CHORUS: MODEL KNOWLEDGE BASE FOR PERFORMANCE MODELING IN DATACENTERS

by

Jin Chen

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2011 by Jin Chen
Abstract

Chorus: Model Knowledge Base for Performance Modeling in Datacenters

Jin Chen
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2011

Due to the imperative need to reduce the management costs, operators multiplex several concurrent applications in large datacenters. However, uncontrolled resource sharing between co-hosted applications often results in performance degradation problems, thus creating violations of service level agreements (SLAs) for service providers. Therefore, in order to meet per-application SLAs, per-application performance modeling for dynamic resource allocation in shared resource environments has recently become promising.

We introduce Chorus, an interactive performance modeling framework for building application performance models incrementally and on the fly. It can be used to support complex, multi-tier resource allocation, and/or what-if performance inquiry in modern datacenters, such as Clouds. Chorus consists of (i) a declarative high-level language for providing semantic model guidelines, such as model templates, model functions, or sampling guidelines, from a sysadmin or a performance analyst, as model approximations to be learned or refined experimentally, (ii) a runtime engine for iteratively collecting experimental performance samples, validating and refining performance models. Chorus efficiently builds accurate models online, reuses and adjusts archival models over time, and combines them into an ensemble of models. We perform an experimental evaluation on a multi-tier server platform, using several industry-standard benchmarks. Our results show that Chorus is a flexible modeling framework and knowledge base for validating, extending and reusing existing models while adapting to new situations.
This thesis is dedicated to my parents for their belief in me.
Acknowledgements

I would like to first acknowledge my advisor Professor Cristiana Amza for her insightful guidance and consistent support during my study. She has always encouraged me to explore the research ideas in depth, and has patiently helped me to improve my research skills step by step. She has also shared her experience both in research and in life with me. I am inspired by her vision, courage, confidence and persistence when I face challenges. I also want to sincerely thank other members of my research committee, Professor Hans-Arno Jacobsen, Professor Nick Koudas and Professor Baochun Li, for their invaluable comments and continuous support. I especially thank Professor Li for offering me several TA positions, and I quite enjoyed the pizza luncheon and chat during the exam marking. I greatly thank Professor Bettina Kemme as my external examiner; her appraisal benefited me a lot toward improving my thesis writing. I also greatly thank Professor Vaughn Betz for agreeing to chair my defense, at the last minute.

I want to deeply thank my collaborators, Gokul Soundararajan, Saeed Ghanbari, Madalin Mihailescu, Adrian Popescu, Daniel Lupei, Bogdan Simion and Mohamed Sharaf for their constant help in research. In addition to academic knowledge, I learned diligence and persistence from Gokul, learned to be calm and handy from Madalin, and learned philosophy from Saeed.

I thank my family for their emotional support and love. I especially thank my husband for his sacrifice, tolerance, understanding during these years. He helped and motivated me to balance research and life. I also thank my parents for their belief in me. They have always stood by my side, and supported me. I particularly thank my sister for helping me to analyze things from a comprehensive view. I thank my daughter Anya for her every smile and hug.

Finally, I also greatly thank my schoolmates, Naiqi Weng, Yilan Gu, Guoli Li, Xin Gu, Weijing Ma, Chuan Wu, Mingxia Xue, Jingrui Zhang, Zheng Li, Chuck Zhao, Zhengdao Xu, Weihan Wang, Yun Liu, Ou Wei, Youmei Liu, Yuan Gan, Xiaojun Bi, Sheng Ma, Mihai Burcea, Kaloian Manassiev, Livio Soares, Ioana Burcea, Alex Depoutovitch, David Tam, Reza Azimi, Adam Czajkowski, Don Pinto, James Huang. I especially thank Rui Yan and Lin Mei for our support of each other during our study. Friendship makes life full of warmth and joy.
## Contents

1 Introduction  
  1.1 Motivation ........................................ 2  
  1.2 Problem Statement .................................. 4  
  1.3 Contributions ...................................... 7  
  1.4 Organization ...................................... 10  

2 Background  
  2.1 Dynamic Resource Provisioning in Shared Datacenters .......... 11  
  2.2 Architectural Background ................................ 14  
    2.2.1 Resource Allocators within the Database Server .......... 14  
    2.2.2 Resource Allocators within the Storage Server .......... 15  

3 Coarse Grained Resource Allocation for Backend Databases  
  3.1 Introduction ........................................ 17  
  3.2 Background .......................................... 21  
  3.3 System Architecture ................................... 22  
    3.3.1 Architecture of the Dynamic Content Server ........... 22  
    3.3.2 Dynamic Replication ................................ 24  
  3.4 Reactive Replica Provisioning ............................ 28  
    3.4.1 Reactive Replica Addition ........................... 28  
    3.4.2 Reactive Replica Removal ........................... 29
3.4.3 Enhancement of Reactive Approach with System Instability Detection 30
3.5 Proactive Replication Provisioning 30
  3.5.1 Overview 30
  3.5.2 Enhancement of Proactive Approach with System Instability Detection 32
  3.5.3 Implementation Details 33
3.6 Experimental Setup 35
  3.6.1 TPC-W E-Commerce Benchmark 35
  3.6.2 Client Emulator 36
3.7 Experimental Results 37
  3.7.1 System Training 37
  3.7.2 Proactive Approach without Stability Awareness 37
  3.7.3 Performance Comparison of the Proactive and Reactive Approaches 40
  3.7.4 Robustness of the Proactive Approach 44
3.8 Summary 48

4 Fine Grained Resource Partitioning of the Cache Hierarchy 50
  4.1 Introduction 50
  4.2 Background 53
  4.3 Cache Partitioning Algorithm 58
    4.3.1 Problem Statement 59
    4.3.2 Overview of Approach 60
    4.3.3 Performance Models 62
    4.3.4 Utility-Aware Iterative Learning Algorithm 62
    4.3.5 Online Adaptation to Dynamic Changes 67
  4.4 Prototype Implementation 67
    4.4.1 Virtual Storage System 69
    4.4.2 Gemini Simulator 70
  4.5 Evaluation 71
5.4.2 Server Platform ................................................. 101
5.4.3 Sampling Methodology ................................. 102
5.5 Cast Studies of Model Templates .................. 103
  5.5.1 Analytical Model Template (A-STOR) for Memory Latency: 103
  5.5.2 I/O Intensive Query Model Template (A-STOR-Q) .......... 107
  5.5.3 Gray-box Inverse Model Template (G-INV) ............... 107
  5.5.4 Gray-box Region Model Template (G-RGN) .......... 107
  5.5.5 Black-box SVM Regression Model Template (B-SVM) .... 108
  5.5.6 Black-box Constant Model Template (B-CNST) .... 108
5.6 Results .......................................................... 108
  5.6.1 Build Models for Predicting Memory Access Latency 109
  5.6.2 Build Models for Predicting Query Latency ............ 113
  5.6.3 Using a Complex Pruning Clue based on Expert Knowledge 115
  5.6.4 Model Extension and Reuse ............................ 118
  5.6.5 Examples of Resource Allocation Using Chorus ....... 118
5.7 Summary ...................................................... 122

6 Related Work ................................................. 123
  6.1 Server Platforms for Dynamic Resource Allocation ........ 123
    6.1.1 Shared Server Pool Platform for Web Applications .... 123
    6.1.2 Replicated Database Cluster ............................ 124
    6.1.3 Fine Grained Resource Multiplexing Platform .......... 126
  6.2 Algorithms and Models for Dynamic Resource Allocation 129
    6.2.1 Analytical Model based Resource Allocation ............ 130
    6.2.2 Black-box Model based Resource Allocation .......... 132
    6.2.3 Hybrid Approaches based Resource Allocation ....... 133
  6.3 Semantic Languages for Performance Analysis ......... 134
7 Conclusion and Future Work

7.1 Conclusion ................................................................. 135

7.2 Future Work .............................................................. 137

Bibliography .................................................................... 139
List of Tables

3.1 An Illustration of Instability .................................................. 38

5.1 Applying Cache Pruning Clue on TPC-W$^{10}$ .......................... 117

5.2 Applying Cache Pruning Clue on TPC-C ................................. 117
List of Figures

1.1 Server consolidation in a multi-tier server environment. We show a typical
datacenter architecture using consolidated storage. ................................. 2
1.2 RUBiS latency surface. ................................................................. 3

3.1 Architecture of dynamic content sites ............................................ 18
3.2 Cluster architecture ................................................................. 23
3.3 Latency instability during replica addition. .................................... 26
3.4 Reactive replication provisioning scheme ...................................... 29
3.5 Proactive replication provisioning scheme .................................... 31
3.6 Sine load function ................................................................. 38
3.7 Provisioning results without stability awareness under a sine load function ... 39
3.8 Provisioning results under a sine load function .............................. 41
3.9 Sudden load spike function .......................................................... 42
3.10 Provisioning results under a sudden load spike function ............. 43
3.11 A browsing workload scenario under a sine load function ............. 45
3.12 Machine allocation results under a step load function .................. 46
3.13 Latency results under a step load function ................................ 47

4.1 Datacenter architecture with shared DBMS and shared storage ........... 54
4.2 We experiment with two cache configurations: LRU/LRU and LRU/DEMOT
...... The results show significant room for improvement. .......................... 56
4.3 Utility functions ................................................................. 66
4.4 Gemini storage architecture: We show the MySQL database server (as a
storage server client) connected to a storage server using NBD protocol. .... 68
4.5 Latency surfaces: Memory access latency for different partitionings of buffer
pool and storage cache. .......................................................... 74
4.6 LRU/LRU ........................................................................... 76
4.7 LRU/DEMOTE ................................................................. 78
4.8 Overload: We show the total revenue for TPC-W/TPC-W for several config-
urations with the light regions showing “high” revenue and the dark regions
showing “low” revenue. We also highlight the optimal cache partitioning set-
tings. .................................................................................. 81
4.9 Comparison of sampling methods ........................................ 83
5.1 Different operating modes of an application. The application varies from I/O-
-intensive (in gray and in top left corner) and CPU-intensive (in black and in
bottom right corner) as the amount of memory and disk resources are varied.
The area in the middle (in white) is the mixed mode. ......................... 88
5.2 Our server platform. It consists of a modified MySQL database server (shown
in left) and a virtual storage prototype Akash (shown in right). ............ 101
5.3 Performance prediction of memory access latency for TPC-W \textsuperscript{10} workload.
Chorus initially matches the A-STOR model, and then incorporates the better
predictions of B-SVM model. ......................................................... 110
5.4 Performance prediction of memory access latency for TPC-C workload. Cho-
rus initially replies on B-SVM model and later frequently selects gray-box
models G-INV and G-RGN for prediction. ..................................... 112
5.5 Performance prediction of query latency for TPC-W \textsuperscript{10} workload. Chorus
mainly matches the prediction of B-SVM model, and later incorporates the
better predictions of G-INV model. ............................................ 114
5.6 Performance prediction of query latency for TPC-C workload. Chorus initially matches the prediction of B-SVM model, and soon incorporates the better predictions of G-RGN model.

5.7 Examples of model reuse using OLTP-A benchmark.

5.8 Examples of resource allocation for running four identical instances.

5.9 Examples of resource allocation for running two TPC-W\textsuperscript{10} Instances and two RUBiS\textsuperscript{10} instances.
Chapter 1

Introduction

Large datacenters, e.g., Amazon Relational Database Service (RDS) datacenter [1], Google App Engine datacenter [38], are highly dynamic environments, co-hosting several applications sharing resources in a multi-tier server environment, as shown in Figure 1.1. Uncontrolled resource sharing between co-hosted applications often results in performance degradation problems, thus creating violations of service level agreements (SLAs) for service providers. Therefore, per-application performance modeling for on the fly adjustment of resource allocation to match per-application SLAs has recently become promising.

With this dissertation, we introduce Chorus, a novel performance modeling framework for dynamic resource allocation and what-if performance inquiry in complex, service-hosting datacenter environments. With the proliferation of Cloud environments, interest in this research area has recently increased from industry and academia alike. Specifically, several recent projects [34, 35, 39, 63, 91, 93, 106] have been very active in the area of performance modeling for datacenters, concurrently with our own [36, 83, 84].

A performance model is a mathematical function that calculates an estimate of the application performance for a range of resource configurations. For example, Figure 1.2 shows a performance model as a 3D surface for the online auctions application RUBiS. The model provides an estimate of the average memory access latency (i.e. average page access latency
Figure 1.1: Server consolidation in a multi-tier server environment. We show a typical datacenter architecture using consolidated storage.

Measured at the database buffer pool) of a MySQL database engine running RUBiS, for all possible memory quotas in a two-level memory hierarchy, consisting of buffer pool and storage cache. Similarly, the average query latency of an application varies as a function of its resource quotas for CPU, buffer pool, storage cache, and disk bandwidth of that application.

As we will show in this dissertation, performance modeling of applications is desirable for automating resource allocation to applications in large datacenters. Specifically, performance models can predict the latency at various sub-systems, or end-to-end, for given resource configuration regions. Hence, performance models can be used for dynamic resource provisioning to applications.

1.1 Motivation

Traditionally, resource provisioning or capacity planning have relied on offline extensive measurements, and/or sysadmin or analyst expertise. However, long-term resource over provisioning for all possible combinations of peak incidental loads is unacceptable in a dynamic environment, due to large cooling and power costs. Moreover, while profiling and monitoring
tools [66, 86] exist for multi-tier server environments, as the example shown in Figure 1.1, the sysadmin may need to extensively profile the application and decompose its end-to-end SLA into per-component SLAs, and bottleneck cases. This is hard, because of inter-dependencies between 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.

Automatic performance model building iterates 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, experi-
mental 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, for instance, cache warm-up time, which make reliable actuation and sampling expensive even for a single configuration point.

For example, for \( N \) modeled resources, and \( M \) increments of sampling for each resource, an application surface model would be an \( N + 1 \)-dimensional hyperplane with \( O(M^N) \) sample points. For our RUBiS example in Figure 1.2, due to the cache warmup effect, 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. It follows that extensive experimental sampling and building fully automated, black-box performance models based on experimental sampling on a live system is too time consuming.

At the other end of the spectrum is using analytical models that rely on sysadmin or analyst’s semantic knowledge of the system and application [39, 84, 93]. However, these analytical models are precise only for restricted parts of the system, specific application workload mix or resource configurations. They are brittle to dynamic changes and require too much domain expertise.

## 1.2 Problem Statement

Our key insight in this dissertation is:

*We propose to leverage automated black-box long-term learning of the system itself, coupled with administrator semantic awareness and expertise, wherever available to find a satisfactory accuracy-time modeling trade-off in most situations.*

We observe that calculating a number of potential models, or fitting known models over the collected sampling data is relatively fast compared to actuating and sampling. We also
observe that each long-running application deployed on the datacenter may experience many
dynamic changes during its deployment. There may be changes in resource availability, due to
dynamic co-hosting of other applications, changes in the load intensity for any given applica-
tion, changes in any application’s own workload mix, changes in the infrastructure itself, due to
component replacements or upgrades. On the other hand, common application characteristics,
infrastructure structural information, or resource availability situations typically show stable, or
repeatable, recognizable patterns over time. For example, the most prevalent workload mixes
of Web applications may show a daily, weekly or seasonal repetition pattern [95]. Furthermore,
most Web applications exhibit non-sequential, but repeatable disk access patterns [85]. Most
caches throughout the system use some variant of LRU cache replacement. Application data
flows through the predefined components of the multi-tier server system. Many other structural
and functional laws of the underlying infrastructure, e.g., Little’s law [42], all stay the same
throughout dynamic changes.

Chorus, our highly flexible performance modeling framework, uses these observations to
provide lightweight sampling and modeling on the fly as well as historic accumulation of
known repeatable patterns. At any given point in time, Chorus provides performance predic-
tions within the application’s current operating zone, i.e., workload mix and resource availabil-
ity. In addition, Chorus allows (i) semantic specification of guidelines for selective, focused,
intelligent sampling for faster model building, (ii) runtime support for validation, management
and refinement of approximate models, and (iii) a knowledge base for storage, and retrieval of
a set of evolving models over time.

In more detail, Chorus receives semantic model guidelines, such as model templates,
model functions, or sampling guidelines, from a sysadmin or a performance analyst, as model
approximations to be learned or refined experimentally. We extend a high-level SQL-like
declarative language, called SelfTalk [36], to express model templates. Model templates can be
in the form of suspected functional relations between metrics, templates for curve fitting, se-
manic dependencies, composable, or nesting of models. SelfTalk allows fuzzy specification
of functional relations between metrics, in a format similar to: "Throughput varies linearly with the memory quota". The context for a functional relation can be specified, as in, "for configurations where the memory quota is greater than 500MB", or left undefined.

While the sysadmin guidance does not need to be accurate, in terms of context of its applicability, or even as functional relation, the more accurate and focused the guidance received, the more selective and focused the experimental sampling in Chorus, the faster the model building.

The Chorus runtime engine allows for model template validation with experimental data, and calibration of parameters for specified configuration ranges. Chorus can also automatically find configuration ranges where a semantic model template fits its sampling data; it further automatically ranks models by accuracy per configuration region and selects the best model per region. In new situations, and wherever model templates are not available, Chorus gathers experimental samples and uses black-box statistical regression to derive models.

In the long term, the Chorus knowledge base accumulates approximate models for different applications and resource configurations, as they dynamically present themselves. For this purpose, Chorus provides tools for storing, retrieving, extending, and refining old models with new samples, and builds an ensemble of models with increasing accuracy over time. Finally, Chorus can answer sysadmin inquiries about any model, resource configuration region and what-if scenario, in a semantically meaningful way.

For the past six years, we have been the designers, developers and system administrators of a datacenter running Chorus and several industry-standard DBMS benchmarks, at University of Toronto. We have been actively using the performance monitoring data and the performance models now stored in Chorus for a variety of purposes, from performance anomaly detection, to resource allocation, to interactive what-if inquiry into resource configurations, to deepening our understanding of our infrastructure and applications.

In this dissertation, 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. In this context, our performance modeling techniques that are now stored in the Chorus knowl-
edge base, and their applications have been evolved in several directions: (i) from offline experimental models [22] to online experimental models [37, 83, 84], (ii) from using performance models to support allocating coarse-grained resources [22, 25, 37], such as server boxes to allocating fine-grained resources [23, 24, 83, 84], e.g., memory, and storage bandwidth, (iii) from online models for resource allocation in one tier [22, 37], i.e., the database tier, to online models for resource allocation in two tiers [23, 83, 84], (iv) from individual performance models of individual components [22, 37], to an ensemble of models combining several per-component models and the causal/structural relationships between them [23].

We briefly summarize our most important contributions along these directions that form the basis for this dissertation.

1.3 Contributions

In this dissertation work, we show that dynamically built performance models can be used to support complex, multi-tier resource allocation, and/or what-if performance inquiry in modern database environments, such as Clouds.

Specifically, we present two case studies for applying performance modeling to dynamic resource allocation in datacenters. We first showcase performance modeling on coarse grained resource allocation for backend databases, and then on fine grained resource partitioning of the cache hierarchy. Both case studies show that model based approaches are successful for resource allocation in datacenters.

Finally, we present our interactive runtime framework, Chorus, for building performance models dynamically, incrementally, on the fly. We also extend an existing semantic language, SelfTalk, to allow sysadmins to express their domain knowledge to Chorus; Chorus can leverage these knowledge to guide and speed up its sampling process. We showcase the refinement process for a model including all resources at the database and storage servers and its use in dynamic resource allocation.
In more detail, our contributions are as follows:

1. **Model-based coarse grained resource allocation for backend database replicas.** We study the problem of proactive resource allocation for replicated backend databases. We propose a novel proactive provisioning scheme based on an offline performance model using statistical machine learning. The performance model allows for adding or removing database replicas dynamically to application allocations. Our model uses lightweight monitoring of essential system and application metrics in order to predict how many databases it should allocate to a given workload. Our proactive algorithm incorporates awareness of system stabilization periods after adaptation in order to improve prediction accuracy and avoid system oscillations. We compare this proactive self-configuring scheme for scaling the database tier with a reactive scheme. Our experiments using the industry-standard TPC-W e-commerce benchmark demonstrate that the proactive scheme is effective in reducing both the frequency and peak level of SLA violations compared to the reactive scheme. Furthermore, by augmenting the proactive approach with awareness and tracking of system stabilization periods induced by adaptation in our replicated system, we effectively avoid oscillations in resource allocation.

2. **Model-based fine grained resource partitioning of the cache hierarchy.** We design and implement a novel coordinated partitioning technique of the database buffer pool and storage cache between applications for any given cache replacement policy and per-application access pattern. We use statistical regression to dynamically determine the mapping between cache quota settings and the resulting per application SLA. An important refinement in this space is to exploit approximation through analytical models, or simulation, wherever applicable, e.g., for the storage cache hierarchy resource [83, 84], to speed up convergence towards the optimal resource allocation solution. Simulation can provide a set of sample points for building an approximate model with sufficient accuracy. We can then selectively refine the model experimentally, while dynamically
actuating resource quotas to applications. A resource controller embedded within the
database engine actuates the partitioning of the two-level cache, converging towards the
configuration with maximum application utility, expressed as the service provider rev-
"nue in that configuration, based on a set of latency sample points. Our experimental
evaluation, using the MySQL database engine, a server farm with consolidated storage,
and the TPC-W e-commerce benchmark, and the RUBiS online bidding benchmark. We
show that our coordinated dynamic partitioning technique provides compliance with the
SLA requirement of applications with strict SLAs, while at the same time maintaining ef-
cient resource usage. As a result, our dynamic cache partitioning technique minimizes
penalties in overload and maximizes the revenue of the service provider in underload.

