Chorus: an Interactive Approach to Incremental Modeling and Validation in Clouds

Jin Chen[⋆], Gokul Soundararajan[⋆], Saeed Ghanbari[⋆],
Zartab Jamil[⋆], Mike Dai Wang [†⋆], Ali B. Hashemi[⋆] and Cristiana Amza [⋆]
Department of Electrical and Computer Engineering
University of Toronto [⋆]
IBM Canada Software Laboratory, CAS Research [†]

Abstract

Performance modeling is an emerging approach towards automating management in Clouds. The need for fast model adaptation to dynamic changes, such as workload mix changes and hardware upgrades, has, however, not been previously addressed. Towards this we introduce Chorus, an interactive framework for fast refinement of old models in new contexts and building application end-to-end latency models, incrementally, on-the-fly.

Chorus consists of (i) a declarative high-level language for expressing expert hypotheses, and system inquiries (ii) a runtime system for collecting experimental performance samples, learning and refining models for parts of the end-to-end configuration space, on-the-fly. We present our experience with building the Chorus infrastructure, and the corresponding model evolution for two industry-standard applications, running on a multi-tier dynamic content server platform. We show that the Chorus on-the-fly modeling framework provides accurate, fast and flexible performance modeling by reusing old approximate models, while adapting them to new situations.

1 Introduction

Cloud environments, such as, Amazon EC2, and Google Apps are large-scale consolidated compute and storage platforms hosting various applications. These shared infrastructures evolve over time through a series of hardware component replacements, software version upgrades, platform utilization changes, and configuration parameter changes. Assuming that the applications hosted on a system are long-running applications, their respective workload mixes and QoS requirements evolve dynamically as well.

Performance modeling is an emerging approach towards automating management of the compute resources and storage hierarchy in data centers and Clouds. In this paper, we explore some of the problems that performance modeling presents in practice, in these dynamic environments. We also present our experience while in the process of finding partial, but practical solutions to a few of these problems.

Much previous work [4, 6, 10, 15, 17, 21] has gone into building performance models from scratch. However, the ultimate performance model, which always covers all possible situations and accurately predicts the answer to anything we may ask of it can never be built for a complex enough system. Hence, the performance models of systems and applications, as well as the human expertize that these performance models incorporate are almost always partial. The modeling framework should evolve end-to-end models from partial or simpler models for parts of the configuration space as they get actuated, and be able to evolve to reflect platform and application changes.

Towards this goal, model validation, reuse, extension, refinement and adaptation to dynamic changes, such as, the region of the configuration space at hand, workload mix changes, and hardware upgrades become very important. Even more important, in practice, is providing some degree of user visibility into the capabilities, contexts and limitations of existing models.

In our experience with building performance models for the storage hierarchy in data centers and Clouds, we found that allowing the user to dialogue with the system through and about existing models was a crucial aspect of a synergistic evolution of the modeling space and user experience over time. In this paper, we describe our hands-on modeling and dialogue experience, towards this overarching goal, through a case study. This case study describes how our modeling and validation framework evolved for a practical scenario. In this scenario, we are concerned with modeling two e-commerce applications towards allowing dynamic resource allocation to these applications on an in-house Cloud platform.

More importantly, as part of this experience, we have built Chorus, an interactive runtime framework for building, storing and querying application latency models for the storage hierarchy, incrementally, on-the-fly, and fast refinement of old models in new contexts.

Chorus consists of (i) a declarative high-level language for expressing expert hypotheses, and system inquiries (ii) a runtime system for collecting experimental performance samples, learning and refining models for parts of the end-to-end configuration space, on-the-fly.

Our experimental evaluation shows that Chorus (i) can integrate the expertise of a system administrator, as approximate model guidance, (ii) dynamically selects the most accurate modeling techniques for different configuration regions of the overall configuration space, (iii) matches or even outperforms the accuracy of the best model per configuration region and (iv) incrementally adjusts existing models on the fly in new resource configurations and workload mix situations.

2 Background and Motivation

A *performance model* is a mathematical function that calculates an estimate of the application performance for a range of resource configurations. For example, Figure 1 shows a performance model as a 3D surface for the on-line auctions application RUBiS (modeled after e-Bay). This example model provides an estimate of the average memory access latency of the MySQL database engine running the RUBiS application, for all possible memory quotas, in a storage hierarchy consisting of buffer pool, and disk.

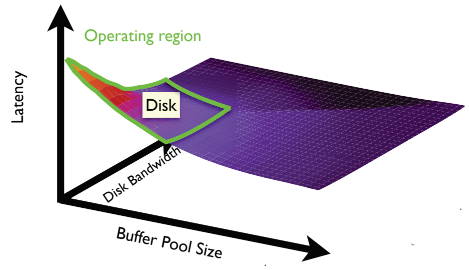


Figure 1: **RUBiS Latency Model and Operating Region of a Partial Model.**

Automatic, black-box performance model building involves iterating through two steps: (i) gathering experimental samples, and (ii) modeling computation. Gathering experimental samples means actuating the experimental system into a given resource configuration, running a specific application workload on the live system (or equivalent) and measuring the application latency. Modeling computation involves mathematical interpolation for building the model on existing sampling data. While modeling computation is typically on the order of fractions of seconds, experimental sampling may take months for mapping out the entire resource configuration space of an application with sufficient statistical accuracy. This is due to dynamic effects, such as, cache warm-up time, which make reliable actuation and sampling of even a single configuration point expensive (on the order of tens of minutes).

For our RUBiS example in Figure 1, due to cache warmup effects, experimental sampling takes around 15 minutes of measurements at *each* of the 1024 (32^2) points of the surface. The total sampling takes approximately 11 days. In an enterprise environment, where 64GB storage caches are common, if we set sampling increments in 1GB units, total sampling would take 2 months.

