Automatic Action Advisor for Storage System Performance Management

Li Yin

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-160
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-160.html

November 28, 2006

Copyright © 2006, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Automatic Action Advisor for Storage System Performance Management

by

Li Yin

B.E. (Tsinghua University) 1998
M.Phil. (Hong Kong University of Science and Technology) 2001

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Randy H. Katz, Chair
Professor Eric Brewer
Professor Steven N. Evans

Fall 2006

The dissertation of Li Yin is approved.

Chair Date

Date

Date

University of California, Berkeley

Fall 2006

Automatic Action Advisor for Storage System Performance Management

Copyright © 2006

by

Li Yin

Abstract

Automatic Action Advisor for Storage System Performance Management

by

Li Yin

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Randy H. Katz, Chair

With growing storage demands and system requirements, storage systems have become more complicated to manage. Management cost now dominates the Total Cost of Ownership. To reduce the cost and adapt to system changes more efficiently, storage systems must be managed more automatically. In this thesis, we aim to develop a framework to automatically and responsively generate an integrated multi-action plan, consisting of *what* actions to invoke, *when* to invoke them and *how* to invoke them, that can maximize the system utility.

To tackle the problem of automatic multi-action plan generation, we face four challenges: how to react rapidly, how to invoke actions proactively, how to compare alternative plans fairly and handle unexpected system changes. To solve these challenges, we use system models and constraint optimization to find the optimal action quickly, perform time-series forecasting and risk management to invoke actions proactively, use the system utility and optimization window to guide the comparison of actions, and apply a defensive strategy to handle unexpected system changes.

Within the general SMART framework, we develop three specific tools. (1) CHAMELEON is an automatic throttling tool that identifies the set of workloads to throttle and the degree of throttling using system models and constraint optimization techniques. (2) SMARTMIG is a proactive migration decision tool that generates the optimal migration plan in three phases: optimization, planning and risk management. Its decision takes into account the future workload demands and the benefit

and risk of migration options. (3) SMART's core design piece, Action Advisor. It integrates multiple action options and generates an action plan to handle expected and unexpected system changes. For predictable system changes, the Action Advisor finds the action plan using a recursive greedy algorithm. For unexpected changes, it applies a defensive strategy based on the ski-rental algorithm to choose actions. We implement SMART in both a real-world distributed file system, GPFS, and a simulator. Our evaluation results show that SMART can improve the system utility significantly. For 90% of the scenarios tested, SMART eliminates 94% of the utility loss. For the remaining 10%, it reduces the utility loss by 70%.

Professor Randy H. Katz
Dissertation Committee Chair

To my parents, my brother and Weidong.

Contents

Contents ii

List of Figures vi

List of Tables ix

Acknowledgments x

1 Introduction 1

1.1 Motivation 1
1.2 Problem Statement and Challenges 6
1.3 Contributions and Structure of The Thesis 9

2 Related Work 13

2.1 Storage System Performance Modeling 13
2.1.1 Simulation-Based Approaches 14
2.1.2 Analytical Models 15
2.1.3 Black-box Approaches 16
2.1.4 Other Works 17
2.2 Automatic Storage System Management 18
2.2.1 Policy-Based Approaches 18
2.2.2 Feedback-Based Approaches 19
2.2.3 Scheduling-Based Approaches 22
2.2.4 Model-Based Approaches 24
2.2.5 Other Works 25
2.2.6 Summary 26

3 Model-Based Automatic Action Advisor **28**
3.1 Motivation 29
3.2 Terminology 29
3.3 Framework for Model-Based Action Advisor: SMART 33
3.3.1 Input Modules 34
3.3.2 Utility Evaluator 36
3.3.3 Single Action Tools 37
3.3.4 Action Advisor 37
3.4 Experimental Exploration of Black-Box Based Component Models 38
3.4.1 Goal of Performance Modeling 39
3.4.2 Procedure of Black-Box Based Modeling 40
3.4.3 Analyzing the Performance Models 41
3.4.4 Experimental Evaluation 44
3.5 Conclusions 52

4 CHAMELEON: An Automatic Throttling Decision Tool for Storage System **54**
4.1 Motivation 55
4.2 Overview 57
4.3 System Models 59
4.3.1 Workload Models 60
4.3.2 Action Models 61
4.4 Reasoning Engine 61
4.4.1 Intuition 62
4.4.2 Formalization in CHAMELEON 63
4.4.3 Workload Unthrottling 65
4.4.4 Confidence on Decisions 66
4.5 Designer-Defined Policies 66
4.6 Informed Feedback Module 67
4.7 Experimental Evaluation 70
4.7.1 Testbed Configuration 70
4.7.2 Evaluation Standard 71
4.7.3 Using Synthetic Workloads 71
4.7.4 Replaying Real-World Traces 74
4.7.5 Decision Overhead of the Reasoning Engine 81

4.8 Working with SMART . . . 82
4.9 Conclusions . . . 84

5 SMARTMIG: An Automatic Proactive Data Migration Decision Tool 86
5.1 Motivation and Related Work . . . 87
5.2 Overview of SMARTMIG . . . 89
5.3 Migration Plan Generator . . . 91
5.3.1 The Optimization Phase: *What* and *Where* . . . 91
5.3.2 The Planning Phase: *How* and *When* . . . 94
How: Choosing Migration Speed . . . 94
When: Choosing the Migration Starting Time t^* . . . 95
5.3.3 Risk Management Phase . . . 98
5.4 Evaluation . . . 100
5.4.1 Experiments Configuration . . . 100
5.4.2 Sanity Check . . . 102
Working of Three Phases . . . 102
Impact of the Optimization Window T . . . 104
Impact of Utility Configuration . . . 104
5.4.3 Efficiency Tests . . . 105
Percentage of the Utility Loss Elimination . . . 105
Computational Overhead of SMARTMIG . . . 106
5.4.4 Sensitivity Test of Performance Model Errors . . . 107
5.5 Summary . . . 109

6 SMART: An Integrated Multi-Action Advisor for Storage Systems 111
6.1 Motivation . . . 112
6.2 Framework Revisited . . . 115
6.3 Decision Algorithms of the Action Advisor . . . 117
6.3.1 Normal Mode: Greedy Pruning with Lookback and Lookforward . . . 119
6.3.2 Risk Management . . . 123
6.3.3 Unexpected Mode: Defensive Action Selection . . . 125
6.4 Experimental Evaluation . . . 128
6.4.1 Evaluation Metric . . . 130
6.4.2 GPFS Prototype Implementation . . . 130

6.4.3 Testbed Configurations 132
6.4.4 Sanity Check 135
6.4.5 Feasibility Test Using the GPFS Prototype 141
6.4.6 Sensitivity Test 144
6.4.7 Decision Overhead of SMART 149
6.5 Conclusion 152

7 Conclusions and Future Work **154**
7.1 Thesis Summary 154
7.2 Future Work 157

Bibliography **160**

List of Figures

1.1 The HDD areal density (left) and the price of various storage devices (right). Source: the evolution of storage systems [61]. 2

1.2 DAS, NAS and SAN 3

1.3 A typical consolidated storage system 5

3.1 Architecture of SMART: a model-based action advisor 33

3.2 A regression tree example 40

3.3 Impact of training size on latency models: single workload 50

3.4 Impact of training size on latency models: multiple workloads 51

4.1 CHAMELEONin the SMART framework 56

4.2 CHAMELEON moves along the line according to the quality of the predictions generated by the internally-built models at each point in time. 56

4.3 Architecture of CHAMELEON 58

4.4 Workload model for SPC. 60

4.5 Overview of constraint optimization. 62

4.6 Workload classification. Region limits correspond to the 100% of the SLO values. . 64

4.7 Operation of the feedback module. 68

4.8 Effect of priority values on the output of the constraint solver. 73

4.9 Sanity test for the reasoning engine (workload W_5 operating from the controller cache.) 74

4.10 The final output of CHAMELEON using a combination of prediction and feedback-based approach 75

4.11 Uncontrolled throughput and latency values for real-world and synthetic workload traces. 76

4.12 Throughput and latency values for real-world workload traces with throttling (without periodic unthrottling.) 78

4.13 Throughput and latency values for real-world workload traces with throttling and periodic unthrottling. 79
4.14 Handling a change in the confidence value of the models at run-time. 81
4.15 Decision overhead of the reasoning engine 82

5.1 Architecture of SMARTMIG 89
5.2 Flow chart of optimization phase 93
5.3 Overall solution utility loss 96
5.4 CDF of the percentage of utility loss with no action invocation for all 86 scenarios 102
5.5 Migration speed vs. component load 103
5.6 Overall utility loss for various migration start times 103
5.7 CDF of the percentage of the saved utility loss 105
5.8 CDF of the percentage of the utility loss with SMARTMIG 106
5.9 Computation overhead of SMARTMIG 107
5.10 Impact of model errors on the accuracy of the predicted utility loss 108
5.11 Impact of model errors on the percentage of the saved utility loss 108