3. Chorus: model knowledge base for resource allocation. We introduce Chorus, an
interactive runtime framework for building performance models incrementally, on the
fly. Chorus builds efficient, accurate models dynamically, reuses and adjusts models
over time, and combines them into an ensemble of models. We perform an experimen-
tal evaluation on several industry-standard benchmarks (TPC-W, TPC-C and OLTP-A)
running on a multi-tier server platform. In our evaluation, we present the case studies of
expressing model templates, and demonstrate several model building examples for pre-
dicting application latency. We also show examples of model evolution where Chorus
seamlessly reuses existing approximate models, and incrementally refines them on the
fly. Specifically, our experimental evaluation shows that Chorus (i) can integrate the ex-
pertise of a system administrator, an analyst, or other statistical, and historical knowledge
base as approximate model guidance, (ii) dynamically selects the most accurate model-
ing techniques for different configuration regions, (iii) matches or even outperforms the
accuracy of the best model per configuration region and simpler modeling approaches,
(iv) incrementally adjusts existing models on the fly in new resource configurations and
workload mix situations, and (v) provides accurate, fast and flexible support for building
and refining model-ensembles for automatic resource allocation on our server platform.
Chapter 3 presents the paper that was published at the 3rd International Conference on Autonomic computing (ICAC’06) [22]. I am the first author of this paper. I proposed and implemented the pro-active algorithm and conducted the experiments independently. Gokul Soundararajan helped me to be familiar with the reactive algorithm and test environments. Chapter 4 presents the paper that was published at the 34th International Conference on Very Large Data Bases (VLDB’08) Conference [83]. I am the second author of this paper. I worked together with Gokul Soundararajan to propose algorithms, conduct experiments and analyze the results. He implemented the main part of the storage server prototype, and I implemented some of functionalities, e.g. resizing, inside the cache module.

The framework of Chorus is described in Chapter 5. I proposed the core ideas of Chorus, implemented most of the Chorus codes, and conducted and analyzed most of experiments. Gokul Soundararajan worked together with me on the early development of Chorus, and continues to provide insightful feedback. He also contributed to propose an analytical performance model A-STOR for the storage system. I implemented the main part of the buffer pool allocator and CPU scheduling techniques in MySQL. Saeed Ghanbari worked with me to extend Self-Talk language for providing modeling guidelines. I also had many discussions with Madalin Mihailescu and Saeed Ghanbari on how to enhance Chorus to cope with the changes of the environments.

1.4 Organization

The outline of this dissertation is as follows. Chapter 2 introduces the background and motivation. Chapter 3 presents the coarse grained resource provisioning in the replicated database cluster. Chapter 4 presents the dynamic partitioning of the cache hierarchy in shared data centers. Chapter 5 presents our novel flexible modeling framework Chorus for resource allocation. Chapter 6 presents related work. Chapter 7 concludes the dissertation and outlines avenues for future work.
Chapter 2

Background

In this dissertation, we design, implement, and experimentally evaluate performance model based approaches for dynamic resource provisioning in large datacenters. We target to reduce the complexity of management, and increase the predictability of application performance in multi-tier server systems. This chapter highlights importance of the dynamic resource provisioning and discusses the challenges, and introduces the architectural background.

2.1 Dynamic Resource Provisioning in Shared Datacenters

Cost-efficiency is becoming one of the major design goals in large datacenters. The costs include the power and cooling expense associated with hardwares, as well as the management cost associated with human. However, resource utilization in large datacenters, e.g. Google datacenters, is usually low. Many statistical data show that, on average, servers spend relatively little time at high load levels; instead, most of the time is spent within 10% ~ 50% CPU utilization usage [10]. Low resource utilization leads to the waste of resource, and hence degrades the efficiency of the datacenter.

To improve resource utilization and reduce cost, server consolidation related techniques have been recognized as crucial methods for managing large datacenters. With them, several concurrent running applications are able to multiplex on shared resources. Virtual Machine
Monitors (VMMs), such as Xen [9] and VMware [97], virtualize the underlying hardware, and provide isolations among the co-located virtual machines. As another alternative, the use of virtualized software services have recently gained increased attention. Amazon Relational Database Service (Amazon RDS) [1] is an example. It provides user an illusion of running her applications on a dedicated database, while in fact her applications could be run on a shared database server with other users. Google App Engine [38], Microsoft Azure Services [60] also provide similar virtualized services.

On the other hand, multiplexing could lead to the performance interference among co-located applications. A large number of resource container or scheduler related mechanisms have been proposed for implementing fairness and priorities in some versions of modern OSes, e.g., in Linux [74, 98]. However, severe interference for resources can still occur, since it is almost impossible to partition all of shared resources that can influence the performance. For example, shared L2 cache of multicore architectures lead to significant performance degradation if multiple applications are running simultaneously [63]. Similarly, network bandwidth and disk bandwidth are usually not partitioned, and thus some aggressive applications may consume most of the bandwidth. As a result, two identical virtual machines may show significant different performance if they are running on different physical servers or located in different zones of a datacenter. Uncontrolled resource sharing thus creates a problem for service providers, because respecting application-level requirements (e.g. average latency bounds or through guarantees) are considered paramount in modern environments. In highly competitive markets, even slight performance degradations may cause serious financial consequences and impact customer trust [31].

Therefore, in state-of-the-art environments, much effort is expended on finessing the application performance versus resource usage trade-off, and investigating performance bottlenecks, usually in a manual, time-consuming fashion. A variety of low-level profiling tools to help highly skilled administrators exist for this purpose. However, in the common case, the administrator still needs to compile, and fine-tune a variety of metrics across the system, by
poring over system logs, or low-level statistics, which is error-prone, expensive in terms of the required level of skill, time consuming, or all three. The alternative solution is to sacrifice resource usage, by conservatively over-provisioning resources to accommodate overall expected peak usage for all applications. While new hardware-level power-saving mechanisms are increasingly being deployed to reduce energy consumption and cooling requirements of idle or partially used servers, neither manual fine-tuning, nor resource over-provisioning is a satisfactory solution in terms of cost-effectiveness.

The optimal way to partition these resources across competing workloads is a non-trivial task. This is challenging due to the following four reasons. First, application QoS is usually expressed as a service level agreement (SLA) or service level objective (SLO), not as per-resource quota. Since applications specify requirements as end-to-end performance goals; this requires the datacenter provider to consider resource allocation in the entire server stack, i.e., from the web server down to the storage. Secondly, we lack the necessary resource isolation infrastructure to solve the problem of uncontrolled sharing. Resource isolation in a single tier, e.g. CPU partitioning in application servers, is not sufficient since the performance of applications could be affected by the resources on the database server and storage server tiers as well. Hence, an end-to-end resource isolation infrastructure is required to ensure strong QoS guarantees for applications. The third reason is the absence of coordination between local resource controllers located within different tiers. This absence of coordination might lead to situations where local partitioning optima far deviate from the global optimum. Last but not the least, sysadmins usually lack the knowledge to understand various types of performance models that are used to allocate resources; and they hardly exploit their domain knowledge to speed up the process of the dynamic resource allocation. We will address these challenges in the following chapters.
2.2 Architectural Background

As shown in Figure 1.1, modern web application systems usually have multiple software layers. The front ends include web servers and application servers. The back ends include database servers and storage servers at the lowest level. In server consolidation environment, each of these software layers hosts multiple applications to reduce operation costs. Within these multi-tier software layers, we focus on studying the resource allocation algorithms for the back ends: database servers and storage system.

In this section, we introduce our end-to-end resource partitioning architecture for managing and allocating resources in database servers and storage servers. We modify MySQL database server and build our own storage server Akash to support resource partitioning. The implementation details are explained in the following subsections.

2.2.1 Resource Allocators within the Database Server

In our design, we allow a single database engine to host multiple applications. Hence, to minimize the interference among applications inside MySQL server, a CPU scheduler is implemented to schedule CPU time, and a database buffer pool allocator is implemented to dynamically adjust the size of the buffer pool partition for each workload.

Workload id: In order to partition resources for different applications, we need to identify them first. By default, queries to one database are considered as requests from one workload. To be flexible, we provide a tool for users to define their workload. Users can choose a set of query templates as their customized workload. We use workload id to identify and track each workload internally. The workload id information is kept in the THD object, which keeps the execution context of each client connection in MySQL. Performance related metrics, e.g., average query latency, throughput, miss ratio, are monitored for each workload separately.

CPU Allocator: We implement our round-robin time quantum scheduler through instrumenting MySQL/InnoDB storage engine’s thread scheduler. If a workload uses out its time
quanta, all threads of its client connections are paused and placed into the sleep queue until the next turn. Our scheduling unit is a group of basic operations, such as fetching a page from the index or updating a single row. The overhead of the scheduler mainly comes from the scheduling granularity. If the group size is set too small, the proportion share is accurate but the overhead of queuing operations is high.

**Buffer Pool Allocator:** We instrument the MySQL/InnoDB buffer pool implementation to support dynamically partitioning the buffer pool. Each workload is given a quota that represents the number of buffer pool pages it can use. Once its quota is running out, new page requests are satisfied only by evicting an old page from this workload’s partition. The implementation of our design is not trivial. In our early development stage, we noticed the original LRU algorithm used in MySQL could cause significant performance slowdown. The root cause is that the new design only allows each workload to evict a page from its own partition during the page replacement phase. But the bottom page of the LRU stack may not belong to the current workload with the eviction request. Thus this workload has to transverse the LRU stack from bottom to top in order to find a qualified page to evict. This search leads to significant time overhead. To solve this problem, we implement a separate LRU stack for each workload so that every workload can quickly find a candidate page to evict during the page replacement phase. With these efforts, our buffer pool size allocator runs well without observable overhead.

### 2.2.2 Resource Allocators within the Storage Server

Network-attached storage (NAS) systems have been widely deployed in datacenters. NAS are often shared by multiple applications as their persistent storage. The storage clients access NAS through TCP/IP network protocols. Motivated by NAS, we develop Akash as our storage system prototype. It supports data accesses to multiple virtual volumes for any storage client, such as, database servers and file systems. It uses the Network Block Device (NBD) driver packaged with Linux to read and write logical blocks from the underlying virtual storage system. NBD is a standard storage access protocol similar to iSCSI, supported by Linux.
It provides a method to communicate with a storage server over the network. We modified existing client and server NBD protocol processing modules for the storage client and server, respectively, in order to interpose our disk controller modules on the I/O communication path. We piggyback the workload id information, passed from the database server through modified I/O system calls, on the NBD packet. In this way, the storage server is able to identify and track each workload.

**Storage Cache Allocator:** Storage cache is a critical component in modern storage server. It directly decides the performance of the I/O requests. In Akash, the storage cache maintains data as a collection of blocks, and supports concurrent accesses from multiple workloads. We implement a storage cache allocator for partitioning the storage cache in Akash. The internal data structure of our storage cache is well designed for supporting multiple workloads, and the cache block list for replacement is separated per workload. Hence, we easily implement the functionality of dynamically adjusting cache size per workload, without causing the global resource contention.

**Disk Bandwidth Allocator:** We use a quanta-based scheduling algorithm to proportionally allocate the disk bandwidth to different workloads in Akash. Each workload is given a quantum of time during which it uses the disk exclusively. This offers a strong isolation between workloads as there are no disk seeks between I/Os of two workloads. It works as follows. When a workload is given a quantum, we estimate the number of requests we can issue to disk such that they complete within the workloads quantum, using measured average service time. First, we issue requests that were enqueued while waiting for the quantum to begin. Then, we issue requests that arrive during the scheduling quantum. We stop issuing requests if we determine that by issuing a request, we will exceed the workloads quantum. In this case, new requests will be enqueued until the next quantum for this workload.

With these resource allocators, we are able to enforce resources quota/quantum for multiple applications online. Our prototype has been used as the server platform to conduct experiments presented in Chapter 4 and Chapter 5, for studying the dynamic resource allocation problems.
In this chapter, we study coarse grained resource allocation for adding and removing database servers from a database resource pool. Specifically, we design and implement a pro-active performance model based resource allocation algorithm for automated provisioning of database replicas.

3.1 Introduction

Autonomic management of large-scale dynamic content servers in datacenters has recently received growing attention [11, 47, 59, 64, 89, 106] due to the excessive personnel costs involved in managing these complex systems. We introduce a new proactive resource allocation technique for the database back-end of dynamic content web sites.

Dynamic content servers commonly use a three-tier architecture (see Figure 3.1) that consists of a front-end web server tier, an application server tier that implements the business logic, and a back-end database tier that stores the dynamic content of the site. Gross hardware over-provisioning for each workload’s estimated peak load can become infeasible in the
short to medium term, even for large sites. Hence, it is important to efficiently utilize available resources through dynamic resource allocation, i.e., on-demand provisioning for all active applications. One such approach, the IBM Tivoli on-demand business solutions, implements dynamic provisioning of resources within the stateless web server and application server tiers. However, dynamic resource allocation among applications within the stateful database tier, which commonly becomes the bottleneck [2], has received comparatively less attention.

Recent work suggests that fully-transparent, tier-independent provisioning solutions can be used in complex systems that contain persistent state such as the database tier as well [11, 47, 89]. These solutions, similar to Tivoli, treat the system as a set of black boxes and simply add boxes to a workload’s allocation based on queuing models [11, 88], utility models [89, 99] or marketplace approaches [64]. In contrast, our insight in this study is that for a stateful system, such as a database tier, offline system training coupled with online system monitoring and tracking system stabilization after triggering an adaptation are the key features for successful provisioning.

Our goal is to keep the average query latency for any particular workload under a pre-defined service level agreement (SLA). Our previous work achieves this goal through a reactive solution [82], where a new database replica is allocated to a dynamic content workload in response to load or failure-induced SLA violations. The reactive solution is a trial-and-error approach. It allocates the database replica once at a time, and then checks the feedback from the measured performance metrics. If the SLA is still violated, the system will then add one
more replica. As a result, the reactive approach is slow in its adaptation to rapid and large load increase, which needs the addition of multiple database replicas.

To overcome the disadvantages of the reactive approach, we introduce a novel proactive scheme that dynamically adds database replicas in advance of predicted need, while removing them in underload in order to optimize resource usage. This proactive approach can add multiple database replicas at the same time so that it can quickly respond to the sharp load increase, and greatly shorten the duration of SLA violations. Our proactive scheme is based on a classic machine learning algorithm, K-nearest-neighbors (KNN), for predicting resource allocation needs for workloads.

We use an adaptive filter to track load variations, and the KNN classifier to build a performance model of database clusters. The learning phase of KNN uses essential system and application metrics, such as, the average throughput, the average number of active connections, the read to write query ratio, the system-level statistics of CPU, I/O and memory usage. We train the performance model on these metrics during a variety of stable system states using different client loads and different numbers of database replicas. Correspondingly, our proactive dynamic resource allocation mechanism uses active monitoring of the same database system and application metrics at run-time. Based on the predicted load information and the trained KNN classifier, the resource manager adapts online and allocates the number of databases that the application needs in the next time slot under varying load situations.

While proactive provisioning of database replicas is appealing, it faces two inter-related challenges: (1) the unpredictable delay of adding replicas, and (2) the instability of the system after triggering an adaptation. Adding a new database replica is a time-consuming operation because the database state of the new replica may be stale and must be brought up-to-date via data migration. In addition, the buffer cache at the replica needs to be warm before the replica can be used effectively. Thus, when adding a replica, the system metrics might show abnormal values during system stabilization, e.g., due to the load imbalance between the old and newly added replicas. We show that a proactive approach that disregards the needed period of system
stabilization after adaptation induces system oscillations between rapidly adding and removing replicas. We incorporate awareness of system instability after adaptation into our allocation logic in order to avoid such oscillations. Some form of system instability detection based on simple online heuristics could be beneficial when incorporated even in a reactive provisioning technique [82]. On the other hand, our proactive technique can detect and characterize periods of instability with high accuracy due to its system training approach. During the training on a variety of system parameters, the system learns their normal ranges and the normal correlations between their values, resulting in more robust instability detection at run-time.

Our prototype implementation interposes an autonomic manager tier between the application server(s) and the database cluster. The autonomic manager tier consists of an autonomic manager component collaborating with a set of schedulers (one per application). Each scheduler is responsible for virtualizing the database cluster and for distributing the corresponding application’s requests across the database servers within that workload’s allocation.

We evaluate our proactive versus reactive provisioning schemes with the shopping and browsing mix workloads of the TPC-W e-commerce benchmark, an industry-standard benchmark that models an online book store. Our results are as follows:

1. The proactive scheme avoids most SLA violations under a variety of load scenarios.

2. By triggering adaptations earlier and by issuing several database additions in a batch, the proactive scheme outperforms the reactive scheme, which adds databases incrementally and only upon SLA violations.

3. Our system is shown to be robust. First, our instability detection scheme based on learning avoids unnecessary oscillations in resource allocation. Second, our system adapts quite well to load variations when running a workload request mix different than the mix used during system training.

The rest of this chapter is organized as follows. We first introduce the necessary background related to the KNN learning algorithm in section 3.2. Next, we introduce our system
architecture and give a brief overview of our dynamic replication environment in Section 3.3. Then, we discuss our reactive and proactive approaches in Section 3.4 and Section 3.5, respectively. Section 3.6 describes our experimental testbed and benchmark. Section 3.7 illustrates our results for applying the two approaches in the adaptation process of database clusters under different workload patterns. Finally, we conclude the chapter in Section 3.8.

3.2 Background

Classic analytic performance models [11] can predict whether or not a system will violate the SLA given the information on future load. These modeling approaches are, however, not amenable to our problem. This is due to the typically time consuming derivation of an analytic model for modeling complex concurrency control mechanisms such as the one in a replicated database system. Furthermore, in our complex system, the average query latency is not only related to the query arrival rate but is also related to the semantics of the query and the particular query workload mix.

Instead, we use the *k*-nearest-neighbor (KNN) classifier, a machine learning approach which considers multiple features in the system. KNN is an instance-based learning algorithm and has been widely applied in many areas such as text classification [13]. In KNN, a classification decision is made by using a majority vote of *k* “nearest” neighbors based on a similarity measure, as follows:

- For each target data set to be predicted the algorithm finds the *k* nearest neighbors of the training data set. The distance between two data points is regarded as a measure of their similarity. The Euclidean distance is often used for computing the distance between numerical attributes, also called as features.

- The distance we use in this chapter is *weighted* Euclidean distance given by the following formula:

\[
Dist(X, Y) = \sqrt{\sum_{i=1}^{N} weight_i \cdot (x_i - y_i)^2}
\]
Here, X and Y represent two different data points, N denotes the number of features of the data, \( x_i, y_i \) denote the \( i^{th} \) feature of X and Y respectively and \( \text{weight}_i \) denotes the weight of our \( i^{th} \) feature. Each weight reflects the importance of the corresponding feature.

- Find the majority vote of the k nearest neighbors. The similarities of testing data to the k nearest neighbors are aggregated according to the class of the neighbors, and the testing data is assigned to the most similar class.

- Cross validation of the training data is an often used criterion that we also use to select the weights of the features and the number of “nearest” neighbors - K.

In KNN, trained models are implicitly defined by the stored training set and the observed attributes. Furthermore, KNN is robust to noisy training data and effective if the training data is sufficiently large. However, KNN’s computation cost grows proportionately with the size of the training data, since we need to compute the distance of each target attribute to all training samples. Indexing techniques (e.g. K-D tree) can reduce this computational cost.

### 3.3 System Architecture

#### 3.3.1 Architecture of the Dynamic Content Server

Figure 3.2 shows the architecture of our dynamic content server. In our system, a set of schedulers, one per application is interposed between the application and the database tiers. The scheduler tier distributes incoming requests to a cluster of database replicas. Each scheduler\(^1\) upon receiving a query from the application server sends the query using a read-one, write-all replication scheme to the replica set allocated to the application. The replica set is chosen by a resource manager that makes the replica allocation and mapping decisions across the different applications.

\(^1\)Each scheduler may itself be replicated for availability [3, 4].
The scheduler uses our *Conflict-Aware* replication scheme [4] for achieving one-copy serializability [12] and scalability. With this scheme, each transaction explicitly declares the tables it is going to access and their access type. This information is used to detect conflicts between transactions and to assign the correct serialization order to these conflicting transactions. The transaction serialization order is expressed by the scheduler in terms of version numbers. The scheduler tags queries with the version numbers of the tables they need to read and sends them to the replicas. Each database replica keeps track of the local table versions as tables are up-
dated. A query is held at each replica until the table versions match the versions tagged with the query. As an optimization, the scheduler also keeps track of versions of tables as they become available at each database replica and sends read-only queries to a single replica that already has the required versions. The scheduler communicates with a database proxy at each replica to implement replication. As a result, our implementation does not require changes to the application or the database tier.

Since database allocations to workloads can vary dynamically, each scheduler keeps track of the current database set allocated to its workload. The scheduler is also in charge of bringing a new replica up to date by a process that we call data migration, during which all missing updates are applied on that replica.

Our goals for resource management in our system are that the resource manager should be:

- **Prompt.** It should sense impending SLA violations accurately and quickly, and it should trigger resource allocation requests as soon as possible in order to deal with the expected load increase.

- **Stable.** It should avoid unnecessary oscillations between adding and removing database servers, because such oscillations waste resources.

### 3.3.2 Dynamic Replication

Next, we provide an overview of the resource manager that implements dynamic replication and briefly introduce the replica addition, removal, mapping as well as data migration mechanisms in our system.

The resource manager makes the replica allocation and mapping decisions for each application based on its requirements and the current system state. The requirements are expressed in terms of a service level agreement (SLA) that consists of a latency requirement on the application’s queries. The current system state includes the current performance of this application and the system capacity. The allocation decisions are communicated to the respective sched-
ulars, which then allocate or remove replicas from their replica sets.

**Replica Addition and Removal**

The resource manager adds or removes a replica to/from an application allocation if it determines that the application is in overload or underload, respectively. Database replica removal needs to be performed conservatively because adding a database to a workload has high overheads. The replica addition process consists of two phases: data migration and system stabilization (see Figure 3.3). Data migration involves applying logs of missing updates on the new replica to bring it up-to-date. System stabilization involves load balancing and warmup of the buffer pool on the new replica. While some of these stages may overlap, replica addition can introduce a long period over which query latencies are high.

