t \in (0, \infty)$ expresses the generic cumulative distribution function of the arrival process to the commit phase, having as $\eta = r_{t,i} = 1/E[\Phi(t, \eta)]$ its average arrival rate, and u_r could be computed as before using equation (19) and equation (20).

More problematic would be, instead, relaxing the assumption that the rate of read operations forms a Poisson process. In equation (17), in fact, we exploited directly the PASTA property [145] of Poisson arrival processes to compute the probability of finding a write-lock busy during a read on a data item x, as the probability for x to be locked at a random instant. However, if one were to assume that the arrival process of read operations on x formed a generic renewal process, one should explicitly account for the dynamic of interleaving between the arrival process of read operations on x and the stochastic process associated with the arrival of transactions that updated x to the commit phase. This would require determining the conditioned probability that, given an arbitrarily small interval $[t - h, t]$, there is a transaction T that is locking the data item x during its commit phase given that a transaction T' issues a read on x in the same time interval, or more formally:

$$lim_{h \to 0} Pr\{X(t - h) = 1 | N(t - h) \geq 1\} \tag{64}$$

where $X(t)$ expresses the number of transactions (that updated x) to be in the commit phase at time t and $N(t)$ is the counting process associated with the arrival of read operations (on x).

3.8 Transaction-Level Model: the ETL Case

We show now how to derive an analytical model of a non-serializable Encounter Locking Time (ETL) scheme, which was used till version 5.0 in Infinispan (and at the time in which the Cloud-TM project started). This model has been used as building block of the performance models of two replication protocols used in the Infinispan data grid, presented in Section 2.3.2.

This concurrency control scheme read objects in a lock-free way and store the read value in the transaction context, which allows ensuring in a lightweight way repeatable read semantics and spares read-only transactions from the possibility of incurring in aborts. The issuing of a write operations, however, triggers, unlike in CTL algorithms,

the *immediate* acquisition of a lock. This may lead to transaction queuing and possibly to deadlocks, which have a particularly detrimental effect at high contention. For this reason, several state of the art STMs [45, 52] adopt a simple technique to achieve deadlock-freedom, namely aborting transactions as soon as they encounter contention. This policy is also supported by Infinispan, and is the one considered to derive the following model.

Let us denote with $\lambda_{Tx,i}$ the frequency of arrival of transactions in the system given that we have i threads concurrently active as:

$$\lambda_{Tx,i} = \frac{i}{r_{t,i}}$$

We can then compute the lock request rate $\lambda^l_{lock,i}$ as:

$$\lambda^l_{lock,i} = \lambda_{Tx,i} \cdot w \cdot \tilde{N_{l,i}}$$

where we have denoted with $\tilde{N_{l,i}}$ the number of locks acquired successfully acquired on average by an update transaction independently from whether it aborts or commits. Recalling that lock contention implies aborting, we can compute the probability of abort during the execution phase of an update transaction as:

$$p_{a,i} = p_{lock,i} = \frac{\lambda^l_{lock,i}}{d} \cdot T_{H,i} \tag{65}$$

By the above probabilities, we can compute the probability that a transaction reaches its commit phase ($p_{cp,i}$) as:

$$p_{cp,i} \quad = \quad (1 - p_{a,i})^{N_{l,i}}$$

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

$$\tilde{N_{l,i}} = P_{p,i} \cdot N_{l,i} + \sum_{j=2}^{N_{l,i}} p_{a,i} \cdot (1 - p_{a,i})^{j-1} \cdot (j-1)$$

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

$$G(j) = \sum_{j=1}^{N_{l,i}} \frac{T_{W,i}}{N_{l,i}} \cdot j$$

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 ($H_{a,i}$)or

successfully completes ($H_{c,i}$).

$$
\begin{aligned}
T_{H,i} &= H_{a,i} + H_{c,i} \\
H_{a,i} &= \sum_{j=2}^{N_{l,i}} p_{a,i} \cdot (1 - p_{a,i})^{j-1} \cdot \frac{1}{j-1} \cdot G(j-1) \\
H_{c,i} &= p_{cp,i} \cdot [T_{commit} + \frac{1}{N_{l,i}} \cdot G(N_{l,i})]
\end{aligned}
$$

where we have denoted with T_{commit} the duration of the commit phase, which coincides with the write-back base and lock realasing time.

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

$$
T^{W,i} = T_{c,i} + T_{a,i}
$$

where

$$
\begin{aligned}
T_{c,i} &= p_{cp,i} \cdot (T_{WR,i} + T_{commit}) \\
T_{a,i} &= \sum_{j=1}^{N_{l,i}} [T_{roll} + (\frac{T_{W,i}}{N_{l,i}} \cdot j)] \cdot p_{a,i} \cdot (1 - p_{a,i})^{j-1}
\end{aligned}
$$

Considering also read-only transaction, whose average service time we denote as T^{RO}, the average service time of a transaction is:

$$
r_{t,i} = w \cdot T^{W,i} + (1 - w) \cdot T^{RO} \tag{66}
$$

3.9 Validation

In this section we provide the results of an evaluation study aimed to verify the accuracy of the proposed modeling methodology. We focus the validation study on the CTL model, as the ETL model will be extensively validated in Section 4, where it will be used as a fundamental building block of two different replication protocol.

The study is based on the comparison between some key performance parameters determined via our analytical model and the corresponding values as obtained by means of a discrete event simulator. The simulation model mimics the execution of a closed system entailing k concurrent worker threads, whose conflicts when executing transactional code portions are regulated by CTL. The simulation results were obtained by repeating a number of independent runs (with different initial seeds for the random generators) until the amplitude of the 90% confidence intervals on the throughput (committed transactions per second) became smaller than 10% of the average throughput value.

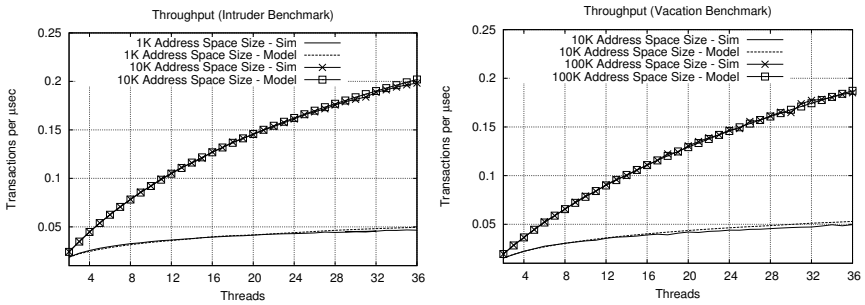


Figure 11: Throughput.

The workload parameters for this study have been selected on the basis of measurement and tracing activities, carried out for the STAMP benchmark [25]. To this end, we have exploited an implementation of TL2 which we have instrumented to trace the data access pattern and the costs associated with the corresponding operations (at the time the validation was performed, Infinispan 5.1, which embeds CTL, was not yet released), as well as the internal operations performed by the STM layer. Measurements have been carried out using a quad-core 2.4 GHz machine equipped with 4 GB of RAM and running the Suse Linux operating system (kernel 2.6.17).

In our study we focus on two of the applications included in the STAMP benchmark, namely Intruder and Vacation. Intruder is a signature-based network intrusion detection system which processes network packets in parallel via a user-tunable number of threads that concurrently update two main data structures, namely a FIFO queue and a self-balancing tree. In this benchmark, each thread spends around 33% of the time executing transactional code, and generates relatively short transactions, belonging to three different classes (capture, reassembly, and detection), the 90% percent of which exhibit a read plus write set made of up to $n = 71$ items, 30% of which are accessed in write mode. Based on our measurements, we set $t_{tcb} = 0.5\mu sec$, $t_{ntcb} = 5\mu sec$ and $t_{commit} = 2\mu sec$.

Vacation, on the other hand, implements an on-line transaction processing system emulating a travel reservation system. The system is implemented as a set of trees that keep track of customers and their reservations for various travel items. Client threads perform a number of sessions, each one enclosed in a coarse-grained transaction (compared to Intruder), which are again differentiated into three classes (reservations, cancellations, and updates), all interacting with the travel system's data layer. In this application, client threads spend almost all their execution time (92%) executing transactions, the 90% percent of which exhibit a read plus write set made of up to $n = 200$ items, 12% of which are accesses in write mode. Based on our measurements, we set $t_{tcb} = 0.2\mu sec$, $t_{ntcb} = 5\mu sec$ and $t_{commit} = 5\mu sec$

In addition to the above parameters, we used our tracing facility to determine also the following set of parameters: $t_{begin} = 0.2\mu sec$, $t_{read} = 0.25\mu sec$, $t_{write} = 0.2\mu sec$, $t_{abort} = 1\mu sec$. Finally, the back-off period, $t_{backoff}$, was set to $2\mu sec$.

By the above description, both the selected benchmark applications entail multi-

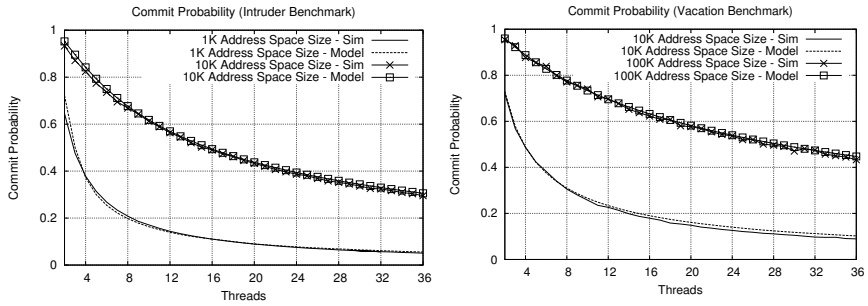


Figure 12: Commit probability.

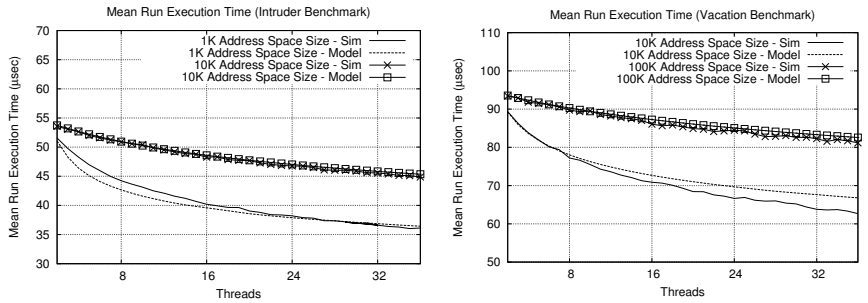


Figure 13: Mean run execution time.

class transactions. Hence the tracing process and the related outcomes have been used in a differentiated manner depending on whether the target is the validation of the single-class or the multi-class model.

To validate the single-class model, we configured the simulator to generate durations of the above mentioned timing activities based on exponential distributions. On the other hand, the validation of the multi-class version of the model, which captures more in detail the execution dynamics of the STM system, has been performed by replaying within the simulator the exact timing of actions as logged in the execution traces.

For what concerns data accesses, the simulator generates them according to a uniform distribution across the total number of d data items/memory words (in compliance with the assumptions of our analytical model). The parameter d is treated as an independent parameter of the validation study. Note that, once fixed the number of worker threads, variations of d allow capturing settings with differentiated levels of contention, which, in their turn, determine different transactions' abort probabilities. Clearly, higher levels of data contention are achieved when the memory is configured with lower values of d, since transactional memory accesses by the worker threads are distributed on a smaller number of distinct memory words. We consider different values for the parameter d, associated, respectively, with reduced and increased values of the benchmarks' data-set size according to the indications provided in [25]. Specifi-

cally, for Intruder, we set d to 1,000 and 10,000, whereas, for Vacation, we set d to 10,000 and 100,000.

3.9.1 Single-class Case

The comparison between analytical and simulation results is based on the following four parameters: (A) the system throughput (Figure 11), (B) the commit probability (Figure 12), (C) the mean execution time evaluated over each single transaction run, independently of whether the run is committed or aborted (Figure 13) and (D) the likelihood of each of the possible causes of transaction abort (Figure 14).

The plots in Figure 11 and Figure 12 point out the accuracy of the presented analytical model, highlighting how analytical and simulation results coincide across the whole considered region of the parameters space, namely low vs high number of worker threads, as well as large vs small address space. By Figure 12, in correspondence with the lower value of d, we can appreciate the accuracy of the analytical model even in high contention scenarios (namely, for very reduced values of the transaction commit probability). By Figure 13, we remark how, when considering the case of smaller address spaces, the relatively high contention probability often leads transactions to be early aborted (i.e., as soon as the first conflicting memory reference is issued), thus contributing to a reduction of the mean value for the run execution time. (Recall that the mean run execution time is evaluated over both committed and aborted run instances.) On the other hand, we observe an increase of the mean run execution time in the configuration with larger address space, where the weight of aborted run instances becomes lower. Note that, due to the aforementioned early abort phenomenon, the variance of the mean run execution time grows in high contention scenarios. The above phenomenon, and their effects on the observed mean value, are correctly captured by our analytical model with very limited error, which is an additional support of the high accuracy of our analytical approach. The only exception is represented by the case of the Vacation benchmark when configured to use the smaller address space. In this case, the accuracy of the analytical model in predicting the mean run execution time is in fact subject to a slight deterioration as the number of worker threads increases. We argue that this is imputable to the fact that the Vacation benchmark comprises transactions whose execution latency is (on average) significantly longer than the Intruder benchmark. This leads to an increase of the variance of the run execution time and to a corresponding amplification of the model's prediction error.

In Figure 14 we evaluate the accuracy of the analytical model in predicting the different causes of aborts for the transactions. Specifically, we set the number of worker threads to eight and report: (i) the probability for a transaction to abort during its execution before reaching the commit phase (recall that this can only happen due to a validation failure during a read operation), denoted as $p_{a,ex} = 1 - p^{o}_{na,n+1}$ (see equations (23-25)); (ii) the probability for a transaction to abort in the commit phase during the writeset lock acquisition, namely p_{wlc} (see equation (26)); (iii) the probability for a transaction to abort in the commit phase due to read-set validation failure, namely p_{rvf} (see equation (33)). Also in this case we observe that the accuracy of the proposed analytical model is very good for the scenarios in which the benchmarks are configured to use the larger datasets. On the other hand, with smaller datasets, namely the ones

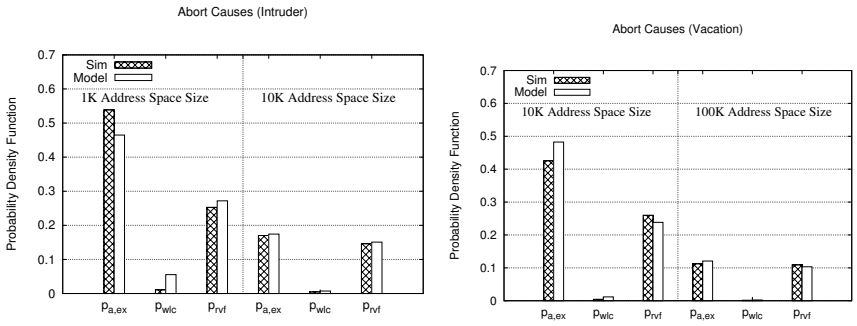


Figure 14: Abort causes.

associated with very high contention rates (note that the probability of abort is around 0.7 and 0.8 in these scenarios), there is a slight degradation of the analytical model accuracy. We argue that this is imputable to the fact that the error introduced by assuming a Poisson assumption for the distribution of the transaction interarrival time to the commit phase, which remains negligible at low/medium contention levels, shows an increasing trend at very high contention levels. This phenomenon is confirmed by the plots in Figure 15, where we evaluate the goodness of this assumption in different workload scenarios by contrasting the empirical density functions of the transaction interarrival time to the commit phase, as computed by the simulator, and the exponential distribution functions whose average value has been computed via the analytical model. More in detail, the plots on the right side of Figure 15 have been obtained by considering moderate contention scenarios obtained by selecting, for each benchmark, the largest address spaces and degree of concurrency equal to eight, that give rise to probability of abort on the order of 20% and 35% for Vacation and Intruder, respectively. On the other hand, the plots on the left side of Figure 15 are associated with a very high (and, arguably, somewhat pathological in practice) contention scenario, in which we select for each benchmark the smallest address spaces and degree of concurrency equal to eight, that give rise to probability of abort on the order of 70% and 80%, for Vacation and Intruder, respectively. The reported results clearly highlight that, up to medium contention levels, there is an excellent match between the empirical and analytical distributions, thus confirming the validity of the Poissionianity assumption for the commit phase arrival in case the timing of actions natively associated with the transactions follows exponential distributions. The left side plots, conversely, highlight a higher discrepancy between the empirical and analytical density functions in very high contention scenarios.

However, it is interesting to highlight that the degradation of the goodness of the poissionianity assumption leads to a (slight) increase of the model's error only when predicting some internal state variables, such as the likelihood of the various abort causes. On the other hand, the model's accuracy in predicting external performance metrics, such as throughput and commit probability, remains very high across every analyzed workload, even those associated with very high contention rate (see Figure 11 and Figure 12).

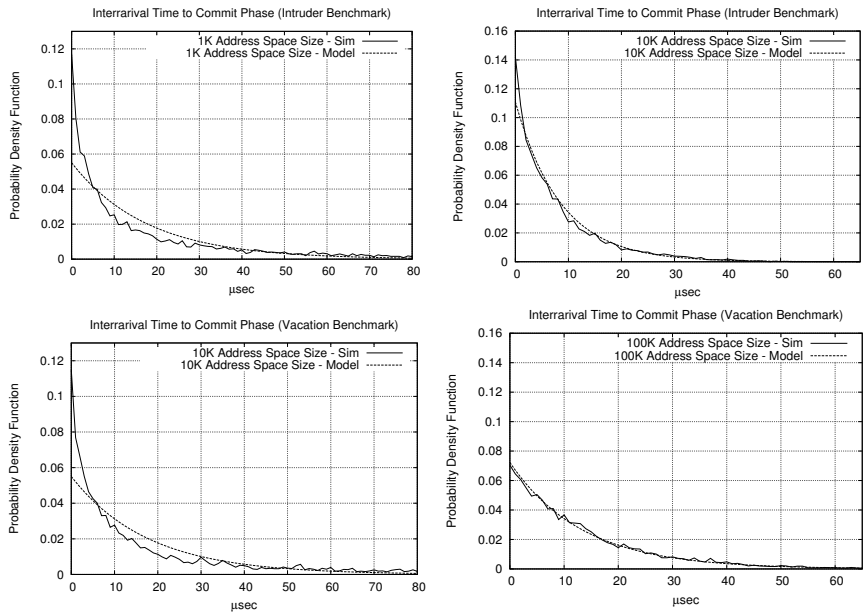


Figure 15: Distribution of the transaction interarrival times to the commit phase.

3.9.2 Multi-class Case

In this section we validate the variant of the analytical model capturing multi-class transactional profiles. To this purpose, the timing of accesses to shared memory data items has been simulated by replaying the execution traces of the Vacation benchmark. On the other hand, we used the reduced data set size selected for this benchmark (i.e., 10,000 data items) in order to stress the accuracy of the model when considering non-minimal contention scenarios. The parameters characterizing this workload are summarized in Table 7.

By the results shown in Figure 16, we have that the analytical model again shows a very good match vs simulative results. In particular, throughput, response time and commit probability for each individual class are evaluated by the model in a very accurate manner while increasing the number of worker threads. Also, the curves show that the matching is good up to a number of worker threads yielding towards flat throughput values.

Parameter	Class 1	Class 2	Class 3
Transaction Class Probability (P^m)	0.898	0.047	0.056
Transaction Class Length (n^m)	154	57	121
Write Probability per Class (p^m_{write})	0.046	0.117	0.080

Table 7: Parameters used for the multi-class study (Vacation benchmark)

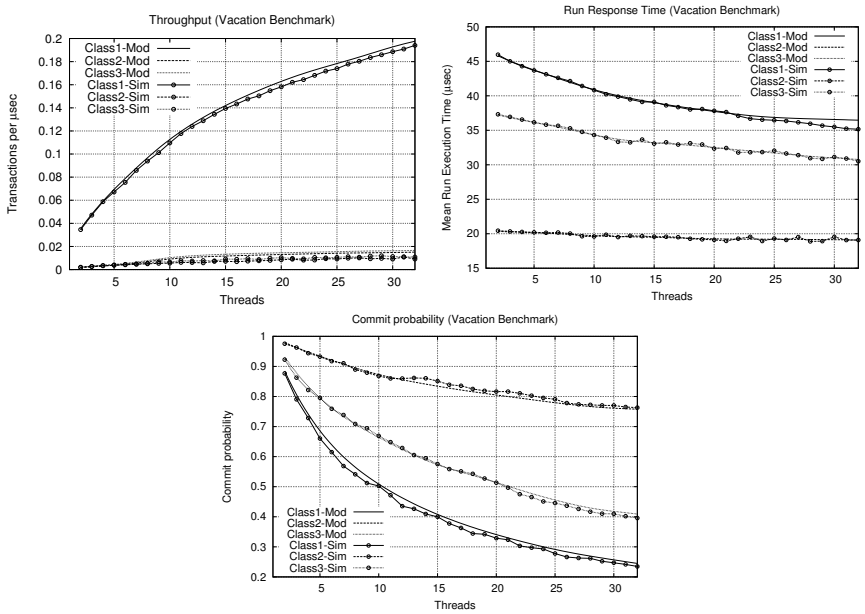


Figure 16: Analytical vs Simulative Results for the Multi-class Scenario.