6.1 Problem with single action 113
6.2 Architecture of SMART: a model-based action advisor 115
6.3 Tree based action schedule generation 119
6.4 Pseudocode for the greedy pruning procedure 121
6.5 Lookback and lookforward optimization 121
6.6 Pseudocode of lookback and lookforward optimization 123
6.7 Pseudocode of the recursive greedy procedure 124
6.8 Example of the unexpected mode 127
6.9 IO rate of workloads as a function of time 133
6.10 Utility functions of workloads 134
6.11 Utility loss if no corrective action is invoked 135
6.12 Action invocation for different utility functions assigned to W_{trend} 136
6.13 Action invocation for different optimization windows 137
6.14 Action invocation for different budget constraints 138
6.15 Action invocation for different risk factor 139
6.16 Action invocation for unexpected case 140
6.17 The cumulative utility loss comparison between no action and SMART's actions . . 141

6.18 The observed utility loss 142
6.19 Difference in utility loss (after filtering): observed value-predicted 142
6.20 Unexpected mode: the cumulative utility loss with migration invoked and with no action 143
6.21 CDF of the percentage of utility loss 145
6.22 CDF of the percentage of the saved utility loss 145
6.23 Impact of model errors on the accuracy of the predicted utility loss 146
6.24 Impact of performance model errors 147
6.25 Impact of future forecasting errors on the accuracy of the predicted utility loss . . 148
6.26 Impact of future forecasting errors 150
6.27 Computational overhead of SMART 151

List of Tables

3.1 Multiple workload environment 46
3.2 All data vs. non-saturated data only 47
3.3 Impact of OSIO 47
3.4 Coarse models vs. fine models: aggregate performance 48
3.5 Coarse models vs. fine models: per-workload performance 49

4.1 Synthetic workload streams. 71

5.1 Constraint optimization for *what* and *where* 91
5.2 Solutions returned by SMARTMIG with a 14 day optimization window 103
5.3 Solutions returned by SMARTMIG with a 7 day optimization window 104
5.4 Five solutions returned by SMARTMIG with different utility configuration 105

6.1 Access characteristics of workloads 132

Acknowledgements

I am truly grateful to everyone who has directly or indirectly helped me in the past five years. Without their help and support, I wouldn't be able to finish my Ph.D. dissertation successfully.

First and foremost, I would like to thank my advisor, Professor Randy Katz, for giving me a chance to work with him, guiding me through these years and teaching me how to be a good researcher. He gave me the freedom to pursue different areas of research. I began my research with Randy on networks, then moved on to the performance study of networked storage before settling on automatic storage system management. I want to thank him for being so supportive and for always giving insightful advices. Despite his busy schedule, Randy managed to meet with me every week. He provided very detailed and insightful feedback on this dissertation with amazing turn-around time, even when he was traveling. This thesis would not have been possible without his guidance.

I would also like to thank my dissertation committee members, Professors Eric Brewer and Steven Evans, for their valuable comments and suggestions on my proposal and thesis. I want to thank Professor Michael Franklin for chairing my qualifying exam committee and providing helpful comments on my work.

In the past two years, I was generously supported by IBM Almaden Research Center as a part time supplemental researcher. I want to thank Dr. Sandeep Uttamchandani for giving me the opportunity to start the collaboration with the lab in Summer 2004, and for being a great mentor. I would also like to thank Dr. John Palmer, Dr. Kaladhar Voruganti and Dr. Honesty Young for their continuous encouragement and support. I am also fortunate to work with many excellent researchers there: Dr. Guillermo Alvarez, Linda Duyanovich, Dr. Madhukar Korupolu, David Pease, Ranami Routray and Aameek Singh.

I have received a lot of help and encouragement from fellow students at Berkeley. I thank the members of the SAHARA and OASIS projects and other friends—Matthew Caesar, Yanlei Diao, Yitao Duan, Ling Huang, Yaping Li, Sridhar Machiraju, Ana Sanz Merino, George Porter, Mukund Seshadri, Lakshminarayanan Subramanian, Mel Tsai, Kai Wei, Fang Yu and Shelley Zhuang.

My life in Berkeley would have been much less fun without my friends. Kaichuan He, Wendy Huang, Jinhui Pan and Xu Zou have given me tremendous help since my first day in the United

States. Yanmei Li, Yaping Li, Fang Yu and Rui Xu have given me continuous support and encouragement in these years. I want to thank all of them for their generous help and support. I am also very grateful to Minghua Chen, Guozheng Ge, Zhanfeng Jia, Wei Wei, Guang Yang, Jing Yang, Min Yue, Jianhui Zhang, Haibo Zeng, Wei Zheng for all the joy they have brought to me in these years.

Finally, I am deeply grateful to my parents, my brother and my husband. My father and mother taught me the importance of being honest, confident and hard-working. My brother took the responsibility of taking care of my mother while I was away from home to pursue my degrees. My husband, Weidong Cui, is my best colleague and friend. His love, encouragement, and faith in me made my years in Berkeley a happy journey. Their constant love and support have always been and will be the source of my strength. I also thank my parents-in-law for their love, support and generosity.

Chapter 1

Introduction

1.1 Motivation

Reduction in system hardware cost and growing system requirements for performance, reliability, availability and security have made management cost the dominant factor in the Total Cost of Ownership (TCO) [36, 37]. Management costs includes the cost of hiring administrators to manage and maintain the system and training support personnel and users. Hawkins [37] states that these management costs now contribute up to 85% of the total cost of ownership. To reduce the TCO, automatic system management has become a major research challenge and technical opportunity. This thesis focuses on automating storage system management because managing storage is now considered the largest component of TCO, estimated at 60% to 80% [36].

Over the recent past, three major trends have driven the evolution of storage systems.

- **Growing storage capacity and decreasing storage cost**. Storage systems are built on Hard Disk Drive (HDD) technology. To understand the evolution of storage systems, it is important to consider the evolution of the HDD. Figure 1.1 plots the HDD areal density and the price of various storage devices since 1980 [61]. We can see that the areal density of the HDD has improved by seven orders of magnitude and the number of bits stored per unit of HDD media is doubling about every year, which is faster than the speed of doubling every 1.5 years in Moore's law. At the same time, the HDD prices have decreased by about five orders of

magnitude and the total cost of storage systems has fallen about 2.5 orders of magnitude in the same period. The decreasing cost of storage systems has enabled new applications. Over time, the cost of the raw storage devices account for a smaller fraction of the total cost of the storage systems.

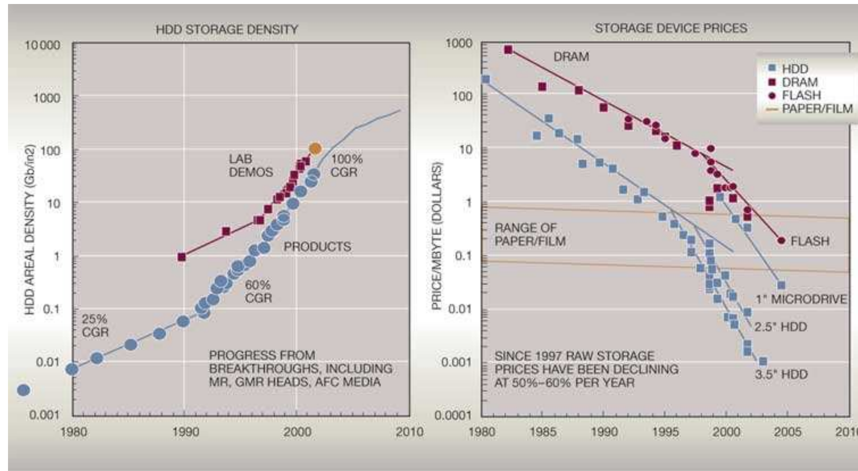


Figure 1.1. The HDD areal density (left) and the price of various storage devices (right). Source: the evolution of storage systems [61].

- **Increasing demand for storage**. Three major factors contribute to this growing demand. First, the decreasing cost of storage systems enables digitization and replacement of legacy media, such as paper and film. As a result of the increased use of digital images, online audio and video presentations, the volume of stored data is increasing exponentially. Second, more data are generated due to business automation, where, the data are generated using digital devices, such as sensors and digital medical imaging, with a much faster speed than the traditional hand-authored data [61]. Another important factor is due to legal regulations. Many companies now must archive e-mail or scan in all documents and store them electronically for a certain period of time to comply with legal requirements. Due to these reasons, the demand of storage increases dramatically. The Seagate IDC 2002 report estimated that the disk storage capacity shipped expected to grow at a 62% compound annual growth rate [91].

- **Growing management cost**. The growing storage demands results in an increasing number of HDDs and makes the management tasks more complicated. For example, a typical data center can have more than 10,000 HDDs. In addition, as the cost of storage systems

decreases, clients are seeking additional features such as performance, reliability, availability and security. To provide these services, technologies such as disk arrays, network-attached storage systems (NAS) [75], and the storage area network (SAN) [23] were developed, thus complicating the management of storage systems. Example technologies include the redundant array of independent disks (RAID) [10], the Network File System (NFS) [43] and the Internet Small Computer Systems Interface (iSCSI) protocols [23]. Figure 1.2 shows several possible ways of accessing data, including the traditional Direct Attached Storage (DAS), NAS and SAN. Taken together, these factors have made the management cost dominant in the TCO of storage systems. [61] estimated that out of three million dollars spent on storage systems, storage administration costs 2 million dollars.