**Potential for Oscillations in Allocation**

Oscillations in database allocations to workloads may occur during system instability induced by adaptations or rapidly fluctuating load. Assume an adaptation is necessary due to a burst in client traffic. Since our database scheduler cannot directly measure the number of clients, it infers the load by monitoring various system metrics instead. In the simplest case, the scheduler infers the need to adapt due to an actual latency SLA violation. However, *during* the adaptation phases, i.e., data migration, buffer pool warmup and load stabilization, the latency will be high or may even temporarily continue to increase as shown in Figure 3.3. Latency sampling during this potentially long time is thus not necessarily reflective of a continued increase in load, but of system instability after an adaptation is triggered. If the system takes further decisions based on sampling latency during the stabilization time, it may continue to add further replicas which are unnecessary, hence will need to be removed later. This is an oscillation in allocation which carries performance penalties for other applications running on the system due to potential interference.

A similar argument may hold for other system metrics measured during adaptation. Their
values will not be indicative of any steady-state system configuration even if the load presented to the system remains unchanged. While rapid load fluctuations may induce similar behavior, simple smoothing or filtering techniques can offer some protection to very brief load spikes. While all schemes presented in this chapter use some form of smoothing or filtering, which can dampen brief load fluctuations, our emphasis is on avoiding the tuning of any system parameter, including smoothing coefficients. Instead we develop techniques for automatically avoiding all cases of allocation oscillation caused by system metric instability.
CHAPTER 3. COARSE GRAINED RESOURCE ALLOCATION FOR BACKEND DATABASES

Replica Mapping

Dynamic replication presents an inherent trade-off between minimizing application interference by keeping replica sets disjoint versus speeding replica addition by allowing overlapping replica sets. In this study, we use warm migration where partial overlap between replica sets of different applications is allowed. Each application is assigned a disjoint primary replica set. However, write queries of an application are also periodically sent to a second set of replicas. This second set may overlap with the primary replica set of other applications. The resource manager sends batched updates to the replicas in the secondary set to ensure that they are within a staleness bound, which is equal to the batch size or the number of queries in the batch. The secondary replicas are an overflow pool that allows adding replicas rapidly in response to temporary load spikes since data migrating onto these replicas is expected to be a fast operation.

Data Migration

In this section, we describe the implementation of data migration in our system. Our data migration algorithm is designed to bring the joining database replica up to date with minimal disruption of transaction processing on existing replicas in the workload’s allocation.

Each scheduler maintains persistent logs for all write queries of past transactions in its serviced workload for the purposes of enabling dynamic data replication. The scheduler logs the queries corresponding to each update transaction and their version numbers at transaction commit. The write logs are maintained per table in the order of the version numbers for the corresponding write queries.

During data migration for bringing a stale database replica up-to-date, the scheduler replays on it all missing updates from its on-disk update logs. The challenge for implementing an effective data migration is that new transactions continue to update the databases in the workload’s allocation while data migration is taking place. Hence, the scheduler needs to add the new database replica to its workload’s replica mapping before the end of data migration.
Otherwise, the new replica would never catch up. Unfortunately, new updates cannot be directly applied to the (stale) replica before migration completes. Hence, new update queries are kept in the new replica’s holding queues during migration. In order to control the size of these holding queues, the scheduler executes data migration in stages. In each stage, the scheduler reads a batch of old updates from its disk logs and transfers them to the new replica for replay without sending any new queries. Except for pathological cases, such as sudden write-intensive load spikes, this approach reduces the number of disk log updates to be sent after each stage, until the remaining log to be replayed falls below a threshold bound. During this last stage, the scheduler starts to send new updates to the replica being added, in parallel with the last batch of update queries from disk.

### 3.4 Reactive Replica Provisioning

Figure 3.4 shows the replica allocation logic and the conditions for triggering an addition of a database replica to an application and for removing a replica from an application. In the following we describe these adaptations in detail.

#### 3.4.1 Reactive Replica Addition

In the Steady State, the resource manager monitors the average latency received from each workload scheduler during each sampling period. The resource manager uses a smoothened latency average computed as an exponentially weighted mean of the form $WL = \alpha \times L + (1 - \alpha) \times WL$, where $L$ is the current query latency. The larger the value of the $\alpha$ parameter, the more responsive the average is to the current latency. If the average latency over the past sampling interval for a particular workload exceeds the HighSLAThreshold, hence an SLA violation is imminent, the resource manager places a request to add a database to that workload’s allocation.
3.4.2 Reactive Replica Removal

If the average latency is below a LowSLAThreshold, the resource manager triggers a replica removal. The right branch of Figure 3.4 shows that the removal path is conservative and involves a tentative remove state before the replica is finally removed from an application’s allocation. The allocation algorithm enters the tentative remove state when the average latency is below the low threshold. In the tentative remove state, a replica continues to be updated, but is not used for load balancing read queries for that workload. If the application’s average latency remains below the low threshold for a period of time, the replica is removed from the allocation for that workload. This two-step process avoids system instability by ensuring that the application is indeed in underload, since a mistake during removal would soon require replica addition, which is expensive. For a forced remove during overload, we skip the tentative removal state and go directly to the removal state. In either case, the database replica is removed from an application’s replica set only when ongoing transactions finish at that replica.
3.4.3 Enhancement of Reactive Approach with System Instability Detection

The resource manager makes several modifications to this basic allocation algorithm in order to account for replica addition delay and protect against oscillations. First, it stops making allocation decisions based on sampling query latency until the completion of the replica addition process. Completion includes the phases of data migration and system stabilization. The scheduler uses a simple heuristic for determining when the system has stabilized after a replica addition. In particular, it waits, for a bounded period of time, for the average query latency at the new replica to become close (within a configurable ImbalanceThreshold value) to the average query latency at the old replicas. Finally, since this wait time can be long and can impact the reaction to steep load bursts, the resource manager uses the query latency at the new replica in order to improve its responsiveness. Since this replica has little load when it is added, we use its average latency exceeding the high threshold as an early indication of a need for even more replicas for that application. The resource manager triggers an extra replica addition in this case.

3.5 Proactive Replication Provisioning

In our proactive approach, the controller predicts performance metrics and takes actions in advance of need to add databases such that the SLA is not violated while resources are used close to optimally.

3.5.1 Overview

Our approach can be summarized as follows. \textit{We predict the status of the application in the next time interval and classify it into two categories: SLA violation or within SLA for a given number of database servers.} By iterating through all possible database set configurations, we
Figure 3.5: Proactive replication provisioning scheme
decide the database set with minimum size that is predicted to have no SLA violations.

In more detail, Figure 3.5 shows the main process of our proactive provisioning scheme. The scheduler of each application works as the application performance monitor and is responsible for collecting various system load metrics and reporting these measured data to the global resource manager (controller). The controller consists of three main components. First, the adaptive filter predicts the future load based on the current measured load information. Next, the classifier finds out whether the SLA is broken given the predicted load and the number of active databases. The classifier directs the resource allocation component to adjust the number of databases to the proper number of databases for this application according to its prediction. The resource allocation component decides how to map this request onto the real database servers by considering the requests of all applications and the available system capacity.

**3.5.2 Enhancement of Proactive Approach with System Instability Detection**

Our classifier is trained under stable states, and our training data is gathered for several constant loads. When the system triggers an adaptation to add a new database or remove a database from a workload’s allocation, the system goes through a transitional (unstable) state. At this time, the system metric values are quite different from the ones measured during stable system states that we use as the training data. As a result many wrong decisions could be made if the classifier uses the system metrics sampled during the period of instability. This could in its turn lead to oscillations between rapidly adding and removing database replicas, hence unnecessary adaptation overheads and resource waste.

To overcome this unstable phenomenon, we enhance our proactive provisioning algorithm to be aware of the instability after adaptation. The system automatically detects unstable states by using two indicators: the load imbalance ratio and the ratio of average query throughput versus the average number of active connections. During our training process, we record the normal range of variations of the load imbalance ratio among different databases. If the mea-
sured ratio is beyond the normal range, we decide that the system is in an unstable state. Second, we select the two features assigned the highest weights during our training phase, which, as we will show, are the throughput and number of connections. If we observe that the ratio of the two metrics is beyond the normal range, we also decide that the system is in an unstable state.

In our approach, if the system is detected to be in an unstable state, we suppress taking any decisions until the system is stable.

3.5.3 Implementation Details

There are various filter and classifier algorithms that can fit into our proactive replication provisioning scheme. We take the ease of the implementation and their promptness as our selection principles.

Filter

We use a critically damped g-h filter [14] to track the variations of our load metrics. This filter minimizes the sum of the weighted errors with the decreasing weights as the data gets older; i.e., the older the error, the less it matters. It does this by weighting the most recent error by unity, the next most recent error by a factor $\Theta$ (where $\Theta < 1$), the next error by $\Theta^2$, the next error by $\Theta^3$, and so on.

KNN Classifier

We select several application metrics and system metrics as the features used by our KNN classifier in order to enhance our confidence about the automatically inferred load information. These system metric are readily available from our environment. They are as follows:

1. Average query throughput - denotes the number of queries completed during a measurement interval.
2. Average number of active connections - counts the average number of active connections as detected by our event-driven scheduler within its select loop. An active connection is a connection used to deliver one or more queries from application servers to the scheduler during a measurement interval.

3. Read write ratio - shows the ratio of read queries versus write queries during a measurement interval. This feature reflects the workload mix.

4. Lock ratio - shows the ratio of locks held versus total queries during a measurement interval.

5. CPU, Memory and I/O usage reported by `vmstat`.

Metrics 1 to 4 are gathered by the scheduler of each application. The traditional system metrics in 5 are measured on each database server.

Although a single metric may reflect the load information to some degree, basing decisions on a single metric could be seriously skewed e.g., if the composition of queries in the mix changes or if the system states are not fully reproducible. We use cross validation techniques in KNN to identify the importance (i.e., weight) of each metric. The weights are automatically determined and they reflect the usefulness of features. Not all features are always useful during online load situations. Their usefulness depends on the characteristics of the current workload mix. For example, for a CPU-bound workload mix, I/O metrics will be irrelevant for the purposes of load estimation.

During our training phase, we run 10-fold cross validation for weight combinations varying within a finite range, and pick the weight setting whose accuracy is higher than our target accuracy (95%) or achieves the highest accuracy in the given range. We can continue to expand the search space by gradient methods until we achieve a good accuracy. This training process assigns weights for all features offline. Less important features automatically get lower weights.
CHAPTER 3. COARSE GRAINED RESOURCE ALLOCATION FOR BACKEND DATABASES 35

3.6 Experimental Setup

To evaluate our system, we use the same hardware for all machines running the client emulator, the web servers, the schedulers and the database engines. Each is a dual AMD Athlon MP 2600+ computer with 512MB of RAM and 2.1GHz CPU. All the machines use the RedHat Fedora Linux operating system. All nodes are connected through 100Mbps Ethernet LAN.

We run the TPC-W benchmark that is described in more detail below. It is implemented using three popular open source software packages: the Apache web server [7], the PHP web-scripting/application development language [72] and the MySQL database server [62]. We use Apache 1.3.31 web-servers that run the PHP implementation of the business logic of the TPC-W benchmark. We use MySQL 4.0 with InnoDB tables as the database backend.

All experimental numbers are obtained running an implementation of our dynamic content server on a cluster of 8 database server machines. We use a number of web server machines sufficient for the web server stage not to be the bottleneck. The largest number of web server machines used for any experiment is 3. We use one scheduler and one resource manager. The thresholds we use in the reactive experiments are a HighSLAThreshold of 600ms and a LowSLAThreshold of 200ms. The SLA threshold used in our proactive approach is 600ms. The SLA threshold was chosen conservatively to guarantee an end-to-end latency at the client of at most 1 second for the TPC-W workload. We use a latency sampling interval of 10 seconds for the scheduler.

3.6.1 TPC-W E-Commerce Benchmark

The TPC-W benchmark from the Transaction Processing Council [92] is a transactional web benchmark designed for evaluating e-commerce systems. Several interactions are used to simulate the activity of a retail store such as Amazon. 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.
The inventory images, totaling 1.8 GB, are resident on the web server. We implemented the 14 different interactions specified in the TPC-W benchmark specification. Of the 14 scripts, 6 are read-only, while 8 cause the database to be updated. Read-write interactions include user registration, updates of the shopping cart, two order-placement interactions, two involving order inquiry and display, and two involving administrative tasks. We use the same distribution of script execution as specified in TPC-W. In particular, we use the TPC-W shopping mix workload with 20% writes which is considered the most representative e-commerce workload by the Transactional Processing Council. The complexity of the interactions varies widely, with interactions taking between 20 ms and 1 second on an unloaded machine. Read-only interactions consist mostly of complex read queries in auto-commit mode. These queries are up to 30 times more heavyweight than read-write transactions.

3.6.2 Client Emulator

To induce load on the system, we have implemented a session emulator for the TPC-W benchmark. A session is a sequence of interactions for the same customer. For each customer session, the client emulator opens a persistent HTTP connection to the web server and closes it at the end of the session. Each emulated client waits for a certain think time before initiating the next interaction. The next interaction is determined by a state transition matrix that specifies the probability of going from one interaction to another. The session time and think time are generated from a random distribution with a given mean. For each experiment, we use a load function according to which we vary the number of clients over time. However, the number of active clients at any given point in time may be different from the actual load function value at that time, due to the random distribution of per-client think time and session length. For ease of representing load functions, in our experiments, we plot the input load function normalized to a baseline load.
3.7 Experimental Results

3.7.1 System Training

In this section, we describe our training phase and its effect on the assigned weights for our pre-selected system features.

We train our system on the TPC-W shopping mix with database configurations of 1 through 8 replicas and client loads from 30 to 220 clients under stable states. The weights of features in the TPC-W shopping mix obtained from the training phase on this mix are listed here in the order of importance: the average number of active connections, the average query throughput, the read/write ratio, the CPU usage, the Lock ratio, the memory usage and the I/O usage.

The TPC-W shopping mix has significant locality of access. Hence, it is not an I/O intense workload. This explains the low relevance of I/O usage. Furthermore, the MySQL database management system does not free the memory pages for TPC-W even if it is in under-load, so memory usage also has low relevance for inferring the load level. On the other hand, contrary to our intuition, the lock ratio does not show a high association with the load level. The lock ratio could, however, show higher relevance for larger cluster configurations.

3.7.2 Proactive Approach without Stability Awareness

In this section, we show the influence of system metric instability during adaptation. Figure 3.7(a) shows an example of such oscillations that happen under the continuous load function shown in Figure 3.6. The oscillations happen due to (incorrect) adaptation decisions taken during system instability when system metrics are varying wildly immediately after an adaptation.

In order to explain the oscillations, Figure 3.7(b) zooms into a small time period of the previous adaptation graph (between times 220 and 260 seconds of the experiment) to illustrate the variation of latency during an adaptation. It shows the latency pattern when the system adapts from a configuration of 2 databases to a configuration of 3 databases i.e., one database
addition. We can see that the latency initially increases during the data migration phase, until End Migration, then stays flat while the buffer pool warms up on the new replica and finally decreases and stabilizes thereafter.

In addition, table 3.1 shows the variation of our highest weighted system metrics, the number of active incoming connections and the throughput just before (i.e. $t_1$) and just after (i.e. $t_2,t_3,t_4$) adding the new database (during 4 time steps). The last row of the table also shows the corresponding stable state values with the 3 database replicas configuration. During stable states, the average number of active connections at the scheduler is closely correlated with the total load on the system induced by active clients. That is, given a fixed number of replicas,

<table>
<thead>
<tr>
<th>No. of db (time)</th>
<th>Throughput</th>
<th>Avg. Active Conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Just before addition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2db ($t_1$)</td>
<td>80.80</td>
<td>118</td>
</tr>
<tr>
<td>After addition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3db ($t_2$)</td>
<td>142.20</td>
<td>105</td>
</tr>
<tr>
<td>3db ($t_3$)</td>
<td>167.10</td>
<td>89</td>
</tr>
<tr>
<td>3db ($t_4$)</td>
<td>211.40</td>
<td>70</td>
</tr>
<tr>
<td>In stable state</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3db</td>
<td>170</td>
<td>70</td>
</tr>
</tbody>
</table>
Chapter 3. Coarse Grained Resource Allocation for Backend Databases

Figure 3.7: Provisioning results without stability awareness under a sine load function

(a) Machine allocation

(b) Latency variation while the system adapts from 2 to 3 databases.
if the number of active connections increases, the throughput also increases until the throughput is saturated. In contrast, as we can see from the table, the number of active connections might register a sudden decrease (from t2 to t4) immediately after adding a new database even while the throughput increases. At t4, the throughput 211.40 is much higher than the normal throughput 170 measured at the stable state when the number of active connection is 70. These effects are due to the various factors at play during system stabilization. For example, as the new replica gets more requests, the overload on existing replicas starts to normalize and many client requests that were delayed due to overload finally complete. As a result, a larger than usual number of clients may get their responses at this time. These clients will be in the thinking state during the next interval explaining the lower number of active incoming connections after adaptation.

These abnormal system metric variations after an adaptation may induce wrong load and latency estimates or predictions. For example, our KNN predictor might interpret the low number of active incoming connections as underload. Wrong decisions, such as removing a database immediately after adding it or vice versa may result. In the rest of the experiments, our KNN-based prediction algorithm is enhanced to suppress taking decisions during intervals when system instability is detected.

### 3.7.3 Performance Comparison of the Proactive and Reactive Approaches

In the following, we evaluate the two autonomic provisioning approaches: reactive and proactive. We first consider a scenario with continuously changing load, where the load variation follows a sinusoid (sine) function. Then we consider a sudden change scenario with a large load spike.

**Load with Continuous Change Scenario**

We use our client emulator to emulate a sinusoid load function, shown in Figure 3.6. As we can see from Figure 3.8(a), the proactive approach triggers replica adding actions earlier than the
Figure 3.8: Provisioning results under a sine load function
reactive approach, because it performs future load prediction. In contrast, the proactive removal is slightly slower than the reactive database removal because we use a conservative decision regarding when the system is sufficiently stable to accurately decide on removal. Figure 3.8(b) shows the comparison of average query latency for these two approaches. As a result of the earlier resource adaptations of the proactive provisioning, this approach registers fewer SLA violations than the reactive approach. Furthermore, the degree of SLA violations, reflected in the average latency peaks, is also reduced compared to the reactive approach.

Sudden Load Spike Scenario

![Figure 3.9: Sudden load spike function](image)

We use our client emulator to emulate a load function with a sudden spike, shown in Figure 3.9. From Figure 3.10(a), we see that neither scheme can avoid SLA violations when the load jump happens, since the change is too abrupt to predict. However, the proactive provisioning has a lower SLA violation peak and duration than the reactive provisioning approach. Specifically, the proactive approach reaches a query latency peak of 2 seconds, and the SLA violations last around 50 seconds while the reactive approach reaches a query latency peak of
Figure 3.10: Provisioning results under a sudden load spike function
CHAPTER 3. COARSE GRAINED RESOURCE ALLOCATION FOR Backend DATABASES

5 seconds, and its corresponding SLA violations last more than 2 minutes.

The reactive provisioning is much slower in its adaptation because it needs to obtain feedback from the system after each database addition. It does not know how many databases it should add, so it has to add the databases one at a time. In contrast, the proactive approach can predict how many databases the system needs for the current and predicted load and it is able to trigger several simultaneous additions in advance of need. Figure 3.10(b) shows that the proactive scheme adds 3 databases in a batch by requesting 3 databases simultaneously, while the reactive approach needs to add the 3 databases sequentially.

3.7.4 Robustness of the Proactive Approach

In this section, we show that our proactive scheme is relatively robust to some degree of online variation in workload mix and different load patterns given a fixed training data set.

Figures 3.11 shows how our proactive approach adapts online under a workload request mix different than the one it has been trained with. In particular, we train the system on a data set corresponding to stable load scenarios for the TPC-W shopping mix as before. We then show online adaptations for running TPC-W with the browsing mix. The browsing mix contains a different query mix composition than the shopping mix (with 5% versus 20% write queries). We use the same sine load function as before. We can see that our proactive scheme adapts quite well to load increases, minimizing the number of SLA violations. It adapts less well to load decreases, however, by retaining more databases than strictly necessary. This effect is most obvious towards the end of the run.

Figure 3.12 and 3.13 shows the robustness of our learning-based approach under a step load function. Figure 3.12(a) shows the step function and the evolution of the instantaneous number of active client connections (as opposed to thinking clients) as measured at the emulator induced by this load function. Figure 3.12(b) shows that the allocation of the proactive scheme is stable while the reactive scheme may register some oscillations in allocation if the thresholds it uses are not tuned. We run the reactive scheme in two configurations, pre-tuned and untuned.
Figure 3.11: A browsing workload scenario under a sine load function
CHAPTER 3. COARSE GRAINED RESOURCE ALLOCATION FOR Backend DATABASES

Figure 3.12: Machine allocation results under a step load function

(a) Step load function

(b) Machine allocation

Figure 3.12: Machine allocation results under a step load function
Figure 3.13: Latency results under a step load function

(a) Average query latency

(b) Average query latency
by using to different values for the ImbalanceThreshold, which governs the scheme’s heuristic instability detection. We use a threshold of 10% load imbalance with a time-out of 2 minutes for a pre-tuned reactive approach and a random value of these parameters for the other reactive graph shown in the Figure. We can see that the reactive scheme registers allocation oscillations which also incur latency SLA violations (shown in Figure 3.13) for both parameter settings with more oscillations for the untuned configuration. More sensitivity analysis results and oscillations induced by different parameters for the reactive scheme, e.g., LowSLAThreshold values are shown elsewhere [82]. Since our proactive scheme learns the normal imbalance range and uses highly relevant system metrics according to the learning phase to determine instability, it is inherently more robust and has no allocation oscillations.

Finally, although the step function makes it slightly harder to predict the load trend compared to the sine load function, for the proactive approach latency SLA violations are mostly avoided in this load scenario as well.

3.8 Summary

In this chapter, we address autonomic provisioning in the context of dynamic content database clusters. We introduce a novel proactive scheme based on the classic K-nearest-neighbors (KNN) machine learning approach for adding database replicas to application allocations based on: (i) load predictions, (ii) extensive offline measurements of system and application metrics for stable system states and (iii) lightweight online monitoring that does not interfere with system scaling.