3.10 Related Work

Perhaps unsurprisingly, the wide majority of existing performance studies on STMs are based on the empirical comparison of different implementation choices, e.g., [45, 51], and on a (very limited) number of simulation-based studies, [68, 98] (the latter one being actually targeted to hardware-implemented transactional-memory systems).

The work in [67] provides an analytical model for STM systems. However, the provided modeling approach suffers from two key limitations, which are overcome by the approach we have presented. First, the model in [67] assumes that applications are constantly executing transactions, while real STM-based applications rely on threads that alternate the execution of transactional and non-transactional code. Second, the model in [67] abstracts over time by describing the execution of a transaction as a sequence of steps whose duration is left unspecified. This restricts the usage of the model exclusively to qualitative comparisons among different STM algorithms, making it infeasible for forecasting fundamental time-related performance metrics, such as response time or throughput (unless when assuming that all the phases of the execution of any transaction have identical, constant duration). Differently, our analytical modeling approach is able to capture the advancement of time according to a continuous timeline. Also, it relies on a detailed workload characterization model, which includes key cost parameters related to both STM internals and STM-based applications, such as the duration of transactional operations (read/write accesses to transactional memory locations, as well as begin/commit/abort operations), and explicitly accounts for the time-interval in between two transactional operations. Additionally, we explicitly

model the relation between the final perceived performance and the time interval spent by each thread outside transactional contexts.

A queuing theory based analytical model is proposed in [66] to evaluate and compare the performance of lock-based and STM-based synchronization schemes. The main limitation of this model is due to the assumption that all transactions (or critical sections) access the same identical memory locations. Conversely, our model captures accesses to distinct locations.

Leveraging on the common notion of atomic transactions, STM algorithms and DBMS concurrency control schemes are naturally closely related. The analytical modeling of concurrency control in database environments has been widely investigated over the last three decades. Analytical modeling approaches have been presented in, e.g., [41, 43, 73, 135–137] for the case of centralized database systems, and in [10, 11] for the case of distributed/replicated databases. However, as already mentioned, the execution time of STM transactions is typically several orders of magnitudes smaller than the counterpart in DBMS scenarios [107], which amplifies the impact of the overhead associated with the STM-specific internal schemes for the management of low-level data-structures (e.g., CTL [45]). These schemes do not have a direct counterpart in the database literature so, consequently, they are not covered by the literature on analytical modeling of concurrency control schemes for database systems. Also, existing performance models of concurrency control schemes do not capture the behavior of applications alternating the execution of transactional and non-transactional phases, as it is conversely typical of STM-based applications.

Finally, concurrency control protocols for database systems, and their impact on performance, have been extensively studied via simulation [1, 3, 27, 120, 121], which is a technique orthogonal to the analytical approach we have provided.

3.11 Integration in the Cloud-TM platform

As discussed, the performance model for CTL presented in this deliverable has been validated against simulative results, mainly due to the unavailability of CTL within Infinispan, unless for very recent releases (namely Version 5.1). Our plan entails further validation steps, in particular when considering analytical vs real data achieved by relying on Infinispan. Also, we plan to use the model mainly as a block for the optimization of the level of concurrency to be adopted on each node hosting an Infinispan cache, to be carried out at the level of the Autonomic Manager of the Cloud-TM platform.

As for the ETL model, validation data against real implementations will be reported in Section 4. In fact, as already hinted, this model has been used as a building block for performance models of already implemented replication protocols, to be presented in that section.

4 Replication protocols

Replication plays a role of paramount importance for in-memory transactional data platforms such as the Cloud-TM platform, both for scalability and fault-tolerant purposes.

Decades of literature and field experience in the area of data replication have brought to the development of a plethora of approaches for state consistency in distributed platforms, and taught a fundamental, general lesson: no universal, one-size-fits-all solution exists that can achieve optimal efficiency across all possible kinds of workloads and for any level of scale of the system. This issue is hence particularly exacerbated in Cloud Computing platforms due to the feature that is regarded as one of the key advantages of the cloud: its ability to elastically acquire or release resources, dynamically varying the scale of the platform in real-time to meet the demands of varying workloads. This means that in order to maximize efficiency (i.e. minimize operational costs, in the pay-for- what-you-use pricing model) data management middleware architectures should be able to adapt their consistency mechanisms in order to ensure optimal performance for every workload and at any scale.

Forecasting the performance of data centric applications deployed on distributed/replicated platforms is extremely challenging. In fact, the performance of distributed data management platform typically exhibit strong non-linear behaviors as the workload and/or the system scale vary.

In this section we report the results so far achieved in the Cloud-TM project in the area of performance modelling of transactional replication protocols.

Our work has encompassed three mainstream replication protocols optimized for different workload types (e.g. conflict-intensive vs low-conflict) and scales of the underlying platform (e.g. small vs large clusters), namely Two Phase Commit (2PC), Primary Backup (PB) and Total Order-based Certification (TOC) protocols.

We start by presenting our results on the modelling of 2PC-based replication in Section 4.1. The PB model is presented in Section 4.2, whereas Section 4.3 reports our results concerning the performance modelling of TOC protocols.

4.1 Two phase commit based replication protocols

This work focuses the issue of forecasting the scalability of a transactional application deployed on in-memory Distributed Transactional Memory (DTM) platforms. We used as reference platform for this study Infinispan, which, as already discussed (see Section 1.1 and deliverable D2.1), represents the backbone of the Cloud-TM platform. Infinispan, like several other recent NoSQL data grids (such as Oracle's Coherence [104] and Apache Cassandra [83]), allows dynamic resizing of the cluster on top of which it is deployed [75, 119]. These platforms allow non-expert users to provision a cluster of virtually any size within minutes. This gives tremendous power to the average user, while placing a major burden on her shoulders. Removing the classic capacity planning process from the loop means in fact shifting the non-trivial responsibility of determining a good cluster configuration to the non-expert user [130].

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 exhibits strong non-linear behaviors. Such behaviors are imputable to the simultaneous, and often inter-dependent, effects of contention affecting both physical (computational, memory, network) and logical (conflicting data accesses by concurrent transactions) resources.

These effects are clearly shown in Figure 17, whose plots report results obtained by running two popular transactional benchmarking frameworks on top of the Infinispan data grid platform [91]: Radargun [3] and TPC-C [4]. We deployed Infinispan over a private cluster encompassing a variable number of nodes and ran benchmarks generating heterogeneous workloads for what concerns number of (read/write) operations executed within each transaction, percentage of read-only transactions, number of items in the whole dataset, as well as size of the individual objects manipulated by each operation.

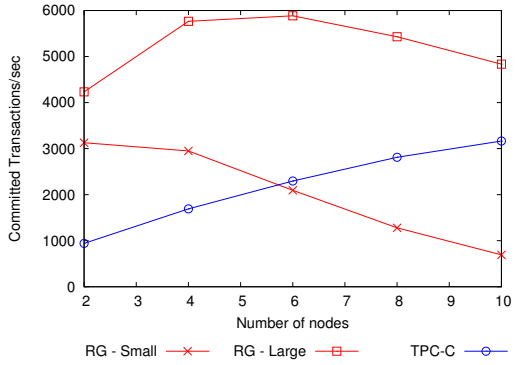
As shown in Figure 17(a), the scalability trends (in terms of the maximum throughput) of the three considered workloads are quite heterogeneous. The TPC-C benchmark scales almost linearly and the plots in Figure 17(b) and Figure 17(c) show that in this case the scalability is limited by a steady increase of contention at both the network and at data (i.e., lock) level. This leads to a corresponding increase of the 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 strongly non-linear 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 section we present Transactional Auto Scaler (TAS), a novel performance prediction methodology based on the joint usage of analytical and machine learning (statistical) models. The analytical model (AM) employed by TAS exploits 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, TAS also exploits black-box, machine-learning (ML) methods to forecast the impact on performance due to shifts in the utilization of system level resources (e.g,. CPU and network) imputable to variations of the system's scale.

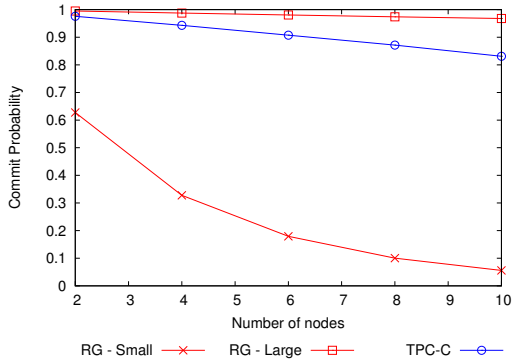
The synergical usage of AM and ML techniques allows TAS to take the best of these two, typically competing, worlds. On the one hand, the black-box nature of ML spares from the burden of explicitly modeling the interactions with system resources that would be otherwise needed using white-box, analytical models. This is not only a time-consuming and error-prone task given the complexity of current hardware architectures. It would also constrain the portability of our system (to a specific infrastructural instance), as well as its practical viability in virtualized Cloud environments where users have little or no knowledge of the underlying infrastructure.

On the other hand, analytical modeling allows to address two well known drawbacks of ML, namely its limited extrapolation power (i.e., the ability to predict previously unobserved scenarios) and lengthy training phase [18]. By exploiting *a priori* knowledge on the dynamics of data consistency mechanisms, AMs can achieve good forecasting accuracy even when operating in previously unexplored regions of

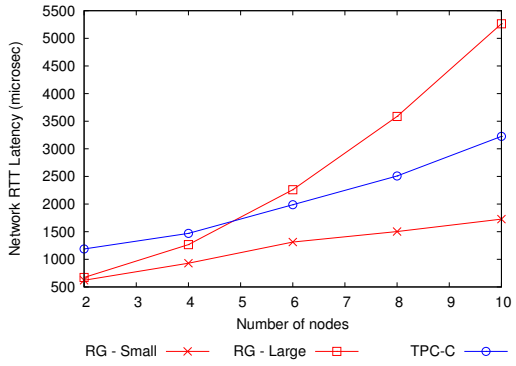
[3] http://sourceforge.net/apps/trac/radargun/wiki/WikiStart
[4] http://www.tpc.org/tpcc



(a) Throughput (Committed Transactions per second)



(b) Transaction Commit Probability



(c) Round-Trip Network Latency

Figure 17: Performance analysis of different data grids applications.

the parameters' space. Further, by narrowing the scope of the problem tackled via ML

techniques, AM allows to reduce the dimensionality of the ML input features' space, leading to a consequent reduction of the training phase duration [18].

While the hybrid AM/ML methodology at the basis of TAS can be applied to a plethora of alternative replication/concurrency control mechanisms, as already mentioned, we have focused our efforts on analyzing the default replication/concurrency control mechanisms used in Infinispan (V.5.0.final). As other recent transactional data grids, e.g., [30, 83], Infinispan opts for guaranteeing a weaker consistency semantic than classic serializability isolation level [15]. Specifically, Infinispan guarantees Repeatable Read [13] by using an encounter based write locking strategy and Two-Phase Commit to detect remote conflicts and enforce transaction atomicity across the whole set of replicas.

One of the key innovative elements of the analytical performance model presented in this section consists in the methodology introduced to characterize the probability distribution of transactions' access to data items. Existing white-box models of transactional systems [33, 42, 134], 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. Even when such approximations are viable, these modeling techniques require complex and time-consuming workload characterization studies, in order to derive the parameters characterizing the data access distributions. In the presented model, conversely, we capture the dynamics of the application's data access patterns via a novel abstraction, which we call *Application Contention Factor* (ACF). ACF exploits queuing theory arguments and a series of lock-related statistics measured in (and dependent on) the current workload/system configuration, in order 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 a large scale evaluation study using both a private cluster and public cloud infrastructures (Amazon EC2), and relying on industry standard benchmarks that generate a breadth of heterogeneous workloads for what concerns contention on both logical and physical resources. The results also highlight that the overhead introduced by TAS' 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.

4.1.1 System Architecture

The reference system architecture for TAS is depicted in Figure 18. Incoming transactions are dispatched by a front-end load-balancer towards the set of nodes composing the data grid. On a periodical basis, statistics concerning load and resource utilization across the set of nodes in the data grid are gathered by a, so called, *aggregator* module. The aggregator module coincides, in the Cloud-TM platform, with the Workload and QoS monitor (see Figure 1 and deliverable D3.1 [113]).

Aggregated statistics are then fed to the *load predictor*, which serves the twofold purpose of forecasting future workload volume and characteristics (e.g., ratio between

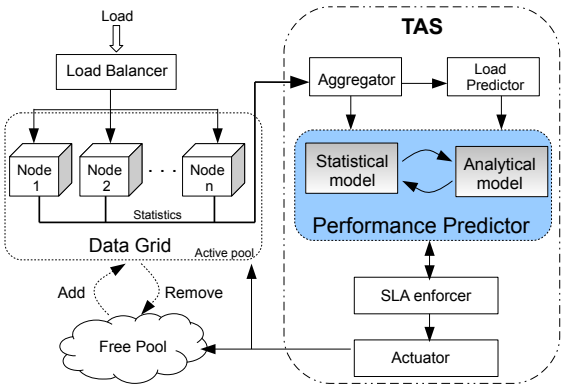


Figure 18: Performance analysis of different data grids applications.

read-only and update transactions, average number of read/write operations for read-only/update transactions), as well as detecting relevant workload shifts. These features are provided by the Workload analyzer of the Cloud-TM platform (see Figure 1 and deliverable D3.2 [114]).

The key innovative point of TAS, which represents the focus of our work, is the methodology employed for predicting the performance of transactional applications when varying the number of nodes of the underlying data grid. More in detail, the *performance predictor* employed by TAS takes as input the workload characteristics (as output 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 (KPI), including average response time, maximum sustainable throughput, transaction abort probability. As shown in Figure 18, TAS relies on the joint usage of a white-box AM (to forecast the effects of data contention) and black-box ML techniques (to forecast the effects of contention on physical resources). A detailed description of the proposed performance forecasting methodology will be provided in Section 4.1.3.

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. As TAS can forecast a number of KPIs (such as response time, throughput, commit probability), our system lends itself to support complex optimization policies involving an ample range of performance (and, of course, cost) constraints.

Finally, the *actuator* is in charge of reconfiguring the system by adding or removing (virtual) servers from the data grid. This component is being developed in the context of deliverable D3.4, and will be described in the associated report. For the moment, it suffices to say that 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, such as Infinispan.

4.1.2 Infinispan Overview

As already mentioned, we selected as target DTM platform for TAS, Infinispan, which is also the DTM platform at the back-bone of the Cloud-TM Data Platform. As TAS employs a white-box analytical model for capturing the effects of data contention on system's performance, in the following we provide an overview of the main mechanisms employed 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. In the following we provide additional details on the concurrency control and replication scheme used by Infinispan, as this will serve as a basis to understand the analytical model presented in Section 4.1.3.

As other recent NoSQL platforms, Infinispan opts for sacrificing consistency in order to maximize performance. Specifically, it does not ensure serializability [15], but only guarantees the Repeatable Read ANSI/ISO isolation level [13]. More in detail, Infinispan implements a non-serializable variant of the multi-version concurrency control algorithm, which never blocks or aborts a transaction upon a read operation, and (up to version 5.0) relies on an encounter-time locking (ETL) strategy to detect write-write conflicts. Write locks are first acquired locally during the transaction execution phase, which does not entail any interaction with remote nodes.

At commit time, Two Phase Commit (2PC) [15] is executed. During the first phase (also called prepare phase), lock acquisition is attempted at all 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 back a commit message, in order to apply the transaction's modifications on the remote nodes, and then it commits locally.

In presence of conflicting, 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 our work, 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 [45] to achieve deadlock freedom. In fact, distributed deadlocks represent a major threat to system scalability, as highlighted by the seminal work in [58] and confirmed by our experimental results.

4.1.3 Analytical modelling of data contention

Our analytical model uses mean-value analysis techniques to forecast the mean probability of transaction commit, the mean transaction duration, and the maximum system throughput. This allows 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 vs 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 of the model. For the sake of simplicity, we will assume these nodes to be homogeneous in

terms of computational power and available RAM, and distinguish only two classes of transactions, namely read-only vs update transactions. A discussion on how to extend the model and relax these assumptions will be provided subsequently.

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 considered 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".

We do not model explicitly the issuing of read operations, as the concurrency control of Infinispan ensures that these are never blocked and can never induce an abort. However, we denote with $T_{localRO}$, resp. $T_{localWR}$, the average time to execute a read-only, resp. update, transaction, namely 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 phase of 2PC, which includes broadcasting the prepare message, acquiring locks at all replicas, and gathering their replies. Note that the value of T_{prep} (and, in principle, also of $T_{localWR}$/$T_{localRO}$) 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). As these phenomena are captured in TAS via machine-learning techniques (described in Section 4.1.4), the analytical model treats T_{prep} and $T_{localWR}$/$T_{localRO}$ simply as input parameters.

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

Data access pattern characterization. In order to compute the response time for a transaction, we need first to obtain the probability that it experiences a local or remote lock contention, that is whether it requires a lock currently held by another transaction. Note that in the modelled concurrency control algorithm, 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 of locking [127, 152], 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 which takes an average time T_H before completing the "service of a lock request" (i.e., freeing the lock) . This level of abstraction allows approximating 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 $U = \lambda_{lock} T_H$ [79] (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 insist on a specific datum. Existing techniques, in fact, assume uniformly distributed accesses on one set citedisanzo-stm (or more sets [134]) of data items of cardinality D (where D is assumed to be *a prior*i known) and compute the probability of lock contention on any of the data items simply

as:

$$P_{lock} = \frac{1}{D}\lambda_{lock}T_H \tag{67}$$

Unfortunately, the availability of information on D, and the assumption on the uniformity of the data access patterns strongly limits the employment of these models in the context of complex applications, especially if these exhibit dynamic shifts in the data access distributions. We overcome these limitations by introducing a powerful abstraction that allows the on-line characterization of the application data access pattern distribution in a lightweight and pragmatical manner. We call this abstraction *Application Contention Factor* (ACF). The ACF captures, roughly speaking, the probability for a transaction to encounter lock contention upon a write operation given that there is one *concurrently executing* transaction that has already acquired all its locks. The ACF has two attractive features that make it an ideal candidate to characterize the data access patterns of complex transactional applications:

1. It is computable on-line, on the basis of the values of P_{lock}, λ_{lock} and T_H measured in the current platform configuration by exploiting Eq. 67:

$$ACF = \frac{P_{lock}}{\lambda_{lock}T_H} \tag{68}$$

   By Eq. 67, 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 incurred in the same contention probability experienced during its actual execution (in which it generated arbitrary, non-uniform access patterns).

2. As we will show in Section 2.3.2, even for applications with arbitrary, complex data access patterns (such as in TPC-C, whose access pattern has strong skew and is very hard to model analytically), ACF is an invariant with respect to the arrival rate, degree of concurrency in the system (i.e., number of nodes/threads generating transactions) and physical hardware infrastructure (e.g., private cluster vs public cloud platform).

The ACF abstraction represents the foundation on top of which we built the AM of the lock contention dynamics, to be discussed shortly. This model allows predicting the contention probability that would be experienced by an application in 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).

Lock contention model. Denoting with λ^l_{lock}, respectively λ^r_{lock}, the lock request rate generated by local, respectively remote transactions, on a given node, we can compute them as:

$$\lambda^l_{lock} = \frac{\lambda_{Tx} \cdot w \cdot \tilde{N}_l}{\nu}, \;\; \lambda^r_{lock} = \tilde{N}_r \cdot \lambda_{Tx} \cdot w \cdot \frac{\nu - 1}{\nu} \cdot P_p$$

where we have denoted with P_p the probability for a transaction to reach the prepare phase (i.e., not aborting earlier), and with $\tilde{N}_l$, respectively $\tilde{N}_r$, the number of locks acquired successfully acquired on average by local, respectively remote, transactions, independently from whether they abort or commit.