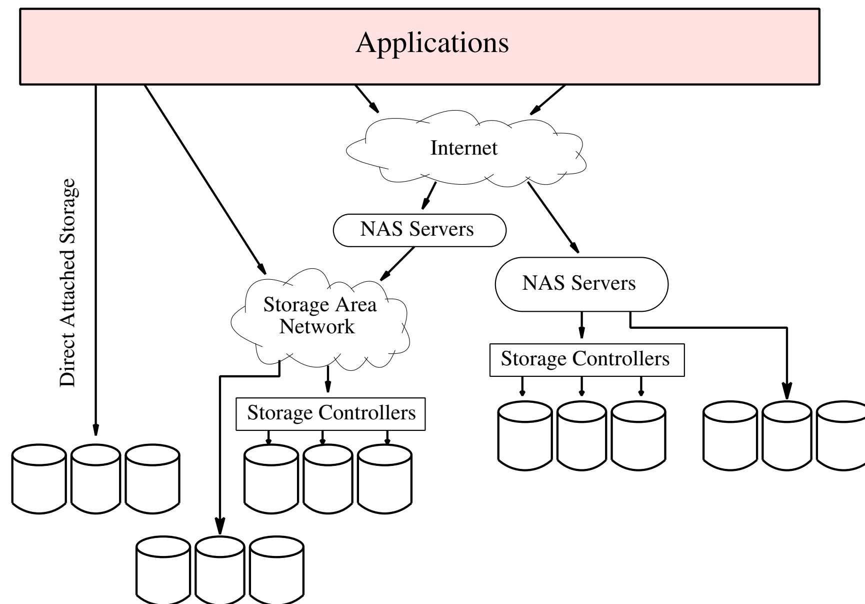


Figure 1.2. DAS, NAS and SAN

As a result of these trends, many organizations are trying to either deploy a centralized corporate data center to consolidate their storage system operations or outsource the data and/or management tasks to a storage service provider (SSP). In both the corporate data center and the SSP environment, the storage system is often shared by multiple independent applications. For example, independent customers share a common storage service provider. Or a given enterprise runs "consolidated" applications on a common storage systems whereas in the past these were independent. Each ap-

plication has different priorities, workload access characteristics and the Service Level Objectives (SLOs). The SLOs are negotiated between the clients and the service provider and are defined on a per-workload basis. They often prescribe the lower bounds for performance, capacity, availability and security that a client expects from the storage infrastructure. In the environment of an SSP, the SLOs also include financial incentives, such as the cost of per GB of storage, and penalties when SLOs are violated by the service provider. For example, when the performance goal is violated, the client might receive a 30% discount. In addition, the data center or SSP is also associated with business-level constraints that determine the "optimization window" and the budget for provisioning addition hardware. The "optimization window" specifies the time window over which the storage resources need to be provisioned and optimized.

From the perspective of the SSP or data center, their goal is to minimize the SLO violations to avoid revenue losses due to penalties. At the same time, they want to improve the storage utilization and reduce their system cost. To improve the storage system utilization and ease its management, they employ storage virtualization technologies to map application data to storage resources independently of the underlying infrastructure (Figure 1.3). This application-to-resource mapping cannot be static, as the correct configuration for optimal performance changes with the workload, application priorities and run-time exceptions such as load surges and hardware failures. As a result, corrective storage system actions must be invoked to change the application-to-resource mappings dynamically. For example, the corrective action *throttling* limits the lower priority requests admitted to the storage system to leave more bandwidth to the higher priority ones. Action *migration* modifies the physical location of workloads to redistribute the bandwidth to applications. *Provisioning* action adds new physical resources to the system.

Currently, system administrators follow a process of Observe-Analyze-Act (OAA) to invoke the corrective actions at run-time to optimize the application-to-resource mapping. In the *Observe* process, they continuously monitor the system state and detect SLO violations. For example, they monitor load and performance of each workload to detect whether these fall below acceptable and agreed upon thresholds. In the *Analyze* process, they analyze the system state and determine the corrective action plan, including *what* actions to invoke, *when* to invoke them and *how* to invoke them. An example action plan is to throttle Workload 1 by 10% immediately and migrate its data

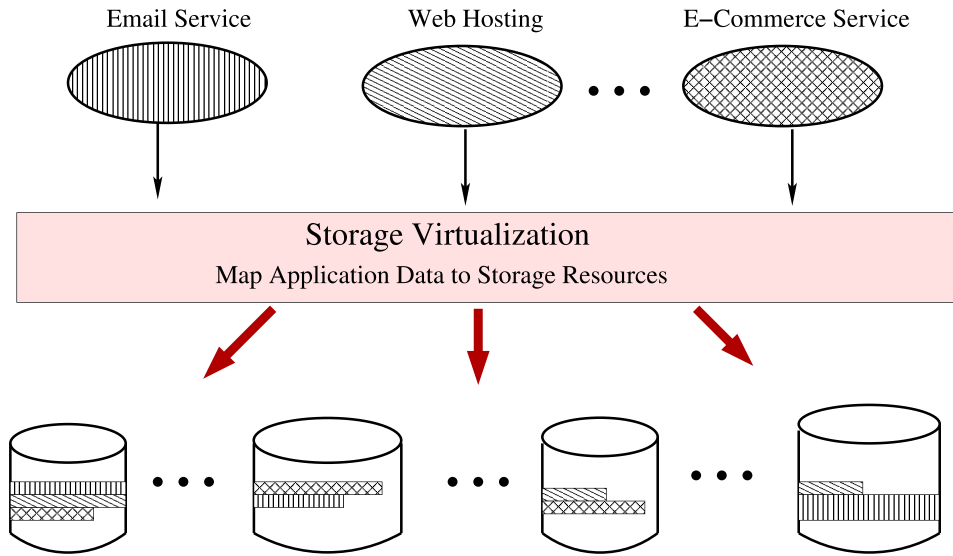


Figure 1.3. A typical consolidated storage system

from device *A* to *B* at 2 am at a speed of 100 migration IOs per second. Finally, in the *Act* part, they trigger the action actuators to correct the SLO violations.

This human-driven OAA loop has three key drawbacks.

- First, it is slow. It can take the administrators hours or days to determine the optimal corrective action plan due to the complexity involved. The decision-making requires processing a large amount of data along with other information, such as workload characteristics, component features and business constraints.

- Second, it frequently leads to a sub-optimal solution. Improving the system utility depends on the knowledge of the administrators and is only as good as they are themselves. In addition, due to the complexity involved, the administrators often settle for a quick, sub-optimal solution. The solution may over-provision the system, making the system under-utilized, or it may fail to solve the root cause of the problem, having the system oscillating around the bad state, with many client SLOs in violation.

- Third, it is expensive due to the high "people" cost. One rule of thumb is that one administrator with an annual hiring cost of $60K to $130K can only manage about 5-20 terabytes of storage, which only cost about $5K-$20K for the raw storage.

To address these problems, we must develop mechanisms to automate the OAA loop to increase the amount of storage managed by an administrator, and to adapt to system changes and exceptions in a *fast* and *cost-effective* way.

In the next section (Section 1.2), we describe the research problems we tackle in this thesis and discuss the research challenges we face. In Section 1.3, we present our contributions and the structure of this thesis.

1.2 Problem Statement and Challenges

Many research efforts have been made to automate the OAA loop. The key functionality of the *Observe* part is to monitor system behaviors and detect problems such as SLO violations. The *Analyze* process is responsible for evaluating system behavior and deciding the corrective action plans to correct the SLO violations. The analysis typically requires processing a large amount of data and combining various inputs such as workload characteristics, component features and business constraints. The corrective action plans include details about *what* actions to invoke, *when* to invoke the actions and *how* to invoke them. The *what* decision evaluates all action options such as throttling, migration and provisioning, and selects the best actions that can solve the system problems. An example *what* decision is to invoke migration as well as throttling. The *when* decision chooses the optimal invocation time for selected actions. For example, throttling needs to be invoked immediately and migration should be invoked at 2 am tonight. The *how* analysis generates the action invocation parameters. For example, the token issue rates of throttling is 150 IOPS for workload 1 and 300 IOPS for workload 2, and workload 1 should be moved from device A to device B at a speed of 100 IOPS. For the *Act* part, the key challenge is to implement actuators to enforce the action behavior as designed. Many research prototypes as well as commercial products have been developed to automate the *Observe, Analyze* or *Act* process. For example, the ControlCenter from EMC [25] and the Total Productivity Center (TPC) from IBM [26] provide an extensive monitoring framework to automate the *Observe* process. They also implement a class of action actuators such as throttling, migration and provisioning tools to assist the *Act* process. However, the *Analyze* process

is still not well-understood. It is either manually controlled or based on simple heuristics, which often lead to sub-optimal solutions.

In this thesis, we focus on automating the *Analyze* process. Our goal is to design a framework that can generate the corrective action plan yielding the maximum system utility in an *automatic* and *responsive* fashion. The system utility here is associated with workloads' performance to reflect the degree of users' satisfaction (see Chapter 3 for the definition). This problem has the following challenges:

- **Rapid reaction**. When the SLOs are violated, the storage service providers are charged a penalty. To reduce revenue loss, it is critical that our framework adjusts the storage system to meet the clients SLOs quickly. Generating the corrective action plan in a short time is a very challenging task due to the large number of workloads, action options and complex interactions between the system and workloads. In such an environment, quickly exploring the search space to find a "near optimal" solution, if not optimal, is difficult. Existing feedback-based solutions [47, 54] trade decision speed for optimality. On the other hand, policy-based solutions [17] trade optimality for speed. In this thesis, we take the model-based approach to find a quick "near optimal" solution. We construct mathematical models to describe the system behaviors. Specifically, we define models for the capabilities of components in the storage system (component models), the workload being presented to them (workload models) and the expected response to different action invocation options (action models). By composing these models (Section 4.4), we can calculate the "expected" system behaviors for a candidate action plan without actually invoking the system subjected to real workloads. In addition, we apply constraint optimization techniques to reduce the overhead of searching the candidate configuration space. In Chapter 3, we give details of the system models. In Chapter 4, Chapter 5, and Chapter 6, we present our approximation algorithms for finding the corrective action parameters and the corrective action plan.