At the other end of the spectrum is using analytical models that rely on sysadmin or analyst semantic knowledge of the system and application [14, 17]. However, these analytical models can be precise only for restricted parts of the system, application workload mix, or resource configurations, are brittle to dynamic changes and require too much domain expertise.

In addition to the cost of building an initial model, adapting a model to new situations automatically, or in an ad-hoc manner, is hard, because of

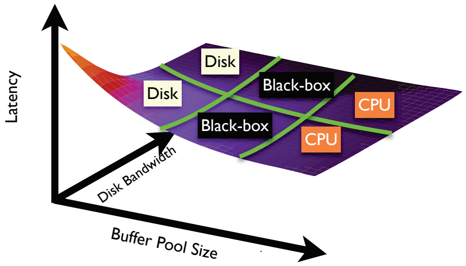


Figure 2: **An Example of Ensemble Learning.**

inter-dependencies between the various resources. For example, for a given application, a particular cache quota setting in the buffer pool, and the replacement policy of the database system influence the number and type of accesses seen at the storage cache, and the storage cache settings further influence the access patterns seen at disk.

Figure 2 shows a simple interactive scenario visually for modeling a workload with Chorus (although visual display would not be possible for higher dimensionality models). Our framework is flexible enough to allow the sysadmin to express model templates, e.g., the belief that a simple Disk-bound and/or CPU-bound model may fit parts of a configuration space shown in the figure and also the operating region(s) of these models, if known. It is important to note that either Disk-bound or CPU-bound may have lower dimensionality than that of the configuration space of the overall model. Alternatively, the sysadmin can ask the Chorus runtime system to automatically find the operation region for any model she provides for validation. Experimental sampling is still required for fitting given model templates to the data and for building black box models from scratch for the regions of the resource configuration space where no model is suspected or known. However, specifying suspected models, or reusing older models that may fit parts of the configuration space is expected to speed up total modeling time, especially for small changes in the system or application, as well as to allow interactive model query and validation.

3 Chorus Design

Chorus is an interactive framework for facilitating management of large scale, complex Cloud environments, subject to dynamic change. We focus on on-line performance modeling learning and model reuse to speed up modeling upon partial system and application changes.

Each long-running application deployed on the Cloud may experience many dynamic changes during its deployment. For example, there may be changes in resource availability, due to dynamic co-hosting of other applications, changes in the application's own workload mix, changes in the infrastructure itself, due to component replacements or upgrades. On the other hand, common application characteristics across e-commerce applications, infrastructure structural information, or resource availability situations typically show repeatable, recognizable patterns over time. For example, the most prevalent workload mixes of Web applications, such as, the Amazon workload, may show a daily, or seasonal repetition pattern, motivating performance model archiving for possible future model reuse. Furthermore, most caches throughout the system use some variant of LRU cache replacement; likewise other system components, e.g., hard disks have stable functional laws of operation across upgrades. Finally, the flow of information between tiers in a multi-tier system is based on known component structure and inter-connectivity.

Therefore, an interactive dialogue between the system and its administrator becomes necessary towards effective, dependable and accountable resource management in Cloud environments.

Towards this, Chorus contains two interrelated components: (i) SysTalk: a high-level declarative interface for human-defined templates guiding the learning of new models, or for pruning the sampling configuration space, and (ii) the Chorus Runtime Engine: a runtime engine for validation of models with experimental data, and for maintenance, inquiry, and reuse of the modeling knowledge base.

3.1 Chorus Overview

The sysadmin or performance analyst provides *model templates* to Chorus in the form of suspected functional relations between metrics, or templates for curve fitting.

The Chorus run-time engine validates the administrator templates with experimental data, as well as accumulates and reuses a library of models over time.

In new situations, and wherever model guides are not available, Chorus gathers experimental

```
TEMPLATE <templateName>
RELATION <relationName> {<metricSet>}
CONTEXT {<contextSet>}
```

Listing 1: Syntax of Model Template

samples and uses black-box statistical regression to derive models.

Over time, the Chorus knowledge base accumulates approximate models for different applications and resource configuration ranges, as they dynamically present themselves. Chorus provides APIs for storing, and querying models as well as automated techniques for retrieving, extending, refining old models with new samples, and combining models to fit larger areas of the configuration space. Chorus can answer analyst *inquiries* about any model, resource configuration region and what-if scenario in a semantically meaningful way. In this paper, we focus on per-application predictive models for the application latency as a function of various resources at the database and storage server tiers in a datacenter.

3.2 SysTalk Model Templates

We introduce *SysTalk*, a high-level, SQL-like declarative language for model guidance and inquiry in Chorus. *SysTalk* offers: model templates tagged with semantic information, clues to the Chorus runtime for pruning the sampling space, and model inquiries. The syntax of a model *template* is shown in Listing 1 (keywords are underlined).

The analyst uses model templates to express her beliefs about analytical performance models for the Chorus runtime to validate with experimental data. Each *template* is identified by a unique name; this allows the template to be saved in a database and later retrieved for future inquiry. The *relation* defines a mathematical function describing the relationship between metrics; it is identified by a relation name and it may be used in several models. A relation could be a suspected mathematical correlation to be validated, curve fitted, or otherwise refined. The *context* is a list of conditions on a set of configurations, in which the analyst believes her template holds. Any parameter, resource configuration, or property that the given relation in the *template* is sensitive to can be specified as an associated expected *context* for that relation. Within the template context, configuration ranges for particular resources can be specified, or left as empty.

```
TEMPLATE CPU_Bound
RELATION Constant(x) {
       x.name=‘disk_bandwidth’
}
CONTEXT (a) {
      a.name=‘memory_size’ and a.value>=2GB
}
```

Listing 2: CPU_Bound Model Template

For instance, a performance model, called CPU_Bound, shown in Listing 2, is given to the Chorus run-time to learn.