When a transaction executes locally, it can experience lock contention (and therefore abort) both with other local transactions and remote ones. By using Eq. 67, we can therefore compute the probability of abort during local transaction execution, P_a^l, as:

$$P_a^l = P_{lock}^l = (\lambda_{lock}^l + \lambda_{lock}^r) \cdot ACF \cdot T_H \tag{69}$$

The probability P_a^r for a remote transaction T to encounter contention upon any lock request issued during its prepare phase with a transaction T' on any node of the data grid can be instead approximated by considering exclusively the probability for T to contend with T' on the node $\nu_{T'}$ that generated the latter transaction. In fact, if T were to contend with T' at a node different from $\nu_{T'}$, then, with very high probability, T would also encounter lock contention with T' also when trying to complete its prepare phase on $\nu_{T'}$. As a consequence we can compute P_a^r as:

$$P_a^r = \lambda_{lock}^l \cdot ACF \cdot T_H^l$$

where T_H^l denotes the mean lock hold time for a local transaction. Thanks to this approximation, we can consider as independent the remote abort probabilities for a transaction on different nodes.

By the above probabilities, we can compute the probability that i) a transaction reaches its prepare phase (P_p), ii) successfully completes its prepare phase on all the $N-1$ remote nodes (P_{coher}), and iii) commits (P_c):

$$\begin{aligned} P_p &= (1 - P_a^l)^{N_l} \\ P_{coher} &= (1 - P_a^r)^{N_l \cdot (\nu - 1)} \\ P_c &= P_p \cdot P_{coher} \end{aligned}$$

We can now compute the mean number of locks successfully acquired by a transaction, $\tilde{N}_l$, taking into account that it can abort during its execution:

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

In order to compute $\tilde{N}_r$ we use a similar reasoning:

$$\tilde{N}_r = (1 - P_a^\dagger)^{N_l} \cdot N_l + \sum_{i=1}^{N_l} P_a^\dagger \cdot (1 - P_a^\dagger)^{i-1} \cdot (i-1)$$

with the exception that in this case we estimate the probability to incur in lock contention taking into account that there cannot be remote contention between two transactions originated by the same node:

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

Where we denoted with $T_H^\dagger$ the average lock holding time of the transactions with which it is possible to experience contention during the prepare phase, which we estimate as:

$$T_H^\dagger = \frac{\lambda_{lock}^l \cdot T_H^l + \lambda_{lock}^r \cdot \frac{(\nu-2)}{(\nu-1)} \cdot T_H^r}{\lambda_{lock}^l + \lambda_{lock}^r \cdot \frac{(\nu-2)}{(\nu-1)}}$$

In order to compute the aforementioned 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 (recalling that we are assuming that the average time between two lock requests is equal to $\frac{T_{localWR}}{N_l}$):

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

We can then 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}), remotely (H_l^{ra}) or successfully completes (H_l^c).

$$\begin{aligned}
T_H^l &= H_l^{la} + H_l^{ra} + H_l^c \\
H_l^{la} &= \sum_{i=2}^{N_l} P_a^l \cdot (1 - P_a^l)^{i-1} \cdot \frac{1}{i-1} \cdot G(i-1) \\
H_l^{ra} &= P_p \cdot (1 - P_{Coher}) \cdot \frac{1}{N_l} \cdot [T_{prep} + G(N_l)] \\
H_l^c &= P_p \cdot P_{Coher} \cdot \frac{1}{N_l} \cdot [T_{prep} + G(N_l)]
\end{aligned}$$

Let us now compute the remote lock hold time, T_h^r. 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:

$$T_h^r = (1 - P_a^\dagger)^{N_l} \cdot [T_{prep} + (1 - P_a^r)^{N_l \cdot (\nu-2)} \cdot T_{com}]$$

where $(1 - P_a^\dagger)^{N_l}$ represents 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 $(1 - P_a^r)^{N_l \cdot (\nu-2)}$ represents the probability for T to successfully acquire its remote locks on the remainder $\nu - 2$ 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^W, is the average among these cases:

$$T^W = T_c + T_a^l + T_a^r$$

where

$$T_c = P_c \cdot (T_{localWR} + T_{prep} + T_{comm})$$
$$T_a^l = \sum_{i=1}^{N_l} [T_{roll} + (\frac{T_{localWR}}{N_l} \cdot i)] \cdot P_a^l \cdot (1 - P_a^l)^{i-1}$$
$$T_a^r = P_p \cdot (1 - P_{coher}) \cdot (T_{localWR} + T_{prep})$$

Considering also read-only transaction, the average service time of a transaction, denoted as T, is:

$$T = w \cdot T^W + (1 - w) \cdot T^{localRO} \tag{70}$$

AM resolution and predicted KPIs. As in previous analytical models of transactional data contention [44, 152], also our model exhibits a mutual dependency between the abort probabilities and other parameters, such as the mean hold time. Prior art copes with this issue by using an iterative scheme in which abort probabilities are first initialized to zero. Next, the depending parameters are computed, and, 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 [152] that this iterative solution technique can suffer from convergence problems at high contention rates. We tackle this issue by adopting a binary search in the bi-dimensional 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 2.3.2, for which we set $\epsilon = 0.001$ and observed convergence in at most 11 iterations.

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

The maximum throughput can be computed by exploiting Little's law [89] in an iterative fashion. At the first step of the iteration, an upper-bound on system throughput is provided as input to the model, computed assuming no conflicts and that all threads in the system constantly execute transactions. This corresponds to setting:

$$\lambda = \frac{\nu\theta}{w \cdot (T_{localWR} + T_{prep} + T_{comm}) + (1 - w) \cdot T_{localRO}}$$

At each step, a new value of λ is fed in input to the model, replacing the denominator of the above equation with the value of T (see Eq. 70) computed in the previous iteration, till convergence to the desired precision is reached.

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 [17] 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, w, of a G/G/θ queue in heavy traffic is exponential and given by :

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

where σ_u is the inter-arrival 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 [132]), λ is the request arrival rate, and T_s is the average service time. The above 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:

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

in which we set the queue average service time (T_s) equal to the average execution time T output by Eq. 70.

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.

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 clustering techniques, such as the one used, e.g., in [57].

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 [152], in which the transaction conflict probability is computed taking into 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 in the Cloud [130, 132], 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 [130], or, whether possible, directly the target application, as in [132]), it is easy to determine scaling factors relating the performance achieved when deploying the application on different type of instances. For example, a medium instance performs 1.5 times better than a small instance, or a large instance

provides 2x 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.

4.1.4 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 physical resources depending on workload's fluctuations or to the re-sizing of the data grid. Developing white-box models capable of capturing accurately the effects on performance due to contention on hardware resources can in fact be very complex (or even non-feasible, especially in virtualized cloud infrastructures), given the difficulty to gain access to detailed information on the exact dynamics of hardware-level components.

In TAS we exploit the availability of a complementary white-box model of 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., [130], 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 physical 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 physical 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 2PC scheme (once reached their commit phase) as well as the percentage of CPU time consumed by the threads in charge of processing local transactions.

As already mentioned, contention on physical resources can have a direct impact on the execution time of two key phases of transactions' execution, namely the duration of the local transaction processing phase, denoted as $T_{localWR}$ and $T_{localRO}$, and the network latency incurred in by transactions while executing the 2PC protocol, denoted as T_{prep}.

However, in all the workloads analyzed during our experimental analysis, it was found that the factor whose variations dominate the system's performance (upon a change of the platform's scale) is by far the network latency associated with the execution of the prepare phase of 2PC (i.e. T_{prep}). This is not surprising, as in normal settings for this kind of platforms, the system bottleneck is typically the network rather than the CPU.

In order to forecast the shifts of T_{prep}, TAS uses the machine learning-based performance model of asynchronous broadcast RPC presented in Section 2.1. As already discussed in Section 2.1, in fact, capturing the performance dynamics of 2PC was one of the key motivations that brought us to develop forecasting models of asynchronous broadcast RPC primitives.

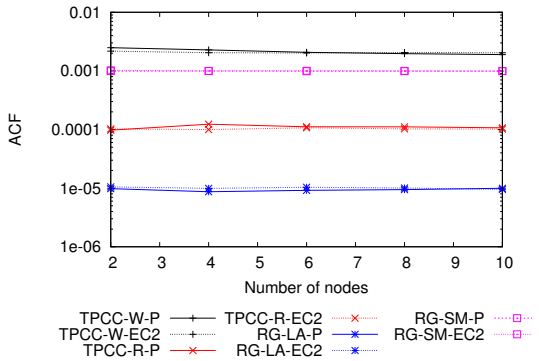


Figure 19: ACF of heterogeneous benchmarks.

4.1.5 AM and ML coupling

By the above 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 the T_{prep} 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 on the level of resource contention in the target configuration.

For simplicity, the current prototype solves this problem by using the following fixed point iterative solution, which, in our experiments, has never shown convergence problems: the AM is initialized with the current values of T_{prep}, it outputs the estimated throughput in the target configuration, and provides it as input feature to the ML to obtain a new value of T_{prep}. The process is repeated till the requested precision is reached. Note that, also in this case, we may have employed a binary search technique analogous to the one described in Section 4.1.3. Such a technique provides stronger convergence properties, at the cost of a higher complexity, as, in this scenario, it would need to operate on a three-dimensional space.

4.1.6 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 experimental platforms used in our study.

Workloads. We consider two well-known benchmarks, already mentioned in Section 4.1, namely TPC-C and Radargun. 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 ones, but have the advantage of being very easily tunable, thus allowing assessing the accuracy

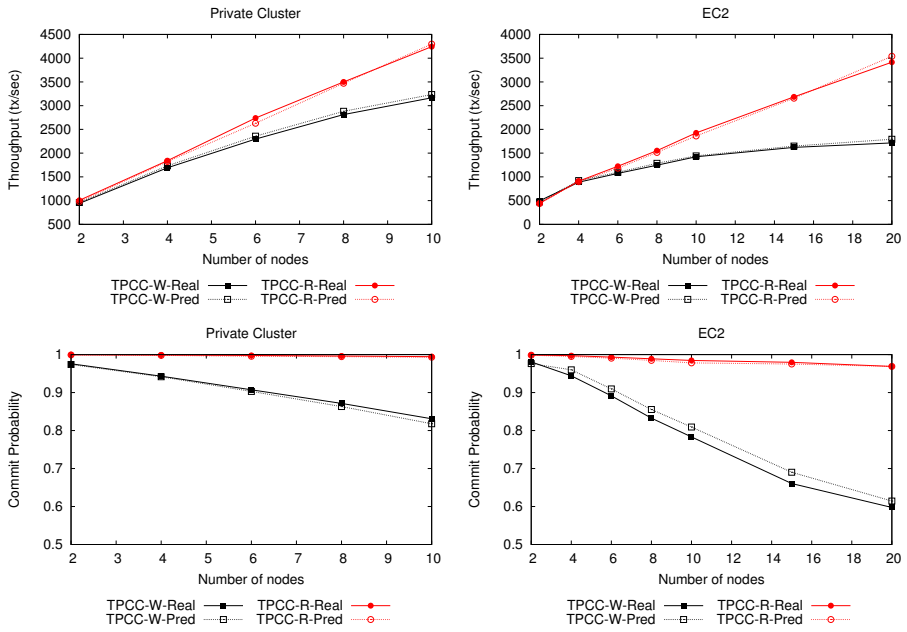


Figure 20: Validation using the TPC-C benchmark

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 (containing 90% read-only transactions) which generates reduced contention on both physical and data resources as the scale of the cluster grows. The second (TPCC-W) include around 50% of update transactions and generate a high data contention level.

Also for Radargun we consider two workloads, denoted as RG-LA and RG-SM. Both workloads generate uniform data access patterns, but RG-LA performs, in each transaction, a single put operation over a set of 100K data items, yielding a very low contention rate. RG-SM, instead, updates in each transaction 10 data items selected over a set of cardinality 1K, thus generating a very high contention probability. We decided to use the Radargun workloads in our evaluation study because their data access patterns are particularly simple and easily predictable, thus allowing us to validate the correctness and semantics of the ACF abstraction.

Experimental Platforms. We use, as experimental test-beds for this study, both a private cluster and Amazon EC2. The private cluster is composed by 10 servers equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) processors and 8 GB 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 4 virtual cores with 2 EC2 Compute Units each.

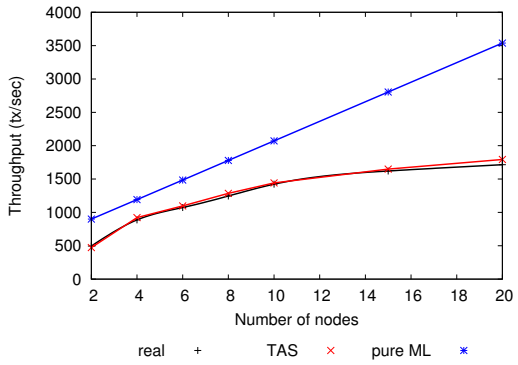


Figure 21: Comparing TAS with a Pure ML approach

ACF validation. In Figure 19 we report the ACFs obtained when running both 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 plots confirm our finding, namely that, once fixed an application workload, the ACF represents an invariant across platforms of different scale, even when deployed on infrastructures of different nature (private vs public). It is noteworthy to highlight that the ACF value is equal to 1E-5, resp. 1E-3, for the workloads RG-LA, resp. RG-SM. We recall that these workloads generate uniform accesses to datasets of size 100K, resp. 1K, items. Therefore, these results confirm that ACF can be interpreted as the inverse of the size of an equivalent, uniformly accessed, dataset.

AM/ML validation Let us now evaluate the accuracy of the final performance predictions output by TAS when jointly using the AM and the ML. We use as KPIs the maximum throughput and commit probability. We report in Figure 20 the forecasts for the TPC-C workloads. We do not include the plots also for Radargun, as they show analogous trends. 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 commit probability. More in detail, TAS achieves a remarkable average relative error (defined as $|real - pred|/real$) on the predicted throughput of 2%, with a maximum of 3.5%.

Comparison with a pure ML approach We conclude 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 [31, 57]. 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, along with number of nodes and active threads per node. As in the previous evaluation study, we use maximum throughput as the output variable. These experiments

were performed using Amazon EC2.

As 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 non-linear scalability trend. As expectable [31], in these conditions, the pure ML-based approach manifests its limits in terms of reduced extrapolation power. In fact, the plots in Figure 21 clearly highlight that the pure ML-based solution tends to mimic the linear scalability trend that it observed during 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, to some extent, addressed 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 systems are influenced by a wide range of parameters [15, 44], 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 [18]).

The AM employed by TAS, on the other hand, can exploit the *a priori* knowledge on the dynamics of data consistency mechanisms to achieve a higher extrapolation power. Further, it allows 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 duration.

As a final remark, it is noteworthy to highlight that, in all our experiments, the performance attained with or without the monitoring framework were indistinguishable. Also, the time required to instantiate and solve a TAS query is on the order of a few hundreds of milliseconds, highlighting the practical viability of the proposed solution to support on-line what-if analysis and automatize elastic scaling.

4.1.7 Related Work

The present work is related to the literature on performance modeling and prediction for transactional systems. This includes both performance models for traditional database systems and related concurrency control mechanisms (see, e.g., [12, 93, 134, 152]), approaches targeting more recent STM architectures (see, e.g., [127]), distributed/replicated transaction processing systems, such as [33], or multi-tier system [40]. With respect to these approaches, TAS presents two key differences: i) it relies on analytical modeling only for capturing data contention dynamics, whereas it relies on black-box statistical methods to model the effects of contention on data resources; ii) from an analytical modeling perspective, in TAS we introduce a novel abstraction (ACF) that allows to concisely characterize and effectively reason about arbitrary transactional data access patterns.

Our work has clearly also relationships with systems that rely solely on ML techniques to automate resource provisioning both in transactional [31, 57, 132, 148] and non-transactional application domains, such as MapReduce [70], VM sizing [142], Grid resource brokering [48] and online gaming [99]. As it will be shown in Section 2.3.2, the joint usage of AM and ML, which represents one of the key innovative characteristics of TAS, allows to enhance the extrapolation power and reducing the training

time of pure ML-based performance predictors.

Control theory techniques are also at the basis of several works in the area of self-tuning of application performance. These solutions often assume a linear performance model, which is possibly updated adaptively as the system moves from one operating point to another. For example, first-order autoregressive models are used to manage CPU allocation for Web servers [143]. Linear multi-input-multi-output (MIMO) models have been applied to manage different kind of resources in multi-tier applications [105], as well as to allocate CPU resource for minimizing the interference between VMs deployed on the same physical node [101]. Compared to these adaptive linear models, the continuous non-linear models used by TAS to forecast both the logical and physical contention can accurately capture the system's entire behavior and allow optimized resource allocation over the entire operating space.

4.1.8 Integration in the Cloud-TM platform

TAS has been designed to forecast the performance of Infinispan (in particular version 5.0, which embeds ETL), which is the reference DTM of the Cloud-TM platform. Thus, TAS is already integrated in the Cloud-TM platform, and our plan is to use it to automate elastic scaling of the Cloud-TM platform.

4.2 Primary-backup replication protocol

As already mentioned, the methodology presented in the previous section (relying on the joint usage of black-box machine learning and white-box analytical modelling) can be used to derive performance models of different replication strategies. In this section we show how this methodology can be applied to the case of Primary Backup (PB) replication [23], a.k.a. passive replication, whose behavior is illustrated by the diagram in Figure 22.

In this case, the front-end load balancer is configured to route update transactions exclusively towards a single node (called *primary* node), and to distribute read-only transactions among the (remaining) nodes of the data platform. The primary node is in charge of regulating the concurrent execution of update transactions and, to this end, it can employ several local concurrency control algorithms. In this work we have considered the concurrency control algorithm used in Infinispan 5.0, as Infinispan, as already discussed, represents the reference DTM implementation for the Cloud-TM project. This concurrency control scheme is, indeed, very similar to the one locally used by each replica when using 2PC, which was described in Section 4.1 and that is briefly recalled here for self-containment. Unlike classic Two Phase Locking (2PL) [15] Infinispan does not use locks for read operations. Conversely, it implements a lightweight, non-serializable variant of the multi-version concurrency control algorithm, which never blocks or aborts a transaction upon a read operation. Upon the issuing of a write operation (whenever a key/value pair is updated/inserted/deleted.), which in the PB approach can only take place on the primary node, locks are locally acquired using an eager strategy.

During the transaction commit's phase, the updates are broadcast to the backups via a FIFO reliable channel [62]. The backups acquire in their turn locks, apply locally the modifications to the updated key/values pars, and acknowledge the master, which waits to gather acks from the whole set of backups before applying locally the updates and release its local locks.

Unlike 2PC-based replication, PB ensures avoidance of distributed deadlocks, being subject exclusively to the possibility of local deadlocks among update transactions executing at the primary node. This allows PB to outperform significantly 2PC in scenarios of high data contention, in which the latter replication scheme becomes subject to severe thrashing.

In Figure 23 we contrast the performance of Infinispan, using either PB or the 2PC scheme described in Section 4.1, while varying both i) the scale of the underlying data platform (on the x-axis), and ii) the transactional workload in input to the system. Our experimental platform for this study is a cluster comprising up to 10 physical machines (each one equipped with 2 quad-core Xeon processors at 2.13GHz and 8GB of RAM), and generated a synthetic workload entailing transactions performing on average 10 data accesses, 10% of which being put (i.e. write) operations, distributed uniformly over keysets of cardinality 1'000, 10'000 and 100'000 keys.

Our experimental data, plotted in Figure 23 highlight that the two replication schemes exhibit dramatic performance differences as the number of replicas and the workload characteristics vary. In fact, by allowing transactions to be executed on each node, 2PCR exhibits the potential for higher scalability at low contention rates, as con-

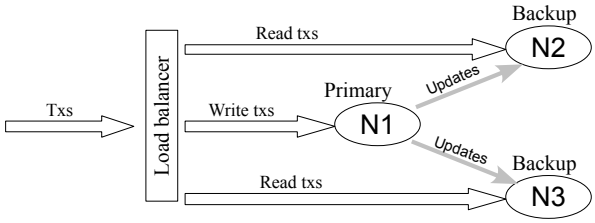


Figure 22: Reference architecture for the primary-backup replication scheme

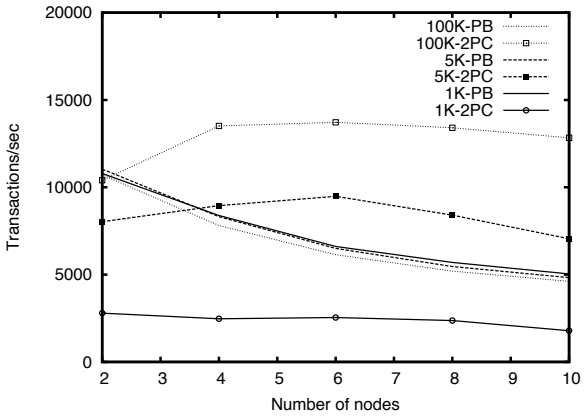


Figure 23: Contrasting the performance of PB and 2PC-based replication protocols.

firmed by the 100K keyset-size scenario. On the other hand, 2PC-based replication protocols are prone to thrashing effects, due to significant increase of the distributed deadlocks' rate when the level of conflict grows beyond some critical threshold, which is what happens for the 1K keyset-size case. In these scenarios, PB can deliver significantly higher performance.