- **Proactive corrective actions**. Existing solutions focus on correcting the system behaviors after the SLOs are violated. However, once the SLO violations happen, the service provider starts losing revenue. Ideally, we want to invoke the corrective actions before the SLOs are

violated to avoid penalty. Existing techniques such as time-series analysis [84] can forecast the future workload demands. By leveraging these, we can predict SLO violations and invoke appropriate corrective actions in advance. Proactively invoking actions have the benefit of avoiding future SLO violations. At the same time, it also faces the risk of paying penalty cost of incorrectly invoking actions. The challenge is how to balance the benefit and the risk. In this thesis, we perform risk management to balance the benefit and the risk of an action plan. Specifically, we combine the time-series forecasting and system models to predict the benefit of an action plan, and scale the benefit with the risk factor, that captures the risk of future uncertainty and action invocation overhead. In Chapter 5 and Chapter 6, we give details of the risk management for migration and corrective action plans.

- **Fair comparison of corrective actions**. Researchers have developed multiple corrective actions for storage systems. When SLOs are violated or will be violated, we need to compare all feasible action options and decide which to invoke. Different corrective actions are designed for different purposes and have different strengths and weakness. For example, throttling can adjust the system behavior in a very short time and is most appropriate for SLO violations due to temporary fluctuations. Migration can solve problems caused by "long-lived" workload changes, such as change of hot-spot, but with a long action execution time due to data movement. Hardware provisioning can solve the root cause of the problem—insufficient resource, but they face the risk of over-provisioning. The challenge is how to compare them without bias.

  To address this problem, the solution in this thesis utilizes two concepts, the "optimization window" and the "system utility", and looks for actions that can maximize the system utility for a given optimization window. By looking for actions that are optimal for the optimization window, we are not biased towards either short-term actions or long-term actions. The actions selected are the most appropriate for the given optimization window. In addition, by using "system utility" (see the definition in Section 3.2), actions with different properties are uniformly evaluated based on their impact on the system utility. The goal of our framework is to find actions to maximize the utility value delivered by the storage system to all workloads. In Chapter 3, we define the system utility, utility functions and the calculation of system utility.

In Chapter 5 and Chapter 6, we describe algorithms for finding the action options that lead to maximum system utility improvement for a given optimization window.

- **Handling unexpected system changes**. Real world storage systems often experience unexpected system changes such as changes of workload characteristics and load surges. Due to the limited information about the system states, generating corrective action plans for unexpected system changes is more difficult than handling the normal system changes that can be forecast based on the history. In this thesis, we propose a defensive strategy to handle unexpected system changes. We borrow the concept of the well-known "ski-rental" problem [48], where we need to make a decision to buy or rent a ski equipment without knowledge on how many times one might go skiing in the future. The commonly used strategy is to stay with the lower cost option, that is, "to keep renting until the amount paid in renting equals to the cost of buying, and then buy". In our solution, we continuously examine the accuracy of future prediction. There exist multiple metrics to evaluate the future prediction accuracy. One example is the absolute prediction error, defined as the difference of the predicted value and the observed value. When the future prediction accuracy is low, we switch to the defensive decision strategy, which always invokes the action with the lowest cost. In the mean time, we continuously collect new observations to improve the time-series prediction accuracy and switch to the normal decision strategy when the prediction is accurate enough. In Chapter 6, we describe our defensive algorithm for handling the unexpected system changes.

In the next section, we highlight our contributions and describe the structure of this thesis.

1.3 Contributions and Structure of The Thesis

In this thesis, we tackle the problem of automatically generating the corrective action plan in the OAA loop. We propose a new framework, SMART, which is a model-based action advisor. It generates action plans by combining system models, time-series forecasting, business constraints and single action options. Specifically, we develop the following specific tools within the more general framework:

- **An Automatic Throttling Decision Tool**: We develop a new throttling tool, CHAMELEON, which automates the decision making of the throttling actions. It uses system models to predict the system behavior for given throttling invocation parameters and applies constraint optimization techniques such as linear programming to scan the candidate space for the optimal settings. In addition, it uses a feedback-loop to control the throttling execution and defines simple heuristics as the fall-back strategy when the system knowledge is insufficient. For example, when the accuracy of system models falls below a given threshold, CHAMELEON determines the invocation parameters using heuristics. We replay traces from production environments in a real storage system, and demonstrate that CHAMELEON makes accurate decisions for the workloads examined. With our testbed and scenarios tested, CHAMELEON can react to and solve performance problems in 3 to 14 minutes using the feedback loop guided by the decision based on system models.

- **A Proactive Risk Modulated Migration Tool**: We develop a new migration tool, SMARTMIG, which can determine the migration invocation parameters, including *what* and *where* to migrate, *how* to migrate and *when* to invoke. SMARTMIG makes the migration decision in three phases. In the optimization phase, SMARTMIG selects the migration data (*what*) and migration targets (*where*). In the planning phase, SMARTMIG determines the migration speed (*how*) and the migration starting time (*when*). In the risk management phase, SMARTMIG applies risk analysis to select migration options leading to the maximum system utility gain and minimum risk. Compared with previous solutions, SMARTMIG's migration decision can account for both the current and future system states. It also considers the risk of migration operation due to future uncertainty and the migration invocation overhead. Experimental results show that SMARTMIG's migration plan can account for various configuration settings. For the scenarios tested, SMARTMIG eliminates 80% of the utility loss that the system may experience when no corrective action is taken. We also show that SMARTMIG can make a decision on the order of minutes.

- **An Integrated Multi-Action Advisor**: In SMART, the Action Advisor is designed to generate the action plan to handle both expected and unexpected system changes. Depending on the accuracy of the future prediction, it operates in either the *normal mode* or the *unexpected*

mode. In the former, it applies a recursive greedy algorithm to generate the action plan and performs the risk management to balance the benefit and risk of an action option. In the latter, it applies a defensive strategy and always selects the action with the lowest cost. In both modes, SMART continuously collects new observations on the system states to improve the accuracy of the future prediction. We test the SMART framework using both the GPFS prototype and a simulator. Our experimental results show that SMART's action plan can adapt to the change of system configuration parameters and generate flexible action schedules to improve the system utility. For our test scenarios, SMART was able to improve system utility, often by substantial margins. For more than 90% of scenarios tested, SMART elimiates more than 94% of utility loss. For the remaining 10% cases, it reduces the utility loss by more than 70%. SMART decision was made without operator intervention. With our testbed, we measured that SMART can generate the action plan in the order of minutes with 100 workloads running in the system.

The rest of this thesis is organized as follows. In Chapter 2, we discuss the related work in the areas of storage system performance modeling and storage system management. It provides the background from which this thesis is developed.

In Chapter 3, we describe the framework of SMART. We introduce the key components of SMART, including the input modules, single corrective action tools and the Action Advisor. In addition, we study one of the key elements of the system, storage performance models in detail. We choose the black-box approach and construct system models using an off-the-shelf regression tree implementation GUIDE [53]. Our experimental results show that, to improve the model accuracy, it is important to filter the data in the saturated region and take into account the number of outstanding I/Os. With three workloads running in a real storage system, we achieve an error rate of 19.3% for the latency models and 4.5% for the throughput models.

In Chapter 4, we describe the design and formulation of CHAMELEON. We discuss the models, reasoning engine, base heuristics and the feedback-control of throttling execution in details. We also present our experimental evaluation results of replaying traces from production environments in a real storage system.

In Chapter 5, we describe the design of SMARTMIG and decision algorithms in three phases: optimization, planning and risk management. We evaluate SMARTMIG's efficiency in terms of utility loss reduced and measure its decision overhead using a simulator.

The core design piece of SMART, the Action Advisor is discussed in Chapter 6. We describe how Action Advisor collects single action options and generate an integrated multi-action plan. We describe the decision algorithms to handle the normal and unexpected system changes in details. We test the SMART framework using both the GPFS prototype and a simulator. Our experiment evaluates how SMART adapts to configuration changes, how SMART performs in real system, the impact of input errors on SMART's decision and the decision overhead.

Finally, in Chapter 7, we summarize our work and contributions, and discuss directions for future work.

Chapter 2

Related Work

In this chapter, we discuss the related work in the areas of storage system performance modeling and storage system management. Instead of covering every piece of work in these areas, we focus on the most relevant ones to provide the background from which we developed this thesis. We start with storage system performance modeling (Section 2.1), describing simulation-based approaches, analytical and black-box models and some other relevant work. In Section 2.2, we summarize the existing research efforts in the area of automatic storage system management, which motivate our work on developing an automatic and responsive framework to generate the corrective action plan.