We use a full prototype implementation of both our proactive approach and a previous reactive approach to dynamic provisioning of database replicas. Overall, our experimental results show that our KNN-based proactive approach is a promising approach for autonomic resource provisioning in dynamic content servers. Compared to the previous reactive approach, the proactive approach has the advantage of promptness in sensing the load trend and its ability
to trigger several database additions in advance of SLA violations. By and large our proactive approach avoids SLA violations under load variations. For unpredictable situations of very sudden load spikes, the proactive approach can alleviate the SLA violations faster than the reactive approach even if SLA violations do occur in this case. Finally, offline training on system-level information is also useful for recognizing periods of instability after triggering adaptations. Detecting unstable system states reduces the prediction errors of the proactive approach in such cases.
Chapter 4

Fine Grained Resource Partitioning of the Cache Hierarchy

4.1 Introduction

The costs of management, power and cooling for large service providers hosting several applications are currently prohibitive, taking up a large portion of the average company budget. This is a major impediment on the efficiency of this industry, by limiting reinvestment, research and development. To achieve cost reductions, automated server consolidation techniques for better resource usage while providing differentiated Quality of Service (QoS) to applications become increasingly important. With server consolidation, several concurrent applications are multiplexed on each physical server of a server farm connected to consolidated network attached storage. The challenge lies in the complexity of the dynamic resource partitioning problem for avoiding application interference at multiple levels of this shared system. For example, the provider may service multiple applications on an infrastructure composed of web servers, database servers and storage servers (as in Figure 4.1). An especially important problem in these environments, which we focus on in this chapter, is controlling application interference in the cache hierarchy across two tiers contributing directly to the performance of consolidated
database applications, namely, (1) the database server tier, and (2) the storage server tier. Towards controlling this interference, we propose a dynamic global cache partitioning scheme that exploits the synergy between the cache at the database server i.e., the buffer pool, and the cache at the storage server.

Previous work in the area of dynamic resource partitioning has focused on controlling interference within a single tier at a time. For example, gold/silver/bronze priority classes within the buffer pool of a database system hosting several concurrent applications have been used to enforce memory priorities [15, 16]. Similarly, storage techniques for partitioning the I/O bandwidth between applications have been developed [40, 55, 96]. Additionally, enforcing per-application CPU quotas through resource virtualization techniques has been studied either at the operating system [9], or at the database system level [68, 69].

The previous approaches fall short of providing effective resource partitioning due to the following two reasons. The first reason is that application QoS is usually expressed as a service level objective (SLO), not as per-resource quotas; there is currently no automatic mechanism to accurately assign resource quotas for applications corresponding to a given application metric. The second reason that prevents these approaches from providing effective resource partitioning is the absence of coordination between different resource controllers located within different tiers. This absence of coordination might lead to situations where local partitioning optima do not lead to the global optimum; indeed local goals may conflict with each other, or with the per-application goals. This resource allocation problem is further complicated when applications define different utilities (or penalties) for meeting (or missing) the specified SLOs. In such settings, the need is even stronger for an SLO-aware coordinated cache partitioning method which maximizes the system utility.

Coordination between the database buffer pool and storage cache has already been shown to be an effective mechanism in the context of cache replacement policies [52, 102]. However, coordinated cache replacement is an efficient mechanism for improving the performance of a single application, whereas in the presence of multiple applications, an orthogonal coordi-
nated cache partitioning mechanism is still required. In this chapter, we show that integrating our cache partitioning solution with current coordinated cache replacements policies provides further performance improvements that are not achievable using replacement policies alone.

Towards addressing the dynamic resource allocation problem in shared server environments, we introduce a novel technique for coordinated cache partitioning of the database server and storage caches. Our technique is independent of the cache replacement policy used at each level and it works with both coordinated and uncoordinated cache replacement policies. Our technique determines per-application resource quotas in each of the two caches on the fly, in a transparent manner, with minimal changes to the DBMS, and no changes to existing interfaces between components. To achieve this, we augment the DBMS with a resource controller in charge of partitioning both the buffer pool and the storage cache between applications. The target is to find a setting that maximizes the overall utility associated with the SLOs of a given set of applications. The resource controller maps the application specified SLO to a target memory access latency, which is the average page access latency measured at the database buffer pool required to meet the SLO.

To decide the right partitioning, the cache controller explores the configuration space through an online simulation of the cache hierarchy. This allows us to converge faster towards an optimal partitioning solution. However, the cache controller actuates the cache partitioning settings periodically, to the current best configuration, and measures performance in the current configuration, in order to validate the simulation. The controller employs statistical regression to dynamically determine per-application performance/utility models as mapping functions between the cache quota settings of the two caches and the corresponding application latency/utility. It then uses these per-application models to answer “what-if” cache partitioning scenarios, for any given set of applications, hence to dynamically converge towards a partitioning that maximizes the perceived overall reward.

We implement our technique in a prototype of a two-level cache controller. In our experiments, we use the MySQL database engine and two applications: the TPC-W e-commerce
benchmark, emulating an online bookstore, such as Amazon.com, and the RUBiS online bidding benchmark, emulating an online auctions site, such as eBay.com. We use our prototype in an experimental testbed, where instances of the two applications share physical servers as well as the storage server, to enforce cache quota allocations for different SLO and load scenarios, and different cache replacement policies. In terms of cache replacement policies, we integrate our coordinated, dynamic cache partitioning technique with (i) classic uncoordinated LRU replacement at each cache level, as well as (ii) coordinated cache replacement based on demote hints from the buffer pool to the storage cache [102].

We show that our coordinated dynamic partitioning technique provides compliance with the SLO requirement of applications with strict SLO’s, while at the same time maintaining efficient resource usage. As a result, our dynamic cache partitioning technique minimizes penalties in overload and maximizes the revenue of the service provider in underload.

The remainder of this chapter is structured as follows. Section 4.2 provides a background on server consolidation in modern data centers highlighting the detrimental effect of interference between two applications. We describe our coordinated cache partitioning algorithm in Section 4.3. Section 4.4 describes our virtual storage prototype. Section 4.5 presents the algorithms we use for comparison, our benchmarks, and our experimental methodology, while Section 4.6 presents the results of our experiments on this platform. Section 4.7 concludes the paper.

### 4.2 Background

Modern enterprise systems consist of multiple software layers including web/application server front-ends, database servers running on top of the operating system, and storage servers at the lowest level. In order to reduce hardware and management costs in large data centers, the storage system is usually shared by a cluster farm, as shown in Figure 4.1. Since slow disk access is the bottleneck in this system, both the database servers and the shared storage server
use memory to cache data blocks, resulting in a two-tier cache hierarchy.

In this chapter, we propose methods for controlling interference among applications in this cache hierarchy. Our techniques are applicable to situations where the working set of concurrent applications does not fit into the cache hierarchy. These situations are, and will remain common in the foreseeable future due to the following reasons. First, while both the buffer pool and storage server cache sizes are increasing, so do the memory demands of applications e.g., scientific and commercial very large databases. Second, efficiently using the combined caching capabilities of database server and storage server is challenging even for a single application. Indeed, the potential for double caching of blocks, and the typically poor temporal locality of accesses that miss in the buffer pool lead to poor cache utilization in the storage level cache [26, 53]. Finally, running several applications on a cluster with consolidated storage, and/or on the same physical server exacerbates the above problems due to application interference for memory, hence the increased potential for capacity misses in the cache hierarchy.

The synergy between buffer pool and storage cache has been exploited through replace-
ment policies for improving cache hierarchy effectiveness for a single application [43, 45, 52, 65, 102, 104]. Specifically, recent work has shown that communication between caches is essential [52, 102, 104] for effective use of multi-tier caches. For example, the DEMOTE [102] scheme sends block eviction hints or explicit demote operations from the client cache e.g., the database buffer pool, to the storage cache with the goal to maintain exclusiveness between the two caches. When the client cache is about to evict a clean block, it sends the clean block to the storage cache using a special DEMOTE operation. The storage cache places the demoted block in its cache, ejecting another block if necessary. The storage cache moves the blocks read by the client to the LRU (least-recently-used) position such that they will be evicted before the demoted blocks. Li et al. [52] and Yadgar et al. [104] extend the DEMOTE idea using DBMS specific information. Their work has shown that these techniques increase the effectiveness of the combined buffer pool and storage caches and are essential to the performance of the database system. However, in the next section, we show that the detrimental effect of application interference in the cache hierarchy offsets the gains obtained from the above advanced replacement policies.

Motivating Example

We present a motivating example to highlight the need for better management of shared multi-tier caches. We use two applications: TPC-W, considered strict SLO, and RUBiS, considered best effort, and schedule the applications such that they share a single DBMS instance, as well as the storage server, as shown in Figure 4.1. We require that the average TPC-W query memory access latency be less than 500\,$\mu$s; in practice, some pre-defined margin of error may be acceptable. We also assume that any reductions in the latency of the best effort application, RUBiS, compared to the worst case scenario are rewarded e.g., through revenue increases for the provider; we consider the worst case scenario for RUBiS to correspond to its incurring the full disk latency on each query data access.

We run the two applications using a single MySQL/InnoDB database engine and a consolidated storage server. Since MySQL/InnoDB does not provide an easily partitionable buffer
pool, we replace its buffer pool with our own implementation. We use a 1GB buffer pool and a 1GB storage cache and we experiment with two cache replacement policies in the two shared caches. We denote a scheme as LRU/LRU a scheme where the classic LRU replacement policy is used in both the buffer pool and the storage cache (Figure 4.2(a)). We denote as LRU/DEMOTE a scheme where a LRU replacement is used at the buffer pool modified to support the DEMOTE cache block eviction hints for the storage cache (Figure 4.2(b)). We provide the details of our storage platform in Section 4.4. We compare two schemes: (1) SHARED

Figure 4.2: We experiment with two cache configurations: LRU/LRU and LRU/DEMOTE. The results show significant room for improvement.
where the applications share both the DBMS buffer pool and the storage cache, with no quota enforcement and (2) IDEAL where we experimentally iterate through all possible partitioning configurations of both caches and choose the optimal setting where we meet the SLO for TPC-W, while minimizing the latency for RUBiS.

Figure 4.2 shows the performance of each benchmark under the above schemes, in addition to the performance for ISOLATED, which corresponds to running each benchmark in isolation, using a 1GB buffer pool at the DBMS and a 1GB storage cache. Figure 4.2(a) shows that under LRU/LRU, the average latency of TPC-W, in isolation, is 420\(\mu s\), while for RUBiS the isolation latency is 304\(\mu s\). This experiment shows that our storage infrastructure is capable of meeting the SLO for TPC-W. Next, we run both TPC-W and RUBiS allowing them to share the buffer pool and the storage cache. In this case, there is no SLO enforcement resulting in TPC-W consistently violating its SLO with an average 715\(\mu s\) memory access latency. This scenario would result in hefty penalties for the service provider. By partitioning the caches, the IDEAL partitioning scheme finds a cache setting that maintains TPC-W within the SLO. This scheme shows the best possible resource usage scenario and the highest revenue for the service provider.

We repeat our experiments with the LRU/DEMOTE cache replacement policy (Figure 4.2(b)). Since the DEMOTE scheme results in a better utilization of the overall cache hierarchy, both TPC-W and RUBiS obtain lower latencies when in isolation, compared to the LRU/LRU case. The average memory access latency is 284\(\mu s\) for TPC-W, while for RUBiS is 143\(\mu s\). While the LRU/DEMOTE policy provides better cache utilization, using the SHARED scheme still results in a SLO violation, since the average memory access latency for TPC-W is 617\(\mu s\). Similar to the results in the LRU/LRU case, the IDEAL scheme maintains the TPC-W latency within the SLO for LRU/DEMOTE as well.

The above results show that the performance of a strict SLO application can severely degrade when two database applications are co-located within the same DBMS instance. These experiments thus motivate coordination in terms of both cache partitioning and replacement...
policy between the two caches. However, the problem of finding the globally optimum partitioning of the two caches to a given set of applications is an NP-hard problem [76]. Let’s consider the time needed to find the IDEAL cache partitioning setting. For example, say we can have 32 possible quota settings for each cache. Then, in order to estimate an application’s performance for all possible cache and storage quota configurations, we need to gather performance samples for $32 \times 32 = 1024$ configurations. Each sample point measurement may take 15 minutes, on average, to ensure statistical significance e.g., due to cache warmup effects and the need to measure latency several times in each configuration. Therefore, in order to compute an accurate performance model for just one application, we will need $1024 \times 15$ minutes, i.e., 256 hours (approximately 11 days)! In our experiment for obtaining the IDEAL cache setting, we reduce this time significantly, by iterating from larger cache quotas to smaller cache quotas for the two caches, for each application, thus amortizing the warm-up time of the larger cache quota configurations for the smaller cache quota configurations. This still results in a total running time for the two applications on the order of days, which is unacceptable for online adaptation.

In the rest of this chapter, we describe the design and implementation of a novel approximate algorithm that partitions both the database buffer pool and the storage cache online for any cache replacement policy and any per-application access pattern.

### 4.3 Cache Partitioning Algorithm

In this section, we describe our approach to providing effective coordinated cache partitioning in two-level caches. Our main objective is to maximize the utility i.e., reward or revenue, derived by the server provider from running a set of applications concurrently on a shared cache hierarchy. Towards this, we use a novel technique, called utility-aware iterative learning, to determine the size of cache quotas at different levels in the system, i.e., the DBMS and the storage server. The key idea is to dynamically determine, through a statistical regression
method, the mapping between a cache partitioning setting for a given set of applications and its corresponding overall utility for the service provider.

In the following, we first introduce the problem statement, and an overview of our approach. Then, we introduce our utility-aware iterative learning approach along with details of its main components.

### 4.3.1 Problem Statement

We study dynamic cache allocation to multiple applications with pre-defined QoS requirements in the cache hierarchy of server farms with network attached storage.

In our model, we assume that the system is hosting $n$ applications, where all applications run on one database engine. The buffer pool is partitioned among applications. Each application has its own buffer pool cache partition. Additionally, the system has a storage cache which is shared among all applications. Finally, we assume that each application is associated with a pre-specified utility, i.e., benefit as a function of the memory access latency perceived by the given application. Thus, the cache partitioning problem translates into allocating each application a buffer pool quota and a storage cache quota in such a way to maximize the service provider’s revenue. Specifically, let’s denote with $r_1, r_2, \ldots, r_n$ the data access latencies of the $n$ applications hosted by the service provider and let $U_i(r_i)$ represent the utility function for the $i^{th}$ application. The goal of the service provider is to maximize the sum of all application utilities i.e.,:

$$\max \sum_{i=1}^{n} U_i(r_i)$$

(4.1)

Finding a practical solution to this problem is difficult, because of the following three reasons:

First, as we have shown in Section 4.2, exhaustively evaluating the application performance for all possible configurations experimentally is infeasible.
Second, effective utilization of the caches depends on several factors, including the (dynamic) access patterns of the applications, the (dynamic) number of applications scheduled on the server farm, and the cache replacement policy used in each cache. Due to the unpredictable impact of these cache and application parameters, implementing an analytical model of performance for guiding the cache partitioning search becomes a daunting task.

Third, accurately evaluating the utility, i.e., benefit gained from an application’s use of its total memory quota within a system component, such as, the database or storage server is non-trivial. Using common cache metrics, such as monitoring the hit rate in each of the two caches is impractical because: (i) the hit rate at the storage cache depends on the behavior of the upper level cache, i.e. the size of the buffer pool and its replacement policy and (ii) their respective access times differ. In more detail, increasing the allocation for an application in the buffer pool usually affects the block accesses seen, hence the hit rate measured, at the storage cache. Moreover, different cache replacement algorithms e.g., LRU versus DEMOTE influence the number and type of accesses seen at the storage cache. Finally, a buffer pool hit is usually more valuable to the application than a storage cache hit, because the storage cache access usually incurs the additional network delay to the storage server. Therefore, simply combining the two cache hit rates for each application does not provide a meaningful overall utility value for memory usage.

4.3.2 Overview of Approach

Our technique determines per-application resource quotas in the database and storage caches, on the fly, in a transparent manner, with minimal changes to the DBMS, and no changes to existing interfaces between system components. For this purpose, we introduce a novel algorithm, called utility-aware iterative learning, which iteratively performs the following two inter-related operations:

1. We build approximate performance models, called application surfaces for mapping cache configuration quotas to the application latency, and its corresponding utility, for
each application, on the fly, and

2. We use the per-application performance models to answer “what-if” cache partitioning scenarios, for any given set of applications, as part of an efficient automatic search for the optimal two-tier cache partitioning solution.

Specifically, we employ statistical support vector machine regression (SVM) [33] for approximating the per-application performance models, based on a set of sample points. Each sample point consists of the application latency for a given cache quota configuration. As sample points are incrementally collected, our algorithm iterates through successive refinement steps, re-approximating the per-application performance models and the optimal solution, until convergence of both the models and the overall optimum occurs.

To achieve this, we augment the DBMS with a resource controller in charge of partitioning both caches between applications. The DBMS cache controller runs our utility-aware iterative learning algorithm to dynamically converge towards a partitioning setting that maximizes the combined application utilities. For each application, the DBMS collects a set of sample points recording the average memory access latency, and its corresponding calculated utility in each cache configuration. In order to speed up convergence, the controller gathers sample points by cache simulation, instead of experimentally. However, the cache controller actuates cache partitioning to better configurations periodically, in order to reduce the penalties incurred by the service provider, whenever the latencies of applications exceed the SLO. When actuating, the controller samples the latency in that cache quota configuration, hence can validate/adjust the respective simulation-based sample point.

In the following subsections, we first describe our performance models in more detail, then introduce our utility-aware iterative learning algorithm.
4.3.3 Performance Models

We dynamically build a per-application latency model, which maps cache partitioning quotas for the respective application to the expected latency. Each per-application model is thus a 3D-surface, of the form \( r_i(q_{i,c}, q_{i,s}) \), or simply \( r_i(q_c, q_s) \), where \( q_c \) and \( q_s \) are the quotas allocated to the application in the buffer pool and storage cache, respectively. Each sample point \( r_i(q_c, q_s) \) is the memory access latency of the application, as estimated by an online simulation of the cache hierarchy. Selective experimental points are also collected to validate or adjust the simulation points.

For the purposes of approximating each function, \( r_i \), based on a set of sample points for that application, we use support vector machine regression (SVM) [33]. Then, for each approximated latency model, \( r_i \), we compute the corresponding utility model as \( U_i(r_i) \), which is another 3D-surface, we call application surface. The application surface represents the service provider’s revenue for hosting application \( i \) for the corresponding application latency function \( r_i \), obtained for different cache configurations \((q_c, q_s)\).

4.3.4 Utility-Aware Iterative Learning Algorithm

For a given set of applications and a cache hierarchy, our goal is to find the cache configuration maximizing the combined application utilities. Towards this, we propose our utility-aware iterative learning algorithm.

In each iteration of our algorithm, we enhance the quality of the latency models described above. Specifically, we employ statistical regression to approximate the per-application latency models, as a set of functions \( r_i \), one such function for each application \( i \), based on a set of latency sample points, \( r_i(q_c, q_s) \), collected for different cache configurations \((q_c, q_s)\). Given the enhanced latency models, we calculate the corresponding application surface for each application. Finally, our goal is to pick from each 3D application surface a single point (i.e., a cache configuration), which:
Algorithm 4.1 Iterative learning for searching the optimal cache partitioning configuration $Q^*$

1: Initialize: $\forall i$, sample set $S_i$ of application $i$, $S_i = \emptyset$
2: repeat
3: for $i = 1$ to $n$ do
4: 1) Add $k$ new samples to sample set $S_i$
5: 2) Use SVM to learn the function $r_i$ using sample set $S_i$
6: end for
7: 3) Map data access latencies $r_i$ to utility values
8: 4) Find $MCU = \max \sum_{i=1}^n U_i(r_i(q_c, q_s))$ for all valid configurations.
9: 5) Actuate to current best configuration $Q^*$ which generates MCU.
10: until Regression error is below a threshold or the MCU value is stable

1. provides $\max \sum_{i=1}^n U_i(r_i)$, and

2. respects the constraint that the sum of the cache quotas (proportions) allocated to applications for each level of cache must be equal to 1.

In order to expedite this search process, we perform a hill-climbing search for the cache configuration settings over the $n$ application surfaces given the requirements above.

Our proposed learning algorithm iterates through successive refinement steps, as more sample points are incrementally added, re-approximating the per-application performance models, as a set of functions $r_i$, as well as the optimal solution, using statistical regression, and hill-climbing, respectively, until convergence of both the models and the overall optimum occurs. The learning algorithm converges when either one of the following conditions occurs: i) adding more sampling points does not increase the accuracy of the regression function i.e., the per-application surfaces vary only within a predefined deviation bound across iterations, or ii) the maximum value of the combined utilities for all applications does not change anymore across iterations, even with increasing the resolution of the regression functions.

Algorithm 4.1 shows the pseudo-code for our iterative learning process. At a given iteration of the algorithm, each application $i$ has a sample set, denoted by $S_i$, initialized to empty (line 1). In each iteration step, for each application $i$, we generate a new set of sample points to expand the current sample set (line 4); we then learn the regression functions $r_i$, based on the current sample set. Based on the regression functions for all applications, we convert
application performance metrics i.e., average memory access latency to the respective utility values (line 7). Next, we employ *hill climbing* to find the maximum combined utility (MCU) value for all valid configurations in the resulting search space (line 8). Next, we actuate to the optimal cache partitioning configuration $Q^*$, which is a set of pairs of cache configurations per application, that is, $Q^* = (q_{1,c}, q_{1,s}), (q_{2,c}, q_{2,s}), ..., (q_{n,c}, q_{n,s})$ (line 9), and we proceed to check for convergence (line 10).

In the following, we describe the main operations in our utility-aware iterative learning algorithm. In particular, we provide the details of steps 1–4 in Algorithm 4.1 above.