In the remainder of this Section, we show how the TAS methodology can be instantiated to predict the performance of PB. We choose to focus exclusively on the analytical modelling of data contention, as , for what concerns the modelling of communication latency (associated with the propagation of the updates from the primary towards the backups), it is possible to use the machine-learning based approach aimed at forecasting Broadcast-based Synchronous RPC presented in Section 2.1 and already successfully used in Section 4.1.

4.2.1 Analytical modelling of data contention

The performance model presented in this sections, being developed using the same framework presented in Section 4.1, employs the same (ACF-based) methodology for capturing transactions' data access patterns and adopts the same notations used in Sec-

tion 4.1.3. Thus, for the sake of brevity, we decided to avoiding replicating this information also here.

Let us denoting with λ^l_{lock} the lock request rate generated by local transactions on the primary node. This can be computed as:

$$\lambda^l_{lock} = \lambda_{Tx} \cdot w \cdot \tilde{N}_l$$

where we have denoted with λ_{Tx} the rate of transactions in input to the system, and with $\tilde{N}_l$ the number of locks acquired successfully acquired on average by an update transaction executing on the primary, independently from whether it aborts or commits. Note that update transactions are serialized on the primary during their execution phase, and that the primary sends a commit request for a transaction T, only if it has received acks from all the backups for every committing transaction T' that conflicts with T. This implies that lock requests on the backups can not fail, and, therefore, lock contention dynamics on the backups are not captured by this model.

When an update transaction executes on the primary, it can experience lock contention and therefore abort[5]. By using Eq. 67 (see Section 4.1.3), we can therefore compute the probability of abort during the execution phase of an update transaction at the primary, P_a, as:

$$P_a = P_{lock} = \lambda^l_{lock} \cdot ACF \cdot T_H \tag{71}$$

By the above probabilities, we can compute the probability that a transaction reaches its commit phase (P_cp) as:

$$P_{cp} \quad = \quad (1 - P_a)^{N_l}$$

We can now 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:

$$\tilde{N}_l = P_p \cdot N_l + \sum_{i=2}^{N_l} P_a \cdot (1 - P_a)^{i-1} \cdot (i - 1)$$

In order to compute the aforementioned 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 (recalling that we are assuming that the average time between two lock requests is equal to $\frac{T_{localWR}}{N_l}$):

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

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 suc-

[5]Recall that we use a contention management strategy that aborts transactions that encounter contention, in order to guarantee deadlock-freedom. See Section 4.1 for a more detailed discussion on this choice.

cessfully completes (H_c).

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

where we have denoted with T_{cd} the duration of the synchronous broadcast RPC issued by the primary to disseminate the updates toward the backup. Analogously to the duration of the prepare phase of 2PC (T_{prep}) in Section 4.1, also in this case we rely on the machine-learning based methodology presented in Section 2.1 to predict the value of T_{cd} given the current workload and scale of the platform.

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

$$T^W = T_c + T_a$$

where

$$T_c = P_{cp} \cdot (T_{localWR} + T_{prep})$$
$$T_a = \sum_{i=1}^{N_l} [T_{roll} + (\frac{T_{localWR}}{N_l} \cdot i)] \cdot P_a \cdot (1 - P_a)^{i-1}$$

Considering also read-only transaction, the average service time of a transaction, denoted as T, is:

$$T = w \cdot T^W + (1 - w) \cdot T^{localRO} \tag{72}$$

The maximum throughput achievable by the system as whole, denoted as λ^{max}, can be computed taking into account that the bottleneck can either be the primary, which processes update transactions at a rate:

$$\lambda^{primary} = \lambda^{max} \cdot w = \theta / T^W$$

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

$$\lambda^{backups} = \lambda^{max} \cdot (1 - w) = \frac{\theta(\nu - 1)}{T^W \cdot (1 - w)}$$

.

By combining the two equations above, we have:

$$\lambda^{max} = \min\{\frac{\theta}{w \cdot T^W}, \frac{(\nu - 1) \cdot \theta}{T^{localRO} \cdot (1 - w)}\} \tag{73}$$

4.2.2 Validation

In this section we report the results of an experimental study aimed which is aimed at evaluating not only the accuracy of the presented performance model of PB, but also at assessing the flexibility of the TAS methodology with alternative replication protocols.

Experimental Platforms. The experimental test-bed for this study is a private cluster composed by 10 servers equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) processors and 8 GB of RAM and interconnected via a private Gigabit Ethernet.

Workloads. We consider three different workloads generated using the Radargun benchmark (which has been already described in Section 4.1.6), which generate very heterogeneous data contention levels. The workloads are generated by injecting write transactions on the primary node via 8 threads that process update transactions in a closed-loop with no think time in between transactions. This allows easily determining the maximum throughput achievable by the primary (in terms of committed update transactions), whose data contention dynamics represent the main focus of the presented analytical model.

In all the considered workloads (update) transactions generate 10 accesses, 10% of which are update operations (hence triggering a lock request on the update datum). The accesses are performed with equal probability (uniform distribution) over data-sets of different sizes, namely 100K, 10K and 1K keys. Clearly, reducing the data-set size translates in an increase of the transaction conflict probability and in an increase of the application's ACF.

As for the backups, the workload is composed by read-only transactions that access 10 uniformly selected data items. Like for the primary, we inject transactions in a closed-loop using a number of threads equal to the number of cores available on the underlying machine (i.e. 8).

We decided to use the Radargun workloads in our evaluation study because their data access patterns are particularly simple and easily predictable, thus allowing us to validate the correctness and semantics of the ACF abstraction.

Experimental Platforms. We use, as experimental test-beds for this study, both a private cluster and Amazon EC2. The private cluster is composed by 10 servers equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) processors and 8 GB 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 4 virtual cores with 2 EC2 Compute Units each.

ACF validation. In Figure 24 we report the ACFs obtained when running the Radargun workloads. The plots confirm the validity used to derive the ACF also in the PB scenario. In particular the curves in the plot show that, even varying the number of nodes in the cluster, the ACF is equal to the inverse of the data set size accessed (uniformly) by the transactions, which coincides with the interpretation of the ACF given in Section 4.1.3. The plots report (in red) also the ACF measured using the 2PC-based replication scheme described in Section 4.1, highlighting how ACF remains invariant

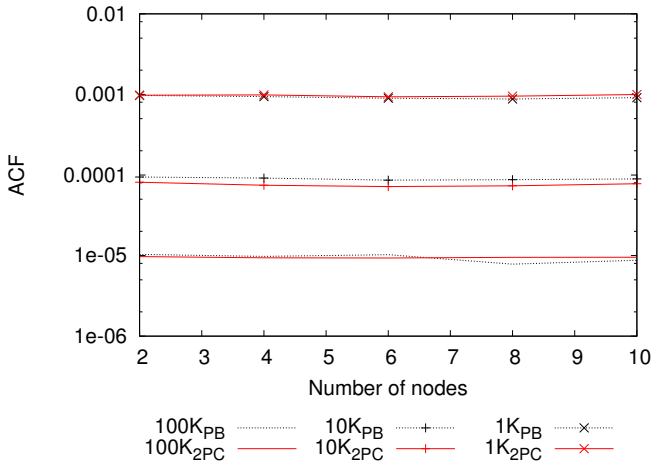


Figure 24: ACF using PB and 2PC-based replication and workloads generating different contention levels.

across different replication protocols if the same workload is provided as input to the system.

This confirms the appropriateness of the ACF to characterize application's data access patterns in a way that is independent from i) the current degree of parallelism in the system (unlike for instance the transaction commit probability), ii) the actual data access pattern distribution, and iii) the replication/concurrency protocol used for ensuring data consistency.

AM/ML validation Finally, Figure 25 reports the results of the forecasting model, by contrasting the maximum throughput on the primary as estimated by the model and the actual maximum throughput of the primary as measured on the real system. To instatiate the model, we compute its input parameters based on measuring gathered by running the cluster with 2 nodes. We then use the analytical model presented in Section 4.2.1 in combination with the machine learning model of synchronous broadcast RPC presented in Section 2.1, to obtain an estimate of the maximum throughput on the primary when deploying the system on an infrastructure of size [4,6,8,10] nodes. We do not report the results of the prediction for read-only transactions, as these simply scale linearly with the number of nodes in the system, given that they do not require any kind of synchronization.

The plots confirm the very good accuracy of the proposed performance prediction scheme both in terms of trends and of relative accuracy. More precisely, the average prediction error results lower than 10% with a maximum of 20% in correspondence of 4 nodes that was found to be due to a drop in the precision of the ML-based predictions of the commit duration.

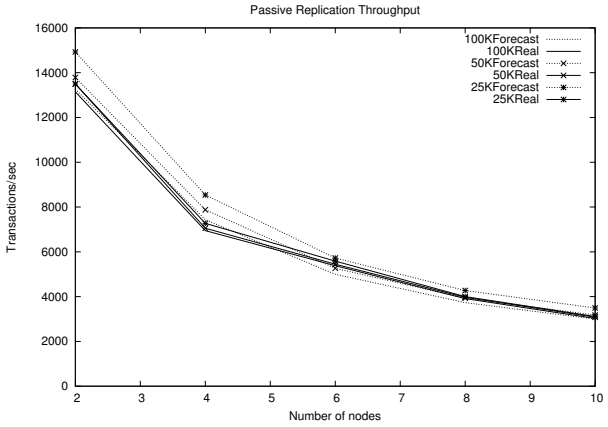


Figure 25: Performance forecasts in the three considered workloads.

4.2.3 Integration in the Cloud-TM platform

In order to develop this performance model and its validation, during WP2 we have extended Infinispan to support a primary-backup replication scheme. At current date, WP3 is working in order to integrate these performance models in the Autonomic Manager.

4.3 Total order based certification protocols

Replication protocols based on Total Order (TO) and certification have garnered a lot of interest in recent literature [77, 108, 110]. Unlike distributed eager-locking schemes, TO-based certification based schemes avoid any onerous replica coordination during the execution phase, running transactions locally in an optimistic fashion. The consistency of replicas (typically, 1-copy serializability [15]) is ensured at commit- time, via a distributed certification phase that uses a single TO to enforce agreement on a common transaction serialization order, avoiding distributed deadlocks, and providing non-blocking guarantees in the presence of failures. This is an extremely important property, as it is widely recognized [58] that distributed deadlocks represent one of key scalability bottlenecks of lock-based multi-master schemes, such as the 2PC-based replication described in Section 4.1. The plots in Figure 26 and Figure 27 contrast the performance of a 2PC-based and a TO-based replication strategy. These two protocols have been integrated in Infinispan in the context of WP2, and were configured to ensure either Read Committed (RC) or Repeatable Read + Write-Skew check (RR+WS) isolation levels. The results highlight precisely that, in workloads with non-minimal data contention, the avoidance of distributed deadlocks allows TO-based certfication schemes to achieve dramatic performance increases with respect to conventional 2PC-based replication schemes.

Of course TO-based certification protocols can also ensure stronger consistency criterion and, in this section, we will focus in particular on the issue of deriving performance models of serializable TO-based certification protocols.

In the design space of 1-copy serializable certification replication protocols a decision that can have a dramatic impact on the actual efficiency and robustness of the system is related to how to address the trade-off between the size of the messages sent via the TO primitive and the number of communication steps required during the transaction commit phase. Depending on how this trade-off is addressed, existing certification-based replication algorithms can be classified into three main categories:

- Solutions that disseminate the whole transaction's read-set to all replicas, called *Non-voting* schemes, allow each replica to certify transactions locally, by sending both the read-set and write-set via an TO primitive. This makes these protocols optimal in terms of communication steps, but also makes them prone to generate very large messages and to overload the Group Communication System.

- *Voting* schemes, which avoid broadcasting the read-set of transactions by sending (via TO) only the write-set, thus drastically reducing the network bandwidth consumption. On the other hand, they incur into the costs of an additional coordination phase along the critical path of the transaction commit , which can hamper significantly performance.

- Approaches relying on the space efficient encoding of Bloom Filters [19] to implement a variant of the non-voting certification mechanism, called *Bloom Filter Certification* (BFC) [35] . Unlike voting mechanisms, BFC determines the outcome of transactions using a single TO, generating smaller messages when compared to non-voting protocols. The probabilistic nature of the Bloom filter

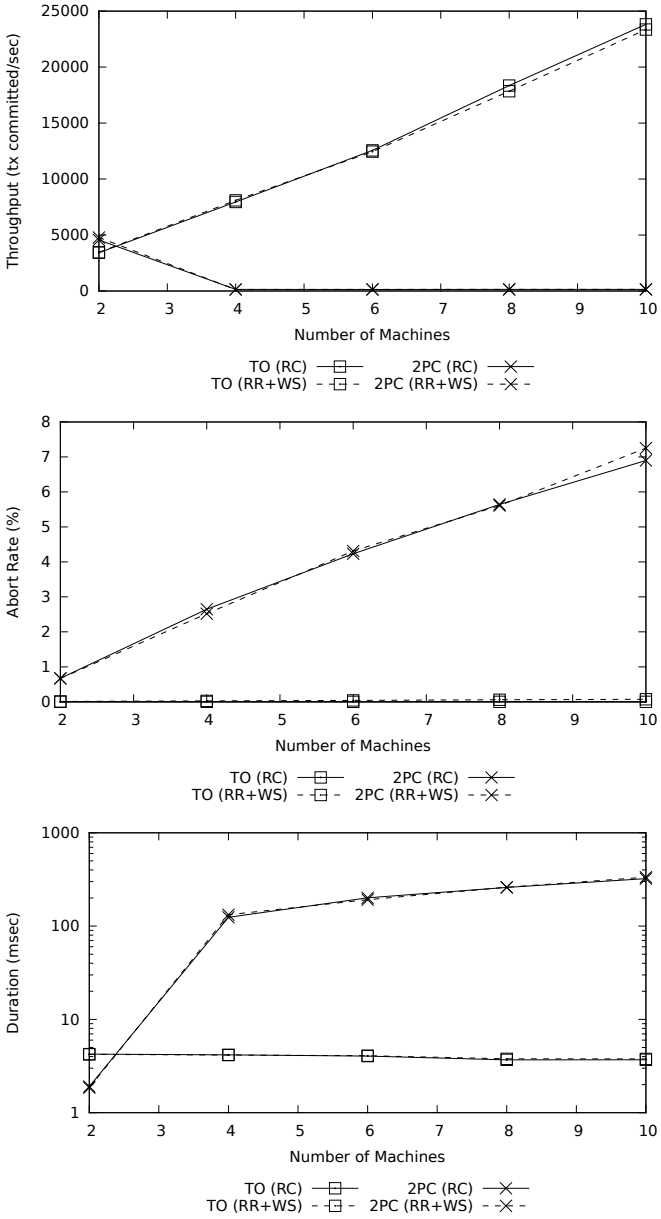


Figure 26: Comparing the performance of a 2PC-based and a TO-based replication scheme. (High contention scenario)

encoding, however, induces false positives in the certification phase, increasing the transaction abort rate.

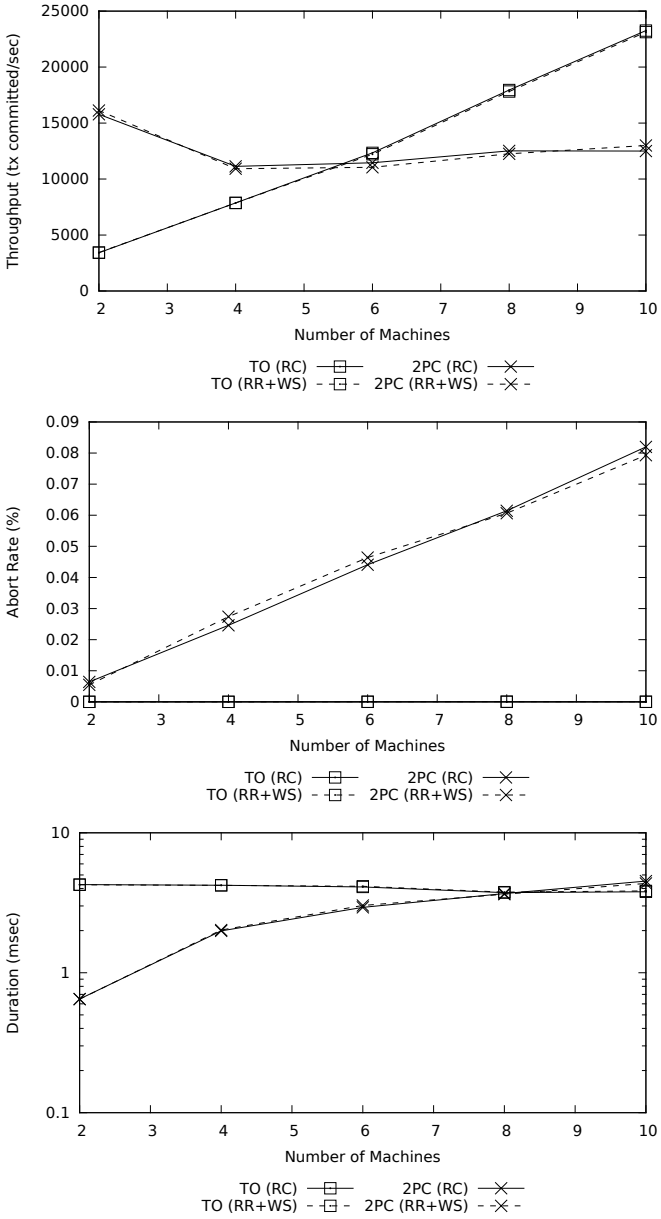


Figure 27: Comparing the performance of a 2PC-based and a TO-based replication scheme. (Low contention scenario)

The above protocols are designed to ensure optimal performance in different workload scenarios and, as we will show in the following, they can exhibit up to 10x differences

in terms of maximum throughput. Our goal is to alleviate the developers/administrators from the hard and time-consuming task of profiling the application and selecting the most suitable replication protocol for each deployment. Furthermore, a static configuration may lead to largely suboptimal configurations in presence of heterogeneous workloads. In these contexts, the employment of a single, statically chosen, replication mechanism, optimized for a specific workload type, will lead to suboptimal performance when processing the transactions that have different characteristics.

One of the key goals of the Cloud-TM project is to develop self-optimizing replication strategies, capable of self-tuning their internal mechanisms in order to meet the actual workload characteristics. A key problem to address to achieve this ambitious goal is to develop performance forecasting techniques capable of predicting which replication to use given the current workload.

In the following we report the main achievements by the Cloud-TM project in this direction, by presenting two machine-learning based techniques aimed to forecast which of the three aforementioned certification protocols to select on a per transaction basis, namely:

- An off-line technique based on regression decision trees [118], that requires a preliminary, computational intensive, feature selection and training phase, but that was shown (see Section 2.2) to achieve high accuracy in forecasting the performance of Total Order algorithms in presence of heterogeneous workloads.

- An on-line reinforcement learning technique, that uses an innovative, parameter-free variant of a very lightweight, but theoretically optimal solution [7] to face the exploration versus exploitation dilemma, i.e. the search for a balance between exploring the environment to find profitable actions while taking the empirically best action as often as possible.

4.3.1 Motivations

As already mentioned, existing certification-based solutions can be classified into three main categories:

- **Non-Voting Certification (NVC):** These solutions [29, 108, 110] disseminate the whole read-set and write-set using the TO service, allowing every replica to determine, upon delivery of the corresponding message, the outcome (commit/abort) of the transaction, by running the certification procedure locally. These schemes are optimal in terms of communication steps, delivering excellent performance when used in workloads characterized by small transaction read-sets. On the other hand, they exhibit very poor performance in presence of transactions reading a significant number of data items. Even worse, in these scenarios, the large network traffic generated by this protocol can saturate and disrupt the proper functioning of the Group Communication Service, leading to network partitions and false failure suspicions.