2.1 Storage System Performance Modeling

Our framework, SMART, is a model-based action advisor. It relies on storage system performance models to predict the system behavior without real action invocation. In this thesis, we do not aim to propose any new modeling techniques. Instead, we apply existing techniques to assist the action selection. In this section, we review the most relevant efforts in storage system performance modeling and identify their weaknesses and strengths for automatic storage system management.

Storage system performance modeling has a long and successful history. The goal is to predict the system's performance characteristics along such dimensions as throughput, latency, and utilization. Based on the approaches taken, existing solutions can be classified into three categories:

simulation-based [33, 93], analytic or white-box techniques [69, 74, 86], and black-box approaches [6, 89]. In the rest of this section, we limit the discussion to several important and relevant efforts in each category. Note that storage systems can consist of a single disk or disks connected using switches and storage controllers. The latter is more complicated to model because disks and controllers interact in a complicated manner. Configurations such as caching and data stripping policies can alter the load seen to each disk and affect the performance of the storage system. For more complicated storage systems, such as networked storage system, it can be modeled by composing the network and the storage subsystem. There exists a large body of literature to simulate and model the network behaviors [3, 4, 63, 64]. We skip the solutions in the domain of network modeling and focus on the models for disks and disk array in this section.

2.1.1 Simulation-Based Approaches

Simulation-based approaches simulate the mechanical seek and rotational behavior of the devices within the storage system and measure the performance for a given configuration to predict performance. Ruemmler and Wilkes [69] developed an accurate simulator based on the mechanical behavior of disks. The simulator considered important features such as data-caching characteristics, data transfer effect including bus transfers, seek-time and head-switching costs, rotational position and data layout. Later on, two well-known simulators, DiskSim [33] and Pantheon [93], were developed to model a storage sub-system. They can support most secondary storage components, including device drivers, buses, controllers, adapters and disk drives. The workload performance achieved using these simulators is close to that of a real system. However, only the disk modules have been carefully validated, but the rest of the components have not.

Developing detailed disk simulators requires both mechanical and behavioral parameters of the component, such as mechanical delays, on-board caching and prefetching algorithm. These characterizations are often not publicly available and are very difficult to acquire. To address this problem, Schindler, *et al.*, developed an automatic tool DIXTRAC [72]. It used a collection of pre-programmed test vectors to measure timing for mechanical and command processing overhead and expert-system-like algorithms to identify the layout and caching policies. DIXTRAC simplifies the parameterization of simulators such as DiskSim and Pantheon.

In summary, simulation-based approaches can achieve detailed and accurate results. However, developing simulators require both mechanical and behavior parameters that may not be publicly available. As modern storage architectures becoming more complicated, it becomes even more difficult to develop an accurate simulator. In general, simulators can be used for model bootstrapping or static provisioning. However, their long execution time make them unsuitable for run-time storage system management, where the decision making mechanisms often need to explore the consequences of multiple design points quickly.

2.1.2 Analytical Models

There is a large body of work on analytical models of disks and disk arrays. Different from simulation-based approach, analytical approaches describe the behavior of storage devices using a set of formulas. These often take workload characterizations as input and output the predicted performance such as system utilization, throughput and response time.

Many analytical models have been developed to approximate the microscope disk performance, such as the seek time, the disk cache impact, and the queuing delay for disks. The seek time was normally modeled as a function of the seek distance [69] or a function of the workload [39]. The most commonly used seek time approximation is a three-part function, which models the acceleration and deceleration of the disk head, the head coasting time and the settling time. Analytical models were also developed to approximate the impact of disk cache with write-back policy [15], write-only policy [77], and to estimate the cache hit rate based on the workload sequentiality [74]. Analysis on the request scheduling was mostly based on the queuing theory. The M/G/1 model was applied to approximate the FCFS scheduling algorithm [92]. Alternative scheduling algorithms like SCAN and LOOK were analyzed in [62, 83].

Shriver, *et al.*, [74] proposed an analytical model for disk drives with read ahead caches and request reordering. They constructed the storage device model by developing sub-models for individual components, including request queues, caches, and disk mechanisms and composing them. Each workload was associated with thirteen attributes describing the request size, the temporal locality, the spatial locality and other measurable. These workload behavior parameters together with

the lower-level device behavior parameters are used to construct the model of a component. They validated their model using Pantheon and show that for disk utilization less than 60%, they can predict the behavior of a variety of real-world devices with mean errors less than 17%.

Previous discussions focus on the analytical models of disk drives. As the disk arrays become more dominant, many analytic models were developed to approximate disk array's behavior in the normal mode [51, 59, 94], the degraded mode and the rebuild mode [58]. Recently, based on Shriver's disk model, Uysal, *et al.*, [86] constructed analytic models to predict the throughput of disk array by decomposing storage system into individual components. They validated their model against a state-of-art disk array for a variety of synthetic workloads and array configurations. The results show that their predictions are within 32% of the actual performance and 15% on average.

In general, compared to simulation-based approaches, analytical models are less accurate, but they are orders of magnitude faster to generate performance prediction. However, defining analytical formulas requires deep understanding of the internal operation of the storage devices and the interaction among workloads. In addition, we often need to develop various analytical models to cover the different configuration of the storage devices, such as the scheduling policies, caching policies and operation mode. As the storage system becomes more complicated, these formulas are often difficult to develop and slow to evolve.

2.1.3 Black-box Approaches

Black box approaches treat the storage systems as black-boxes and require minimal or no device specific information. Most of them construct models by correlating the input measurable with the output observables using machine learning techniques. Anderson, *et al.*, [6] proposed a table-based approach. They bootstrapped the model with various data points and recorded the corresponding performance in the form of a table. Each data point consists of system state and workload features. The performance of a new point was predicted using interpolation techniques such as closet point, nearest neighbor averaging and hyperplane interpolation. Experimental results show that their approach can achieve an error rate of 5% to 20%, depending on the size of performance table. However, this approach is not scalable with respect to the dimensions of the input vector. In addi-

tion, they represented the model with a table, which limits the flexibility of applying mathematical techniques such as constraint optimization.

Hidrobo, *et al.*, [38] applied machine learning techniques to model the response time as a function of the request type, the block address, the reference distance and the request size. They evaluated three models: linear, quadratic and neural network. Their study show that the neural networks outperforms the other two with an error below 10%.

Wang, *et al.*, [89] proposed a solution based on the Classification And Regression Trees (CART) algorithm. The CART model uses piece-wise constant functions to approximate the performance functions on a multi-dimensional Cartesian space. They studied both the request level and the workload level models and found that the request level prediction achieves better accuracy because the input information is more precise. At the workload level, the latency prediction has approximately 40% error.

In summary, black-box approaches require minimal or no device specific information. Models are constructed by correlating the input measurable with the output observables. Compared with the simulation-based approaches and analytical models, they are easier to develop. In addition, by continuously updating the training data, they can evolve with changes in the system behavior, workload characteristics and action effects automatically. On the other hand, black box approaches are less accurate. The model accuracy depends on the quality of training data. When the storage system becomes complicated, black-box approaches may require a long bootstrap time to achieve a high prediction accuracy.

2.1.4 Other Works

In a storage system with multiple workloads, the I/O requests interleave together and change the behavior of each individual workload. Most works in storage system performance analysis treat the mixed workloads as one stream and focus on analyzing the performance of the whole system such as overall throughput and average latency. Limited research has been done on analyzing the performance of individual workloads. As the degree of resource sharing grows, the per-workload performance prediction becomes more important to provide the service guarantee on a per-workload

basis. The most relevant work on this problem is the study by Borowsky, *et al.*, [12]. They applied queuing theory to approximate the bound of the response time. Their formulation requires prior-knowledge on the correlation between workloads and the mean and variance of the service time for each individual workload. Evaluation results showed that the approximated bound can accurately determine if a given bound can be satisfied. Although their work did not construct an analytical function to predict the exact performance, it provided a feasible solution to detect if a workload can meet its SLO requirement in the shared system.

2.2 Automatic Storage System Management

With the growing complexity of storage system management, many commercial products and research prototypes have been developed to manage the storage system automatically. Management software, such as the Total Productivity Center (TPC) [26] from IBM and the ControlCenter from EMC [25], provided an extensive monitoring framework and also implemented various action actuators. Other solutions [21, 22, 66] can detect and diagnose system problems using machine learning techniques. In this section, instead of covering all work related to automatic storage system management, we focus on the specific literature of automatic decision making for storage system performance optimization. Existing solutions fall in the taxonomy of policy-based, feedback-based, scheduling-based and model-based approaches. In the rest of this section, we cover the most relevant work in each.

2.2.1 Policy-Based Approaches

In policy-based approaches [42, 87], system administrators encode policies as sets of Event-Condition-Action (ECA) rules that are triggered when some precondition is satisfied. An example of the precondition is that one or more system metrics go beyond a predetermined threshold. Most current commercial tools for automatic resource allocation belong to this category. For example, BMC Patrol [76], IBM Total Productivity Center [26] and EMC ControlCenter [25] are all policy-based solutions. Rules are a clumsy, error-prone programming language. They are written by experts with many years of experience as system architects and administrators. However, as systems

grow in complexity, even experts find it difficult to develop such rules. Administrators are expected to account for all relevant system states, to know which corrective action to take in each case, to specify useful values for all the thresholds that determine when rules will fire, and to make sure that the right rule will fire if preconditions overlap. Moreover, simple policy changes may translate into modifications of a large number of rules, which makes it difficult to adapt to system changes. Verma, *et al.*, developed a variation [88] based on case-based reasoning. It relies on iterative refinement to derive rules from a *tabula rasa* initial knowledge base. *Tabula rasa* refers to the development of autonomous agents that are provided with a mechanism to reason and plan toward their goal, but no "built-in" knowledge-base of their environment. This approach does not scale well to real systems because it explores the search space in an unstructured way.