This model is designed by a performance analyst to model a CPU intensive database application. This type of application's latency will only be affected by CPU resources, and won't be affected by the change of the disk bandwidth. Hence, this model is declared that application latency has constant relation with the value of the disk bandwidth. Since this constant relation only holds when the memory resource is sufficient to contain the whole working set (i.e. 2GB in this example), in the context of this model, the analyst declares that the memory size (i.e. buffer pool size) allocated to the application needs to be no less than 2GB according to her knowledge.

In our experimental prototype, in order to gather performance samples, Chorus dynamically varies the settings of system configuration parameters (e.g. disk bandwidth, buffer pool size) and measure performance parameters (e.g. latency, throughput). Based on gathered samples, Chorus validates this model and computes confidence scores and error rates for this model.

The configuration threshold for memory size could be left unknown. In unknown case, the a.value in this model is left empty as "*". Chorus will gather experimental samples, validate, find the memory size of the working set through a systemically trial and error approach.

Chorus stores each model in conjunction with a *feature set*. The *feature set* helps Chorus to detect matches between old and new modeling situations; it consists of a set of metrics and their respective value ranges that are believed to be most representative for the model they are associated with. In addition to any metrics used to define the model's

```
INQUIRY CPU_Bound
CONTEXT (a) {
  a.name='memory_size' and a.value='*'
}
RELATIVE_ERROR < 0.2
ORDER BY RELATIVE_ERROR DESC
```

Listing 3: Inquiry on CPU_Bound Model

context by the analyst, each model's feature sets includes environmental parameters or workload characteristics such as, a list of software and hardware components, and the read/write ratio, average working set size, and I/O intensity of the workload.

A model template thus expresses incomplete domain knowledge with or without specifying a concrete context. It is the task of the Chorus runtime to validate and calibrate the associated model, and to find out the context where that model holds.

Within any *model template*, Chorus can refer standard mathematical relations (e.g., linear, exponential, inverse) or machine learning algorithms (e.g. SVM regression), as well as customized analytical models that are provided by analysts. Moreover, the models previously learned for one application online can be later provided as a starting approximation i.e., *template* for new applications, or for new scenarios of the same application.

3.3 SysTalk Inquiries

Finally, upon an analyst *inquiry*, Chorus can provide automated feedback regarding the degree of fit, error rates, or confidence of any model template for any area of the configuration space. A time bound, and accuracy requirement can be specified for each modeling task, or inquiry.

For example, the sysadmin may ask for all available memory sizes where a pre-defined CPU_Bound model registered average relative error rates of below 20% for a given application. This inquiry is shown in Listing 3.

3.4 Operation of the Chorus Runtime

The processing flow of the runtime engine is shown in Figure 3. The module matching uses a feature-based similarity detection technique to recognize situations that are similar to those previously modeled. Chorus can thus minimize the number of additional experimental samples needed in modeling new workload by leveraging samples taken from adjacent areas of the configuration space and/or for historical models with similar features in their feature sets.

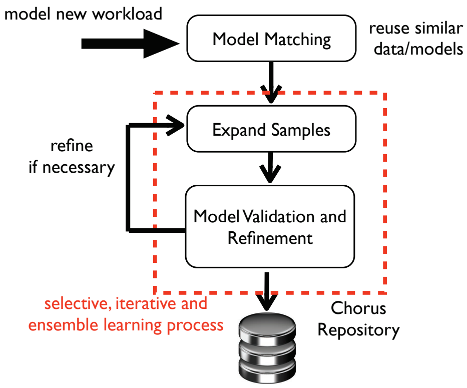

model new workload
Model Matching
reuse similar data/models
Expand Samples
refine if necessary
Model Validation and Refinement
selective, iterative and ensemble learning process
Chorus Repository

Figure 3: **Runtime Engine Processing Flow.**

If the historical models or any newly provided model templates cannot provide the desired accuracy, Chorus enters the iterative learning process where it expands its sample set and checks the validity of each model template with monitored historical data, and live experimental sampling.

For the configuration ranges where the applicability of several models overlaps, Chorus automatically finds the best model within each region. As a fall-back model, wherever guidance from the sysadmin is either unavailable, or too inaccurate to fit to the data, Chorus uses a default, fully automated, black-box model; the black-box model is based on statistical regression for interpolation over experimental sampling.

Thus, both the dialogue between system and administrator in SysTalk and the resulting incremental knowledge accumulation allow Chorus to optimize its sampling of the search space, by focused sampling in estimated areas of interest, or by pruning uninteresting areas.

3.5 Model Validation Process

Algorithm 1 shows the model validation process in Chorus. The analyst or sysadmin specifies model templates, clues or inquiries into text template files. These templates are inputs to the system as a collection of performance model templates. The outcome is an overall model for C called an *ensemble* model. The algorithm uses performance samples gathered at runtime to refine each approximate model, and it ranks the models by accuracy per re-

Algorithm 1 Iterative algorithm to build an ensemble of models for a configuration space C.

```
Initialization:
Select a collection of performance model templates $M$
Divide configuration space evenly into $l$ regions
Select $v$ samples to construct test sample set $\mathcal{S}_v$.
Training set $\mathcal{S}_t = \varnothing$, $m = size(M)$
Iterative Training:
repeat
  /* Expand the training sample set */
  Add $t$ new samples to the training sample set $\mathcal{S}_t$.
  /* Build ensemble of models */
  Set ranking set $\mathcal{W} = \varnothing$.
  Partition the training set $\mathcal{S}_t$ into $k$ subsets.
  for $i = 1$ to $k$ do
    1) Use $i^{th}$ subset as validation set $\mathcal{S}'_v$
    2) Use other $k - 1$ subsets as training set $\mathcal{S}'_t$
    for $j = 1$ to $m$ do
      3) Train each base model $M_j$ on $\mathcal{S}'_t$
      4) Test $M_j$ on $\mathcal{S}'_v$
    end for
  end for
  Derive rank per region from cross validation results
  Build an ensemble from the rank
  Test the ensemble of models on $\mathcal{S}_v$
until stop conditions are satisfied
```

gion within C. Our samples can be gathered on the live testing system by actuation into the desired configuration and taking several measurements of the application latency. In industry practice, it is common that a production system has one or multiple replicated testing systems, feeding either with live data or recorded real trace.