**Step 1: Sampling**

We experiment with two methods for generating sample points: (1) *random* sampling, and (2) *greedy* sampling. In *random* sampling, a sample is selected randomly from all possible cache partitioning configurations; every possible sample has an equal chance of being selected. Random sampling is not goal oriented, hence can lead to relatively slow convergence. *Greedy* sampling optimistically predicts that the current optimum found at a given iteration is close to the global optimum. It thus preferentially adds sample points within a gradually increasing radius of the current optimum, seeking rapid convergence. A variant of our *greedy* algorithm is a gradient guided approach, which predicts that the global optimum is on the path with the highest increase in utilities. Hence it adds sample points along the direction with the highest gradient, in order to achieve fast convergence.

**Step 2: Statistical Regression**

For the purposes of approximating each function, $r_i$, based on a set of sample points for that application, we use *support vector machine regression* (SVM) [33]. SVM is a non-linear regression algorithm that is tolerant to measurement errors (small noise) in the sample set, as well as generalizing for the regions that are not sampled. SVM maps the regression problem to a quadratic optimization, finding the optimum solution.
Given a set of training points \( \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\} \), SVM finds a function \( \overline{f}(\vec{x}) \) that has a small deviation (\( \epsilon \)) from the targets \( y_i \) for all training data points. The estimated function \( \overline{f}(\vec{x}) \) takes the form:

\[
\overline{f}(\vec{x}) = \sum_{i=1}^{m} \alpha_i y_i K(\vec{x}_i, \vec{x})
\]  

(4.2)

To build our latency model per-application, each training point \( i \) represents one of the sample points, where \( \vec{x}_i \) is the cache configuration for that point (i.e., \((q_c, q_s)\)) and \( y_i \) is the latency corresponding to that configuration. Each training point \( \vec{x}_i \) is associated with a variable \( \alpha_i \) that represents the strength with which the training point is embedded in the final function. The points which lie closest to the hyperplane, denoting \( \overline{f}(\vec{x}) \), are called the support vectors.

\( K(\vec{x}_i, \vec{x}) \) is a kernel function which maps the input into a high dimensional space, called feature space, where linear support vector regression is applied. We use radial basis functions (RBFs) as our kernel functions.

**Step 3: Mapping Latency to Utility**

The utility function corresponding to the performance of any given application (e.g., [89, 99]) varies since it depends on the contract between the service provider and the client and the costs for the service provider to host the application. Our algorithm does not depend on the exact specification of the utility function. Thus, without loss of generality, for the purposes of this paper, we classify applications in two categories: strict SLO (or high priority) applications and best effort applications.

Figure 4.3(a) depicts the utility function we use for strict SLO applications. For this application class, the provider pays a penalty whenever the application’s SLO i.e., its average memory access latency (denoted as response time in the Figure), is violated beyond a small margin of error called slack. On the other hand, the provider has no benefits for providing service better than the pre-agreed SLO for the application. As shown in the Figure, as long as the
Figure 4.3: Utility functions

application’s response time is less than a deadline $D$ with some slack $\epsilon$, the utility is constant at zero. Beyond this value, the provider starts paying penalties for SLO violations, proportional to the magnitude of the violation, until another threshold $D'$ considered to be unacceptable to the customer.

Figure 4.3(b) shows the utility function for the best effort application class. The provider pays no penalties, regardless of the level of service for an application in this class. Hence, the baseline level of performance with response time beyond $D'$ has the utility value zero. This baseline level would correspond to the application performance for 100% cache miss rates for
any level of cache in our case. However, we assume that performance above the baseline carries a reward for the service provider, which increases proportionally to the level of service until reaching a maximum performance level, after which no more benefits accrue.

**Step 4: Finding the Maximum Combined Utility**

In order to achieve a near-optimal performance, we need to select from each 3D application surface a cache configuration so that the total application utilization is maximized. This results in a combinatorial search space where finding the optimal solution is not feasible. Hence, we use the greedy hill climbing algorithm with random restarts to find the point where the combined utility is the maximum.

**4.3.5 Online Adaptation to Dynamic Changes**

After our utility-aware iterative learning algorithm converges, we obtain accurate per-application surfaces, and the optimal $Q^*$ cache partitioning configuration, for the current set of applications running on the infrastructure. Depending on the type of dynamic change, the entire algorithm, or selected parts of it, may need to be re-executed. For example, if a new application is co-scheduled on the same infrastructure, we need to sample the latency and compute the application surface, only for the new application. Then, we re-compute the new optimum, $Q^*$, cache partitioning configuration by hill climbing, based on the new set of application surfaces. If the access pattern of a given application changes e.g., as detected by significant changes in its miss ratio curve, we need to build a new application surface from scratch for the given application, and recompute the global optimum configuration.

**4.4 Prototype Implementation**

We implement our dynamic cache partitioning algorithm within MySQL and in our Linux-based virtual storage prototype, Gemini. We run a database server using a networked storage
server. The architecture, shown in Figure 4.4, includes a two-level cache hierarchy, consisting of a buffer pool and a storage cache.

MySQL communicates with the virtual storage device through standard Linux system calls and drivers, either iSCSI or NBD (network block device), as shown in the Figure 4.4. NBD is a standard storage access protocol similar to iSCSI, supported by Linux. It provides a method to communicate with a storage server over the network. We modified existing client and server NBD protocol processing modules for the storage client and server, respectively, in order to interpose Gemini modules on the I/O communication path.

In the following, we first describe the interfaces and communication between the core modules, then describe the role of each core module in more detail. Finally, we describe the dual role of the Gemini prototype as an online cache simulator, where the same modules which service an I/O request are used concurrently to explore the configuration space faster and with minimal overhead.
4.4.1 Virtual Storage System

Gemini is a modular virtual storage system which can be deployed over commodity storage firmware. It supports data accesses to multiple virtual volumes and it can interface through Linux with either a storage controller for a RAID system or a single hard disk. Finally, we design a database system plug-in to enable coordination between the database system and the storage server.

Storage clients, such as MySQL, use NBD for reading and writing logical blocks. For example, as shown in Figure 4.4, MySQL/InnoDB mounts the NBD device (/dev/nbd1) on /dev/raw/raw1. The Linux virtual disk driver uses the NBD protocol to communicate with the storage server. An I/O request from the client takes the form <type, offset, length> where type is a read or write. The I/O request is passed by the OS to the NBD kernel driver on the client, which transfers the request over the network to the NBD protocol module running on the storage server.

The storage server is built using different modules. Each module consists of several threads processing requests. The modules are interconnected through in-memory buffers. The modular design allows us to build many storage configurations by simply connecting different modules together.

**Disk module**: The disk module sits at the lowest level of the module hierarchy. It provides the interface with the underlying physical disk by translating application I/O requests to the virtual disk into `pread()`/`pwrite()` system calls, reading/writing the underlying physical data. We disable the operating system buffer cache by using direct I/O i.e., the I/O `O_DIRECT` flag in Linux.

**Cache module**: The cache module allows data to be cached in memory for faster access times. The cache module is portable to different environments by providing a simple hashtable-like interface (modelled after Berkeley DB) supporting `get()`, `put()`, `delete()` and `flush()` operations. It supports different block sizes, dynamic resizing, asynchronous I/O, several cache replacement algorithms and several prefetching policies. For the purposes
of this paper, the cache maintains data as a collection of blocks, implements two cache replacement policies, either LRU or DEMOTE, and manages accesses from concurrent threads. Since MySQL/InnoDB does not support buffer pool partitioning, we embed our caching library into MySQL, replacing MySQL’s buffer pool manager. The server cache is located on the same physical node as the storage controller. The two instances of the cache module create a two-tier cache hierarchy.

**NBD Protocol module:** We modify the original NBD processing module on the server side, used in Linux for virtual disk access, to convert the NBD packets into our own internal protocol packets, i.e., into calls to our Gemini server cache module.

### 4.4.2 Gemini Simulator

The Gemini system has two modes: normal and simulation. In simulation mode, in addition to run the system normally under its current setting, it also estimates the performance of the system under other settings through simulation. The goal of the simulation is to explore the search space of cache partitioning settings for any given cache replacement technique employed by our prototype.

**GEMINI** system records the most recent page access trace collected at the level of the MySQL buffer pool, as the input of the simulator. The GEMINI simulator consists of a trace module and two cache simulators. The trace module replays the most recent page access trace file. The cache simulator uses the same cache code, as described for the caching module. But the I/O requests dispatched by the storage cache simulator does not direct to the real disk, instead we simulate the disk delay and network delay by advancing the virtual clock of the simulator with the average respective access delays measured on-line. More detailed disk simulation, such as employed in DiskSim [17] is not necessary in our system because we use the simulation only to guide convergence by estimating the performance under promising configurations.

We validate our simulation by comparing the predicted latency with the measured latency
when running in a specific configuration. As we will show in Section 4.6.8, the dual nature of the Gemini code allows for very accurate estimates, where the latency predicted by the simulator is within 5% of the latency measured by the runtime system.

4.5 Evaluation

In this section, we describe several cache partitioning algorithms we use in our evaluation, as well as the benchmarks and our evaluation platform.

4.5.1 Algorithms used in Experiments

We implemented a prototype of our utility-aware iterative learning algorithm (Section 4.3), which we will call DYNAMIC and compared it to the following schemes:

**CONSERVATIVE:** We take the conservative approach and allocate both the buffer pool and the storage cache to the high-priority application. To the low-priority application, we allocate only a minimum cache quota, such that its data accesses can still occur i.e., 32MB in our implementation, and dedicate the rest of the cache space to the high-priority application.

**PROFILE:** We profile each application offline to determine the amount of buffer pool it needs in order to meet its SLO. We assign each application the respective amount of buffer pool cache, whereas the storage cache is shared among all applications with no-quota enforcement. Hence, this scheme is SLO-aware, however, it is oblivious to the presence of the second-level cache.

**MRC:** A miss-ratio curve (MRC) estimates the page miss ratio for an application given a particular amount of memory. It has been applied to effectively allocate memory to several applications [108]. We extend MRC for the purpose of partitioning a two-level cache hierarchy. Specifically, at the buffer pool level, we partition the buffer pool using the MRC computed for each application. Similarly, at the storage cache level, we partition the storage cache by building an MRC for each application using its missed data accesses (i.e., accesses that are not
satisfied by the buffer pool cache).

**IDEAL**: We perform offline experiments iterating through all possible partitioning configurations of the two caches and choose the setting which maximizes the revenue.

**SHARED**: We allow applications to share both the DBMS buffer pool and the storage cache with no quota enforcement.

**DYNAMIC**: This is our cache partitioning scheme described in Section 4.3.

### 4.5.2 Benchmarks

We use two industry-standard benchmarks, TPC-W and RUBiS, to evaluate our proposed algorithm.

**TPC-W**$^{10}$: The TPC-W benchmark from the Transaction Processing Council [92] 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. We use the *shopping* workload that consists of 20% writes. To fully stress our architecture, we create TPC-W$^{10}$ by running 10 TPC-W instances in parallel creating a database of 40 GB.

**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 distinguish between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users are only allowed to browse. During a buyer session, in addition to the functionality provided during the visitor sessions, users can bid on items and consult a summary of their current bid, rating, and comments left by other users. 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.
4.5.3 Evaluation Platform

We run our Web based applications on a dynamic content infrastructure consisting of the Apache web server, the PHP application server and the MySQL/InnoDB (version 5.0.24) database storage engine. We run the Apache Web server and MySQL on Dell PowerEdge SC1450 with dual Intel Xeon processors running at 3.0 Ghz with 2GB of memory. MySQL connects to the raw device hosted by the NBD server. We run the NBD server on a Dell PowerEdge PE1950 with 8 Intel Xeon processors running at 2.8 Ghz with 3GB of memory. The storage uses a direct-attached SAS enclosure with 15 10K RPM 250GB hard disks configured to use RAID-0. We install Ubuntu 6.06 on both the client and server machines with Linux kernel version 2.6.18-smp. We configure our caching library to use 16KB block size to match the MySQL/InnoDB block size.

4.6 Results

In this section, we present an experimental evaluation of our multi-tier cache allocation technique. We conduct experiments on our prototype storage system to evaluate the performance of our approach. We use two applications: TPC-W as the strict SLO (high-priority) application, and RUBiS as the best-effort application. We express the SLO in terms of average memory access latency. A memory access latency SLO of less than $500\,\mu s$ provides an average query response time below 500ms for both our benchmarks, which closely approximates values used as QoS for the two e-commerce applications in previous studies [85]. Thus, in our utility function, we set $D = 500\,\mu s$, $D' = 3500\,\mu s$ (the average disk access time), $U_{\min} = -100$ and $U_{\max} = 100$. In addition, we experiment with relaxing the SLO by varying the slack ($\epsilon$) from 10% ($D + \epsilon \leq 550\,\mu s$) to 100% ($D + \epsilon \leq 1000\,\mu s$). We explore the effects of different application access patterns and the effect of different replacement policies on the optimal cache partitioning. We use 1 database server and 1 storage server, each configured with a 1GB cache.
Chapter 4. Fine Grained Resource Partitioning of the Cache Hierarchy

4.6.1 Latency Surfaces

Figure 4.5: Latency surfaces: Memory access latency for different partitionings of buffer pool and storage cache.
In Figure 4.5, we show the latency surface of two applications: (1) TPC-W, and (2) RUBiS. The figure shows the memory access latency for different settings of the buffer pool size and the storage cache size. The light areas indicate configurations with high memory access latency, whereas the dark areas indicate configurations with low memory access latency. These applications have varying working sets. TPC-W, having a small working set, obtains low data access latencies even with small allocations of cache space. RUBiS, with a larger working set, requires more cache space in the cache hierarchy to obtain low data access latencies.

### 4.6.2 Latency under LRU/LRU

In Figure 4.6, we compare the performance of the different cache partitioning schemes when both the database buffer pool and the storage cache use the LRU cache replacement policy. Figure 4.6(a) shows that, under the **SHARED** and the **MRC** schemes, the SLO of our high-priority application (i.e., TPC-W) is violated. For instance, the average memory access latency of TPC-W under **SHARED** is $715\mu s$, as opposed to the pre-specified SLO of $500\mu s$. This is mainly because both schemes are oblivious to the SLO requirements.

On the other hand, both the **CONSERVATIVE** and the **PROFILE** schemes satisfy the SLO requirements of TPC-W. However, both schemes over-allocate cache resources to TPC-W to the detriment of the best-effort application (RUBiS). Between the two schemes, RUBiS performs worse under **CONSERVATIVE**, which allocates all the available cache to TPC-W. In contrast, under **PROFILE**, RUBiS achieves a better performance, since **PROFILE** allows RUBiS to share the storage cache with TPC-W.

The **IDEAL** scheme, and similarly our **DYNAMIC** scheme, are both able to strike a fine balance between satisfying the TPC-W SLO requirement while providing an acceptable performance to RUBiS. This is simply due to the fact that under **LRU/LRU**, the storage cache typically includes blocks already contained in the database buffer pool. Thus, there is no additional benefit for an application, if its partition in the storage cache is smaller than its partition in the buffer pool. Since both **CONSERVATIVE** and **PROFILE** are oblivious to this
This is wasteful, since TPC-W does not derive any additional benefit from the storage cache.
allocation, while the respective allocation of storage cache could have been of a significant benefit if allocated to RUBiS. Our DYNAMIC scheme dynamically recognizes this trade-off and it detects that more revenue is achievable if the storage cache is allocated to RUBiS. Thus, by accurately computing the overall utility function, DYNAMIC chooses a near-optimal cache partitioning setting, where most of the database buffer pool is allocated to TPC-W (the high-priority application) and most of the storage cache is allocated to RUBiS (the best-effort application). With near-optimal settings, using DYNAMIC, we reduce the latency of RUBiS to 1193µs (versus 1844µs for PROFILE).

4.6.3 Revenue under LRU/LRU

The gains provided by our DYNAMIC scheme are even more prominent when the provided latencies are mapped to the corresponding revenues, as shown in Figure 4.6(b). The figure also shows that with larger slack, we are able to further increase revenue. For instance, DYNAMIC increases the revenue from 43 (with 0% slack) to 87 (with 100% slack). This increase is achieved by reducing the RUBiS memory access latency from 1193µs to 612µs. On the other hand, the PROFILE scheme is unable to take advantage of the slack to the same degree. For example, there is no additional revenue generated when the slack is 0%, and the revenue generated with larger slacks is significantly lower than the revenue generated using the DYNAMIC scheme.

4.6.4 Latency under LRU/DEMOTE

In Figure 4.7, we repeat the previous experiment using the LRU/DEMOTE scheme, where the database buffer pool informs the storage cache of block evictions, and the storage cache uses the DEMOTE cache replacement policy. The DEMOTE policy maintains exclusiveness between the database buffer pool and the storage cache. Thus, the DEMOTE scheme results in a better utilization of the overall cache hierarchy, leading to both TPC-W and RUBiS obtaining lower
latencies even when in isolation, compared to the LRU/LRU case.

Under SHARED, both applications compete for the cache space, causing TPC-W to incur
higher cache misses at both the buffer pool and the storage cache; this in turn leads to an average memory access latency of $615\mu s$ for TPC-W, which is 23% higher than the pre-specified SLO. The fact that the best effort RUBiS is doing well under this scheme does not matter since the provider incurs substantial loss for violating TPC-W SLO.

For fairness of comparison, we modify the MRC algorithm to support DEMOTE policy and we use our modified MRC algorithm to allocate the memory to applications. Specifically, the MRC algorithm analytically derives the miss-ratio curve by tracking cache contents using an LRU stack. Upon a read/write request, it moves the accessed block to the top of the LRU stack. In the presence of DEMOTES, to model the policy correctly, we modify MRC so that to place blocks referenced in a DEMOTE request to the top of the LRU stack instead of blocks referenced in I/O reads. Finally, I/O writes are handled the same for both LRU and Demote cache policies. Under our modified MRC, the average memory access latency for TPC-W is within the SLO, while the RUBiS latency is still higher than our DYNAMIC scheme ($1095\mu s$ under MRC vs. $903\mu s$ under DYNAMIC).

While, the CONSERVATIVE and the PROFILE algorithms maintain TPC-W’s latency within the SLO, they over-provision the cache resources, leading to high latencies for RUBiS, $3508\mu s$ for the CONSERVATIVE scheme and $1476\mu s$ for the PROFILE scheme. The PROFILE scheme allocates the buffer pool to TPC-W assuming that the storage cache provides no additional benefit. While this assumption is true for the LRU/LRU layout, it is false for the LRU/DEMOTE layout, where by using the DEMOTE algorithm, the storage cache provides a significant benefit to TPC-W. Hence, the PROFILE scheme provides a memory access latency of $376\mu s$, even while profiling to meet the $500\mu s$ SLO.

By accurately modeling the effect of two-tier caching, our DYNAMIC scheme selects a near-optimal partitioning setting, where TPC-W is allocated enough buffer pool space such that it meets its SLO. With this allocation, the TPC-W latency is within the SLO and the RUBiS latency is $903\mu s$. 
4.6.5 Revenue under LRU/DEMOTE

With larger slack, as shown in Figure 4.7(b), we can further reduce RUBiS latency, from \(903 \mu s\) to \(284 \mu s\), thereby increasing the revenue from 74 (with 0% slack) to 95 (with 100% slack). The PROFILE scheme also generates higher revenue compared to the LRU/LRU layout, due to higher utilization of the storage cache. However, the DYNAMIC scheme provides a higher revenue than the PROFILE scheme. Hence, integrating our cache partitioning DYNAMIC scheme with the DEMOTE coordinated cache replacement policy provides further revenue improvements that are not achievable using DEMOTE with other comparison schemes.

4.6.6 Performance under Overload Scenario

To better understand the improvement in performance achieved by DYNAMIC, we experiment with an overload scenario where two high-priority applications are scheduled. Specifically, we use two instances of the high-priority TPC-W application (denoted A and B) sharing the database and storage cache, thus creating an overload case, where the available resources are not sufficient to meet the SLO, given no slack. In this case, no additional revenue can be generated, and all schemes simply strive to minimize the losses. With two equally high priority applications, the CONSERVATIVE, PROFILE, and MRC schemes divide the database buffer pool and the storage cache equally (50/50) among the two TPC-W instances. Under the LRU/LRU layout, this leads to an average memory access latency of \(833 \mu s\), while our DYNAMIC scheme matches the IDEAL by obtaining an average memory access latency of \(743 \mu s\). DYNAMIC achieves this improvement by dynamically selecting an optimal cache configuration, which exploits the inclusiveness in LRU/LRU.

To provide an insight into the optimal partitioning, in Figure 4.8, we show the revenue function. The \(x\)-axis shows the fraction of the storage cache given to application A and the \(y\)-axis shows the fraction of the buffer pool given to application A. Since only two applications are running, Application B is given the remaining cache space. Figure 4.8 shows the revenue
Figure 4.8: Overload: We show the total revenue for TPC-W/TPC-W for several configurations with the light regions showing “high” revenue and the dark regions showing “low” revenue. We also highlight the optimal cache partitioning settings.
for different cache partitioning settings. The “low” revenue settings are shown in dark colors, and the “high” revenue is shown in light colors. The contour lines highlight the near-optimal settings. For example, as shown in Figure 4.8(a), the LRU/LRU layout has two optimal configurations. One optimal setting (top-left of the figure) is when Application A is given most of the buffer pool and a small fraction in the storage cache. The other optimal setting (bottom-right of the figure) is when Application B is given most of the buffer pool and very little of the storage cache.

If LRU/LRU is used, then the storage cache only provides a marginal benefit to the application given a large proportion of the database buffer pool. Thus, the plot shows that the optimal setting is achieved when the buffer pool is allocated to one application (A or B), and the storage cache allocated to the other (B or A). On the other hand, using the LRU/DEMOTE scheme, the storage cache benefits both applications equally leading to an optimal partitioning of 50/50 (Figure 4.8(b)).