Sleds [17] is a distributed controller that provides statistical performance guarantees for a storage system. It uses a leaky bucket filter [31] to shape and throttle I/O flows from each client. It defines policies to specify the set of workloads supposedly responsible for the SLO violations and the degree of throttling. Sleds can provide service isolation and differentiation by selectively throttling workloads. It also scales well with its decentralized architecture. However, in Sleds, the policies are predefined by administrators and hard-wired into the system. As a result, it cannot adapt to system changes at run-time.

In summary, policy-based approaches front-load all the complexity into the work of creating the rules, in exchange for simplicity of execution at run-time. They can only provide a coarse-grained optimization, as good as the human who wrote the rules and they can not adapt to system changes automatically.

2.2.2 Feedback-Based Approaches

A huge body of existing works fall into this category. Feedback-based approaches based on the control theory make minimum or zero assumptions on the system behavior and workload characteristics. The action decision is solely based on the most recent performance samples and the desired system states.

Diao, *et al.*, [29] discussed the control theoretic foundations for self-managing systems. They

made an analogy between the elements of autonomic systems and those in control systems, and argued that control theory approach provides properties such as stabilities, accurate regulation and short settling times. These are important properties for self-managing systems. Stability means that for any bounded input over any amount of time, the output will also be bounded [67]. In the domain of self-managing system, it indicates that for any bounded configuration parameters, the performance of the storage system is also bounded. It is an important feature for mission critical applications. Accurate regulation ensures the measured output will converge to the objectives. An example objective is "maximizing the throughput without exceeding the latency constraints". Settling time is the time from the change in input to when the measured output is sufficiently close to its new steady state value. For example, the settling time for throttling is the time from when the new throttling decision is enforced to when the latency of the storage system is stable. A short settling time indicates that the system converges quickly to its steady state value. They also pointed out a set of challenges, including measurement delays and noises, long convergence time to meet the objectives, and development of effective resource models to characterize the behavior of storage systems.

Chase, *et al.*, proposed Muse [19], which applies economic models to manage the energy and sever resources in a data center. In the Muse system, customers "bid" for resources as a function of delivered performance. Muse continuously monitors the load and plans resource allotments by estimating the value of their effects on service performance and allocating resources to their most efficient use. Muse uses a feedback loop to adjust the resource prices to balance the supply and demand. Experimental results showed that Muse can adapt to offered load and available resources, and can reduce the server energy usage by 29% or more for a typical Web workload.

Lumb, *et al.*, proposed a virtual storage controller, Façade [55], which sits between hosts and storage devices in the networks and controls the scheduling of interleaved I/O requests. Façade maintains per-workload latency targets and a queue, which is shared by interleaved workloads. It adjusts the queue length based on the workloads' changing latency targets and the observed latency in previous window. When the observed latency is higher than the latency target, it reduces the queue length to admit less requests into the system. This is equivalent to throttling. However, Façade does not support complex decisions about which subset of the workloads should be left alone. Therefore,

unexpected demand surge of one competing workload will cause all workloads being throttled, and high priority workloads are throttled equally with low priority ones. As a result, it fails to provide service isolation and differentiated service.

Triage [47] is another feedback-based throttling scheme. It defines a number of bands based on the aggregated throughput. Each band specifies policies on how workloads should share the additional throughput. At run-time, Triage keeps track of the performance band of the system and uses an on-line feedback loop to control the sending rate and adjust the performance band to meet the latency goal of each workload. The concept of band helps to capture the relative importance between workloads and can provide a certain level of performance isolation and differentiation. However, as the number of workloads increases, the number of bands grows. It becomes difficult to specify the detailed sharing policies for each band.

Aqueduct [54] is an on-line migration algorithm based on control theory. It constructs a feedback loop to dynamically adjust the migration speed such that the performance guarantees on the application can be satisfied in the presence of workload and system variations. Its prototype was evaluated on a real storage system. The experimental results showed that Aqueduct successfully provides the performance guarantees in terms of bounded average latencies. It reduced the average I/O latency by as much as 76% and the migration was performed very close to the maximum migration speed allowed by the latency contract.

Hippodrome [8] is a tool to automate the design and configuration process of storage systems. It is structured as an iterative loop: it first analyzes a workload to determine its requirements, then creates a better storage system configuration to better meet these goals. It then invokes migration to migrate the existing system to the new design. Hippodrome repeats the loop until it finds a storage system configuration that satisfies the workload's I/O requirements. Unlike previous examples that use feedback for parameter tuning, Hippordrome's feedback loop involves data migration. Therefore, adapting system changes using Hippodrome can be both costive and time-consuming.

In general, compared with policy-based approaches, feedback-based solutions require minimum or no prior-knowledge on the controlled system and workloads characteristics. They can work well for parameter-tuning decisions with limited scale. However, they are not well-suited for

decision-making with multiple variables [82] due to the large solution space and the possible long convergence time. For decisions other than parameter tuning, feedback-based approaches may have a slow feedback loop and suffer from a long convergence time. Lastly, feedback-based approaches rely on system signals to trigger actions and adjust the decisions. Thus, they lack the ability to invoke actions and determine action parameters proactively.

2.2.3 Scheduling-Based Approaches

Scheduling-based approaches control resource allocation through managing job scheduling. It has a successful history in the field of providing quality of service in networks. However, because request dropping is not an option in storage as it is in networks, scheduling algorithms for storage systems normally work differently. In this section, we focus our review on scheduling mechanisms in domains other than networks.

Sullivan, *et al.*, [81] proposed a framework for proportional-share resource management. The goal is to provide flexible resource sharing and secure isolation at the same time. They extended the standard lottery-scheduling scheme to support both hard shares and soft shares. By hard shares, we mean absolute resource reservations, such as 10 GB of bandwidth. Soft shares are proportional allocations, such as 10% of bandwidth. In addition, they introduced the *ticket exchange* mechanism to allow applications to modify their own shares without compromising the rights of other resource principles. They have prototyped their framework in the VINO operating system and have used it to manage the CPU time, physical memory and disk bandwidth. The experimental results demonstrated that the extended lottery-scheduling framework can improve the performance of server applications.

Scheduling-based mechanisms are also used in storage systems to enforce fair share and provide isolation. Jin, *et al.*, [45] designed scheduling schemes to enforce fair share by throttling request flows and reordering I/O requests. They studied two scheduling schemes based on the Start-time Fair Queuing (SFQ) and a request window algorithm. Their results showed that their algorithms can achieve efficient resource usage and performance isolation. However, they found that scheduling

schemes cannot ensure tight bounds on the performance, which is essential for satisfying the SLO constraints.

Lan, *et al.*, proposed a multi-dimensional storage virtualization system, Stonehenge [40], which uses a learning-based bandwidth allocation mechanism to map SLOs to virtual device shares dynamically. One of its key component is the CSCAN-based Virtual Clock (CVC) scheduler, an efficiency-aware real-time disk scheduling algorithm. The CVC scheduler maintains two disk request queues. One, the QoS queue, orders requests based on the Virtual Clock queuing algorithm and the other one, the utilization queue, orders requests based on the standard CSCAN order. To pick the next disk request for service, the CVC scheduler checks whether servicing the head request in the utilization queue first will violate the finish time of the head request of the QoS queue. The head request of the utilization queue will only be dispatched if no finish time guarantee is violated. Otherwise, the head request in the QoS queue is scheduled first. The results showed that CVC achieves 25% to 80% better disk utilization efficiency than generic VC scheduling algorithm. However, although Stonehenge allows more general SLOs with multiple dimensions, such as bandwidth, capacity, latency and availability, it can only arbitrate accesses to the storage device, not to any other bottleneck components in the system.

QoSMig [28] is an adaptive rate-control scheme for migration. It leverages the traffic shaping ability from Sleds [17] and uses a unified admission control framework to control the migration speed. QoSMig defines the reward models for I/O and migration requests to capture both the performance requirements of clients I/Os and the constraints associated with migration. The migration speed is controlled through the admission control algorithm that admits request with maximum system rewards first. The trace-driven experimental study demonstrates that QoSmig can provide significant better I/O performance as compared to existing migration methodologies.

In general, scheduling-based approaches provide fine-granularity rate-control and can enforce isolation and fairness successfully. However, they are not sufficient for automatic decision making because they cannot ensure tight bounds on the workloads' performance.

2.2.4 Model-Based Approaches

Model-based approaches describe the system behaviors using models and apply them to make decisions. It was applied to construct a portable high-level library in 1995 [13]. Recently, it has been applied to make dynamic resource allocation decisions in server farms [18, 57] and to assist the storage system provisioning decision making [5, 9].

Brewer [13] used statistical models to make decisions for portable high-level optimizations such as data layouts and algorithm selection. He constructed models automatically from profiling information and used them to select the optimal parameters and implementations. He studied the models of the performance of the stencil module and the sorting algorithms and applied them to select the optimal parameters and implementations in different platforms. The results showed that both models were very accurate and can make correct decision. Finally, a portable high-level library was constructed using this method.