We build and rank the models using the training set and evaluate our approach using the testing set; the samples are gathered using user specified sampling methods (e.g. random sampling, greedy sampling, etc.) from the configuration space. The refinement occurs *iteratively*, by adding new samples until the stop condition (time, accuracy or both) is met. The size of the testing set is fixed.

Chorus ranks the models per region based on their cross-validation results, and keeps the rank results into its region table. The best ranked model in each region is selected to predict the performance for this region. In details, at each iteration, we use the samples in the training set to construct our *ensemble* model. We use a standard technique k-fold cross validation for the ranking process to avoid over-fitting, where k is 5 in our implementation. Finally, we test our *ensemble* of models on the testing set, and report its average relative error rate. If the test results satisfy the stop conditions, *Chorus* is ready to be used for prediction on C; otherwise, the iterative training process continues.

4 Platform and Methodology

In this section, we describe the benchmarks, platform, test and sampling methodology we use in our evaluation.

4.1 Testbed

We use one synthetic workload OLTP-A and two industry-standard benchmarks (TPC-W, TPC-C) to present our experience and evaluate *Chorus*.

OLTP-A: OLTP-A is a set of OLTP-like workloads we generate using ORION (Oracle I/O Calibration Tool) [11]. OLTP-A is characterized by many random I/O accesses of 16KB. We generate a set of OLTP-A workloads by configuring ORION with different arguments. Specifically, the number of outstanding IO is varied from 1 to 16, and the write ratio of data subsectionaccesses is varied from 0% to 100%. The sizes of raw disk partitions are varied as 512MB, 1GB and 2GB.

TPC-W: The TPC-W benchmark from the Transaction Processing Council [16] is a transactional web benchmark designed for evaluating e-commerce systems. Several web interactions are used to simulate the activity of a retail store. The database size is determined by the number of items in the inventory and the size of the customer population. We use 100K items and 2.8 million customers which results in a database of about 4 GB. In this chapter, we use the *browsing* workload, and create TPC-W^{10} by running 10 TPC-W instances in parallel creating a database of 40 GB.

TPC-C: The TPC-C benchmark [12] simulates a wholesale parts supplier that operates using a number of warehouse and sales districts. Each warehouse has 10 sales districts and each district serves 3000 customers. The workload involves transactions from a number of terminal operators centered around an order entry environment. There are 5 main transactions for: (1) entering orders (*New Order*), (2) delivering orders (*Delivery*), (3) recording payments (*Payment*), (4) checking the status of the orders (*Order Status*), and (5) monitoring the level of stock at the warehouses (*Stock Level*). Of the 5 transactions, only *Stock Level* is read only, but constitutes only 4% of the workload mix. We use 128 warehouses, which gives a database of 32GB.

RUBiS^{10}: We use the RUBiS Auction Benchmark to simulate a bidding workload similar to e-Bay. The benchmark implements the core functionality of an auction site: selling, browsing, and bidding. We are using the default RUBiS bidding workload containing 15% writes, considered the representative of an auction site workload. We create a scaled workload, RUBiS^{10} by running 10 RUBiS instances in parallel, which is about 30GB.

4.2 Server Platform

Our server platform is inspired by a Cloud service, Amazon Relational Database Service (Amazon RDS) [1], and consists of a storage hierarchy where we can allocate buffer pool, storage cache and disk bandwidth to applications. The details of our server platform are presented in the Appendix.

In our platform, we partition buffer pool and storage cache for different workloads and adjust memory quota dynamically. Proportion-share schedulers are used in the database server and the storage server for allocating CPU time quanta and disk bandwidth quanta proportionally. Similar resource partitioning mechanisms have been used in several recent studies [10, 14, 19], in order to relieve interference among concurrent workloads.

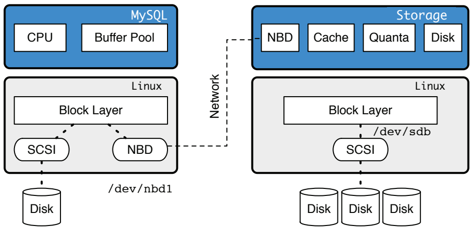


Figure 4: **Our server platform. It consists of a modified MySQL database server (shown in left) and a virtual storage prototype Akash (shown in right).**

As shown in Figure 4, our platform consists of a database server running modified MySQL code and a virtual storage prototype, called *Akash*. We modify the MySQL/InnoDB to have a quanta based scheduler for CPU usage allocation, and modify its buffer pool implementation to support dynamic partitioning and resizing for each workload partition. The database server connects to Akash through the network, using Network Block Device (NBD). Akash contains a storage cache which supports dynamic partitioning, and has a quanta based scheduler, which allocates the disk bandwidth to different workloads.

We configure Akash to use 16KB block size to match the MySQL/InnoDB block size. In addition, we use the Linux `O_DIRECT` mode to bypass any OS-level buffer caching and use the `noop` I/O scheduler to disable any OS disk scheduler. We run MySQL/InnoDB (version 5.0) database and storage server on Dell PowerEdge SC1450 with dual Intel Xeon processors running at 3.0 Ghz with 2GB of memory. To maximize I/O bandwidth, we use RAID 0 on 15 10K RPM 250GB hard disks.

Our platform provides strong isolation between workloads, and hence we are able to measure the performance impact of resource allocations for every workload through dynamically setting its resource quanta. Using this platform, multiple applications can be hosted on the same database server, and share the underlying storage.