### 4.6.7 Sampling Convergence

In Figure 4.9, we compare the speed of convergence of two sampling strategies: (1) greedy sampling and (2) random sampling. In greedy sampling, we gather samples near the currently found optimal configuration. In random sampling, we select a set of random samples at each iteration. The benefit of greedy sampling is intuitively in potentially faster convergence towards the optimal cache configuration.

In Figure 4.9, the x-axis shows the number of samples selected for our statistical regression, and the y-axis shows the deviation from optimal. The deviation from optimal is the difference in revenue by using the estimated optimal partitioning, as opposed to the revenue generated using ideal cache partitioning. Initially, with a small number of samples, both the greedy approach and the random approach are far from optimal. However, after 64 samples, the greedy approach starts converging to the optimal, reached with only 160 samples (on average). On the other hand, the unguided random sampling converges only after 352 samples. Our sampling
approach is efficient; we can collect 350 samples in simulation within 30 minutes. During this period of time, on average, two actuations take place, hence two experimental latency points are also collected.

### 4.6.8 Simulation Accuracy

We also evaluate the accuracy of our simulations by comparing the predicted latency with the measured latency when running in a specific configuration. In this experiment, we ran several configurations from small caches (64MB) to large caches (1GB). For each configuration, we compared the predicted latency obtained from simulation and the measured latency by running our prototype system. In all configurations the predicted latency is within 5% of the measured latency.
4.7 Summary

In order to reduce the costs of management, power and cooling in large data centers, operators co-schedule several applications on each physical server of a server farm connected to a shared network attached storage. Determining and enforcing per-application resource quotas on the fly in this context poses a complex and challenging resource allocation and control problem due to (i) the strict Quality of Service (QoS) requirements of many database applications, (ii) the unpredictable resource needs and/or access patterns of applications in modern environments with dynamic application co-scheduling and (iii) the existing interdependency between tiers, such as the effects of cache replacement policies on application patterns at different levels.

Our contribution in this chapter is to introduce a novel approach for controlling application interference in the cache hierarchy of shared server farms. Specifically, we design and implement a technique for partitioning the buffer pool and storage caches adaptively, online; a cache controller embedded into the DBMS actuates the partitioning of the two caches with the goal to dynamically converge towards a partitioning setting that minimizes the perceived application penalties. At the same time, the controller allocates any spare resources to best effort applications in order to maximize the revenue for the service provider.

Our method is implemented in a Linux-based prototype, called Gemini, which requires minimal DBMS instrumentation, and no changes to existing interfaces between commodity software and hardware components. Our experimental evaluation shows the effectiveness of our technique in enforcing application SLOs as well as maximizing the revenue of the service provider in shared server farms. In contrast, all other techniques we evaluated suffer from either violations of the SLO requirement of strict SLO applications, missed revenue opportunities, or both.
5.1 Introduction

As we have seen in the previous chapters of this dissertation, server consolidation allows the datacenter provider to amortize costs by transparently hosting several applications on a set of shared resources. However, the sharing of the underlying physical resources across multiple applications makes it very difficult to guarantee performance goals to applications.

As previously discussed, towards addressing this problem, in this dissertation, we build adaptive performance models on the fly; we leverage our performance models for dynamic resource allocation in large data centers. A performance model is a mathematical function that calculates an estimate of the application performance for any possible resource configuration. For example, a model can provide an estimate of the average latency of an application, given certain memory and disk bandwidth fractions.

In the previous chapters, we have shown successful applications of our modeling approach to coarse grained resource allocation for backend databases, shown in the Chapter 3; and also on fine grained resource partitioning of the cache hierarchy, shown in the Chapter 4. Our
previous studies used individual models for parts of the system. We estimate that a collection of such individual performance models, collected over time, forms a solid knowledge base supporting the sysadmin in finding the appropriate accuracy-speed trade-off points for various workloads and user requirements.

Towards offering this level of support, we have built a general and flexible performance modeling framework, called Chorus. Chorus accumulates, searches and extends previously learned models and uses them for providing an estimation of the performance for each application, given certain resource configurations.

In this chapter we first discuss the challenges that model selection and evolution for a collection of models entail. Next, we introduce the Chorus framework and its implementation details. We then present an evaluation of the Chorus functionality for storing and refining previously learned models. We show how our previously introduced cache hierarchy model from Chapter 4 can be extended to model the whole storage hierarchy, including disk bandwidth. We finally show how we applied this extended storage hierarchy model to the problem of dynamic resource allocation.

**Challenges with Choosing the Appropriate Model**

Performance models can be evolved either analytically or by using a black-box approach. The trade-offs between these two approaches are as follows.

With the existing profiling and monitoring tools [66, 86], an analytical performance modeling approach would likely require cumbersome application profiling for the insight of the system administrator to blend into a high dimensional surface modeling all resources. Moreover, modeling the performance of the memory hierarchy may reward with a relatively simple, monotonically decreasing surface, as previously shown in Figure 1.2 for RUBiS. For a high-dimensional surface, modeling CPU, memory, and disk resources of multiple tiers, however, well-behaved properties of application surfaces, such as monotonicity, or convexity are not guaranteed.

It is therefore highly unlikely that a system administrator would ever have the expertise and
stamina to either (i) define a sufficiently accurate single analytical end-to-end model for the system, or (ii) meaningfully, and resource consciously stitch together various analytical models to form an end-to-end model. For highly-varied resources or highly complex workloads, deriving an analytical model may not be feasible; however, for simple cases, where an accurate analytical model of the system is available, this is the ideal solution in terms of modeling speed for a desired accuracy.

On the other spectrum, black-box models have the advantages of generality, and can be easily applied in various conditions. However, it is a time intensive proposition. Specifically, we need to dynamically actuate the system with different configurations, and experimentally gather performance samples, then interpolating the gathered performance samples using statistical regression. The time spent on sampling may take weeks, even months, depends on the range of the resource configurations. In addition, the sysadmin’s domain knowledge about the system is completely lost in black-box models due to the lack of any manual sysadmin intervention in modeling.

With Chorus, we make the assumption that the sysadmin or analyst may know relatively simple models with good enough accuracy for parts of the configuration space, or for modeling the operation of certain resources in isolation. For example, the analyst may know that when the application is disk bound, the relation “disk latency varies inverse exponentially with the disk bandwidth quanta” applies. A similar function may apply for the cases where the application is CPU bound.

Figure 5.1 shows these scenarios visually; the application’s behavior is classified into different operating modes (i.e. I/O intensive, CPU intensive and mix modes) for different memory allocations (rows) and disk bandwidth fractions (columns). On the other hand, the analyst may or may not be able to give a rough estimate about what parts of the configuration space correspond to a disk bottleneck, or a CPU bottleneck, for each application. Experimental sampling would still be required for regions of the resource configuration space where no model is known.
Figure 5.1: Different operating modes of an application. The application varies from I/O-intensive (in gray and in top left corner) and CPU-intensive (in black and in bottom right corner) as the amount of memory and disk resources are varied. The area in the middle (in white) is the mixed mode.

We design a runtime system for our Chorus framework, which takes high level guidelines, such as, baseline models, whether black-box, analytical, or gray-box, and automatically optimizes the experimental sampling and total modeling time. By validating with experimental sampling and ranking the different models on the fly, Chorus automatically finds the two shaded configuration regions in Figure 5.1, where the simpler CPU bound, and disk bound analytical models fit. Overall, this helps speed-up modeling convergence by focusing the sampling to in-between areas where no predefined model fits.

Chorus uses model guides such as the above CPU bound model, wherever available, or its repository of models derived previously for other applications. Chorus thus incrementally tunes and blends approximate, incomplete models together over time into increasingly refined performance models for each application. The level of expertise of the analyst, the validity and accuracy of known model relations may affect the convergence time towards eventual prediction accuracy of Chorus modeling. Conversely, Chorus provides a starter hands-on training tool about the system and application for the sysadmin or analyst.
We evaluate our Chorus through extensive experiments, we run the industry standard TPC-C benchmark, which models a wholesale part supplier, the OLTP-A storage workload, and the e-commerce TPC-W benchmark. We show case studies where Chorus seamlessly reuses existing approximate models, and incrementally refines them on the fly. Specifically, our experimental evaluation shows that Chorus (i) can integrate the expertise of a system administrator, an analyst, or other statistical, and historical knowledge base as approximate model guidance, (ii) dynamically selects the most accurate modeling techniques for different configuration regions, (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.

5.2 Chorus Design

Chorus is an interactive framework for learning new performance models for applications, on the fly, and extending existing models of applications. The design of Chorus takes into account that large datacenters environments are dynamic: hosted applications, and application workload mixes for each application change over time; system components may change as well e.g., as a result of system upgrades. On the other hand, while many workloads may be hosted on the same infrastructure within some period of time, their behavior may be similar.

To address these challenges and opportunities, Chorus is designed for incremental, interactive, efficient learning of new models and/or refinement of old models through intelligent guided sampling of the resource configuration space. Towards this, Chorus contains two interrelated components: (i) SelfTalk: 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: a runtime for validation of models with experimental data, and for maintenance, inquiry, and reuse of a modeling knowledge base.

In the following, we introduce an overview of the Chorus high-level declarative interface
and the Chorus operation.

### 5.2.1 Chorus Overview

Our key idea is that some basic information about the structural semantics of the infrastructure, common application flows through its multiple tiers, and old models derived for previous workloads are known. This old knowledge about parts of the system can then be composed, refined or reused in new situations. For example, an old model for the storage system can be later reused by refitting the storage latency curve with samples from a new application. Cases where the application workload mix changes, but the primary resource bottleneck for that application doesn’t change would typically require even fewer model refitting samples. Finally, even more intelligent, selective sampling may be possible if a priori expert knowledge about a specific application, or the inner working principles of components exists. However, ideally, the expert should not need to accurately specify which resource impacts which application configuration region, and which models apply where in a highly dimensional configuration space.

#### Chorus Operation

Chorus uses human high-level guidance and semantic tags for storing, retrieving, and evolving a repository of performance models learned under various dynamic conditions.

Specifically, a sysadmin may propose a learning template to Chorus tagged with the application name and/or human intelligible name(s) of the part(s) of the system and the specific metrics the learning task is related to. Chorus gathers samples and trains a model based on the guidance. Chorus also validates the model and develops an accuracy score, based on the fit to experimental data. Once a template proposed by an analyst becomes a validated model, this model is stored in the Chorus knowledge base and can later be searched and reused.

Thus, for each long-running application, the Chorus modeling knowledge base contains derived models for parts of the configuration space e.g., for modeling the latency of the cache hierarchy, or for modeling the latency of the storage system; it also contains previously gath-
ered samples for the same application. Model reuse can be effective in reducing sampling
time in new situations, such as, when the application workload mix changes, or when resource
availability changes e.g., new applications are deployed on the system. Model reuse can also be
helpful when a part of the system is added, replaced or upgraded e.g., an SSD-device becomes
part of the storage hierarchy, or replaces an existing hard drive.

Chorus uses semantic tags e.g., application or component name, and a set of features pro-
vided by the analyst to store and search models in its modeling knowledge base. It keeps
the tags and models as (key,value) pairs into the database. Chorus retrieves models through
matching the target tag with keys . More advanced information retrieval methods [57] can be
used to index and retrieve models if the model knowledge space becomes large. Our thesis
concentrates on modeling language, methods, framework and applications. Hence, an in-depth
discussion of model indexing and retrieval in Chorus is beyond the scope of this study. Instead
we focus on presenting the novel tool-box we use in Chorus for retrofitting or combining old
models to/in new situations. These tools include the following.

- **Model calibration, refinement and extension**: Chorus adds new samples incrementally
  and selectively to tune model parameters, and extend the range of configurations and
  workload mixes used for prediction in an old model.

- **Model ensemble**: Chorus ranks various possible models and automatically selects the
  best fit for any given configuration range, workload, and time/accuracy trade-off.

- **Semantic-aware inquiry**: Chorus answers queries about application modeling accuracy,
  validated models for any given configuration regions, and what-if scenarios, in a seman-
tically meaningful way.

### 5.2.2 SelfTalk Model Templates

We introduce *SelfTalk*, a high-level, SQL-like declarative language for model guidance and
inquiry in Chorus. *SelfTalk* 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 5.1 (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. In context, configuration ranges for particular resources can be specified, or left as empty.

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. Let’s look at an example of a learning task for modeling the latency of the multi-tier server system shown in Figure 1.1. The platform consists of a MySQL database server, and a network attached storage server. It hosts several concurrently running applications. To avoid interference among these applications, a proportion-share scheduler in the database server allocates CPU time in proportion (denoted by CPU quanta) to their respective CPU share. Similarly, a quanta based disk scheduler in the storage server allocates disk bandwidth in proportion (denoted by disk quanta) to each application. As a result of the resource scheduling, the performance of each application in
this platform is mainly controlled by its assigned resource quanta. For any given application, the analyst may provide the simple template shown in Listing 5.2 to Chorus. Specifically, a performance model, called CPU_Bound, is given to the Chorus run-time to learn i.e., gather experimental samples, validate, curve-fit, find the configuration settings where the relation applies, if any, and compute confidence scores and error rates.

The relation is Inverse_Exponential, which is a pre-defined mathematical relationship in Chorus, implemented as follows:

$$\hat{y}_{\alpha, \beta}(x) = \frac{\alpha}{x^\beta}$$  \hspace{1cm} (5.1)

The parameters $\alpha$ and $\beta$ are estimated by Chorus. This relation represents that the memory latency is inverse exponential with the allocation of the cpu quanta. This model is declared sensitive only to one parameter: the memory size 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 “*”.

Within any template, Chorus can refer standard math relations, such as, linear, exponential, inverse, machine learning algorithms, as well as non-standard 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.
5.2.3 SelfTalk Sampling Guidelines (Clues)

We define a clue as a condition-action guideline regarding sampling actions for the Chorus runtime. The clue indicates areas of the search space which can be, or should be pruned when sampling, because they are: already known, known to be noisy, or known to be invalid configurations.

For instance, from our experience, whenever the disk quanta given to an application is too small e.g., less than 32ms (about 0.1 fraction of the total disk quanta) in our storage server, the disk latency shows unacceptable instability. This is due to the lack of sufficient disk access time for I/O burst. As a result, these small quanta are not allocated in real system. Hence we wrote a clue for Chorus to avoid sampling configurations with small disk quanta as shown in Listing 5.3. This clue guides Chorus to take Prune action once the fraction of the allocated disk quanta is less than 0.1. Prune action is a pre-defined action in Chorus, which prunes the configuration space specified by the condition. As a result of this clue, about one-tenth of the sampling time is reduced during the modeling process. In general, analysts can use clues to prune un-interesting configurations, where their application could perform poorly, e.g., they can write clues to prune small cache size configurations for a memory intensive workload.

5.2.4 SelfTalk 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 any given application, SelfTalk allows the sysadmin to issue inquiries regarding the error rates or confidence score of any pre-defined model. For example, he/she can ask for all available memory sizes where the pre-defined CPU_Bound model registered average relative error rates of below 20% for a given application. This inquiry is shown in Listing 5.4.

The analyst tagging of system parts, attributes and contexts with explicit semantic meaning in the model templates are thus helpful for specifying either focused or generic inquiries about modeling tasks. For example, the accuracy of all throughput models for all configuration ranges, or the throughput of specific components e.g., just for the MySQL database engine, and specific configuration ranges e.g., for buffer pools greater than 500MB, can be inquired into. Finally, the feedback reflected back to the analyst is semantically meaningful, because it refers to either generic or focused inquiries with analyst recognizable semantic tags, such as names of measurement units, or relevant component names.

### 5.2.5 Operation of the Chorus Runtime System

For each template provided, Chorus expands it into a performance model, by checking the validity of the template with monitored historical data, or live experimental sampling. New sampling is performed only as needed in the configuration regions that match the specification, or for the resources the model is sensitive to. Where a mathematical relation is given, Chorus automatically finds the appropriate parameters for curve fitting, and computes error rates for the accuracy of fit. Whenever the accuracy for a template fulfills the analyst’s requirement within the given contexts, Chorus stores the template as a model together with its error rates in its modeling knowledge base.
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 semantic help 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. It is slow to build, but generically and automatically covers all scenarios where no simpler analytical or gray-box model provides sufficient accuracy.

Thus, both the dialogue between system and administrator in SelfTalk 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.

Finally, Chorus 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 new situations by leveraging samples taken from adjacent areas of the configuration space and/or for models with similar features.

### 5.2.6 Chorus Summary

Chorus accumulates a modeling knowledge base, as a library of standard, or learned models and their contexts, models that can be later queried, refined or reused to build new models. Chorus automatically adjusts model parameters by fitting over old and new data. It reuses and blends models together into incrementally built more complex and more accurate models. From an ensemble of archived, or newly provided models, Chorus automatically determines the most accurate model per resource configuration region. Chorus can give semantically meaningful feedback to its administrator about the accuracy of any model, or its predictions within any region of the resource configuration space and application.

Overall, the benefits of this approach are: (i) faster convergence towards an accurate model, (ii) tunable and extensible modeling over time, as new models are provided, new configurations are sampled, new workload mixes dynamically occur, and (iii) facilitating analyst hands-on
learning and gathering insights about the system and application based on meaningful feedback from Chorus.

5.3 Prototype Implementation

Our prototype of Chorus is implemented in a combination of C and Python. It consists of multiple components: a frontend parser to understand users’ inputs, a modeling engine, a profiling engine, as well as a database backend. The analyst or sysadmin can specify model templates, clues or inquiries into files. The language parser translates them into an internal representation, and then the modeling engine conducts the modeling operations online. The profiling engine controls the setting of the resource quota or quantum through resource controllers; it is also responsible for collecting data samples from applications, under specific resource configurations. The backend database stores the sampling data and information of validated models (e.g. model relation, model parameters, and model accuracy for specific workloads, etc.) for later retrieval. Next, we explain our prototype implementation in detail.

5.3.1 Model Ensemble

We assume that a number of model templates have been provided for Chorus to learn/validate within a configuration space \( C \). We focus on describing how Chorus trains and ranks performance models for \( C \). For this purpose, Chorus assumes that the configuration space is divided into multiple regions, controlled by a configurable parameter \( D \), which defines the number of divisions along each dimension. This results in dividing the configuration space into \( D^N \) regions, where \( N \) is the number of resource dimensions.

Algorithm 5.1 shows the learning process in Chorus. 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 region within \( C \). Our samples are gathered on the live system by actuation into the desired configuration and taking
Algorithm 5.1 Iterative algorithm to build an ensemble of models for a configuration space $C$.

1: **Initialization:**
2: Select a collection of performance model templates $M$
3: Divide configuration space evenly into $l$ regions
4: Select $v$ samples to construct test sample set $S_v$.
5: Training set $S_t = \emptyset$, $m = \text{size}(M)$
6: **Iterative Training:**
7: repeat
8: /* Expand the training sample set */
9: Add $t$ new samples to the training sample set $S_t$.
10: /* Build ensemble of models */
11: Set ranking set $W = \emptyset$.
12: Partition the training set $S_t$ into $k$ subsets.
13: for $i = 1$ to $k$ do
14: 1) Use $i^{th}$ subset as validation set $S'_v$
15: 2) Use other $k - 1$ subsets as training set $S'_t$
16: for $j = 1$ to $m$ do
17: 3) Train each base model $M_j$ on $S'_t$
18: 4) Test $M_j$ on $S'_v$
19: end for
20: end for
21: Derive rank per region from cross validation results
22: Build an ensemble from the rank
23: Test the ensemble of models on $S_v$
24: until stop conditions are satisfied

several measurements of the application latency.

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. At each iteration, we use the samples in the training set to construct our ensemble model. To do this, we further partition the original training set into two sets: a build set for refining baseline models and a validation set for ranking them. Once the models are built, we test their prediction accuracy using the validation set. We use a standard technique $k$-fold cross validation for the ranking process to avoid over-fitting, where $k$ is 5 in our implementation.
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. 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.

5.3.2 Model Extension and Reuse under Dynamic Changes

We observe that if a workload change is not significant, the performance surface (model) for that workload will likewise register little change. Chorus can thus adapt to workload changes through the reuse of old models, and old sampled data. Towards this, Chorus monitors several features of the workload, such as, the read/write ratio for data accesses, outstanding I/Os, the size of the working set, etc. The similarity of workloads is defined by the Euclidean distance in their feature space.

The process of model extension consists of the following two steps. First step is to search potential similar models from the Chorus knowledge base. We select potential historical models from the Chorus repository, based on a similarity distance check in their feature space to the current workload. The second step is to selectively refine models per region. We gather new sample data through experiments for the new workload. Then, we compute the prediction accuracy per region. If the accuracy of a region is satisfied, we accept the corresponding model and will use its prediction for this region. Otherwise, Chorus refines models for each region whose accuracy is not met. Chorus iteratively gathers new samples in these regions to calibrate and rebuild models for them using the ensemble algorithm just described, until the overall target accuracy is met.
5.4 Evaluation

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

5.4.1 Benchmarks

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

OLTP-A: OLTP-A is a set of OLTP-like workloads we generate using ORION (Oracle I/O Calibration Tool) [67]. 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 accesses 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 [92] 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\textsuperscript{10} by running 10 TPC-W instances in parallel creating a database of 40 GB.

TPC-C: The TPC-C benchmark [75] 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.

### 5.4.2 Server Platform

Our server platform is inspired by a Cloud service, Amazon Relational Database Service (Amazon RDS) [1]. 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 [63, 84, 96], in order to relieve interference among concurrent workloads.

![Diagram](image)

Figure 5.2: 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 5.2, 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 extends the functionalities of the storage server Gemini which we built in section § 4.4.1. It 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.

### 5.4.3 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 are stored in files on disk.

## 5.5 Cast Studies of Model Templates

In this section, we show several case studies of using model templates. Our analyst provides the following model templates in our SelfTalk language. Since she is uncertain which model can reach high accuracy for new workloads, she uses Chorus to validate them on the fly.