- **Voting Certification (VC):** These solutions [78] disseminate exclusively via TO the transaction write-set, thus avoiding the issues incurred in by Non-voting

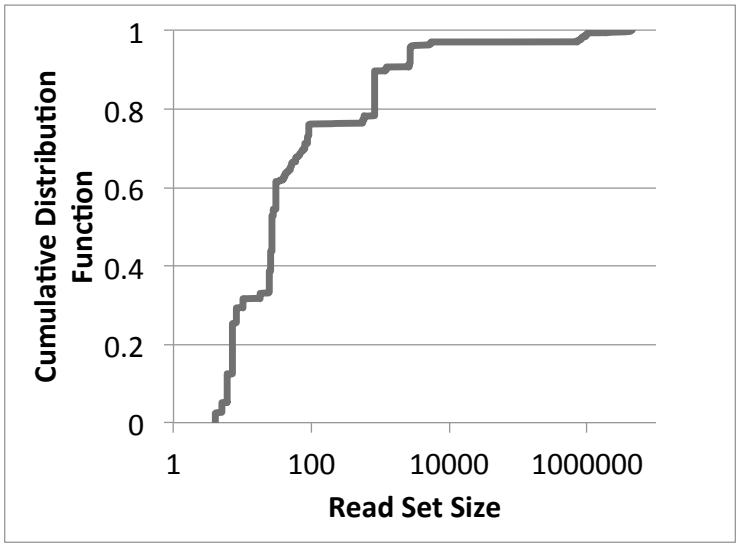


Figure 28: Distribution of transaction read-set size in the STMBench7 Benchmark.

schemes with large transaction read-sets. On the down side, the transaction can only be certified at the site in which it was originated. This implies the need for an additional communication phase, executed using a Uniform Reliable Broadcast (URB) [62] (a lighter communication primitive when compared to TO), which is triggered by the replica where the transaction was originated in order to inform the remaining replicas of the final outcome of the transaction. This extra communication phase, which requires at least two communication steps, has a negative impact on the latency of the commit phase, which represents by far the dominating cost for small transactions. By introducing additional latency in the critical path of the commit phase, which needs to be run sequentially for conflicting transactions, these schemes can adversely affect the maximum throughput achievable by the platform [128].

- **Bloom Filter Certication (BFC):** An alternative approach, denoted as Bloom Filter Certification (BFC) [35], consists in encoding the read-set of the transaction in a Bloom filter [19], a space-efficient data structure that allows compressing the messages disseminated via the TO service, while still allowing every replica in the system to deterministically certify the transactions. Unlike Voting schemes, BFC avoids additional communication steps during the commit phase. In terms of generated network traffic, even though BFC generates larger messages than voting protocols, it typically reduces significantly the size of the messages exchanged via the TO service when compared to non-voting schemes. On the down side, BFC can suffer from false positives due to the probabilistic nature of Bloom filter-based encoding, which ultimately leads to an additional rate of aborted transactions.

From the above discussion, the performance of each of these three alternative certification mechanisms is strongly dependent on the actual distribution of the size of the read-sets generated by the transactional application. Unfortunately, realistic transactional applications can exhibit very heterogeneous workloads encompassing read-sets whose sizes range from less than ten to hundreds of thousands of objects. We have experimentally observed this phenomena, as illustrated in Figure 28, which depicts the distribution of the read-set size for a widely used benchmarking application for in-memory transactional systems (in particular Transactional Memories), namely the STMBench7 [60] benchmark.

In Figure 29 we show the results of a sensitivity analysis aimed at assessing the actual impact of the read-set size distribution on the performance of the three certification schemes described above. The results were obtained using a simple synthetic benchmark adapted from the Bank Benchmark originally used in [35]. This benchmark simulates the concurrent transfer of funds from different bank accounts (modelled as a simple array of doubles), and was altered to vary the number of items read within a transaction in the range [1,100'000]. Further, to focus only on the effects due to variations of the read-set size, which represents the goal of this sensitivity analysis, we configured the benchmark to never generate conflicts among transactions. The only aborts experienced in the system are therefore those determined by false positives with the BFC scheme (which was configured to have an additional abort-rate of 1%, as in [35]). These results were obtained running on a cluster of eight nodes, each one equipped with two Intel Quad-Core XEON at 2.0 GHz, 8 GB of RAM, running Linux 2.6.32-26-server and interconnected via a private Gigabit Ethernet (which represents the reference experimental platform used in the remainder of this study). The in-memory transactional data grid and the certification protocols were implemented in JAVA. The system uses two main components: i) a state of the art Software Transactional Memory (STM), namely JVSTM [24], used to manage local concurrency, and ii) a replicated key/value store, used to maintain associations between unique object identifiers and object instances. Further details on the system architecture will be provided in Section 4.1.1.

Our experimental results highlight that no-one-fits-all solution exists that maximizes the throughput across all the considered workloads. On the contrary, NVC provides the best performance in the scenario with small read-sets, BFC in the scenario with 1000 items in the read-set, and VC is by far the best performing protocol with large read-sets. Further, the relative difference in the performance between the best and worst performing protocol for each scenario ranges from a factor 2.5x (BFC vs VC, read-set size equal to 1000) to 10x (VC vs NVC, read-set size equal to 100'000).

4.3.2 Performance models using off-line trained regressor decision trees.

In order to forecast the time necessary for committing a transaction with each of the three considered certification strategies, we start by forecasting the size, m_p, of the TO message that would be generated by each of the certification protocols $p \in \{NVC, BFC, VC\}$. This corresponds to the number of bytes required to transmit the transaction read-set and write-set with NVC, the transaction write-set with VC, and the write-set and the Bloom filter based encoding of the transaction read-set with BFC.

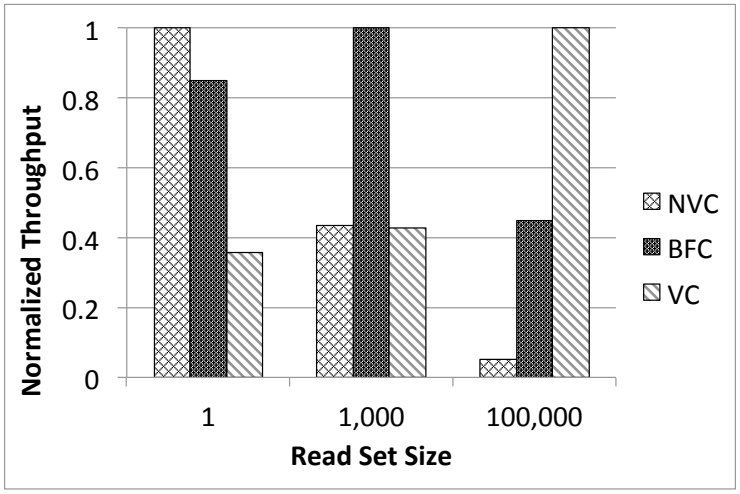


Figure 29: Throughput of three certification strategies with different read-set sizes.

Next, we forecast the time for marshalling and validating a transaction with each of the considered certification schemes. To this end, we maintain, for each certification strategy, a moving average of the average marshalling time per byte, denoted as T_p^m, and of the validation time, denoted as T_p^v, for all $p \in \{NVC, BFC, VC\}$. Further, for BFC, we maintain moving averages of the time required to build a Bloom filter that encodes the read-set (normalized by the read-set's size), denoted as T^{BF}. Finally, for VC, we maintain also the moving averages of the self-delivery latency of the URB convoying the vote from the transaction's initiator, denoted as T^{URB}. Note that the choice of measuring self-delivery latencies allows us to avoid distributed clock-synchronization mechanisms, which in our preliminary experiments revealed not to be sufficiently accurate for our purposes.

Using the metrics above, the commit latency T_p for a transaction using certification protocol p is then forecast as follows:

$$T_{NVC} = T_{NVC}^m \cdot m_{NVC} + T_{NVC}^v + T^{TO}(m_{NVC}) \quad (74)$$

$$T_{BFC} = T_{NVC}^m \cdot m_{BFC} + T_{VC}^v + T^{BFC} \cdot rsSize + T^{TO}(m_{BFC}) \quad (75)$$

$$T_{VC} = T_{VC}^m \cdot m_{VC} + T_{VC}^v + T^{TO}(m_{VC}) + T^{URB} \quad (76)$$

where $T^{TO}(m)$ is the forecast latency for self-delivering a message of size m using the TO primitive. To this end, we exploit our recent results in [37], where we presented and evaluated a series of (off-line) machine learning techniques to forecast TO's performance, including neural-networks [65] and support vector machines [131]. In the light of the results achieved in [37], we integrated in our system a regression technique relying on the Cubist© [115] decision-tree regression algorithm, which proved to be the most accurate and robust predictor among those evaluated.

Analogously to classic decision tree based classifiers, such as C4.5 and ID3 [118],

Metric	Description
freeMem	Free memory in the Java Virtual Machine
tLGC	The time since the last garbage collection
pLGC	% of time since the last GC cycle w.r.t. the time between the last 2 GC cycles
undelivMsgs	#TO Broadcast msgs and not yet self-delivered
bytesUp_x	#Bytes received over a x msec. time window
bytesDown_x	#Bytes sent over a x msec. time window
TOBUp_x	#TOB deliver events over a x msec. time window
TOBDown_x	#TOB broadcast events over a x msec. time window
totCPU_x	% total CPU utilization over a x msec. time window
esCPU_x	% CPU utilization by ES thread over a x msec. time window
TCPqueue	Outgoing messages queued at the Transport Layer

Table 8: List of metrics (features) collected by the Monitoring Layer.

Cubist© builds decision trees choosing the branching attribute such that the resulting split maximizes the normalized information gain (namely the difference in entropy). However, unlike C4.5 and ID3, which contain an element in a finite discrete domain (i.e. the predicted class) as leafs of the decision tree, Cubist© places a multivariate linear model at each leaf, which we use to predict the TO self-delivery latency (expressed in microseconds).

In order to generate the training data for the decision tree regressor we ran the synthetic benchmark described in Section 4.3.1, for each of the three considered certification protocols, varying every 3 minutes the read-set size of the generated transactions in the set {10,100, 1'000, 100'000}. Overall the training data set gathered by each replica is constituted, on average, by approximatively 25'000 samples, reporting the self-delivery latency for each TO message along with the message size, and a total of 53 different metrics (i.e. context information), also referred to as features, including averages on multiple time scales and time series of a plethora of metrics (listed in Table 8) concerning the utilization of various system resources (CPU, RAM and Network). The choice of this synthetic benchmark to generate the training data set has the following rationale: since this benchmark generates transactions with very heterogeneous read-set sizes, it allows gathering a good a-priori knowledge on the performance of a wide range of possible workload scenarios that the system may face when running more complex, realistic applications.

To minimize the effects of overfitting, which are likely to occur given the high dimensionality of the feature space, we run a greedy feature selection algorithm (Forward Selection [63]) aimed at discarding loosely correlated features and boosting the predictor's accuracy. Feature selection is by far the most time consuming phase of the off-line training, taking on average 45 minutes (per replica) when run on a PC equipped with Intel Core 2 CPU with a 2.2GHz clock-rate and 2GB of RAM. On the other hand, feature selection allows to achieve a significant improvement in the accuracy of the predictions, as highlighted by the results shown in Table 9, which report the correlation factor and mean absolute error using 10-fold cross-validation before and after performing feature selection.

Metric	Before FS	After FS
Relative Absolute Error	0.81	0.30
Correlation Coefficient	0.17	0.76

Table 9: Prediction accuracy of the decision tree regressor before and after feature selection.

4.3.3 Performance models using UCB online learners.

Our second performance forecasting technique employs an on-line learning technique. Therefore, it does not require an a-priori computational intensive off-line training. Instead, it relies on a lightweight reinforcement learning (RL) technique that updates the model as the system is running.

This model relies on a customized, self-tuning version of a state of the art RL algorithm, called UCB (Upper Confidence Bounds), which solves (in a theoretically optimal manner) a classical on-line learning problem, known in literature as the *multi-armed bandit* [123]. In this problem, a gambling agent is faced with a bandit (a slot machine) with k arms, each associated with an unknown reward distribution. The gambler iteratively plays one arm per round and observes the associated reward, adapting its strategy in order to maximize the average reward. Formally, each arm i of the bandit, for $0 \leq i \leq k$, is associated with a sequence of random variables $X_{i,n}$ representing the reward of the arm i, where n is the number of times the lever has been used. The goal of the agent is to learn which arm i maximizes the criterion:

$$\mu_i = \sum_{n=1}^{\infty} \frac{1}{n} X_{i,n}$$

that is, achieves maximum average reward. To this purpose, the learning algorithm needs to try different arms in order to estimate their average reward. On the other hand, each suboptimal choice of an arm i costs, on average, $\mu^* - \mu_i$, where μ^* is the average obtained by the optimal lever. Several algorithms have been studied that minimize the *regret*, defined as

$$\mu^* n - \mu_i \sum_{i=1}^{K} E[T_i(n)]$$

where $T_i(n)$ is the number of times arm i has been chosen. Building on the idea of *confidence bounds*, the UCB algorithm creates an overestimation of the reward of each possible decision, and lowers it as more samples are drawn. Implementing the principle of *optimism in the face of uncertainty*, the algorithm picks the option with the highest current bound. Interestingly, this allows UCB to achieve a logarithmic bound on the regret value not only asymptotically, but also for any finite sequence of trials [7].

More in detail, UCB assumes that rewards are distributed in the [0,1] interval, and associates each arm with a value:

$$\overline{\mu_i} = \overline{x_i} + \sqrt{2 \frac{\log n}{n_i}} \tag{77}$$

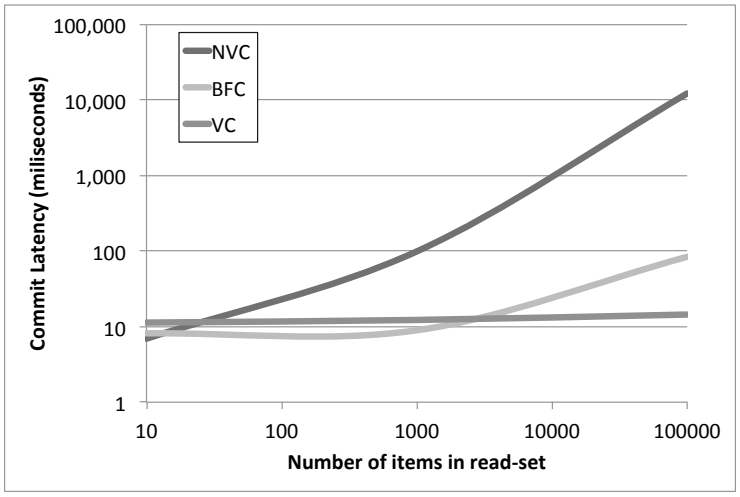


Figure 30: Commit latency as a function of the read-set size.

where $\overline{x_i}$ is the current estimated reward for arm i, n is the number of the current trial, n_i is the number of times the level i has been tried. The right-hand part of the sum in Eq. 77 is an *upper confidence bound* that decreases as more information on each option is acquired. By choosing, at any time, the option with maximum $\overline{\mu_i}$, the algorithm searches for the option with the highest reward, while minimizing the *regret* along the way.

In order to apply the UCB technique to our problem, we had two face two main issues, which we discuss in the following paragraphs.

State space discretization. As we have illustrated in Section 4.3.1, the performance of certification depends on the workload characterization. Thus, using a single UCB instance, having as arms the three alternative protocols for all possible scenarios is clearly not a viable solution. This observation raises the problem of discretizing the workload state space into a set of distinct, representative, classes of workload scenarios. This allows, in fact, to associate a different instance of a UCB learner with each discretized interval of the workload's parameter space, and to train each instance to choose among the 3 considered protocols under specific workload conditions.

The discretization process involves a delicate trade-off: a finer (i.e. denser) discretization can lead, eventually, to more accurate predictions across the entire state space, but requires the training of a larger number of UCB instances, which can lead to a considerable increase of the learning time. In order to determine an appropriate discretization strategy, we analysed the average commit latency of each of the three protocols as a function of the read-set size using the synthetic benchmark introduced in Section 4.3.1. The results, reported in the log-log plot of Figure 30, highlight that, for NVC and BFC, the read-set size and commit latency exhibit a heavy-tail relationship. At the light of this observation, we opted to use the read-set size as the reference variable to discriminate different workload situations,and we discretized it using ex-

ponentially increasing intervals, where each sampling interval is defined by the range $[10^i, 10^i + 1]$ with $i \in \{1 \ldots 6\}$). This choice allowed us to partition the state space into a small number of intervals, thus reducing learning time, while associating each discretized interval with fluctuations of approximately the same relative amplitude in the commit latency, even for the case of the NVC, whose commit latency is the most sensible to variations of the read-set size.

Definition of the reward function. UCB is based on the assumption that rewards are distributed in the [0,1] interval, whereas, as we have seen in Figure 30, the commit latencies are distributed over a very large domain. This required defining a mapping function, denoted as $R(t)$, taking as input a commit latency, t, and outputting a value (the reward) distributed in the [0,1] interval. In order to preserve the relative distance among samples before and after applying the mapping function we employed the following linear transformation:

$$R(t) = \frac{maxLatency - min\{maxLatency, t\}}{maxLatency}$$

which relies on the parameter $maxLatency$, defining a threshold for the commit latency, above which the reward is mapped to the value 0. Based on our preliminary experiments, we observed that the correct definition of the $maxLatency$ parameter value has a fundamental impact on the effectiveness of UCB: excessively low or high values would in fact lead to saturating the reward function, preventing UCB to distinguish sensibly the performance of the various protocols. Also, the manual tuning of this parameter is an extremely time-consuming task, given that the setting of $maxLatency$ was found to depend strongly on the characteristics of the user level application. For instance, we noted that, when testing this approach with the STMBench7 benchmark, we had to increase the value of $maxLatency$ by a factor approximately 27x larger than when using the synthetic benchmark described in Section 4.3.1.

This led us to define a self-tuning mechanism to define the value of the $maxLatency$ parameter. This mechanism is based on the observation that the (average) commit latency when using VC is i) largely unaffected by the read-set size (given that it does not disseminate the read-set), and ii) lower than that of *both* NVC and BFC for sizes of the read-set larger than some threshold (this threshold being unknown and dependant on the application and deployment scenario). In other words, VC's commit latency represents a consistent upper bound for NVC's and BFC's commit latencies below a given read-set's size threshold, in which the two protocols typically exhibit alternate performances. On the other hand, it represents a lower bound for NVC's and BFC's commit latency for high read-set's size, a scenario in which it is actually unnecessary to be able to accurately predict their performance, given that VC outperforms them significantly.

This makes the VC's average commit latency, denoted as T_{VC}, a good reference point for UCB's $maxLatency$ parameter value. This insight led us to define the following rule:

$$maxLatency = \overline{T_{VC}} \cdot (1 + \sigma(T_{VC}))$$

where $\sigma(T_{VC})$ denotes the standard deviation (more precisely the squared root of the sampling variance) of T_{VC}. In order to instantiate this formula, upon boot-strapping of

the system, we execute transactions using the VC scheme until the following stopping condition is reached:

$$\sigma(T_{VC}) < 2 \cdot \overline{T_{VC}}$$

which in our experiments typically implied a few tens of transactions (and that was however upper bounded to 100 transactions to ensure robustness in the presence of highly disperse sampling data). To minimize the impact of this (typically quite short) bootstrapping phase on the learning time, we provide the observed sampling data to the corresponding UCB's instances also during this phase (in which UCB's instances are not being queried to choose the replication protocol), thus allowing them to gather statistical information concerning the reward of the arm associated with the VC protocol.

A further optimization that we designed in order to minimize learning time is to have the replicas periodically exchange and merge the locally gathered statistical information concerning the reward distributions of UCB's arms. This allows the replicas to mutually benefit from the statistical knowledge that they have gathered so far, narrowing the upper confidence bounds of the UCB's instances and accelerating their convergence. To minimize the overhead, we piggyback periodically (e.g. each 10 seconds in our experiments) the state of the 6 UCB's instances maintained at each replica (encoded by the tuple $< \overline{x_i}, n_i, n >$ for each of its three arms $i \in \{NVC, BFC, VC\}$, and globally accounting to around 100 bytes) to the TO messages generated by the PRM. As soon as updated statistical information from a different replica is received, the information concerning the local UCB instances is updated by setting, for each arm i:

- the value of $\overline{x_i}$ to the average of the local and remote values of $\overline{x_i}$, weighted proportionally to the number of times i was played locally and remotely, namely:

  $$\overline{x_i} = w_i^{loc}\overline{x_i} + w_i^{rem}\overline{x_i^{rem}}, \text{ where } w_i^{loc} = \frac{n_i}{n_i + n_i^{rem}} \text{ and } w_i^{rem} = 1 - w_i^{loc}$$

- the value of n_i and n to the sum of their, respectively, local and remote values, namely $n_i = n_i + n_i^{rem}$ and $n = n + n^{rem}$

4.3.4 Experimental Evaluation

In this section we report the results of an experimental study aimed at validating the proposed machine-learning based self-optimizing mechanisms.

We start by considering the synthetic benchmark already used in Section 4.3.1 that, thanks to its simplicity and predictability, allows to access the accuracy of the proposed performance models in precisely identifiable workload scenarios. All the throughput results reported in the following were obtained averaging over a number of runs sufficient to ensure that the width of the 90% confidence intervals for the throughput was less than 10% of the corresponding average value.