4.3 Sampling Methodology

To evaluate our approach, we gather a large amount of performance data logs that are sampled from our platform over a period of *nine* months. We use these data to construct our training and testing data sets. During the data sampling process, for each resource configuration, we wait for cache warm-up, until the application miss-ratio is stable (which takes approximately 10 minutes on average in our experiments). Once the cache is stable, we monitor and record the average of application latency every 10 seconds for total 15 minutes to get about 90 sample latency points for each configuration. Once measured, sample points for a given configuration of an application as well as validated models are stored in a database on disk.

5 Results

In subsequent sections, we will present our experience for modeling the latency of applications hosted on our multi-tier server system. Our platform can host several concurrently running applications. To avoid interference among these applications, a proportional-share scheduler in the database server allocates CPU time in proportion (denoted by **CPU quanta**) to their respective CPU share. Similarily, a quanta based disk scheduler in the storage server allocates disk bandwidth in proportion (denoted by **disk quanta**) to each application.

In this section, we introduce some preliminary models that already exist in Chorus or are specified by the analyst and will be used in the experiments.

Black-box SVM Regression Model Template (B-SVM) Chorus uses a black-box model template to cover all scenarios where no model is known, or to refine areas where other models do not provide sufficient accuracy. In this case study, Chorus uses a well-known machine learning algorithm: *Support Vector Machine regression* [3] (B-SVM) as its default, fully automated, black-box model template. SVM estimates the performance for configuration settings we have not actuated, through interpolation between a given set of sample points. SVM is shown to scale well for highly-dimensional, non-linear data. Radial basis functions (G-RBFs) are used as kernel functions.

Gray-box Inverse Exponential Model Template (G-INV) Inverse_Exponential is a pre-defined mathematical relationship in Chorus, defined as follows:

$$\hat{y}_{\alpha,\beta}(x) = \frac{\alpha}{x^{\beta}} \tag{1}$$

In this formula, the parameters α and β are to be curve-fitted by Chorus.

Gray-box Region Model Template (G-RGN) The analyst believes that while the performance models of applications are complex in general, they are simple within a small range of configurations, i.e., constant, linear, or polynomial. Hence, we can model the performance using simple curve fitting within a region (i.e., a subset of configurations). While any function can be provided, for this model template, the analyst specifies the use of the average function for Chorus to fit the samples in each region.

Analytical Model Template (A-STOR) This is a memory latency model for our multi-tier storage hierarchy. This A-STOR template uses the relation *Analytical_Mem_Latency_Pred* for predicting the memory latency. The argument list (i.e. l,c,s,d) of this relation implies that the memory latency l is a function of the buffer pool size c of MySQL and storage cache sizes s of the storage server Akash, and the disk quanta d of Akash. Other system parameters not in the argument list have no impact on the memory latency in this model. This relation, which is implemented as the analyst-defined plug-in program implementing the (non-trivial) average memory latency $\mathcal{L}_{mem}$ formula shown below, is validated by the Chorus run-time, just like any standard relation. More details of this analytical model can be found in [14].

$$\begin{aligned}\mathcal{L}_{mem}(\rho_c, \rho_s, \rho_d) &= \underbrace{\mathcal{M}_c(\rho_c)\mathcal{H}_s(\rho_c, \rho_s)L_{net}}_{\text{I/Os satisfied by the storage cache}} \quad (2)\\ &+ \underbrace{\mathcal{M}_c(\rho_c)\mathcal{M}_s(\rho_c, \rho_s)L_d(\rho_d)}_{\text{I/Os satisfied by the disk}}\end{aligned}$$

where ρ_d is the allocated fraction of disk bandwidth (i.e. disk bandwidth quanta); and ρ_c, ρ_s are the buffer pool, and storage cache quota allocated to the application.

Black-box Constant Model Template (B-CNST) This is a very simple model which uses a simple average relation which returns the average value of all training samples to predict performance. The predicted latencies are the same for all configurations. In contrast, G-RGN uses average function for each region, and hence the prediction values are usually different in different regions.

Next, we will first show how Chorus builds and validates preliminary models dynamically, for synthetic applications. We then show how Chorus refines these preliminary models for new situations and builds an ensemble of models with high accuracy for TPC-C and TPC-W. We then show the benefits of using semantic guidance through Chorus for speeding up the sampling process in two cases of dynamic change: i) workload mix changes and ii) a hardware change. We psubsectionresent a use case for the models developed with Chorus for a resource allocation scenario in the Appendix.

```
TEMPLATE DISK
RELATION Inverse_Exponential(x,y) {
       x.name='disk_quanta' and
       y.name='memory_latency'

}
CONTEXT (a) {
      a.name='memory_size' and a.value='*'
      b.name='disk_bandwidth' and b.value='*'
}
```

Listing 4: DISK Model Template

5.1 Building a Preliminary Disk Model (DISK)

The analyst is aware that our storage server uses a quanta based scheduler to proportionally allocate the disk bandwidth among multiple applications. A larger fraction of the disk bandwidth allocated usually leads to the shorter delay of requests. Hence, the analyst provides the inverse model template based on inverse exponential, as in the existing G-INV model, as a standard mathematical relation automatically supported by the Chorus model template library.

The template for the DISK model is shown in Listing 4. This model is declared sensitive only to two parameters: the memory size allocated to the application and the disk bandwidth allocated to the application. The configuration range for which this model may apply is left unknown. Hence, the context in this model is left empty as "*". The Chorus run-time gathers experimental samples, validates, and curve-fits the model, finds the configuration settings where the relation applies, if any, and computes confidence scores and error rates.

Assuming that we allocate the disk bandwidth in 32ms quanta slices, and the disk quanta range from the minimum 32ms quanta to the maximum 256ms quanta with 8 settings, and statistical measurements taken at each setting takes around 15 minutes, validating this model for all configurations of an application e.g., OLTP-A, takes Chorus approximately 120 minutes.