### 5.5.1 Analytical Model Template (A-STOR) for Memory Latency:

Shown in the Listing 5.5, the analytical model template A-STOR is tagged by semantic information through variable names (e.g. `memory_latency`) and component names (e.g. `mysql`) in the metric sets. These semantic tags are useful for semantic-aware inquiries about models in Chorus. 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. This relation, which is implemented as an analyst-defined plug-in program, is validated by the Chorus run-time, just like any standard relation. In the following, we explain the derivation of this `Analytical_Mem_Latency_Pred` relation.

**Multi-level caching relation:** In a multi-level cache hierarchy using the standard (uncoordinated) LRU replacement policy at all levels, any cache miss from cache level `i` will result in bringing the needed block into all lower levels of the cache hierarchy, before providing the requested block to cache `i`. It follows that the block is redundantly cached at all cache levels, which is called the inclusiveness property [102]. Therefore, if an application is given a certain cache quota `ρ_i` at a level of cache `i`, any cache quota `ρ_j` given at any lower level of cache `j`, with `ρ_j < ρ_i` will be mostly wasteful. Based on these observations, we make the following simplifications to approximate the overall miss-ratio of a two-level cache, i.e., `\( \hat{M}(ρ_c, ρ_s) \), based on a
single-level cache model.

In an uncoordinated LRU cache hierarchy, only the maximum size quota given at any level of cache matters; therefore, we approximate the miss-ratio of a two level cache, consisting of a buffer pool (with quota $\rho_c$) and a storage cache (with quota $\rho_s$) by the following formula

\[
\hat{M}(\rho_c, \rho_s) \approx M_c(\max[\rho_c, \rho_s])
\] (5.2)

**I/O scheduler relation:** Our storage system, Akash, uses the quanta-based scheduler to divide the storage bandwidth among several virtual volumes. The quanta-based scheduler partitions the bandwidth by allocating a time quantum where one of the workload obtains exclusive access to the underlying disk. For modeling the quanta latency, we observe that the typical storage server system is an *interactive*, closed queueing network. This means that, the number of users in the system is constant during periods of stable load. Then, according to the *interactive response time law* [42]:

\[
L_d = \frac{N}{X} - Z
\] (5.3)
where \( L_d \) is the response time of the storage server, including both I/O request scheduling and the disk access latency, \( N \) is the number of application threads, \( X \) is the throughput of the I/O requests, and \( Z \) is the think time of each application thread issuing requests to the disk. We then use this formula to derive the average disk access latency for each application, when given a certain fraction of the disk bandwidth. We assume that think time per thread is negligible compared to request processing time, i.e., we assume that I/O requests are arriving relatively frequently, and disk access time is significant. Then, through a simple derivation, we arrive at the following formula

\[
L_d(\rho_d) = \frac{L_d(1)}{\rho_d}
\]

(5.4)

where \( L_d(1) \) is the *baseline disk latency* for an application, when the entire disk bandwidth is allocated to that application, and \( \rho_d \) is the allocated fraction of disk bandwidth (i.e. disk bandwidth quanta). This formula is intuitive. For example, if the entire disk was given to the application, i.e., \( \rho_d = 1 \), then the storage access latency is equal to the underlying disk access latency. On the other hand, if the application is given a small fraction of the disk bandwidth, i.e., \( \rho_d \approx 0 \), then the storage access latency is very high (approaches \( \infty \)).

**Overall Memory Access Latency** – Analytical_Mem_Latency_Pred : This relation assumes that the hit access latency in the buffer pool is negligible; the overall memory hierarchy latency is determined by the accesses that miss in the buffer pool and either (i) hit in the storage cache or (ii) miss in the storage cache, hence access the disk. It also assumes that the access latency for a hit/miss in the storage cache is approximately the network/disk latency, i.e., \( L_{net}/L_d \), respectively. Therefore, the average memory latency \( \mathcal{L}_{mem} \) is approximated by the formula:

\[
\mathcal{L}_{mem}(\rho_c, \rho_s, \rho_d) = \frac{\mathcal{M}_c(\rho_c)\mathcal{H}_s(\rho_c, \rho_s) L_{net}}{\text{I/Os satisfied by the storage cache}} + \frac{\mathcal{M}_c(\rho_c)\mathcal{M}_d(\rho_c, \rho_s) L_d(\rho_d)}{\text{I/Os satisfied by the disk}}
\]

(5.5)
where $\rho_d$ is the allocated fraction of disk bandwidth; and $\rho_c, \rho_s$ are the buffer pool, storage cache quota allocated to the application. The miss and hit ratio at the storage cache, i.e., $M_s(\rho_c, \rho_s)$ and $H_s(\rho_c, \rho_s)$, is a function of both the quota at the first level cache ($\rho_c$), and the quota at the second level cache ($\rho_s$), while the miss-ratio of the buffer pool, $M_c(\rho_c)$, is only a function of $\rho_c$.

We can further approximate the fraction of accesses that miss in both levels of cache, hence reach the disk, i.e., $M_c(\rho_c)M_s(\rho_c, \rho_s)$ as

$$M_c(\rho_c)M_s(\rho_c, \rho_s) = \tilde{M}(\rho_c, \rho_s) \hspace{1cm} (5.6)$$

By using the previously derived models for $\tilde{M}(\rho_c, \rho_s)$ in the case of uncoordinated LRU (Equation 5.2), we obtain

$$M_s(\rho_c, \rho_s) = \frac{M_c(\max[\rho_c, \rho_s])}{M_c(\rho_c)} \hspace{1cm} (5.7)$$

Therefore, we can approximate the miss-ratio in the storage cache, $M_s(\rho_c, \rho_s)$, in terms of the miss-ratio of a single-level cache. To derive miss/hit ratio for a single level cache, we conduct page access trace collection at the database buffer pool and compute the miss ratio curve (MRC) using Mattson’s Stack algorithm [58] for the database buffer pool. Finally, by replacing the respective miss and hit ratio of the storage cache in Equation 5.5, we derive the overall memory access latency.


5.5.2 I/O Intensive Query Model Template (A-STOR-Q)

The I/O-intensive query model is a simple analytical model designed for I/O intensive workloads for predicting the query latency. This model does not consider the time at the CPU because if a workload is I/O-bound then, most of the query processing time is in fact spent on waiting for I/O completion. In this model, our analyst leverages the query selectivity, which we obtain using our own statistics of the number of page accesses (i.e. $N_{acc}$) to the database buffer pool.

The query latency model is therefore:

$$L_{query}(\rho_p, \rho_c, \rho_s, \rho_d) = N_{acc} \times L_{mem}(\rho_c, \rho_s, \rho_d)$$  \hspace{1cm} (5.8)$$

where $\rho_p, \rho_c, \rho_s, \rho_d$ is the CPU, buffer pool, storage cache and disk bandwidth quota allocated to the application; $N_{acc}$ is the average number of page accesses made for each query in the workload; and $L_{mem}$ is the average memory access latency for each page as derived in Equation 5.5.

5.5.3 Gray-box Inverse Model Template (G-INV)

The analyst is aware that our storage server uses a quanta based scheduler to proportionally allocate the disk bandwidth among multiple applications. Larger fraction of the disk bandwidth allocated usually leads to the shorter delay of requests. Hence, the analyst uses an inverse model template based on the relation Inverse_Exponential, which is a standard mathematical relationship automatically supported by the Chorus model template library.

5.5.4 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.

5.5.5 Black-box SVM Regression Model Template (B-SVM)

Chorus uses a black-box model template to cover all scenarios where no simple analytical or gray-box model (e.g. curve fitting model) provides sufficient accuracy. In this case study, Chorus uses a well-known machine learning algorithm: Support Vector Machine regression [33] (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 (RBFs) are used as kernel functions.

5.5.6 Black-box Constant Model Template (B-CNST)

This model uses a simple average relation which returns the average value of all training samples to predict performance. The predicted latencies are same for all configurations. Due to this, it usually performs poorly since the actual performance surface is not constant. In contrast, G-RGN uses average function for each region, and hence the prediction values are usually different in different regions. We use B-CNST to evaluate that whether a bad model template with high errors affects the performance of the Chorus.

5.6 Results

In this section, we first show how Chorus validates models dynamically, and builds an ensemble of models with high accuracy. We then show the benefits of using semantic guidance through Chorus for speeding up the sampling process; we further show how Chorus extends models for workload mix changes.
5.6.1 Build Models 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 and TPC-C workloads, running on our server platform. Based on historic training, Chorus has accumulated the following model templates in its model knowledge base: A-STOR, G-INV, G-RGN, B-SVM, B-CNST. Chorus uses these model templates in its knowledge base, without any further sysadmin guidance to learn for modeling the memory latency of the two workloads.

Modeling TPC-W Memory Access Latency

We vary the size of the DBMS buffer pool from 128M to 960M with 15 settings, the storage cache size from 128M to 896M with 8 settings. 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. The size of the testing set is 10% of the original set size. The number of regions used on each resource dimension is 4, hence the whole configuration space is divided into 64 regions.

Figure 5.3 presents our results. On the x-axis, we 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 an error of 23%. 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. Chorus performs well matching the A-STOR model initially then incorporating the better predictions of B-SVM with more training time. The results are fur-
Chapter 5. Chorus: Model Knowledge Base for Performance Modeling

Figure 5.3: Performance prediction of memory access latency for TPC-W workload. Chorus initially matches the A-STOR model, and then incorporates the better predictions of B-SVM model.
ther supported by Figure 5.3(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.

Predicting TPC-C Memory Access Latency

We vary the size of the DBMS buffer pool from 128M to 960M with 15 settings, the storage cache size from 128M to 896M with 10 settings, and the disk bandwidth quanta from 32ms to 256ms with 8 settings. The training time of exhaustive sampling is about 13 days for 1200 configurations. The size of the testing set is 10% of the original set size. The number of regions used on each resource dimension is 4, hence the whole configuration space is divided into 64 regions.

Figure 5.4 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. 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. 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 5.4(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
Figure 5.4: Performance prediction of memory access latency for TPC-C workload. Chorus initially replies on B-SVM model and later frequently selects gray-box models G-INV and G-RGN for prediction.
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\textsuperscript{10}, and also that there is no single model that works best for all time deadlines.

### 5.6.2 Build Models for Predicting Query Latency

Next, we evaluate our Chorus approach in predicting query latency. We model the DBMS query latency, for two workloads, TPC-W\textsuperscript{10} and TPC-C, running on our server platform. Chorus validates and builds the ensemble of the following model templates in its model knowledge base: A-STOR-Q, G-INV, G-RGN, B-SVM, B-CNST.

#### Predicting TPC-W\textsuperscript{10} Query Latency

We vary the size of the DBMS buffer pool from 128M to 960M with 15 settings, the storage cache size from 128M to 896M with 8 settings, and the disk bandwidth quanta from 32ms to 256ms with 8 settings. The training time of exhaustive sampling is about 10 days for 960 configurations. The size of the testing set is 10% of the original set size. The number of regions used on each resource dimension is 4, hence the whole configuration space is divided into 64 regions.

Figure 5.5 presents the results. The black-box B-SVM model performs well for this dataset. After about 10 hours of the training, the average relative errors quickly drops to about 15%. G-INV takes a longer training time (about 30 hours) to start predicting well. Chorus mainly matches the prediction of the B-SVM model, and then later it incorporates some better predictions from G-INV model. On the other hand, the analytical model A-STOR-Q model performs worse than B-SVM model, with a high average error rate of 43%. The composition of Chorus is shown in Figure 5.5(c). The B-SVM model contributes most to Chorus as it is ranked as the best model in more than 40/64 regions all the time. The G-INV model achieves slightly higher accuracy than B-SVM model after gathering sufficient training samples; as a result, it is ranked as the best model with more than 10 regions after 35 hours.
Figure 5.5: Performance prediction of query latency for TPC-W \(^{10}\) workload. Chorus mainly matches the prediction of B-SVM model, and later incorporates the better predictions of G-INV model.
Predicting TPC-C Query Latency

We vary the size of the DBMS buffer pool from 128M to 960M with 15 settings, the storage cache size from 128M to 896M with 10 settings, and the disk bandwidth quanta from 32ms to 256ms with 8 settings. The training time of exhaustive sampling is about 13 days for 1200 configurations. The size of the testing set is 10% of the original set size. The number of regions used on each resource dimension is 4, hence the whole configuration space is divided into 64 regions.

Figure 5.6 shows the results. The black-box model B-SVM performs well initially, and hence Chorus matches its performance. After about 20 hours, Chorus gradually outperforms B-SVM with lower error rates. This is due to that the grey-box model G-RGN starts to offer better predictions in many regions, and contributes more to Chorus. From Figure 5.6(c), the composition of Chorus shows that it mostly selects B-SVM model or G-RGN model as the best ranked model per region for predictions. Through effectively combining the merits of both models, Chorus outperforms each individual one. On the other hand, analytical model A-STOR-Q does not perform well with a high average error rate of 77%, and is not selected into the composition of Chorus. A secondary, but important point is that from the figure we can see that no individual model always wins for all time; and hence it is crucial to dynamically validate each model.

5.6.3 Using a Complex Pruning Clue based on Expert Knowledge

An example of expert knowledge that helped Chorus prune the configuration space significantly is as follows. The analyst knows that the database buffer pool cache and storage cache in our server platform, use standard (uncoordinated) LRU replacement. Due to the cache inclusiveness property of LRU caches [84], she knows that any cache miss from the buffer pool results in the block being cached into both caches; the cache content is essentially replicated in both caches. Therefore, assuming the hit latency in either cache is about the same, whenever one of
Figure 5.6: Performance prediction of query latency for TPC-C workload. Chorus initially matches the prediction of B-SVM model, and soon incorporates the better predictions of G-RGN model.
the caches is larger than the other cache, by at least some threshold value, the smaller cache is redundant, hence irrelevant. Based on this knowledge, the analyst can specify a cache pruning clue to use the same experimental sample point (e.g. a measured latency value) for a range of cache configurations that are roughly equivalent to its configuration by the above rule. For instance, using this clue, the latency value for configuration (768MB in buffer pool, 256MB in storage cache) can be approximated using the latency value measured at the configuration (768MB in buffer pool, 384MB in storage cache).

Table 5.1 shows the effects of applying this cache pruning clue. The time reduction refers to the modeling time for the memory access latency for the TPC-W\textsuperscript{10} workload. If the threshold for pruning is 256MB, the modeling time is reduced by 47.3% without any accuracy loss. The modeling average error rate is 12% as the result without applying this clue. If the threshold is 0MB, the pruning causes a 6% increase in the average error rate, but at a dramatic modeling time reduction (83.6%).

<table>
<thead>
<tr>
<th>Threshold (MB)</th>
<th>256</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Reduction</td>
<td>47.3%</td>
<td>83.6%</td>
</tr>
<tr>
<td>Error Rate Increase</td>
<td>0%</td>
<td>6%</td>
</tr>
</tbody>
</table>

We apply the same clue for modeling TPC-C memory latency and results are shown in Table 5.2. If the threshold for pruning is 256MB, the modeling time is reduced by 46.4%. The modeling average error rate increases by 10% comparing with the error rate (20%) without applying the pruning clue. If the threshold is 0MB, the pruning causes 14% increase in error rate, but at a dramatic modeling time reduction (90.6%).

<table>
<thead>
<tr>
<th>Threshold (MB)</th>
<th>256</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Reduction</td>
<td>46.4%</td>
<td>90.6%</td>
</tr>
<tr>
<td>Error Rate Increase</td>
<td>10%</td>
<td>14%</td>
</tr>
</tbody>
</table>
5.6.4 Model Extension and Reuse

Chorus can extend the range of configurations and workload mixes used for prediction in an old model. Here, we give examples of reusing old models when the workload mixes are changed using OLTP-A benchmark. Specifically, we design a series of experiments to show the impact of reusing historical models when the write ratio of data accesses is changed. In detail, initially an old model trained for the workload with the write ratio of $a$ has been saved into the Chorus model repository, and then the write ratio of data access of the workload decreases to $b$. The Euclidean distance which reflects the similarity between the new and the old workload is denoted by variable $\text{dist}$, 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 5.7 shows the results. x axis lists a series of tests with the same $\text{dist}$. 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 normalized to the time without reusing the old model. For example, 0.2 means the modeling time is 20% of the time which models this new workload completely from scratch.

When the similarity distance $\text{dist}$ is 0.1, shown in Figure 5.7(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 $\text{dist}$ to 0.2, shown in Figure 5.7(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.

5.6.5 Examples of Resource Allocation Using Chorus

In this section, we show three examples of using the Chorus performance models that we build in Section 5.6.1, for allocating resources in our server platform. Here, the goal of the resource allocation is to minimize the sum of memory access latencies for all applications.
Our Chorus resource allocation scheme first builds a performance model for each application, using the Chorus model knowledge base. Similarly as the allocation algorithm we used in Section 4.3.4, Chorus allocation scheme exploits hill climbing algorithm with random restarts, to find the resource partitioning setting which gives the optimum i.e., lowest combined latency in our examples. The hill climbing algorithm is an iterative search algorithm that moves towards the direction of decreasing combined latency value for all valid configurations at each iteration. To avoid sticking into a local optimum, we conduct several searches from different starting points chosen randomly, and use the resource allocation setting that yields the best
Figure 5.8: Examples of resource allocation for running four identical instances.

We compare the performance results (i.e. the sum of the latencies) using Chorus resource allocation scheme that we described above with the following allocation schemes:

MRC scheme uses miss ratio curve (MRC) to partition cache independently at the buffer pool and the storage cache, based on access traces seen at that level. The disk bandwidth is equally divided among all applications.

DISK scheme assigns equal portions of the cache to all applications at buffer pool and the storage cache and explores all the possible configurations at the disk level.
Equal scheme assigns equal portions of the resources for each application. If the running instances are identical, then MRC scheme and DISK scheme will have the same allocation result as the Equal scheme.

Ideal* scheme finds the configuration with the best overall latency by an exhaustive search through all resource configurations. This scheme is not feasible in practice, since it needs to exhaustively sample all configurations and may take weeks or even longer to finish.

We first run four instances of the TPC-W\(^{10}\) on our server platform, sharing the database and the underlying storage server. We set the size of both the buffer pool and the storage cache as 1GB; and the disk bandwidth is partitioned among these instances. As shown in Figure 5.8(a), Equal has the worst performance with the overall latency 21.36\(\text{ms}\) since it blindly partitions each resource by equal share, without considering the performance impact. In contrast, Chorus uses the TPC-W\(^{10}\) performance models (with 16 hours of training) to guide its allocation decision. As a result, the overall latency is 14.78\(\text{ms}\) by Chorus scheme, very close to the Ideal* scheme (13.66\(\text{ms}\)).

We observe similar trends for running four TPC-C instances on our server platform with the same resource constraints as the TPC-W\(^{10}\) allocation experiments. Shown in Figure 5.8(b), Equal performs worst with the highest overall latency 16.40\(\text{ms}\); and the resource allocation scheme using Chorus models for TPC-C (with 64 hours of training) achieves the total average latency 5.82\(\text{ms}\), almost as low as the Ideal* results (4.91\(\text{ms}\)).

We also run two TPC-W\(^{10}\) instances and two RUBiS\(^{10}\) instances on our server platform with the same resource constraints as previous allocation experiments. As shown in Figure 5.9, MRC and DISK both perform poorly with high overall latency (above 28\(\text{ms}\)) because both schemes only target to optimize local resource allocations. The resource allocation scheme using Chorus models achieves the same performance as the Ideal* result (20\(\text{ms}\)) because Chorus builds performance models for end-to-end resources and targets to achieve global optimum; and hence outperforms MRC scheme and DISK scheme. All of the above examples show that Chorus modeling framework can effectively support sysadmins to find
near-optimum resource allocation.

![Image](image_url)

Figure 5.9: Examples of resource allocation for running two TPC-W\textsuperscript{10} Instances and two RUBiS\textsuperscript{10} instances

## 5.7 Summary

We design, implement and deploy Chorus, a novel interactive runtime system and knowledge base for modeling the performance of datacenter applications. Our key idea is to provide an interactive high-level environment, where both the modeling system and its administrator can communicate semantically meaningful information about the overall model and its parts. Specifically, the sysadmin or the analyst 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, and blends models together to incrementally construct an ensemble of models. Through a set of case studies, we show that Chorus can successfully validate, extend and reuse existing models under new situations.
Chapter 6

Related Work

Our dissertation work is built on previous research in several areas: techniques for resource partitioning and allocation in datacenters, techniques for modeling application performance, and techniques for designing semantic languages for systems. We will describe the related works in each category in this chapter.

6.1 Server Platforms for Dynamic Resource Allocation

We focus on introducing the advancement of solving dynamic resource allocation problems for multi-tier server systems. In this section, we describe the development of server platforms for resource multiplexing in this recent decade.

6.1.1 Shared Server Pool Platform for Web Applications

In the late nineties, many web sites typically use separate server pool for each application. Various web applications, for instance, search, email, are hosted on dedicated server pools. Similarly, organizations often run separate server pool for internal clients and for external clients. While this approach achieves performance isolation, it typically results in lower average utilization of servers and potential higher average request latencies, because resources that are not
currently utilized by one application cannot be used by others.

Therefore, large datacenters gradually move to host multiple applications, such as e-commerce, auctions, news, search, email, games, concurrently in a shared server pool. If web sites experience daily patterns with peak loads for each application occurring at a different time, hardware resources can be re-assigned from one application to another. Thus, instead of gross over-provisioning for each application’s estimated peak load, the web service provider can efficiently multiplex datacenter resources across applications through dynamic resource allocation, i.e., on-demand provisioning. Several research works [11, 90, 93, 99] investigate dynamic provisioning of resources within the (mostly) stateless web server and application server tiers. In their designs of datacenter [11, 90, 99], a local resource manager (or application manager) collects local measurements, computes local utility functions, and cooperates with the global resource controller (or resource arbiter) to implement server redeployments. Their experiments are conducted on the web server tier and application server tier, with the assumption that the database tier is not replicated.

However, dynamic resource allocation among applications in the stateful database tier, which commonly becomes the bottleneck [2], has received comparatively less attention. Next, we describe the replication and resource provisioning related literatures for the database tier.