The bar plot in Figure 31 reports the normalized throughput (with respect to the optimal non-adaptive workload) for each of the workloads generated by the Bank benchmark, contrasting the performance of static certification protocols, with the one

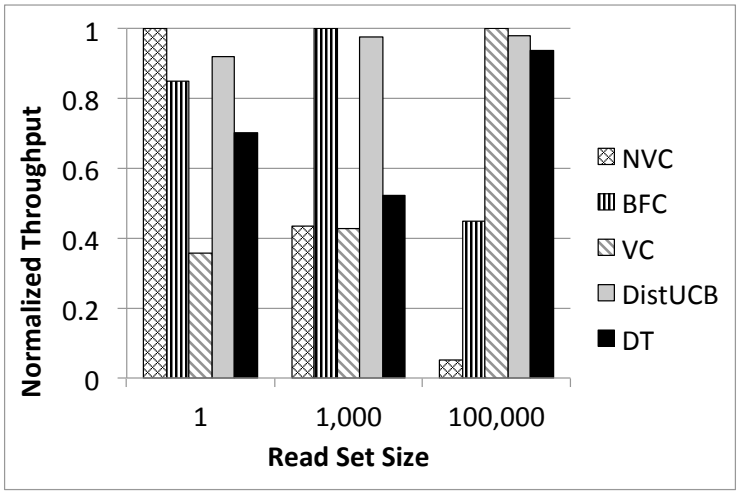


Figure 31: Normalized throughput of the adaptive and non-adaptive protocols (Bank benchmark).

achieved by following the indications of the performance models using regressor decision trees and UCB (respectively dubbed as DT and DistUCB in the plot). These experiments were run by switching the workload every three minutes, thus the reported performance incorporates also data gathered during the initial phases during which we bootstrap the statistical information of UCB.

Our experimental data shows that the on-line learning model using UCB (with the optimization for periodically exchanging statistical information among replicas enabled) achieves a performance very close to the corresponding optimal protocol for each scenario, namely on average around 5% less than the optimal solution and in the worst case, the scenario where the transaction's read-set size is set equal to 1, less than 10% from the optimum. In this scenario, UCB alternates between BFC and NVC, whose performances are quite close (differing by around 15%); in several runs some replicas eventually converged towards the choice of BFC. In all the remaining scenarios, after a short bootstrapping phase, the replicas converged consistently towards the choice of the optimal certification protocols, which explains why they achieved performance almost indistinguishable from those of an optimally tuned non-adaptive protocol.

On the other hand, the performance achieved by the performance model based on regressor decision trees was significantly worse. When using DT, the performance resulted approximately 25% worse than that of the corresponding optimal non-adaptive scheme (across the three workloads). Note that, DT was still able to outperform the second best non-adaptive protocol (but not the optimal choice). A main source of inefficiency in the implementation of the DT performance model is the following: it relies on the Java Native Interface (JNI) to query the decision tree-based model generated by Cubist, implemented in C. The overheads due to JNI are negligible in the scenario with read-set size equal to 100K, whose transactions have a local execution time in the order

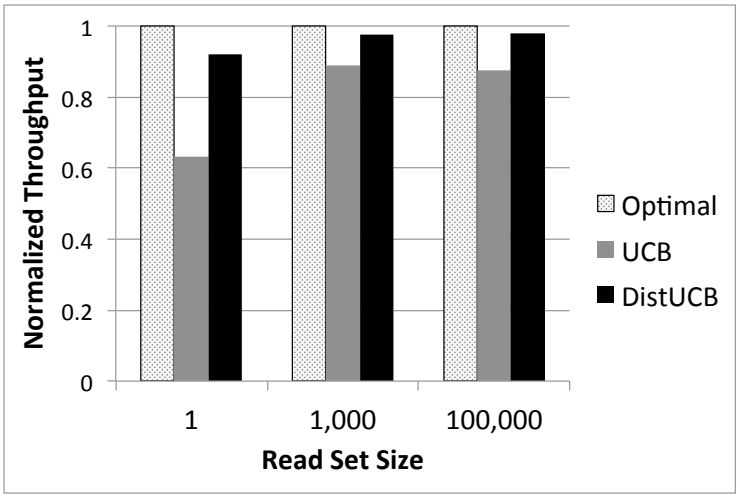


Figure 32: Normalized throughput of UCB and DistUCB over a three minute run (Bank benchmark).

of a few tens of milliseconds. On the other hand, JNI's overheads have a negative impact on performance in the scenarios with smaller read-set sizes, in which transactions have a local execution time on the order of just a few tens of microseconds. The performance of DT is lower in the scenario with a read-set size of 1000, as in this case the DT performance model had a lower accuracy in forecasting the TO self-delivery time, and erroneously biased its decisions towards the voting protocol (which is chosen in approximately 30% of the cases on average).

In Figure 32 we contrast the performance of the UCB-based performance model (again in terms of normalized throughput vs the optimal non-adaptive protocol), over a three minute run, with and without enabling the optimization of exchanging periodically (each 10 seconds) statistical information among replicas to improve learning. The data clearly shows the effectiveness of this optimization, with speed-ups larger than 25% due to the fastest convergence towards the optimal non-adaptive solution. Figure 33 provides more detailed insights on the speed of convergence of UCB and DistUCB versus the optimal solution, reporting the average throughput over 10 seconds time windows, achieved by the two protocols. The plots clearly highlight the positive effects, in terms of learning time reduction, due to the exchange of statistical information occurring, in particular, at the time instants 10, 20 and 30 (seconds), that nearly halves the time required to converge to the optimal choice.

4.3.5 Related Work

Our work is clearly related to the vast literature on replication of transactional systems, and in particular to the more recent works relying on TO to achieve a replica-wide agreement on the transaction serialization order [77, 78, 108, 110]. None of them however considered the problem, tackled in our work, of autonomically determining, on a

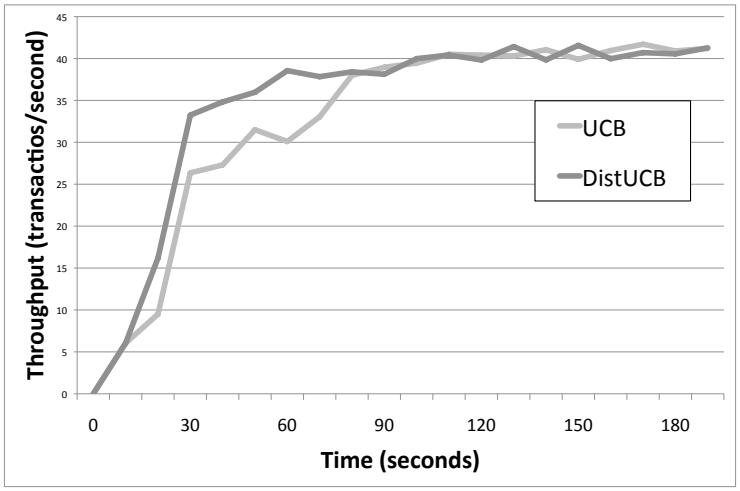


Figure 33: Evolution of throughput over time with UCB and DistUCB (Bank benchmark - 100K read-set size scenario).

per-transaction basis, the most adequate replication protocol to employ using machine-learning techniques.

Machine learning techniques have already been used to predict the performance of computer systems in several contexts. These include works aiming at forecasting the throughput of TCP flows [96] and Pub-Sub systems [56], solutions aimed at automatically classifying traffic based on semi-supervised learning techniques [49], at automatizing the allocation of resources in cloud-computing infrastructures [149], or generating software aging models to be used in the context of rejuvenation frameworks [4]. Also, as noted in the text, the regressor decision tree performance model exploits our previous results in the area of machine-learning performance prediction of TO protocols, presented in Section 2.2 and recently published in [37].

Our work is clearly related to the body of research on autonomic computing, and in particular to the field of self-optimizing databases. In this context, several approaches have been proposed based on the idea to automatically analyse the incoming workload, e.g. [92], to automatically identify the optimal database physical design or self-tune some of the DBMS inner management schemes, e.g. [22]. However, none of these approaches investigated the issues related to autonomically adapt the replication scheme. We argue that this is mainly due to the fact that current DBMSs, because of the high complexity of their architecture, lack the flexibility required to dynamically adapt such low level mechanisms.

5 Data Access Pattern Characterization

Large-scale applications are usually characterized by having both elaborate business logic and large volumes of data to be processed. For such systems to perform well, they must exhibit good data locality, but achieving this locality requires very good knowledge about how the system operates and performs its functionalities. A programmer equipped with this knowledge about the application may explore the proper distribution, sharing, and loading/unloading of data, leading to an optimized system operation with minimal levels of overhead. Unfortunately, due to the size, complexity, and constant evolution of large-scale applications, it is practically impossible for programmers to identify manually the necessary course of action to achieve good data locality. Thus, we claim that the solution to this problem is to have an automated approach that is capable of monitoring, modelling, and predicting the data access patterns of the system and then, based on these predictions, taking the most adequate measures for optimizing the system's performance.

For the purpose of performing the analysis and prediction of the data access patterns of an object-oriented application, we developed three alternative stochastic model implementations. The models are based on Bayesian Inference, Importance Analysis, and Markov Chains, and they model the access patterns at the class and class field's level only.

In the following section we briefly describe each of the three models used to characterize the data access patterns of an object-oriented application. Then, in Section 5.2 we evaluate the accuracy of each of the three models on a given benchmark. This evaluation of the accuracy is complemented in Section 5.3 with an evaluation of the performance overhead introduced by the injected code to monitor the data access patterns of an application, which is needed to build each prediction model. In Section 5.4 we briefly discuss some of the related work. Finally, in Section 5.5 we describe how this work has been integrated into the Cloud-TM platform.

5.1 Overview of the modelling techniques

5.1.1 Bayesian Inference

Bayesian analysis techniques are used for parameter estimation. They give an estimate of the statistical uncertainty of the estimated parameters (corresponding, in our work, to the likelihood of reading/writing a given field of an application class) and can update them when new information becomes available.

If observations of one (or more) of the stochastic variables $\mathbf{X}$ are available, the probability density function can be updated. Consider a stochastic variable X with density function $f_X(x)$. If $\mathbf{q}$ denotes a vector of parameters defining the distribution for X, the density function of the stochastic variable X can be written as $f_X(x, \mathbf{q})$.

It is assumed that n observations (realizations) of the stochastic variable X are available making up a sample $\widehat{\mathbf{x}} = \left(\widehat{x}_1, \widehat{x}_2, \ldots, \widehat{x}_n\right)$. The realizations are assumed to be independent. The updated density function $f''_{\mathbf{Q}}\left(\mathbf{q}|\widehat{x}\right)$ of the uncertain parameters $\mathbf{Q}$, given the realizations, is denoted the posterior density function and is given by (see

textbook on Bayesian statistics, for example [20] or [88]):

$$f''_{\mathbf{Q}}\left(\mathbf{q}|\widehat{x}\right) = \frac{f_X\left(\widehat{x}|\mathbf{q}\right) f_Q\left(\mathbf{q}\right)}{\int f_X\left(\widehat{x}|\mathbf{q}\right) f_Q\left(\mathbf{q}\right) d\mathbf{q}} \tag{78}$$

where $f_X\left(\widehat{x}|\mathbf{q}\right) = \prod_{i=1}^{n} f_X\left(\widehat{x}_i|\mathbf{q}\right)$ is the probability density at the given observations assuming that the distribution parameters are $\mathbf{q}$. The integration in equation 78 is over all possible values of $\mathbf{q}$.

To implement the Bayesian Inference Model, two sets of statistical data are used to generate the predictions. The first set is called prior set and it contains data about access patterns observed in the past, up to a given point in time. This time reference corresponds to the moment when the model prediction was updated last. The second set is called current set, and it contains data from the point in time when the prior set ends, to the moment when the new updated prediction is to be generated.

The model prediction updating is performed at regular time intervals. The exact duration of these intervals is application dependent and should take into consideration several factors. The time intervals should be long enough to allow the accumulation of a representative volume of behavioural data. In other words, sufficient time should be given to the target application to perform an adequate amount of operations to correspond to a sample of typical system operation. Depending on the target application, the duration of the time interval could vary from several minutes to several hours, based on the amount of work being performed by the target system. If the time periods are too short, the model prediction updating in the end of these (intervals) may lead to two potential problems. The first of these problems is that if the current data set collected during this time period is not representative of the typical application behaviour, when the current set is used to update the model prediction, it could make the model conclude something that is not true about the target system behaviour. The second problem relates to the fact that very frequent recalculations of the model predictions may add up to a performance overhead that could be otherwise avoided

When the moment for updating the model prediction comes, the probability density functions of the prior and current data sets are determined. These functions describe the behaviour of the target application, in terms of data accesses performed for the periods of time to which the sets belong. Using the current probability density function, the prior function is updated, obtaining thus the so-called posterior probability density function. The posterior function corresponds to the prediction generated by the model, and describes what is the most likely future behaviour of the target application, in terms of the data it manipulates. Based on the posterior function, the actual access probabilities for all of the application domain class fields are calculated.

5.1.2 Importance Analysis Techniques

An Importance Analysis is a procedure for analysing the potential failure modes within a system by classifying them based on the severity or the effect of failures on the system. It is widely used in many industries during various phases of the system life cycle

(e.g., see [76, 86, 103]). A failure mode can be defined as the manner by which a failure is observed and it generally describes how the failure occurs. Some of the tools used in the design stage for identifying failures and determining their consequences are Risk Priority Numbers (RPN), Occurrence/Severity Matrix (OSM), Risk Ranking Tables (RRT), and Criticality Analysis (CA). For the current work, these methods are adapted to indicate which groups of fields are more critical or important for the operation of the considered target application.

The Risk Priority Number (RPN) system is a relative rating system that assigns a numerical value to the issue in each of three different categories: Severity (S), Occurrence (O), and Detection (D). The three ratings are multiplied together to determine the overall RPN for the issue. The criteria used in each rating scale are determined based on the particular circumstances for the item that is being analyzed. Because all issues are rated according to the same set of rating scales, this number can be used to compare and rank issues within the analysis. However, because the ratings are assigned with regards to a particular analysis, it is generally not appropriate to compare RPN numbers among different analyses.

Failure mode, effect analysis, and criticality analysis techniques are used throughout industry for a variety of applications and consist in a flexible analysis method that can be performed at various stages in the system life cycle. These analyses can be employed to support design, development, manufacturing, service, and other activities to improve reliability and increase efficiency.

The implementation of the Importance Analysis model considers the gathered access pattern statistical data and calculates the *local* and *global* access probabilities for all domain class fields. The *local* access probability corresponds to the likelihood of accessing a certain piece of information within the scope of a given *context*, which has been identified to exist during the execution of the target system. The *global* access probability is defined by the likelihood of accessing a given datum at the level of the whole application. Once the *local* and *global* access probabilities are calculated, they are submitted to a normalization process that generates an access probability classification, within which all access probabilities are placed. If a given access probability has the value of p, then its associated access probability class i is calculated according to the following formulae:

$$i = floor\left[\left(p + \left(p_{\max} - p_{\min}\right)\right) / \left(p_{\max} - p_{\min}\right)\right] \tag{79}$$

where $p \in [0, 1]$ and $p_{\min}$and $p_{\max}$ correspond to the minimum and maximum access probabilities observed for that classification type (*local* or *global*). The local and global access probability class indices are two of the three input arguments used to generate the final result of the Importance Analysis model.

The third and last input is called the *impact factor*. The *impact factor* corresponds to an expert judgement coefficient whose value should be provided by someone who has solid knowledge of how the application operates (e.g. developers). It is a subjective indication of how important a given piece of domain data is for the execution of the target application, from an expert point of view. If no impact factor is provided for a given application class field, the implementation of the model provides a default neutral value to be used instead.

The RPN value (for a given domain class field) can be calculated by multiplying the local access probability class, the global access probability class and the *impact factor*. The RPN values give an indication of how important are their respective fields to the execution of the target application.

5.1.3 Markov Chains

The Markov Chain [94] model analysis procedure is composed of several phases. During the first phase, it builds a transition matrix. This matrix contains the probabilities of navigating from one system state to another (automata theory). As one of the main goals of this work is to allow the identification and prediction of access patterns performed by object-oriented applications, the states correspond to the manipulation of a given domain class field. In the transition matrix $\mathbf{T} = [t_{ij}]$, the cell t_{ij} contains the probability of manipulating field i immediately after having manipulated field j.

To calculate these probabilities, all *contexts* (that have been identified so far) keep a hash table containing the statistical data concerning the sequences of observed field accesses. The keys of the hash table correspond to *PFields*, whereas the associated values are sets of *PFields*. The *PField* sets contain access information for the situations when they have been accessed immediately after the hash table key (*PField*) to which they are associated.

All the required input for the analysis is collected during a training period. The training is to be performed only once, as long as it is representative of the application service life. During this period, every time there is a field access, the surrounding context is determined. Subsequently, the last field that has been accessed in that *context* is identified and used to index the *context*'s hash table containing statistical data about what access sequences have been seen. After this, the data is updated to reflect the current field access and, finally, the last accessed field is updated to the current one. This allows the gathering of data about which fields are accessed after a given one and the number of times this has been observed.

When the accumulated statistical data is representative, the transition matrix can be built. First, a square matrix $\mathbf{T}$ of size $n \times n$ is created, where n is the number of fields in all of the application's domain classes. This is used as a base for the calculations, resulting in the transition matrix, and, subsequently, in the stationary vector.

For each column $j \in [1, n]$ of the matrix $\mathbf{T}$, each of its $i \in [1, n]$ cells contains the number of times that the field i has been accessed immediately after j. After computing these values, they are normalized: For a column j, the value of each of its cells is divided by $\sum t_{ij},\ i \in [1, n]$, obtaining the normalized transition matrix, $\overline{\mathbf{T}}$.

This matrix does not present all the necessary properties to be suitable for the *Power method* [2]. This method consists in calculating the powers of the transition matrix to obtain the stationary distribution.

So, before applying the *Power method*, we need to use the *Random Surfer Model* [21] to compute a new suitable matrix. This is accomplished by identifying the columns whose elements are all zero and, for any such column j, setting its elements to $t_{ij} = 1/n$.

Following this, a perturbation matrix $\mathbf{E}$ with size $n \times n$ is constructed. All of its cells have the value of $1/n$. Once this is done, the $\overline{\overline{\mathbf{T}}}$ matrix is defined as $\overline{\overline{\mathbf{T}}} = \alpha\overline{\mathbf{T}} + (1-\alpha)\mathbf{E}$,

where $0 \leq \alpha \leq 1$.

Finally, given the $\overline{\overline{\mathbf{T}}}$ matrix, we may now apply the *Power method*, thereby calculating the stationary distribution matrix. Any of the columns of the distribution matrix yields the stationary vector, whose elements represent the global probability of accessing or manipulating a given domain class field of the application.

5.2 Accuracy of the prediction models

To evaluate the effectiveness of each method for predicting data access patterns in object-oriented applications, two different benchmarks are employed. The first benchmark is the TPC-W, which was originally presented by Smith [133] and specifies an e-commerce workload that simulates the activities of a retail store website. Emulated users can browse and order products from the website.

The second of the benchmarks is the oo7, firstly presented by Carey [26]. It is often used to assess the performance of object-oriented persistence mechanisms. It strives to present a broad set of operations, allowing for the building of a comprehensive performance profile. The oo7 was designed to boast properties common to different CAD/CAM/CASE applications, although in its details it does not model any specific application. The evaluation is performed through the execution of a series of traversals, updates, and query operations over the underlying object model.

Due to the fact that the results from the two benchmarks are very similar, both in terms of the precision of the predictions generated by the models as well as with regards to the performance overhead due to the gathering of statistical information, in the following we present and discuss only the results obtained from the oo7 benchmark.

There are some random behavioural elements in the oo7 benchmark, but these are not sufficient to demonstrate the validity of a system that aims to predict the behaviour of a target application. So, we resorted to Monte Carlo simulations [14] to make the oo7 benchmark behave like a stochastic process. A stochastic process is one whose behaviour is non-deterministic in that a system's subsequent state is determined both by the process's predictable actions and by a random element. This is done by modelling the number of invocations of the methods that compose the benchmark. A triangular distribution is used to perform the modelling. This type of distribution is chosen because it is the one most commonly employed when dealing with a system about which there is little or none information regarding its behaviour. Three different triangular distributions are generated to model the benchmark invocations, namely with left, middle, and right mode locations.

5.2.1 Bayesian Inference

The Bayesian inference technique uses two sources of data to make a prediction. One of them is the *previous*, and the second one is the *current*. The current data is used to update the previous collected information, by making it reflect the patterns that have been most recently observed.