Menascç, *et al.*, [57] proposed a model-based scheme to tune resource configuration parameters, such as the number of threads and the maximum queue size of requests at each server. They used queuing theory to predict the system performance for given parameters, and applied the hill-climbing algorithm to find the optimal configuration values.

Chandra, *et al.*, [18] proposed an on-line dynamic resource allocation algorithm to provide service guarantees to web applications running on shared data centers. They applied time-series analysis techniques to predict expected workload parameters from measured system metrics and employed a constrained non-linear optimization technique to dynamically allocate the server resources based on the estimated application requirements.

Doyle, *et al.*, [30] designed a resource provisioning scheme for web service utility using performance models. It predicts the performance with multiple models, including server memory, storage I/O rate, storage response time and service response time, and adjusts the storage and memory allotment dynamically to satisfy the SLOs constraints in terms of response time.

Minerva [5] is an automated resource provision tool for storage systems. The goal is to find the minimum set of storage devices that can satisfy the performance requirements of input workloads. It solved this problem by constructing analytical device models to predict the performance for a

given configuration, and then using constraint optimization solver and optimizer to find the optimal configuration. Minerva targeted at automating the initial configuration of storage systems instead of run-time system management. Our work is complimentary because our system makes on-line action decisions, which can be viewed as a dynamic tuning of workload placement in response to system changes.

Ergastulum [9] is another storage system configuration tool, which was proposed after Minerva as part of the Hippodrome loop [8]. It takes the input workload, a set of allowed device types and a goal as input and generates a storage system design. Ergastulum uses analytic storage system performance models to evaluate whether a candidate design supports a given workload, and uses a generalized version of the best-fit bin packing algorithm with randomization and backpacking to search for the optimal configuration. In addition, to avoid local optima and to find-tune the design, it performs re-assignments to undo existing parts of the assignment at different granularity. Compared to Minerva, Ergastulum can find a system configuration with similar costs in a shorter execution time. It typically runs in 15% - 25% of the time of Minerva.

In summary, model-based solutions enable the systems to make complicated decisions through convenient mathematical formulations. They can explore a large number of design alternatives in a very short time. In addition, model-based approaches are not depending on any system signals, thus, they can detect and correct the system changes proactively. However, generating accurate models is still a challenging problem even after decades of research efforts. A wrong model can lead to decisions that make the system even worse. When making decisions using system models, we need to either derive accurate models, or design schemes that are not sensitive to model errors.

2.2.5 Other Works

Some existing works are relevant, but do not belong to any category just discussed. We briefly review them here.

Polus [85] is a framework for storage system QoS management. It combines declarative specification and predictive calculus as in the field of logic based programming. In Polus, system administrators do not need to specify details, such as action thresholds and parameters. Instead, they

first input knowledge in the form of “rules of thumb”. For example, prefetch requires memory and improves throughput. After that, they use machine learning techniques to quantify the relationship defined in the rules-of-thumb. For example, prefetch improves throughput when available memory is greater than 20 percent. The reasoning engine in Polus is expressed in first-order predicate calculus. When exceptions such as SLO violations happen, Polus triggers the reasoning engine and determines the corrective action to invoke.

Ganger, *et al.*, [32, 80] developed an analogy between human organizations and storage systems and proposed the self-* storage system. Their proposal is based on a set of interesting observations from human organizations, such as complaint-based tuning, try it and see, observe-diagnose-repair loop and negotiation. These observations are applied as the design guide for the Self-* storage project.

2.2.6 Summary

In this chapter, we reviewed the related work in the areas of storage system performance modeling and the automatic storage system management. We observe that the black-box approach is promising for performance modeling due to its ability to adapt to system changes automatically and the minimal requirements it places on device specific information. Existing results in storage system performance modeling improve our confidence to design a model-based management framework for storage systems.

In the domain of automatic storage system management, we observe that most existing solutions are policy-based, feedback-based, scheduling-based or model-based. Policy-based approaches front-load all the complexity into the work of creating the rules at design time, in exchange for simplicity of execution at run-time. In addition, they cannot adapt to system changes easily. Feedback-based approaches require minimum assumptions about the system. But they are not well-suited for decision-making with multiple variables [82] due to the large search space and possible long convergence time. Scheduling-based approaches can enforce isolation and fairness successfully. But they cannot ensure tight bounds on the workload’s performance. As a result, they are not sufficient to provide SLO guarantee. Model-based approaches can explore the configuration space and choose

the optimal action quickly and can predict the system state proactively to prevent the system from running into the bad state. The main challenge is to construct accurate system models. In this thesis, we develop mechanisms to select the invocation parameters that can enforce the SLO bounds automatically and quickly by combining model-based decision making and a feedback-based action execution (Chapter 4).

In addition, existing solutions focus on choosing the invocation parameters for single corrective action only. they lack the ability to generate an action plan with more than one action, which is necessary when the system can not invoke the optimal action immediately due to resource limitation or business constraints. Moreover, due to continuous changes of workloads and systems, there is no one-size-fit-all solution. The decisions about *what* corrective action(s) to invoke and *when* to invoke them need to be made dynamically. In this thesis, we develop framework and algorithms to generate an integrated multi-action plan automatically. Our plan consists of the optimal corrective actions with their optimal invocation time and invocation parameters (Chapter 6).

In the next chapter, we present the framework of SMART, a model-based action advisor that can generate an integrated multi-action plan in an automatic and responsive fashion. Details of the framework are given in Chapter 4, Chapter 5 and Chapter 6.

Chapter 3

Model-Based Automatic Action Advisor

The main goal of our work is to design a framework and algorithms to generate the corrective action plan in an automatic and responsive fashion such that the system utility can be maximized. In this chapter, we describe the design of SMART, a framework to generate the optimal corrective action plan, and give a brief introduction to its key building blocks including input modules such as system models and time-series forecasting, utility evaluator, single action tools and action advisor. In addition, we study in details one of the key elements of the framework, storage performance models. Our experimental results show that using the off-the-shelf GUIDE algorithm, by filtering the data in the saturated region and including the number of outstanding I/Os as a parameter, we can achieve an error rate of 19.3% for latency models and 4.5% for throughput models. In the following chapters, we first discuss the design of single action tools. In particular, we use throttling (Chapter 4) and migration (Chapter 5) as examples to illustrate how to choose action invocation parameters using system models, time-series analysis and constraint optimization techniques. We then describe the design of SMART's core piece, the Action Advisorand give details on how it generates an integrated multi-action plan in Chapter 6.

3.1 Motivation

With increasing storage demands and system requirements for performance, reliability and security, storage system management becomes more complicated. Today's run-time system management is handled by skilled system administrators who continuously *observe* the system behavior, *analyze* it and *invoke* corrective actions to adjust the application-resource mapping at run time. This human-driven *Observe-Analyze-Act* (OAA) loop can be slow, expensive and inefficient. To reduce the management cost and adapt to the system changes more responsively, we need mechanisms to automate the OAA loop. Existing commercial products [25, 26] and research prototypes provide excellent monitoring and execution frameworks. However, the "analyze" process is still not well-understood. We aim to develop a framework and algorithms to automate the *analyze* part to connect the gap. In particular, our work focuses on the invocation decision of corrective actions. As described in Chapter 2, existing solutions lack the ability of integrating multiple actions. Our framework, which we call SMART, aims to generate an integrated multi-action plan that simultaneously answers the questions of *what* actions to invoke, *when* to invoke them and *how* to invoke them.

In addition, since model-based approaches can explore the configuration space and choose the optimal action quickly, and proactively prevent the system from running into the bad state, our work falls into this category. SMART uses system models to estimate the resulting performance of each action option and generate an optimal action plan. In the rest of this chapter, we begin by first introducing the terminology we will use throughout the thesis (Section 3.2). We will then describe the framework of our automatic action advisor, SMART in Section 3.3. In Section 3.4, we will demonstrate our study and experimental results of performance models in storage systems.

3.2 Terminology

This section covers the terminology used in the rest of this thesis.

- **Workloads**: A typical consolidated storage infrastructure often is shared by several applications, such as online transaction, decision support, scientific computing, e-mail and web services. The I/O requests from one or more related applications are logically grouped together

and referred to as a *workload*. A workload is characterized by its I/O *access characteristics*, typically defined in terms of request rate, read/write ratio, random/sequential ratio, request size and the footprint size. The read/write ratio is the ratio of read requests to write requests. The random/sequential ratio is the percentage of I/O requests that access the data sequentially versus at random. The request size is the average transfer size of the I/Os, and the footprint size is the size of the data set accessed by this workload. These characteristics are not static and may change continuously at run-time. In our work, we assume the workloads are independent. That is, changing of one workload will not alter the characteristics of another workload.

- **Service Level Objectives (SLOs)**: SLOs are defined on a per-workload basis. They often prescribe lower bounds or service requirements for capacity, performance, availability and security that the clients expect from the storage infrastructure. For example, the capacity constraint specifies the minimum storage capacity that the client requires and is often in the unit of GB. The performance SLOs often specify the thresholds on received throughput and latency. For example, the workload should have a minimum throughput of 10 MB/s and a less than 10 ms average latency. The availability SLO has multiple parameters, such as the number of 9s. The security SLO specifies detailed security requirements such as authorization mechanisms, encryption requirements and access control schemes. Beyond these requirements, SLOs may also include information on the financial incentives and penalties when SLOs are violated. In our work, we focus on performance management, where the SLOs define the minimal required throughput and the maximum average response time.