5.2 Using the DISK model in a Multi-Tier Model (A-STOR)

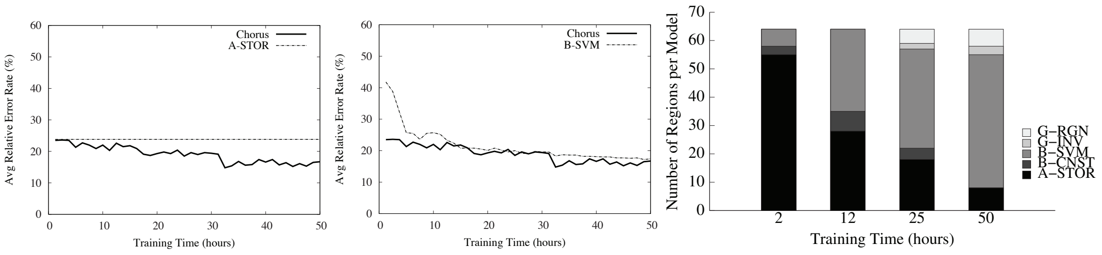
After validating simpler models, such as DISK and G-INV, the analyst may ask Chorus to validate the multi-tier analytical model template A-STOR. Given that this is an analytical model with few additional parameters to fit, its training time is minimal. Since the disk latency is a component of the A-STOR formula, the previously derived DISK model can be reused to improve the accuracy and speed of modeling A-STOR. In this case, the analyst may know that the disk latency curves demonstrate inverse exponential shapes, with different parameters for different applications. Chorus will refit the parameters of the DISK model for each new application under consideration. The more generic, G-INV model, which indicates an inverse exponential relation can also be provided as a template for approximating either Disk-bound or CPU-bound portions of the overall configuration space, as we show next.

5.3 Model Reuse for Predicting Memory Access Latency

In this section, we evaluate Chorus for modeling memory access latency (i.e. average buffer pool page access latency), for TPC-W^{10} and TPC-C workloads, running on our server platform. In this example, based on preliminary training, as described above, Chorus has already accumulated the following models in its model knowledge base: G-INV and A-STOR. The analyst provides new models for validation and fitting: G-RGN, B-SVM, B-CNST. We show how Chorus refits previous models from its model archive and validates new model templates, without any further sysadmin guidance to model the memory latency of the two new workloads.

The configuration space of resources is designed as follows. We vary the size of the DBMS buffer pool from 128M to 960M with 15 settings, the storage cache size from 128M to 960M with 8 settings for TPC-W^{10}, and 10 settings for TPC-C. We allocate the disk bandwidth in 32ms quanta slices, and the disk quanta range from the minimum 32ms quanta to the maximum 256ms quanta with 8 settings. The training time of exhaustive sampling is about 10 days for 960 configurations of TPC-W^{10}, and 13 days for 1200 configurations of TPC-C. The size of the testing set is 10% of the original set size. The number of regions we divide on each resource dimension is 4, hence the whole configuration space is divided into 64 regions in our ensemble algorithm.

Modeling TPC-W^{10} Memory Access Latency: Figure 5 presents our resutls. On the *x*-axis, we



(a) A-STOR (b) B-SVM (c) 'Voices' of Individual Models in Chorus.

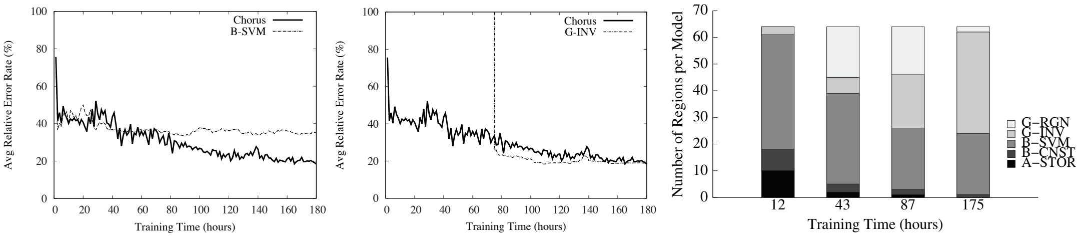
Figure 5: **Performance prediction of memory access latency for** TPC-W^{10} **workload.** Chorus initially matches the A-STOR model, and then incorporates the better predictions of B-SVM model.

show the training time and the *y*-axis shows the average relative error between the predicted and the measured performance for the testing set. For clarity, we show a trend curve comparison only between Chorus and the best individual models. We also show how the accuracy of models changes over time in the ensemble model. With sufficient insight into the system, the A-STOR performs well with a constant error of 23%, shown in Figure 5(a). The black-box and gray-box models perform poorly initially but improve over time. Specifically, B-SVM initially has lower accuracy than the A-STOR model, but gradually improves to lower errors (below 20%). Other models, B-CNST, G-RGN and G-INV, while improving over time, perform poorly compared to A-STOR and B-SVM in most of the configuration space. As shown in Figure 5(a) and 5(b), *Chorus* performs well matching the A-STOR model initially then incorporating the better predictions of B-SVM with more training time. The results are further supported by Figure 5(c) that shows that the A-STOR model is selected to predict performance for most regions initially, and then it contributes less to the *Chorus* over time; the B-SVM model replaces the A-STOR over time. Specifically, after 50 hours of training time, the B-SVM model is selected as the best model in 50/64 regions of the configuration space. We can see again that no individual model always wins for all time deadlines and regions.