Based on the Bayesian method, both the physical uncertainty related to the considered variable as well as the statistical uncertainty related to the model parameters can

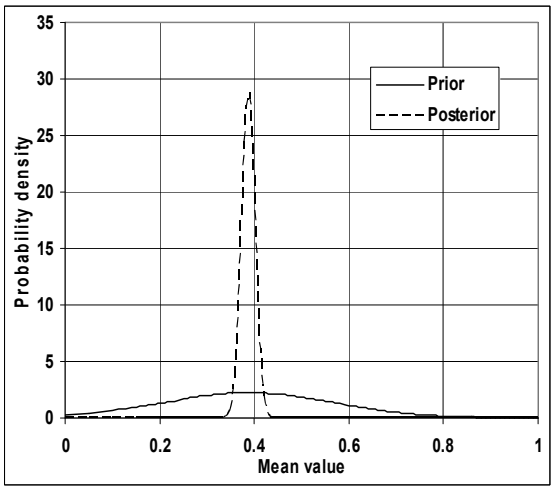


Figure 34: Uncertainty reduction after applying Bayesian inference

be quantified. An important property of the Bayesian analysis is that the uncertainty of the prediction is reduced. This may be seen in Figure 34.

In Figure 35 we show the results obtained with the Bayesian inference approach. The x-axis of the histogram represents the index of the probability classes to which a field belongs. The index i is defined by $i = floor\,[(p + 0.1)/0.1]$ where $p \in [0, 1]$. The bars in the previous and current histograms indicate the number of fields whose observed probability of being read/written belongs to a given class, whereas the predicted histogram reflects the fluctuation in the relative importance of each of the classes, which is expected to be seen in the following executions of the application (see textbook on Bayesian statistics, e.g. [20] and [88]).

As these results show, the great majority of the application's fields belong to low probability classes, while there are only a few, which present the maximum probability of being accessed. Such a field distribution simplifies the process of making optimizations because most of the fields are not likely to be accessed and, as a result of that, the set of fields and associated classes that need to be dealt with is compact and easier to handle.

To evaluate the precision of the results generated by the system, we compared the predictions generated during a given run of the benchmark with the actual patterns observed during its following run. To check whether the mean of the prediction is significantly different from the one observed in the next execution, we used a null hypothesis test.

The null hypothesis states that the mean value of the prediction is not significantly different from the mean value of the patterns that were actually observed during the subsequent execution. The Z test statistic for this type of check can be found in [20] as:

$$Z = \frac{\mu_1 - \mu_2}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}} \tag{80}$$

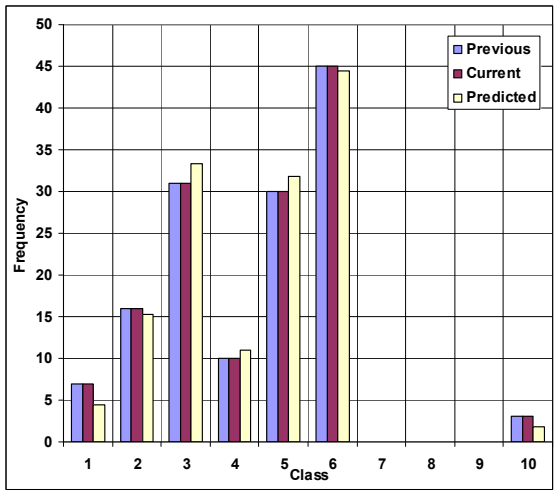


Figure 35: Bayesian inference, left mode location

where μ_1 and μ_2 are the mean values, σ_1^2 and σ_2^2 are the variances and n_1 and n_2 are the sample sizes, for the predicted and the observed, respectively. This is also known as the decision rule.

The critical value for Z is defined as ± 1.96, which corresponds to 0.05 level of significance to reject the null hypothesis, and equals to 95% of the area below the curve of the normal probability density function. We chose the value of 0.05 for the level of significance because it corresponds to the most common value used in engineering practice for such tests.

For the three different input modes tested here, the calculated Z values ($Z_l = -0.33$, $Z_m = 0.34$, and $Z_r = 1.49$) belong to the area of 95% level of confidence. This means that at the 0.05 level of significance, there is no significant difference between the mean values of the predicted and subsequently observed (patterns). Consequently, we may conclude that the predictions created by the system are precise: They are able to indicate correctly the tendencies that are actually observed in the next executions of the application.

5.2.2 Importance Analysis

In this section we present the results obtained with the Importance Analysis model on the oo7 benchmark. But, before the presenting the results, we need to introduce the notion of the criticality rank table.

As has been explained in the theoretical section of the model based on Importance Analysis, tables such as the one presented in Figure 36 are used as guidelines for the decision making process of determining whether a given item should be considered as important or critical, with regards to the type of analysis being performed. In the context of the current work, such an item corresponds to the field of a given application class, whereas its importance or criticality is associated with the probability of that field

L/G	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

Figure 36: Criticality rank table

being accessed by the application in a given context. Thus, if, by using the RPN value of a given field, the position obtained belongs to the dark grey area, then the field is considered highly important for the operation of the application. If, on the other hand, the position of a field is within the non-shaded area, then it is very unlikely that it will be needed by the application. For the remaining fields in the intermediate area, their importance is deemed as average.

Passing on to the actual results, these may be observed in the histograms presented in Figure 37, where the left mode location is shown. The z-axis of this histogram corresponds to the number of fields whose RPN values equal the number obtained by multiplying their associated x and y values. The x-axis indicates the local probability access class to which a given field belongs, whereas the y-axis is relative to the global probability access class of the fields.

According to these results, most of the application fields belong to the lowest probability classes, while, at the same time, very few of them belong to the higher probability classes. This trend is similar to the one observed in the Bayesian Inference model results, and the benefits ensuing from it are the same, namely a smaller data set of likely to be needed data is easier to manage when trying to perform optimizations.

5.2.3 Markov Chains

In the following, we present the results obtained with the third model used in this work—the Markov Chain model. The two main outputs of this analysis technique are an access probability transition matrix and an access probability stationary vector. The matrix contains the observed probabilities of accessing a given field immediately after having accessed another field. The stationary vector presents the global access probabilities, which, in the work presented here, is calculated based on the *Power Method*.

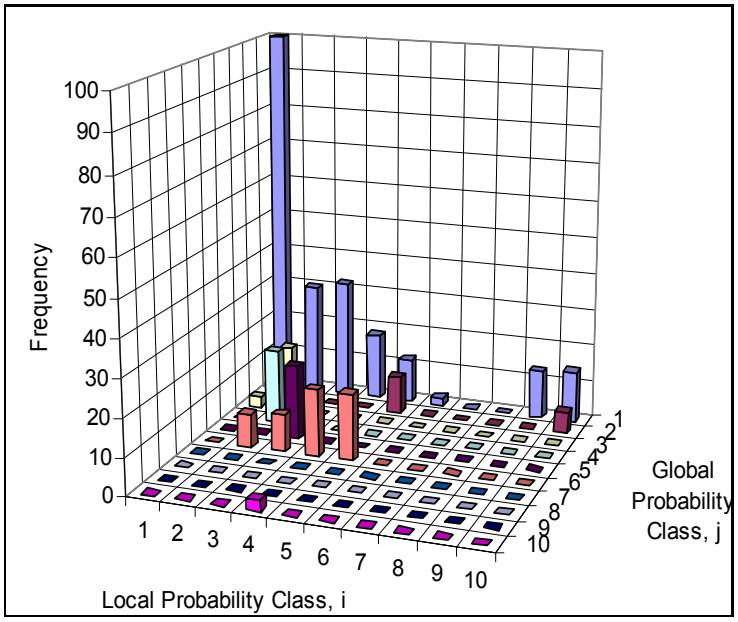


Figure 37: Results for the Criticality model in the oo7 benchmark using the left mode location

The results achieved from executing the left mode location of the oo7 benchmark may be observed in Figure 38 and Figure 39.

Figure 38 presents the access probability matrix for the left mode location. The value present in a cell with coordinates (i, j) corresponds to the probability of accessing field i immediately after field j has been accessed (the diagonal formed from the lower left corner towards the upper right corner results from consecutive accesses to the same field, which correspond, most likely, to iterative access patterns).

Given that the matrix generated has a size of 90, corresponding to the number of fields present in the classes of the benchmark, it is not possible to present the numerical values of the cells of the matrixes. Thus, the format employed only shows precisely where the probability is greater than zero and gives an indication of how high that probability is based on the degree of shading applied to the cell under consideration.

Figure 39 shows the access probability stationary vector for the left mode location. On the x-axis of the histogram we have the field ids, whereas the y-axis indicates the global probability of each field being accessed by the application. The y-axis is normalized, and, as such, all values belong to the interval $[0, 1]$. It is important to note that the minimum of the histograms, along the y-axis, is adjusted so that the bars of any field whose probability of being accessed equals 0.00344 are not displayed. This is done with the aim of presenting in a clearer way the fields that have been effectively accessed during the execution of the benchmark: As a result of the application of the *Random Surfer Model*, any field that has never been accessed ends up with a minimal residual probability of being accessed, which, in the current situation, equals 0.00344.

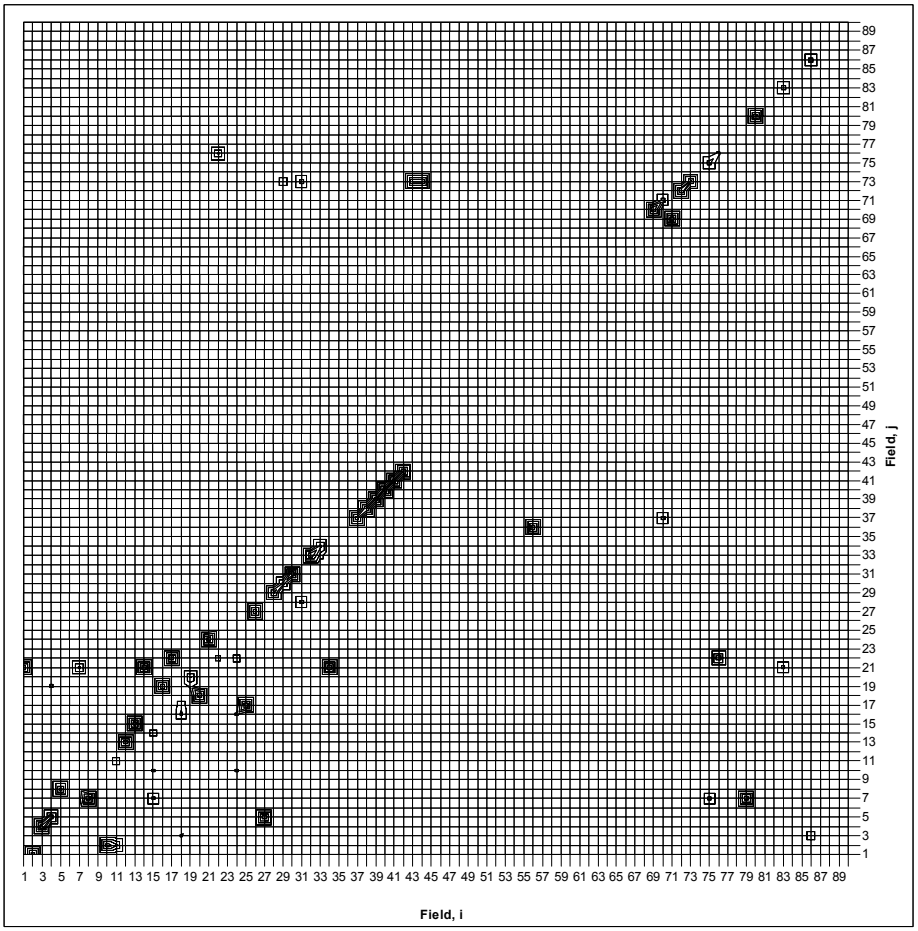


Figure 38: Access probability transition matrix, left mode

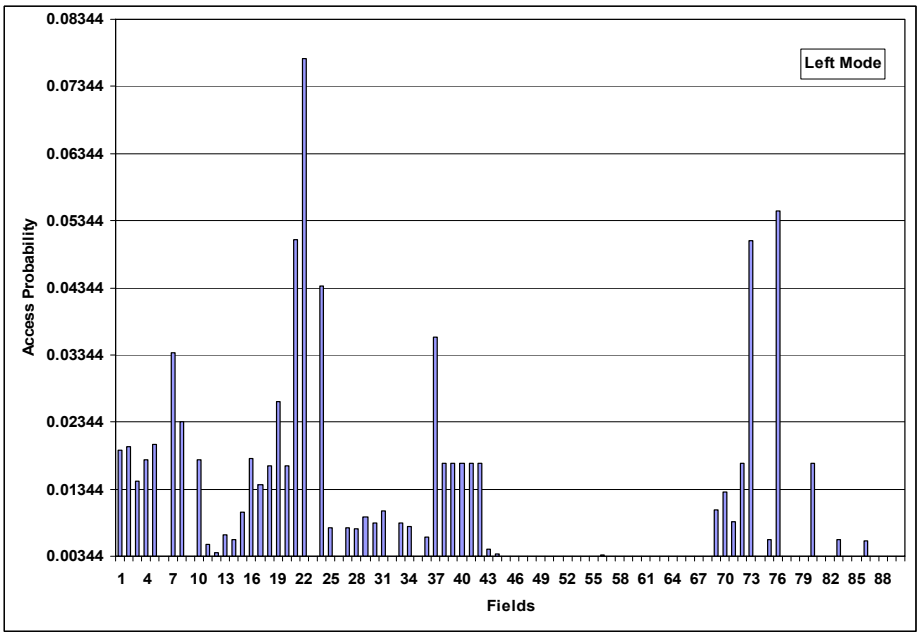


Figure 39: Access probability stationary vector, left mode location

Again, as seen in the two previous models, most of the fields have very low probability of being accessed, whereas the fields with the highest probability form a small and compact set.

Finally, to evaluate the accuracy of this model, we used a null hypothesis test, which was formulated in the same way as the one used for the Bayesian Inference model: We compare the prediction generated by the system, under the form of the stationary vector with the field access probabilities, to the actual access probabilities observed during a subsequent execution of the benchmark, modelled with the same location mode invocations. The data on which the test is based can be seen in Figure 40, where the results for the left mode location are presented. The Z values obtained for the three mode location invocations of the benchmark are all practically equal to zero (belonging to the area of 95% level of confidence). This leads us to the conclusion that the predictions generated by the system, while employing the Markov Chain model, are very precise—that is, they are able to indicate correctly the actually observed tendencies in future executions of the target application.

5.3 Overhead of the data access monitoring

There is a significant amount of code injected in the application to collect all the information needed to build the models. The execution of this extra code introduces overheads and, therefore, may penalize the performance of the application, in comparison with its non-instrumented version. Thus, it is important to measure those overheads

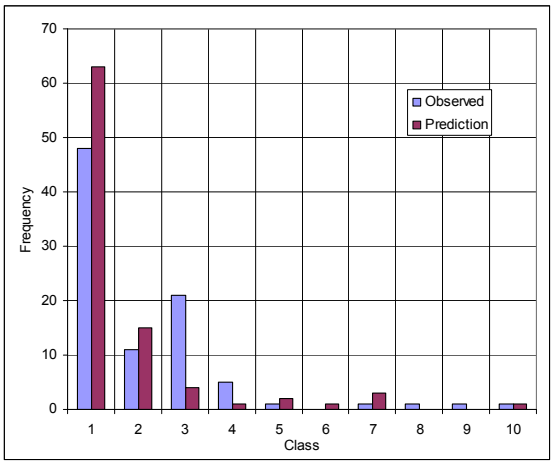


Figure 40: Populations used for Z value test, left mode

to determine if they are acceptable in the context of the normal operation of the application.

The mechanisms for gathering the input data for the Importance and Bayesian analysis models are significantly different from those employed for the Markov chain analysis, and, so, their respective overheads shall be considered separately. Additionally, whereas the data acquisition mechanisms for the former two analysis models are expected to operate while the application is operating normally, the gathering of data for the Markov chain analysis takes place only during the training period of the model. Consequently, the negative effects of the overheads introduced by the Markov chain data collection are less significant than the ones for the other two models.

We start with the analysis of the overheads for the Bayesian and Importance Analysis models. The weighted average, $\overline{overhead}$, is calculated as:

$$\overline{overhead} = \frac{\sum_{i=1}^{n} overhead_{method,i} \times t_{method,i}}{\sum_{i=1}^{n} t_{method,i}} \tag{81}$$

where n is the total number of methods, $overhead_{method,i}$ is the overhead, in percentage, associated with the method with index i and $t_{method,i}$ is the execution time of the method indexed by i.

The weighted average of the performance overhead caused by the collection of the data access patterns was 5.15%. This means that the instrumented version of the application, for either the Bayesian model or the Importance Analysis model is, on average, about 5% slower, in its execution, when compared with the original, non-instrumented application.

In terms of the overheads introduced by data gathering for the Markov Chain model, the weighted average, again calculated by Equation 81, is 9.14%.

Processing all of the input data and generating the prediction models by any of the three approaches takes at most a few milliseconds to terminate, and, as such, does not incur significant overheads to the performance of the target application.

Finally, we point out that the above overhead measurements correspond to the worst case scenario, because all of the existing methods and classes have been instrumented. In most situations, it would be of interest to instrument only a subset of these, abstracting away less important aspects of the operation of the target application.

5.4 Related work

Most of the work on data access patterns characterization has been performed as part of some other work where the authors are trying to explore that knowledge to optimize some aspect of an application execution. Typically, the emphasis of that work is on the optimization technique rather than on the data access patterns models themselves, and, thus, the models used to predict the data access patterns are relatively simple. To the best of our knowledge, there is no previous published work that uses stochastic methods to predict the behaviour of object-oriented applications, in what concerns the domain data manipulations they perform at runtime. Still, we borrowed some of the ideas from this body of previous work.

For instance, Han et al. [64] presented a new prefetching technique where one of the fundamental ideas is that the data access patterns can be modelled in terms of the attribute references that are made when manipulating objects instead of the pattern of object references. In their work, they introduced the concepts of type-level access locality and type-level access pattern. Applications traverse the graph of objects, which represent the domain entities, through the references existing among objects. Whenever distinct objects of the same type are navigated, it is frequent for the same attributes to be referenced repeatedly. This repeated referral to the same attributes in objects of the same type is what defines the type-level access locality. The type-level access pattern corresponds to the pattern of attributes that are referenced in such situations. Similarly to this work of Han and colleagues, our data access models are class based, rather than instance based, but we use different modelling techniques to create the models of the data access patterns.

Another important concept used in our models is that of a context, which is similar to ideas used in other works on prefetching. One such case is the work of Ibrahim and Cook [72], which proposed the Autofetch system. Their system is implemented as an extension to Hibernate and adds automatically specifications for the realization of prefetch operations. One of the key concepts introduced in their work is that of a traversal of the graph of domain objects of an application. A traversal is defined as the subgraph of objects and associations that are accessed as a consequence of processing the results of a given query. The traversals capture the way that applications navigate through the graph of objects returned by the queries and they are aggregated into traversal profiles. Each of the nodes of a tree that represents a profile keeps information about the number of times that a given object was loaded from the database as well as its potential. The potential of an object is defined as the number of times that an object that has a direct reference to the considered object was accessed. Each query is classified with the aim of allowing several distinct queries to share the same

traversal profile. The criterion that is used to classify effectively the queries is the stack of invocations at the moment of execution of the query. In this way, queries that share an execution context are grouped into the same profile, because they will have similar traversals associated. Then, each time a query is to be executed, the system consults the associated traversal profile, and defines a prefetch specification. The concrete objects that will make part of the prefetch are determined based on the accumulated statistics for their nodes in the profile. When the quotient between the number of times that the object was loaded and its potential exceeds a pre-established threshold, the object in consideration will be a part of the concrete prefetch specification. The traversal profiles and how they are identified are similar to our contexts, but, again, we introduced more sophisticated stochastic models to capture the data access patterns. For instance, unlike the simple model presented in their work, the Bayesian model that we propose can adapt to variations in the workload. Other examples of prefetching systems that introduce the idea of a context are those proposed in [16, 144].

An example of a system that uses a more sophisticated model is that proposed by Knafla [81]. It uses a discrete-time Markov Chain to model the relationships between the objects. The method that Knafla developed is designated as hitting times and its final result is the probability for a certain page of data to be accessed, as well as the average access time. The probabilistic calculation is performed taking into account the structure of the existing relationships between the objects. The probabilities associated to the (possible) transitions between objects are used for accessing adjacent pages, from the current position in the graph of objects. If the calculated probability exceeds a threshold limit defined by taking into account cost/benefit parameters (that include not only the benefits obtained in case the estimate is correct but also the penalty associated with an unnecessary prefetch), then the page is a candidate for prefetching. This work of Knafla departs from the class based approaches described above, even though it cannot be seen as a pure instance based approach either, as it deals with memory pages rather than individual objects.