6.1.2 Replicated Database Cluster

Web server and application server can be easily replicated since most of them track very few stateful information, and the time of warming up a server is comparatively short. However, bottleneck in database tier is complex to relieve since it has strict consistency requirement and working sets of many enterprise workloads are usually large. On the other hand, database replication is crucial for relieving hotspots of databases that are very common in e-commerce or online audition sites. In this section, we introduce the recent advancements in replicated database clusters.
Conflict-Aware Scheduler

Amza et al. propose a lazy read-one, write-all replication algorithm for database replicas to reduce conflict waiting time [3]. A middleware tier, i.e. conflict-aware scheduler, is introduced between application server clusters and database clusters, and transparent to them. When the scheduler receives a read query, it delivers the request to one of the lightest load replicas which have finished all previous operations in conflict with the read. When the scheduler receives a lock, write or commit, it must dispatch the request to all replicas. A response is sent back as soon as the scheduler receives a result from any replica. A unique sequence number is assigned by a global sequencer to each transaction when the first lock request of the transaction arrives. Each database replica maintains a local queue to ensure conflicting transactions are executed in sequence order, and thus 1-copy serializability is assured. A limitation of this scheme is the use of conservative two-phase locking which, while avoiding deadlocks, may severely limit concurrency. The authors’ following paper [4] integrates distributed versioning with a conflict-aware scheduler to improve performance. Open source software C-JDBC [18] implements this conflict aware scheduler following JDBC standard so that it supports any database engine through any vendor supplied JDBC driver.

Other Database Replication Schemes

Kemme and Alonso implement Postgres-R [49], an eager replication protocol, by applying group communication. With Postgres-R, a transaction normally executes locally. Read queries are executed locally; for a write query, after its local execution, a writeset is extracted and multicasted to all replicas through group communication primitives.

With the popularity of Snapshot Isolation (SI), which provides weaker consistency than 1-copy serializability, several database replication schemes are proposed to support SI. Plattner and Alonso propose Ganymed [73], a master/slave replication scheme. It has a scheduler that assigns a global database version number to all transactions. Instead of sending write transactions to all replicas, it only sends them to the master replica. Then, the master replica
extracts the writesets, and sends them to slave replicas. Recently, Lin et al. [54] propose a middleware based replication scheme SI-Rep providing SI without a master node, through group communication primitives.

In Chapter 3, we use the Conflict-Aware Scheduler [3] as our database replication infrastructure, and develop our resource provisioning project based on it. The principle of our pro-active resource provisioning algorithm can also be applied on other database replication schemes.

### 6.1.3 Fine Grained Resource Multiplexing Platform

Resource multiplexing techniques we introduced in previous sections are conducted on coarse grained level, where the unit of the resource provisioning is a single server. To further utilize resources and reduce costs of ownership, larger service providers now run many concurrent applications on the same physical server. Hence, effective fine grained resource multiplexing techniques are crucial in order to reduce performance interruption among co-located applications. Next, we introduce these techniques, with an emphasis on CPU, Memory and Disk multiplexing.

**CPU Scheduling/Throttling:** The task of properly allocating the CPU has traditionally been the responsibility of the operating system (OS). Within the OS, many algorithms, such as round robin, priority, shortest job first (SJF), lottery scheduling [98], and multi-level feedback queues have been extensively studied and implemented in variety of operating systems, such as Linux, BSD, and Microsoft Windows. More recently, virtual machines monitors (VMM), e.g., Xen [9], have implemented their CPU schedulers within the hypervisor to provide performance isolation for their virtual hosts. However, the OS is unaware of the application performance thus it relies on other metrics such as instructions-per-cycle (IPC) to determine the relative benefit of different CPU allocations. This issue has been addressed by commercial database vendors. Hence commercial databases e.g., Oracle [78] and Microsoft SQL Server implement CPU resource allocators within the database server. They allow the database administrator to
specify CPU limits per application. However, the database systems do not automatically adjust CPU allocation across applications and do not provide allocations across multiple servers.

**Dynamic Memory Partitioning and Management:** Dynamic memory allocation has been studied in the VMWare ESX server [97]. The algorithm estimates the *working-set* sizes of each VM and periodically adjusts each VM’s memory allocation.

The area of adaptive cache management based on application patterns, or query classes has been extensively studied in database systems. For example, the DBMIN algorithm [28] uses the knowledge of the various patterns of queries to allocate buffer pool memory efficiently. The LRU-k [65], and its variant 2Q [45] cache replacement algorithms prevent useful buffer pages from being evicted due to sequential scans running concurrently. Brown et al. [15, 16] study schemes to ensure per-class response time goals in a system executing queries of multiple classes by sizing the different memory regions. Recently, IBM DB2 adds the self-tuning memory manager (STMM) to adjust the size of different memory regions [87]. However, the above works target only the memory regions within the DBMS. In our study, we have shown that optimally partitioning multi-tier caches results in significant performance gains.

Several works pass explicit hints from the client cache to the storage cache [21, 52, 70]. For example, these hints can indicate the reason behind a write block request to storage, and whether a block is about to be evicted from the client cache and should be cached at the storage level [52], explicit demotions of blocks from the storage client to server cache [102], or the relative importance of requested blocks [26]. These techniques modify the interface between the storage client and server, by requiring that an additional identifier be passed to the storage server. As opposed to our work, these techniques need thorough understanding of the application internals and changes to the kernel API and the storage protocol. For example, Li et al. [52] require the understanding of database system internals to distinguish the context surrounding each block I/O request. Similarly, Wong et. al [102] require the addition of a DEMOTE command to the SCSI protocol.

Transparent and gray-box techniques for storage cache optimization include inferring ac-
cess patterns of the upper tier by observing characteristics of I/O requests [8, 27, 46], or using meta-data available at the file system layer. Schindler et al. investigate ways for providing the DBMS with more knowledge of the underlying storage characteristics [79]. The drawback of these techniques is that they are DBMS-specific or specific to the storage hardware. This may not be feasible in a datacenter.

**Disk Bandwidth Partitioning:** Large datacenters may contain petabytes or more of data in their storage system. Recently, direct-attached storage (DAS) have been consolidated to store data from different applications/clients. In order to provide performance isolation among them, dynamic allocation of the storage bandwidth have been studied to provide QoS at the storage server. SLEDS [19], Façade [55], SFQ [44] add a scheduling tier above the existing storage devices in order to control the I/Os issued to the underlying storage device. Specifically, SLEDS takes periodic performance samples, and delays I/Os from overly-demanding clients whenever other clients experience inadequate performance. It uses a leaky bucket filter to shape and throttle I/O flows. Façade converts QoS requirement to the latency deadline of each I/O requests, and uses an Earliest Deadline First (EDF) I/O scheduler for dispatching packets. It utilizes surplus resources to improve service quality for active I/O flows. Jin et al. [44] uses the adaptation of SFQ (Start-time Fair Queuing) algorithm to approximate proportional sharing of the storage device. These works all assume that the cost of each I/O is known in advances; however, this knowledge is usually difficult to acquire. The I/O cost may vary abruptly as a result of disk seek and the interference from concurrent I/O requests. Argon [96] and our own work [84] propose to use a quanta based scheduling algorithm for partitioning the disk bandwidth. It gives each workload a quantum of time during which it uses the disk exclusively. Our work shows it offers strong performance isolation among workloads since it minimizes the effect of potential large disk seeks between two concurrent workloads, and hence greatly reduces the inter-application interference in the storage tier.

**Multi-resource Partitioning:** Multi-resource partitioning is an emerging area of research where multiple resources are partitioned to provide isolation and QoS for several competing
applications. Wachs et al. [96] show the benefit of considering both cache allocation and disk bandwidth allocation to improve the performance in shared storage servers. However, the resource allocation is done after modeling applications through extensive profiling. Chanda et al. [20] implement priority scheduling at the web and database server levels. Wang et al. [101] extend the SFQ [44] algorithm to several storage servers. Padala et al. [69] study methods to allocate memory and CPU to several virtual machines located within the same physical server. However, these papers focus on either (i) dynamic partitioning and/or quota enforcement of a single resource on multiple machines [20, 101] or (ii) allocation of multiple resources within a single machine [69, 96]. In our own work in the area of dynamic partitioning, we have investigated either partitioning memory, through a simulation-based black-box search approach [83] (introduced in Chapter 4), or partitioning storage bandwidth, through an adaptive feedback-loop approach [81], or a mathematical model based approach [84] for partitioning cache and storage bandwidth together. Finally, our Chorus (introduced in Chapter 5) provides a flexible framework for modeling the performance impact of multiple resources across servers.

### 6.2 Algorithms and Models for Dynamic Resource Allocation

Many algorithms are proposed for allocating resources in large datacenters. As we mentioned in Chapter 3, Soundararajan et al. [82] use a reactive resource provisioning method to allocate resources for replicated database cluster. Their algorithm adds only one database replica at a time when the SLO violation is detected. If the SLO is still broken after the new addition, the resource manager will add one more replica; otherwise, it will stabilize using the current resource allocation. Zheng et al. [103] also propose a reactive provisioning approach to manage web servers. However, the reactive approach cannot decide how many resources are needed in advance. It relies on the "trial-and-error" feedback loop for monitoring the application’s performance and deciding their next action. As a result, its resource provision actions may
behave too conservatively and respond slowly to the load spike.

Since it is time-consuming to evaluate and find proper resource configurations through the "trial-and-error" approach, the pro-active model based approach, introduced in Chapter 3, is favoured for providing prompt resource allocation. It relies on a performance model to decide how many resource it needs to allocate for meeting the SLO requirement. Base on the prediction of the workload density and the performance model of this workload, it can allocate resources in appropriate amount quickly. Existing techniques for building accurate performance models range from analytical models \([11, 93, 94, 80]\), to black-box models based on machine learning algorithms \([35, 100, 107]\). There are also gray-box models in related studies \([91, 106]\).

### 6.2.1 Analytical Model based Resource Allocation

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 examples of advanced analytical queuing models for specialized cases: Uysal et al. derive an analytical throughput model for modern disk arrays \([94]\). Their model is built on a hierarchical decomposition of the relevant parts of the disk array’s internal architecture. The relative errors in predictions are 15% on average for common cases, and this level of the accuracy has been shown to meet their requirements for provisioning storage systems. Doyle et al. \([32]\) model web applications serving static web content using a queueing model. Their model connects a CPU server queue to a storage server queue, with a cache component in between to absorb a portion of requests. They assume the requests follow a Zipf-like popularity distribution, while this assumption breaks for dynamic web objects. More sophisticated queueing models are developed by Urgaonkar et al. \([93]\) for modeling web servers. They present a model based on a network of queues, where the queues represent different tiers of applications. Each queue represents a service tier and the underlying server that it runs on. Their model works well under light load, however, fails to handle very diverse workloads (e.g. I/O intensive databases).
Kounev and Buchmann [51] show Queueing Petri Net (QPN) models can be used for modeling the performance of multi-tier server system. QPN can model simultaneous resource possession, synchronization, blocking and contention for software resources. However, due to its complexity, QPN related solution methods are still needed to be developed in order to use QPN for performance analysis and prediction in real systems. Jung et al. [48] provide an end-to-end dynamic resource management solution for multi-tier applications in virtualized environment, using layered queuing networks. They model Apache, Tomcat, and MySQL as M/M/n queues, and model the workload as a set of independent open Poisson processes, one for each transaction type. Their models are used to estimate the CPU utilization of each tier for a given workload. The parameters for models are measured in an offline measurement phase. Their approach has been shown to effectively improve CPU utilization and is recently extended to improve power efficiency [47].

Soror et al. [80] study the resource allocation problem for running database systems on multiple virtual machines, without considering the storage resource. They 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. They need to use actual performance measurements to correct the cost models’ inaccuracies. Various query monitoring techniques [56, 61] estimate query latency based on detailed statistics information of queries, such as query plan, cardinality, number of groups, for current resource configuration.

It is common that analytical models often have regions of configurations in which they work well and regions in which their assumptions break and they do poorly. Lack of the robustness to workload changes and resources availability could lead analytical models to produce inaccurate prediction. This brittleness severely prevents them from being trusted by many system practitioners [88, 91].
6.2.2 Black-box Model based Resource Allocation

Due to the inherent brittleness of analytical models, some system researchers and practitioners believe that it is almost impossible to predict the performance of complicated server workloads (e.g. large databases) under dynamic changes, before actually trying them [34, 105]. As a result, experiment driven sampling approaches are often considered to provide the most accurate results. However, because the resource configuration space for a multi-tier application is multi-dimensional (i.e. relying on multiple resources) in large datacenters, leading to combinatorial explosion in the number of possible configurations/experiments. An exhaustive sampling approach may take several months to build a complete performance model [84] for a database application running on a shared network attached storage (NAS).

Therefore, many statistical models are proposed to enhance purely experimental approaches either through optimizing the sampling process [34] or through applying various machine learning methods to extrapolate [29, 35, 41, 83]. With a few exceptions [88, 91], these models do not make a priori assumptions on system structure and workload behavior, and consider the whole system as a "black-box". With sufficient time of gathering samples as training data, statistical models will approach the accuracy of purely experimental methods. Hence, they are more robust than their analytical peers and can be applied to general workloads.

For instance, Wang et al. use a machine learning model, Classification And Regression Trees (CART), to predict performance for storage device [100]. It requires no knowledge of the device internals, and predicts per-request response times and aggregated values as a function of input workloads. Their results show the relative error of their prediction is as low as 19% in most of cases. Recently, Ganapathi et al. [35] propose Kernel Canonical Correlation Analysis (KCCA) to predict execution time for database queries. They are able to predict individual query execution time within 20% of its actual time for 85% of the test queries for decision support benchmark TPC-DS. Their further study suggests that the same technique can be applied for map-reduce jobs with the customization of the feature vectors for new system. Black-box model is also applied in sampling techniques. iTuned [34] proposes an adaptive
sampling method that proactively brings in appropriate data through planned experiments to find high-impact parameters and high-performance configurations. It automatically selects experimental samples guided by utility functions and uses Gaussian process Representation of a response Surface (GRS) to build their performance model.

Despite of many advantages of statistical models, their efficiencies are not satisfying especially when performance surfaces are bumpy or noisy. Under these conditions, in our experience, they usually require to conduct many experiments in order to gather sufficient training data, and it leads to a long training period before producing high-fidelity predictions.

### 6.2.3 Hybrid Approaches based Resource Allocation

To overcome the disadvantages of analytical models and black-box models, hybrid approaches [91, 107] have been recently studied. Zhang et al. [107] extend their previous black-box model approach, Tree-Augmented Bayesian Networks (TAN) models [29], and use the ensemble of TAN models to capture the performance behavior of systems under changing workloads and conditions. They shown that an ensemble of such models can achieve rapid adaptation to workloads changes, infrastructure changes, and external disturbances. As another example, IRONModel [91] uses a series of analytical models, including CPU model, network model, buffer pool model and disk model, as its expectation-based models. These models on their own are not accurate due to the complexity of caching algorithms, etc. Hence, it uses a decision tree based machine learning model, called Z-CART, to continuously tune the parameters for these analytical models designed for their storage system, and refine their predictions with time.

While our approach has some similarities with these approaches, Chorus (introduced in Chapter 5) provides a general modeling framework for sysadmins to express their domain knowledge, and uses the ensemble learning for automatically validating various models per region. Chorus is flexible to incorporate common semantic information, and high-level guidance in a human intelligible query language to its automated runtime system. The runtime system leverages different types of models, each of which may work well in a different operating
mode, to provide an ensemble of models that can make robust performance predictions in the whole configuration space. In addition, our runtime system conducts intelligent sampling and selective model refinement based on sysadmin guidance and historical information.

### 6.3 Semantic Languages for Performance Analysis

SelfTalk [36] is a declarative language that allows analysts to query and understand the status of a large scale system. Analysts use it to pose hypotheses about system behavior and query its run-time to validate system behavior. In Chapter 5, we extend SelfTalk to the area of the performance modeling. We exploit model templates to express performance models and system structure information, and use inquiry to validate these models. We further propose clue into SelfTalk to guide Chorus run-time engine to trim configuration space. These new features enable SelfTalk to be a two-way communication tool between sysadmins and our performance modeling run-time engine.

Other semantic language based approaches include MACE [50], PSpec [71] and Pip [77]. They allow programmers to express their expectations about the systems communication structure, timing, and resource consumption. PSpec [71] is a performance checking assertion language that allows system designers to verify their expectations about a wide range of performance properties. The type of assertions of PSpec are similar to SelfTalk comparison relations. Also, similar to SelfTalk, PSpec uses a relational approach to represent and query monitoring data. However, PSpec lacks the ability to use mathematical functions as the basis of checking the behavior of the system. In general, in contrast to the existing language based approaches, SelfTalk mainly targets users such as sysadmins or performance analysts who have some insight into the systems behavior but lack the knowledge of the details and have no access to the systems source code.
Chapter 7

Conclusion and Future Work

In this chapter, we summarize our main techniques, and further discuss several promising research directions to extend our works.

7.1 Conclusion

This dissertation has presented Chorus, an interactive performance modeling framework and knowledge base, which can be used for dynamic resource allocation and what-if performance inquiry in large datacenters. This work is inspired by the observation that sysadmins or performance analysts lack effective tools to help them understand the performance characteristics and resource demand of running applications. Moreover, we observe that many applications have similar workflow patterns through system components, and similar performance model relations when accessing the same part of the system, such as disk. Finally, while a variety of performance models may be available (e.g., based on system administrator expertise and guidance, or historic data), not all of them may be equally suitable for the current workload, user requirements or system configuration.

For example, black-box models are automatic, but time-consuming to build, while analytical models require extensive domain knowledge and are brittle to environmental changes. In large datacenters, changes in hosted applications, workload mixes, and system components
are common. Ideally our performance modeling techniques would (i) benefit from historic knowledge about the system and long-running applications to select and refine an appropriate model for a new workload and (ii) adapt the model quickly to the dynamic changes such that the overall system’s response is in accordance with the performance requirements of all hosted applications.

With Chorus we dynamically learn, validate or revalidate, and combine various types of performance models to build an ensemble of models. The models exploit the insight or belief of sysadmins or analysts about the system and applications, as well as information automatically gathered from the live system through experimental sampling.

Specifically, Chorus provides a high-level SQL-like declarative language interface for sysadmins or analysts to express their knowledge as model templates, model relations, or sampling guidelines. These proposed model templates and relations are validated or refined experimentally by the Chorus runtime engine on the fly. Chorus ranks models by accuracy per configuration region, and selects the best model per region to build an ensemble of models. Historical models and experimental data are continually accumulated in the Chorus knowledge base, and can be reused or extended for modeling new workloads.

This dissertation shows the evolution of the work on the Chorus project, from learning simple offline models to refined online models. In our experimental evaluation of performance modeling in Chorus, we used several industry-standard benchmarks: TPC-W, TPC-C, OLTP-A, RUBiS. We have shown analytical, simulation-based or black-box models can be used for resource allocation in one or more tiers of a datacenter. Specifically, our results showed that model based approaches for dynamic resource allocation are effective for coarse grained resource allocation in the database back-end (Chapter 3) and for fine grained resource partitioning of the two-level cache hierarchy in a datacenter (Chapter 4). Moreover, the case studies on model templates demonstrate that the language interface provided by Chorus is flexible for sysadmins and analysts to express their modeling guidelines. We further showed the detailed results of ensemble modeling and model evolution, and their application to resource allocation...
using Chorus (Chapter 5).

We conclude that our framework provides accurate, fast and flexible support for performance modeling of applications sharing resources in datacenters.

7.2 Future Work

Our study can be extended to explore the semantic knowledge of sysadmins for other server platforms and other types of applications, as well as towards a wider reuse of the Chorus knowledge base as model guidelines across applications and platforms. We believe that these new research areas will enrich the knowledge base of Chorus, and the Chorus knowledge base, in its turn, can help sysadmins to better manage their applications and fully utilize their resources.

Specifically, we plan to extend our use of Chorus in the following research directions:

- **Exploring the workload similarity across different types of workloads**

  In this work, we reuse or extend the historical models when the workload mix has changed for the same application. As we know, large datacenters host more than tens of thousands of applications on their computing servers, and keep billions of objects in storage. We expect that different applications running on the same server platform might have many common workload features, e.g. read/write ratio, the number of outstanding I/O, cache miss ratio, etc. Hence, the performance samples/models running in the past might be reused for predicting performance for the newly scheduled applications, although these applications may belong to different users or different types of applications. Therefore, using Chorus, we can discover common features among various types of applications for model reusing.

  We believe that, through the reuse of historical performance data across applications, not just for different workloads of the same application, the time for building performance models for new workloads would be further shortened.
• **Applying Chorus to other types of applications**

In this work, we focus on studying database applications. We can further extend our Chorus framework to model other types of applications. For example, we can use Chorus to build performance models for MapReduce [30] type of distributed applications (e.g., Apache Hadoop [6]). A MapReduce job usually splits the input data-set into chunks, and process them in parallel. Chorus can be used to model its job execution time given the number of compute nodes and storage nodes, and other job related information. Another interesting application is Cassandra [5] which is widely used in Facebook, Twitter data-centers as a highly scalable distributed database. We can apply Chorus to predict the query execution time in Cassandra, and sysadmins can provide their knowledge about the replication and consistency policies as model guidelines.

• **Applying Chorus on other types of server platforms**

In this work, our applications are running through a shared database server connected to a network attached storage server. We enhance this server platform with the capabilities of the resource partitioning/scheduling to control resource sharing. We plan to apply Chorus on other popular server platforms, such as Virtual Machine Monitor environments (VMMs). Though VMMs have inherent resource isolation mechanisms for CPU and memory, performance degradation still occurs due to the other shared resources, e.g., network and storage bandwidth, shared cache in multi-cores. We can use Chorus to model applications’ performance in VMMs environments, and further guide the virtual machine or storage migration, and application placement algorithms.
Bibliography


[31] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s Highly Available Key-Value Store. In Proceedi-


[53] Shuang Liang, Song Jiang, and Xiaodong Zhang. STEP: Sequentiality and Thrashing Detection Based Prefetching to Improve Performance of Networked Storage Servers.