Modeling TPC-C **Memory Access Latency:** Figure 6 shows our results. The black-box B-SVM model and the gray-box model G-INV contribute the most towards *Chorus*. B-SVM performs with very high error (above 40%) for over 45 hours, then performs better with more training time until reaching about 30% error rate. G-INV takes a longer training time (more than 80 hours), and after sufficient training data, it is able to predict well. In fact, it has the lowest average error rates (about 20%) of any base model in the end. G-RGN has a similar trend as B-SVM, albeit with slightly higher average errors. On the other hand, the A-STOR and B-CNST models perform worse. Shown in Figure 6(a) and 6(b), *Chorus* combines the positives of G-INV and B-SVM to achieve better performance. Specifically, it has a 20% prediction error (compared to 30% of B-SVM). In addition, by dynamically ranking the models, *Chorus* performs better than G-INV for a long time (80 hours), and then matches the performance of the fully trained G-INV. The number of regions contributed by each base performance model is shown in Figure 6(c). It shows that initially the B-SVM is selected to predict performance for most of regions around 12 hours, and then it contributes less over time. While the G-INV is rarely selected initially, it becomes more frequently selected after a sufficiently long training time i.e., after 175 hours. The remaining regions are predicted using the G-RGN. We can see that the model composition is different from modeling TPC-W^{10}, and also that there is no single model that works best for all time deadlines.

5.4 Model Extension for Workload Mix and Hardware Changes

Chorus can reuse and extend trained historical models that are saved in Chorus repository when the hardware changes and workload mix changes as we explain next.



(a) B-SVM (b) G-INV (c) 'Voices' of Individual Models in Chorus.

Figure 6: **Performance prediction of memory access latency for** TPC-C **workload.** Chorus initially relies on B-SVM model and later frequently selects gray-box models G-INV for prediction.

Workload Mix Changes

We conduct a series of experiments to show the impact of reusing historical models when the workload mix changes, using OLTP-A benchmark. Specifically, initially a model trained for the workload with the write ratio of a has been saved into the Chorus model repository, and we refer this workload as the historical or old workload. Then the write ratio of the data accesses decreases to b, and we refer this workload as the target or new workload, which we want to model. The Euclidean distance which reflects the similarity between the new and the old workload is denoted by variable $dist$, which equals $a - b$. We set the target error rate requirement for the modeling as 10%, and report the time of modeling for this new workload identified by the write ratio b.

Figure 8 shows the results. The x axis lists a series of tests with the same $dist$. The value of x axis refers to the write ratio b in the new workload. y axis shows the modeling time for the new workload with model reuse normalized to the time without reusing the historical model. That is, the modeling time is 1 if we train the new workload completely from the scratch. Each column corresponds one test. For example, first column of Figure 7(a) shows that when we train a read-only workload from the old workload with 10% write ratio, and the normalized modeling time is reduced to about 0.32.

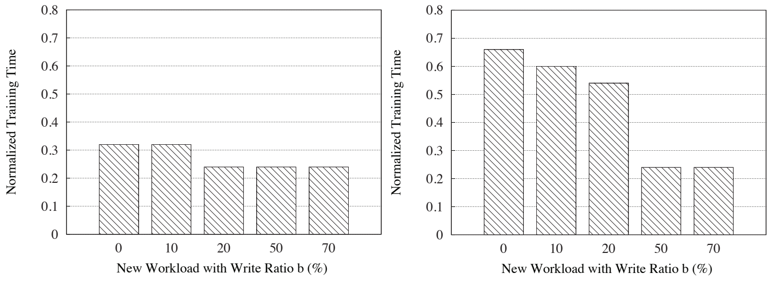
When the similarity distance $dist$ is 0.1, shown in Figure 7(a), the modeling time is significantly shorter (in the range of 0.2∼0.35) than the case without the model reuse. We further increase the distance $dist$ to 0.2, shown in Figure 7(b), the modeling time for the new workload are longer than previous tests with shorter distance (i.e. 0.1) due to the larger difference between the old and the new workloads, but overall the reduction of the modeling time are still significant, in the range of 0.2∼0.7.

Hardware Component Upgrades

In this section, we show an example where Chorus reuses a model previously derived for application latency using a hard disk drive (DISK) for the case of a disk replacement with an SSD device. We use the storage benchmark Fio [5] with random I/O to generate the workloads.

As shown in Figure 8(a), the archived models for the hard disk show that the I/O latency is linear with the increase of the number of outstanding I/O (i.e. I/O depth) under various read ratios. Upon the hardware change, Chorus verifies whether the known linear models for disk fit for the SSD case; towards this it samples for a given test configuration (50% read ratio in our example), and determines that the model does indeed fit. This allows Chorus to dramatically reduce further sampling. In our example, to model the curves with 0%, 25%, 75%, 100% read ratio, Chorus needs to take only 2 samples to fit each curve. The new SSD models are shown in Figure 8(b). Compared with exhaustive sampling from scratch upon disk replacement, the time reduction using Chorus is **53%**. We verified that the average error rate for new SSD models of these four curves is **4%**.

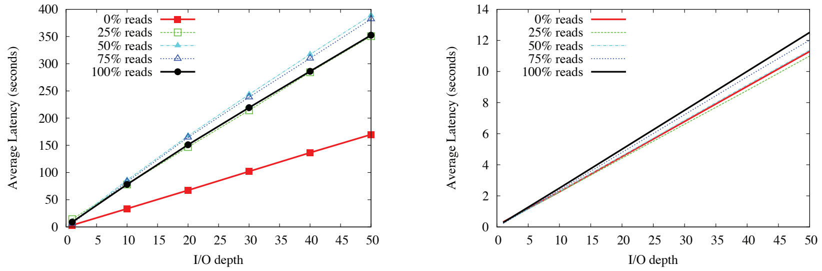
The verification phase is important, since not every archived HDD model for this application can work to model the performance of the new SSD device. For example, in the case of an attempted reuse of a model that shows the impact of the I/O randomness degree on latency, the Chorus verifi-



(a) similarity distance is 0.1

(b) similarity distance is 0.2

Figure 7: **Examples of model reuse when workload mix changes.**



(a) Hard disk models

(b) SSD models

Figure 8: **Examples of model reuse when hardware changes.**

cation phase shows that the model derived for the hard drive does not fit the SSD case. The mix of random and sequential I/O in the workload has an impact for the HDD, but not for the SDD. A new model needs to be built from scratch for the SSD for modeling this metric correlation.