The Fido system, proposed by Palmer and Zdonik [106], on the other hand, is a system based on recognizing patterns of object references to predict future references. They used an associative memory to predict future requests based on access patterns that approximately match a pattern in the training trace.

Finally, a work worth mentioning here is the work by Madhyastha and Reed [90], which used neural networks and a hidden Markov model to predict future client behaviour, in terms of input/output patterns. They concluded that the Markov model offers a more precise control over caching and prefetching policies than neural network access pattern classification.

5.5 Integration in the Cloud-TM platform

The stochastic models described in this section for characterizing the data access patterns of an object-oriented application were developed in a generic way, meaning that they can be employed in any application containing a well identified domain model. Likewise, most of the Java code developed for instrumenting an application, collecting the statistics about the data accesses, and computing the models based on those statistics is self-contained and independent of any underlying platform.

However, this code has been integrated into the Cloud-TM platform to facilitate its use by applications built on top of this platform, and also to integrate it with the Autonomic Manager. At the moment of writing, we have a working prototype where the Workload Monitor of the Autonomic Manager may monitor the statistics collected about the data access patterns performed by an application developed on top of the Data Platform.

References

[1] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4), December 1987.

[2] J. Aitchison and I. Dunsmore. *Statistical Prediction Analysis*. Cambridge University Press, 1980.

[3] N. Al-Jumaha, H. Hassaneinb, and M. El-Sharkawia. Implementation and modeling of two-phase locking concurrency. *Information and Software Technology*, 42(4):257–273, March 2000. Elsevier Science.

[4] A. Andrzejak and L. Silva. Using machine learning for non-intrusive modeling and prediction of software aging. In *Proc. NOMS*, Apr 7–11 2008.

[5] M. Ansari, C. Kotselidis, I. Watson, C. C. Kirkham, M. Lujin, and K. Jarvis. Lee-tm: A non-trivial benchmark suite for transactional memory. In A. G. Bourgeois and S.-Q. Zheng, editors, *Proc. of the Int. Conf. on Algorithms and Architectures for Parallel Processing (ICA3PP)*, volume 5022 of *LNCS*, pages 196–207. Springer, 2008.

[6] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[7] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, 2002.

[8] A. Bartoli, C. Calabrese, M. Prica, E. A. D. Muro, and A. Montresor. Adaptive message packing for group communication systems. In R. Meersman and Z. Tari, editors, *Proceedings of the Workshop on Reliable and Secure Middleware (WRSM'03)*, number 2889 in Lecture Notes in Computer Science, pages 912–925. Springer-Verlag, 2003.

[9] D. M. Bates and D. G. Watts. *Nonlinear Regression Analysis and Its Applications*. Wiley, 1988.

[10] B.Ciciani, D.M.Dias, and P.S.Yu. Analysis of concurrency-coherency control protocols for distributed transaction processing systems with regional locality. *IEEE Transactions on Software Engineering*, 18(10):pp. 899–914, October 1992.

[11] B.Ciciani, D.M.Dias, and P.S.Yu. Dynamic and static load sharing in hybrid distributed-centralized systems. *Computer Systems Science and Engineering*, Volume 7(1):pp. 25–41, January 1992.

[12] M. Bennani and D. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 229 –240, june 2005.

[13] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, 1995.

[14] B. Berg. *Markov Chain Monte Carlo Simulations And Their Statistical Analysis*. World Scientific, 2004.

[15] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[16] P. A. Bernstein, S. Pal, and D. Shutt. Context-based prefetch for implementing objects on relations. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 327–338. Morgan Kaufmann Publishers Inc., 1999.

[17] U. N. Bhat, M. Shalaby, and M. J. Fischer. Approximation techniques in the solution of queueing problems. *Naval Research Logistics Quarterly*, 26(2):311–326, 1979.

[18] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1st ed. 2006. corr. 2nd printing edition, Oct. 2007.

[19] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[20] G. Box and G. Tiao. *Bayesian inference in statistical analysis*. Wiley classics library. Wiley, 1992.

[21] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the seventh International Conference on World Wide Web*, pages 107–117. Elsevier Science Publishers B. V., 1998.

[22] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *Proc. of the International Conference on Data Engineering (ICDE)*, pages 826–835, 2007.

[23] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. *Distributed systems (2nd Ed.)*, chapter The primary-backup approach, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[24] J. a. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, Dec. 2006.

[25] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[26] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 12–21. ACM, 1993.

[27] M. J. Carey and W. A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems*, 4(4):338–378, November 1986.

[28] B. Carmeli, G. Gershinsky, A. Harpaz, N. Naaman, H. Nelken, J. Satran, and P. Vortman. High throughput reliable message dissemination. In *Proc. SAC*, pages 322–327, 2004.

[29] N. Carvalho, P. Romano, and L. Rodrigues. Asynchronous lease-based replication of software transactional memory. In *Proc. Middleware*, pages 376–396. Springer Berlin / Heidelberg, 2010.

[30] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, 2006.

[31] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. In *Proceedings of the International Conference on Autonomic Computing*, ICAC '06, pages 231–242, Washington, DC, USA, 2006. IEEE Computer Society.

[32] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[33] B. Ciciani, D. M. Dias, and P. S. Yu. Analysis of replication in distributed database systems. *IEEE Trans. Knowl. Data Eng.*, 2(2), 1990.

[34] A. Coccoli, P. Urban, A. Bondavalli, and A. Schiper. Performance Analysis of a Consensus Algorithm Combining Stochastic Activity Networks and Measurements. In *Proc. DSN*, pages 551–560, 2002.

[35] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D^2STM: Dependable distributed software transactional memory. In *Proc. PRDC*. IEEE Computer Society, 2009.

[36] M. Couceiro, P. Romano, and L. Rodrigues. A machine learning approach to performance prediction of total order broadcast protocols. In *SASO'10: Proceedings of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, Budapest, Hungary, Sept. 2010.

[37] M. Couceiro, P. Romano, and L. Rodrigues. A machine learning approach to performance prediction of total order broadcast protocols. In *Proc. SASO*, pages 184–193, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[38] F. Cristian, R. D. Beijer, and S. Mishra. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering*, 1:177–201, 1994.

[39] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[40] J. Dejun, G. Pierre, and C.-H. Chi. Resource provisioning of web applications in heterogeneous clouds. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, pages 5–5. USENIX Association, 2011.

[41] P. di Sanzo, B. Ciciani, F. Quaglia, and P. Romano. A performance model of multi-version concurrency control. In *Proceedings of the 16th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2008)*, pages 41–50. IEEE Computer Society, 2008.

[42] P. di Sanzo, B. Ciciani, F. Quaglia, and P. Romano. A performance model of multi-version concurrency control. In *MASCOTS*, 2008.

[43] P. Di Sanzo, R. Palmieri, B. Ciciani, F. Quaglia, and P. Romano. Analytical modeling of lock-based concurrency control with arbitrary transaction data access patterns. In *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 69–78, New York, NY, USA, 2010. ACM.

[44] P. di Sanzo, R. Palmieri, B. Ciciani, F. Quaglia, and P. Romano. Analytical modeling of lock-based concurrency control with arbitrary transaction data access patterns. In *WOSP/SIPEW*, 2010.

[45] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[46] T. Dietterich. Overfitting and undercomputing in machine learning. *ACM Comput. Surv.*, 27(3):326–327, 1995.

[47] R. Ekwall and A. Schiper. Modeling and validating the performance of atomic broadcast algorithms in high-latency networks. In *Proc. Euro-Par*, Lecture Notes in Computer Science, pages 574–586. Springer, 2007.

[48] E. Elmroth and J. Tordsson. A grid resource broker supporting advance reservations and benchmark-based resource selection. In *State-of-the-art in Scientific Computing*, 2006.

[49] J. Erman, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson. Offline/real-time traffic classification using semi-supervised learning. *Perform. Eval.*, 64(9-12):1194–1213, 2007.

[50] P. Felber, C. Fetzer, R. Guerraoui, and T. Harris. Transactions are back—but are they the same? *SIGACT News*, 39(1):48–58, 2008.

[51] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, New York, NY, USA, 2008. ACM.

[52] P. Felber, T. Riegel, C. Fetzer, M. Subkraut, U. Muller, and H. Sturzrehm. Transactifying applications using an open compiler framework. In *In Proc. of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[53] E. Frank, M. A. Hall, G. Holmes, R. Kirkby, B. Pfahringer, and I. H. Witten. *Weka: A machine learning workbench for data mining.*, pages 1305–1314. Springer, Berlin, 2005.

[54] R. Friedman and E. Hadad. Adaptive batching for replicated servers. In *Proc. SRDS*, pages 311–320, 2006.

[55] T. Friedman and R. V. Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *Proc. of HPDC*, pages 233–, 1997.

[56] L. Garces-Erice. Admission control for distributed complex responsive systems. In *Proc. ISPDC*, pages 226–233, Washington, DC, USA, 2009. IEEE Computer Society.

[57] S. Ghanbari, G. Soundararajan, J. Chen, and C. Amza. Adaptive learning of metric correlations for temperature-aware database provisioning. In *Proceedings of the International Conference on Autonomic Computing*, ICAC '07, pages 26–, Washington, DC, USA, 2007. IEEE Computer Society.

[58] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the Conference on the Management of Data (SIGMOD)*, pages 173–182. ACM, 1996.

[59] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proc. of PPOPP*, 2008.

[60] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. *SIGOPS Operating Systems Review*, 41(3):315–324, 2007.

[61] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.

[62] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[63] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, 2003.

[64] W. Han, K. Whang, Y. Moon, and I. Song. Prefetching based on the type-level access pattern in object-relational DBMSs. In *Proceedings of the 17th International Conference on Data Engineering*, pages 651–660. IEEE, 2001.

[65] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1994.

[66] Z. He and B. Hong. On the performance of commit-time-locking based software transactional memory. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, pages 180–187, Washington, DC, USA, 2009. IEEE Computer Society.

[67] A. Heindl and G. Pokam. An analytic framework for performance modeling of software transactional memory. *Comput. Netw.*, 53(8):1202–1214, 2009.

[68] A. Heindl and G. Pokam. Modeling software transactional memory with anylogic. In *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[69] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41(10):253–262, 2006.

[70] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[71] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[72] A. Ibrahim and W. R. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 50–73. Springer-Verlag, 2006.

[73] I.K.Ryu and A.Thomasian. Analysis of database performance with dynamic locking. *Journal of the ACM (JACM)*, 37(3):pp. 491 – 523, July 1990.

[74] B. Jiang, X. Zhang, and T. Cai. Estimating the confidence interval for prediction errors of support vector machine classifiers. *J. Mach. Learn. Res.*, 9:521–540, 2008.

[75] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, 1997.

[76] D. Kececioglu. *Reliability engineering handbook*. Prentice-Hall, 1991.

[77] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS)*, page 156. IEEE Computer Society, 1998.

[78] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *Proc. of the Very Large Data Base Conference (VLDB)*, pages 134–143, Cairo, Egypt, 2000. ACM.

[79] L. Kleinrock. *Queueing Systems. Volume 1: Theory*. Wiley-Interscience, Jan. 1975.

[80] L. Kleinrock. *Queuing Systems (Vol1 and Vol2)*. Wiley-Interscience, 1975.

[81] N. Knafla. Analysing object relationships to predict page access for prefetching. In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3): Advances in Persistent Object Systems*, pages 160–170. Morgan Kaufmann Publishers Inc., 1999.

[82] F. Korn, B.-U. Pagel, and C. Faloutsos. On the 'dimensionality curse' and the 'self-similarity blessing'. *IEEE Trans. Knowl. Data Eng.*, 13(1):96–111, 2001.

[83] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44, 2010.

[84] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[85] L. Lamport. Lower bounds for asynchronous consensus, 2004.

[86] J. Langford. *Logistics: principles and applications*. McGraw-Hill, 1995.

[87] S. T. Leutenegger and D. Dias. A modeling study of the tpc-c benchmark. *SIGMOD Rec.*, 22(2):22–31, 1993.

[88] D. Lindley. *Introduction to Probability and Statistics: From a Bayesian Viewpoint*. Cambridge University Press, 1976.

[89] J. D. C. Little. A proof for the queuing formula: L= ? w. *Operations Research*, 9(3):383–387, 1961.

[90] T. M. Madhyastha and D. A. Reed. Input/output access pattern classification using hidden Markov models. In *Proceedings of the fifth workshop on I/O in parallel and distributed systems*, IOPADS '97, pages 57–67. ACM, 1997.

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

[92] P. Martin, S. Elnaffar, and T. Wasserman. Workload models for autonomic database management systems. In *Proc. of the International Conference on Autonomic and Autonomous Systems (ICAS)*, page 10, Washington, DC, USA, 2006. IEEE Computer Society.

[93] D. A. Menascé and T. Nakanishi. Performance evaluation of a two-phase commit based protocol for ddbs. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '82, 1982.

[94] S. Meyn and R. Tweedie. *Markov Chains and Stochastic Stability*. Cambridge University Press, 2009.

[95] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. ICDCS'01*, pages 707–710. IEEE, Apr. 2001.

[96] M. Mirza, J. Sommers, P. Barford, and X. Zhu. A machine learning approach to tcp throughput prediction. In *Proc. SIGMETRICS*, pages 97–108, New York, NY, USA, 2007. ACM.

[97] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[98] M. Moir, K. Moore, and D. Nussbaum. The adaptive transactional memory test platform: a tool for experimenting with transactional code for rock (poster). In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 362–362, New York, NY, USA, 2008. ACM.

[99] V. Nae, A. Iosup, and R. Prodan. Dynamic resource provisioning in massively multiplayer online games. *IEEE Transactions on Parallel and Distributed Systems*, 22(3):380–395, March 2011.

[100] J. Nagle. Congestion control in ip/tcp internetworks. *SIGCOMM Comput. Commun. Rev.*, 14:11–17, October 1984.

[101] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, 2010.

[102] R. Nelson. *Probability, stochastic processes, and queueing theory: the mathematics of computer performance modeling*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[103] U. S. D. of Defense. *Procedures for Performing a Failure Mode, Effects and Criticality Analysis*. MIL STD 1629A : 1980. U.S. Department of Defense, 1980.

[104] Oracle. Oracle Coherence. `http://www.oracle.com/technetwork/middleware/coherence/overview/index.html`, 2011.

[105] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, 2009.

[106] M. Palmer and S. B. Zdonik. Fido: A cache that learns to fetch. Technical report, Brown University, Providence, RI, USA, 1991.

[107] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho. Evaluating database-oriented replication schemes in software transactional memory systems. In *Proc. of International Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, Oct. 2006.

[108] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proc. of the International Conference on Distributed Computing (DISC)*, pages 315–329. Springer-Verlag, 2000.

[109] K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(6):559–572, 1901.

[110] F. Pedone, R. Guerraoui, and A. Schiper. The Database State Machine Approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[111] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[112] C.-T. Project. Deliverable d2.2: Preliminary prototype of the workload monitor, 2011.

[113] C.-T. Project. Deliverable d3.1: Prototype of the workload monitor, 2011.

[114] C.-T. Project. Deliverable d3.1: Prototype of the workload analyzer, 2012.

[115] J. R. Quinlan. Cubist. `http://www.rulequest.com/cubist-info.html`.

[116] J. R. Quinlan. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348. World Scientific, 1992.

[117] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[118] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[119] N. Raghavan and R. Vitenberg. Balancing the communication load of state transfer in replicated systems. In *SRDS*, 2011.

[120] D. Ries and M. Stonebraker. Effects of locking granularity in a database management system. *ACM Transactions on Database Systems (TODS)*, 2(3), September 1977.

[121] D. R. Ries and M. Stonebraker. Locking granularity revisited. *ACM Transactions on Database Systems (TODS)*, 4(2), 1974.

[122] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[123] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–35, 1952.

[124] P. Romano, N. Carvalho, and L. Rodrigues. Towards distributed software transactional memory systems. In *Proc. LADIS'08)*, Watson Research Labs, Yorktown Heights (NY), USA, Sept. 2008. (invited paper).

[125] D. E. Rumelhart, R. Durbin, R. Golden, and Y. Chauvin. *Backpropagation: the basic theory*, pages 1–34. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1995.

[126] N. Santos and A. Schiper. Tuning paxos for high-throughput with batching and pipelining. In *Proc. ICDCN*, 2012.

[127] P. D. Sanzo, B. Ciciani, F. Quaglia, and P. Romano. Analytical modelling of commit-time-locking algorithms for software transactional memories. In *Proc. 35th International Computer Measurement Group Conference (CMG)*, 2010.

[128] N. Schiper, P. Sutra, and F. Pedone. P-Store: Genuine partial replication in wide area networks. In *Proc. of the Symposium on Reliable Distributed Systems (SRDS)*, pages 214–224, Washington, DC, USA, 2010. IEEE Computer Society.

[129] F. B. Schneider. *Replication management using the state-machine approach*. ACM Press/Addison-Wesley Publishing Co., 1993.

[130] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[131] S. K. Shevade, S. S. Keerthi, C. Bhattacharyya, and K. R. K. Murthy. Improvements to the smo algorithm for SVM regression. *IEEE-NN*, 11(5), September 2000.

[132] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proceedings of the International conference on Autonomic computing*, ICAC '10, pages 21–30, New York, NY, USA, 2010. ACM.

[133] W. Smith. TPC-W: Benchmarking an ecommerce solution. White paper of the Transaction Processing Performance Council, 2001.

[134] Y. C. Tay, N. Goodman, and R. Suri. Locking performance in centralized databases. *ACM Trans. Database Syst.*, 10, 1985.

[135] Y. C. Tay, R. Suri, and N. Goodman. A mean value performance model for locking in databases: the no-waiting case. *J. ACM*, 32(3):618–651, 1985.

[136] A. Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Computing Surveys*, 30(1), March 1998.

[137] A. Thomasian and I. Ryu. Performance analysis of two-phase locking. *IEEE Transactions on Software Engineering*, Volume 17(5):386 – 402, May 1991.

[138] S. Thompson. *Sampling*. Wiley series in probability and statistics. Wiley, 2002.

[139] A. A. Tomusyak. Computation of an ergodic distribution of markov and semi-markov processes. *Kibernetika*, (1), 1969.

[140] Transaction Processing Performance Council. *TPC Benchmark^{TM} C, Standard Specification, Revision 5.1*. Transaction Processing Perfomance Council, 2002.

[141] J. S. Vetter and D. A. Reed. Managing performance analysis with dynamic statistical projection pursuit. In *Proc. Supercomputing '99*, page 44, New York, NY, USA, 1999. ACM.

[142] L. Wang, J. Xu, M. Zhao, Y. Tu, and J. A. B. Fortes. Fuzzy modeling based resource management for virtualized database systems. In *MASCOTS*, 2011.

[143] Z. Wang, X. Zhu, and S. Singhal. Utilization and slo-based control for dynamic sizing of resource partitions. In *DSOM*, 2005.

[144] B. Wiedermann and W. R. Cook. Extracting queries by static analysis of transparent persistence. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '07, pages 199–210. ACM, 2007.

[145] R. W. Wolff. Poisson arrivals see time averages. *Operations Research*, 30(2):223–231, 1982.

[146] D. H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Comput.*, 8(7):1341–1390, 1996.

[147] P. Wu, M. M. Michael, C. von Praun, T. Nakaike, R. Bordawekar, H. W. Cain, C. Cascaval, S. Chatterjee, S. Chiras, R. Hou, M. Mergen, X. Shen, M. F. Spear, H. Y. Wang, and K. Wang. Compiler and runtime techniques for software transactional memory optimization. *Concurr. Comput. : Pract. Exper.*, 21(1):7–23, 2009.

[148] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. HacigümüŞ. Activesla: a profit-oriented admission control framework for database-as-a-service providers. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. ACM.

[149] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing*, 11(3):213–227, 2008.

[150] L. Yang, J. M. Schopf, C. L. Dumitrescu, and I. Foster. Statistical data reduction for efficient application performance monitoring. In *Proc. CCGRID '06*, pages 327–334, Washington, DC, USA, 2006. IEEE Computer Society.

[151] P. S. Yu, D. M. Dias, and S. S. Lavenberg. On the analytical modeling of database concurrency control. *Journal of the ACM (JACM)*, 40(4):831–872, September 1993.

[152] P. S. Yu, D. M. Dias, and S. S. Lavenberg. On the analytical modeling of database concurrency control. *J. ACM*, 40, 1993.

[153] S. Zhang, I. Cohen, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *Proc. DSN '05*, pages 644–653, Washington, DC, USA, 2005. IEEE Computer Society.