6 Related Work

In this section, we discuss previous work closely related to our own in the areas of performance modeling. Existing techniques for predicting performance range from analytical models [2, 13, 17, 18], to black-box models based on machine learning algorithms [6, 20, 22]. There are also previous gray-box models in related studies [15]. Furthermore, Zhang et al. use a regression-based analytical model for capacity planning of multi-tier applications [21]. Compared to these existing systems, Chorus seamlessly combines analytical performance models that have been derived for specialized systems with generic types of models.

Building complete analytical models requires an in-depth understanding of the underlying system, however, which may not always be possible in multi-tier server systems. As an example of advanced analytical models for specialized cases: Uysal et al. derive an analytical throughput model for modern disk arrays [18] and queuing models [2, 17] have been explored for CPU-bound web servers. Soror et al. [13] use query optimizer cost estimates to determine the initial allocation of CPU resources to virtual machines and to detect changes in the characteristics of the workloads. Various query monitoring techniques [9, 8] estimate query latency based on detailed statistics information of queries, such as query plan, cardinality, number of groups, for current resource configuration.

With the complexity of modern systems, machine-learning based approaches have been explored to model these systems. These previous works either target providing a best model for some specific type of system or workload, or provide fully automated modeling/sampling methods. For example, Wang et al. use a machine learning model, CART, to predict performance for storage device [20]. Ganapathi et al. predict DBMS performance by using a machine learning algorithm [6] called KCCA. Zhang et al. [22] discuss how to use ensemble of a group of probability models for automated diagnosis of system performance problems. IRONModel uses a hybrid decision-tree based machine learning model, called Z-CART, to predict the parameters for analytical models designed for their storage system [15]. iTuned [4] proposes an adaptive sampling method which automatically selects experimental samples guided by utility functions. Ghanbari et al. propose to use a query language for evaluating system behavior [7].

While our approach has some similarities with these approaches, we add the capability to incorporate semantic information, and high-level guiding in a human intelligible declarative language to our automated runtime system. We focus on developing a run-time system providing intelligent sampling and interactive model refinement based on sysadmin guidance and historical information.

7 Conclusion

We design, implement and deploy Chorus, a novel interactive runtime system for incremental, on-the-fly performance modeling of datacenter applications. We notice that patterns of long running application classes are repeatable and hardware upgrades are sometimes localized to a few components. Our key idea is to provide an interactive high-level environment geared towards reuse and refinement of old models in new data center situations.

With Chorus, both the modeling system and its administrator can communicate semantically meaningful information about the overall application model and its parts. Specifically, the sysadmin provides model templates and sampling guidelines through a high level declarative language. Chorus validates these model templates using monitored historical data, or live sampling for the resources the model is sensitive to; thus Chorus incrementally constructs an ensemble of models semantically tagged for the purposes of inquiry or adaptation to new situations. Through a set of case studies, we show that Chorus can successfully validate, extend and reuse existing models under new situations. Furthermore, we show that these models are accurate enough for on-line resource management purposes.

References

[1] Amazon Relational Database Service (Amazon RDS). http://aws.amazon.com/rds/.

[2] M. N. Bennani and D. A. Menascé. Roesource Allocation for Autonomic Data Centers using Analytic Performance Models. In *Proceedings of the 2nd International Conference on Autonomic Computing (ICAC'05)*, pages 229–240, 2005.

[3] H. Drucker, C. J. C. Burges, L. Kaufman, A. J. Smola, and V. Vapnik. Support Vector Regression Machines. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'96)*, pages 155–161, 1996.

[4] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. *Proceedings of the VLDB Endowment*, 2:1246–1257, 2009.

[5] Fio storage benchmark. http://freecode.com/projects/fio.

[6] A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. A. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *Proceedings of the 25th International Conference on Data Engineering (ICDE'09)*, pages 592–603, 2009.

[7] S. Ghanbari, G. Soundararajan, and C. Amza. A Query Language and Runtime Tool for Evaluating Behavior of Multi-tier Servers. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'10)*, pages 131–142, 2010.

[8] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke. Increasing the Accuracy and Coverage of SQL Progress Indicators. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 853–864, 2005.

[9] C. Mishra and N. Koudas. The design of a query monitoring system. *ACM Trans. Database Syst.*, 34(1), 2009.

[10] 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)*, pages 237–250, 2010.

[11] ORION - Oracle I/O Calibration Tool. http://www.oracle.com.

[12] F. Raab. TPC-C - The Standard Benchmark for Online transaction Processing (OLTP). In *The Benchmark Handbook*. Transaction Processing Council, 1993.

[13] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst.*, 35(1), 2010.

[14] G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza. Dynamic Resource Allocation for Database Servers Running on Virtual Storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, pages 71–84, 2009.

[15] E. Thereska and G. R. Ganger. Ironmodel: Robust Performance Models in the Wild. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'08)*, pages 253–264, 2008.

[16] Transaction Processing Council. http://www.tpc.org.

[17] B. Urgaonkar, G. Pacifici, P. J. Shenoy, M. Spreitzer, and A. N. Tantawi. An Aonalytical Model for Multi-tier Internet Services and Its Applications. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, pages 291–302, 2005.

[18] M. Uysal, G. A. Alvarez, and A. Merchant. In *Proceedings of the 9th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'01)*.

[19] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, pages 61–76, 2007.

[20] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with CART models. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'04)*, pages 412–413, 2004.

[21] Q. Zhang, L. Cherkasova, N. Mi, and E. Smirni. A Regression-based Analytic Model for Capacity Planning of Multi-tier Applications. *Cluster Computing*, 11(3):197–211, 2008.

[22] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of Models for Automated Diagnosis of System Performance Problems. In *International Conference on Dependable Systems and Networks (DSN'05)*, pages 644–653, 2005.