- **Utility Functions**: The storage systems typically cannot always satisfy the SLOs for all the workloads. As such, the concept of *utility function* was borrowed from the domain of economics and was used to evaluate the degree of user's satisfaction [46]. The utility functions represent preferences of different consumptions, such as throughput received. In SMART, the utility function is defined on a per-workload basis and it associates the achieved throughput and latency with a utility value. A higher utility value indicates the corresponding performance is more preferred by the user. The goal of our management framework is to maximize the utility value delivered by the storage system to all the workloads.

- **Business-Level Constraints**: These are the constraints the storage infrastructure needs to meet due to business reasons. It may include constraints on the available resources, budgets and acceptable time overhead. They are often specified by the administrators as input to management softwares. In SMART, we consider two business constraints, the optimization window and a budget. The optimization window is the time over which the storage resources need to be provisioned and optimized. The budget constraint specifies the amount of money that can be spent on new hardwares.

- **Corrective Actions**: A corrective action changes the application-to-resource mapping to changes the utility value delivered to one or more workloads. The intent is to maximize the overall utility value delivered by the system. Many corrective actions have been developed for storage systems. For example, *throttling* action improves the system utility by rate-limiting the low-utility workloads to leave more resource to the high utility ones. *Cache repartition* modifies the workloads' share of the cache to increase the performance of the high utility ones. *Migration* moves the data of some workloads to other physical locations to redistribute the load to storage devices. *Replication* action also improves the performance by modifying the physical location of workloads. It creates additional copies of the data such that the load can be shared among the replicas. *Hardware provisioning* involves adding new physical resources to the system.

  Each action has *invocation parameters* which specify how the action should be invoked. For example, the migration action has four invocation parameters, including *what* data to migrate, *where* to place them, *when* to invoke migration and *how* fast to move data. In addition, each action also has a *cost* in terms of resource overheads, a *benefit* in improving the cumulative utility over a certain window of time, and a *lead time* to come into effect. For example, the data migration action has the cost of moving the data, the benefit of balancing the load among the components and improving the utility.

  In general, depending on their invocation overhead and impact on the storage system, corrective actions can be classified into three categories: short-term actions, long-term actions and hardware provisioning. Short-term actions, such as throttling and cache repartition, can adjust the system behavior in a very short time and have very low invocation cost. They are

most appropriate to solve system problems caused by temporary fluctuations. However, for problems due to "long-lived" workload or system changes, short-term actions often lead to sub-optimal system utility. Long-term actions, such as migration and replication, can achieve higher system utility in these situations. However, they often experience a long action lead time due to data movement. Compared to short-term actions, they often lead to higher benefit, but are not easily reversible due to the higher invocation cost. Hardware provisioning involves adding new hardware to the system. It can solve the root cause of the problem, insufficient resource. But hardware provisioning faces the risk of over-provisioning. For example, the demands may drop after new hardware arrives, thus making the new hardware un-utilized. In this thesis, we use three commonly used corrective actions, throttling, migration and hardware provisioning, to illustrate how to generate an integrated multi-action plan. We develop decision making mechanisms for throttling (Chapter 4) and migration (Chapter 5).

- **System State**: This loosely means the run-time details of the system. In SMART, system state $\mathcal{S}$ is a triple, $\mathcal{S} = \langle \mathcal{W}, \mathcal{C}, \mathcal{M} \rangle$, where $\mathcal{W}$ represents the access characteristics of the set of workloads in the system, $\mathcal{C}$ is the set of components in the system and $\mathcal{M}$ represents the mapping of workloads to components. The values in the system state typically represent the average over a sampling window, which is typically a window of 1200 seconds in data-center environment. In addition to the average value, it is important to record the variance or the histogram of the value distribution, especially for workloads with a bursty access pattern.

After describing the terminology, we next present our framework, SMART, and describe its key building blocks including input modules, utility evaluators, single corrective action tools and the action advisor. We will briefly introduce how these components work together to generate the action plan. The details of the single corrective action tools and the action advisor are covered in Chapter 4, Chapter 5, and Chapter 6.

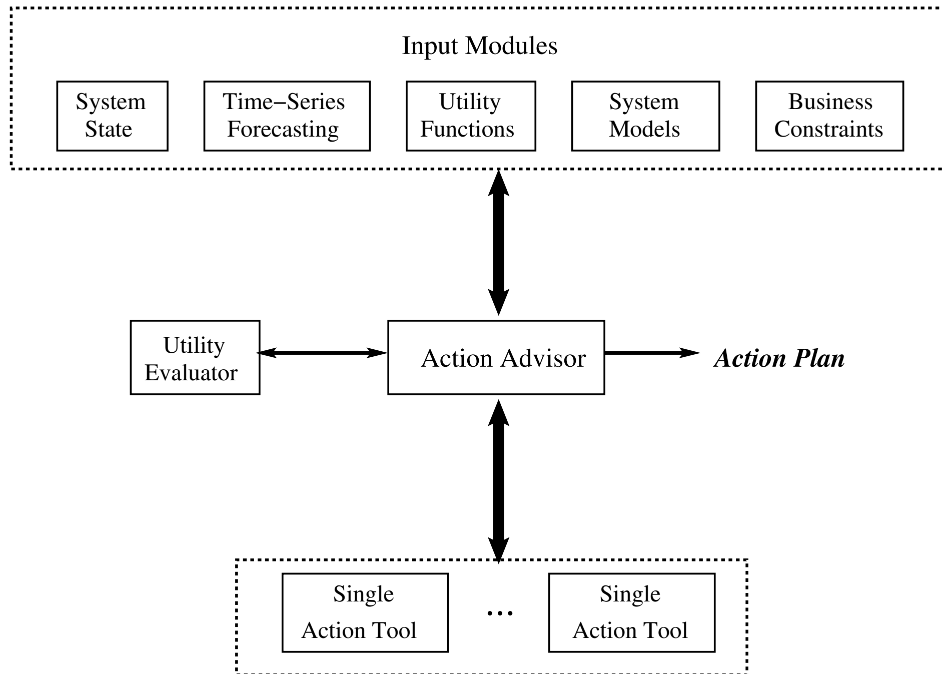


Figure 3.1. Architecture of SMART: a model-based action advisor

3.3 Framework for Model-Based Action Advisor: SMART

This section describes the framework of SMART, a model-based action advisor. SMART is a general framework and can be deployed in file systems, storage resource management software and storage virtualization boxes.

Figure 3.1 shows the architecture of SMART. It has four key components: input modules, utility evaluator, single action tools and action advisor. It takes five types of information as input so that it can react to system changes rapidly and proactively, and can compare different actions fairly (see Chapter 1 for detailed discussion). The output of SMART is the corrective action plan, which consists of the set of actions and their invocation time and parameters. The details of the key components are as follows:

- **Input modules**: They include sensors monitoring the system state $\mathcal{S}$, specifications for administrator-defined business-level constraints such as budget constraints and optimization window, utility functions, time-series forecasting of workload request-rate, and system models for the storage devices.

- **Utility evaluator**: It calculates the overall utility value in a given system state. To do so, it uses system models to interpolate the I/O performance values and then maps them to the utility delivered to the workloads.
- **Single action tools**: These tools decide the optimal invocation parameters for a corrective action in a given system state. In Chapter 4 and 5, we will use throttling and migration as examples to illustrate the design of single action tools to choose action invocation parameters automatically. Separating the single action tools from the action advisor does not only allow easy upgrade in the future, It also allows SMART to leverage existing tools for individual corrective actions when the appropriate interface is provided.
- **Action Advisor**: The Action Advisor aims to choose an optimal action schedule to improve storage system utility for a given optimization window and business-level constraints. It interacts with the single action tools and generates a time-based action schedule with details of *what* action to invoke, *when* to invoke and *how* to invoke. We will present the decision algorithm for generating an action schedule in Chapter 6.

In the rest of this section, we will describe the details of the key components of the framework.

3.3.1 Input Modules

For the input modules described below, several different well-known techniques are available. Rather than developing new techniques, we focus instead on demonstrating how these building blocks work together to generate the action plan.

- **Time-series Forecasting**. The forecasting of future workload demands is based on extracting patterns and trends from historical data. There are several well-known approaches for time series analysis of historic data such as ARIMA [84] and Neural Networks [11]. The general form of time-series functions is as follows:

$$y_{t+h} = g(X_t, \theta) + \epsilon_{t+h} \tag{3.1}$$

where: y_t is the variable(s) vector to be forecast, t is the time when the forecast is made, X_t is the predictor variable(s) vector, which usually includes the observed and lagged values of y_t till time t, θ is the vector of parameters of the function g, and ϵ_{t+h} is the prediction error.

With the assistance of Time-series forecasting, SMART can evaluate an action's impact on the future system state and suggest action options optimizing the entire optimization time-window.

- **Utility functions**. The concept of *utility function* has been used in the domain of Economics to evaluate the degree of user's satisfaction [46]. We borrow the concept the represent the client's preference of service received. Utility functions are defined on a per-workload basis. There are several different techniques to specify them. For SMART, the utility function associates workloads performance with a utility value, which reflects the user's degree of satisfaction. The utility function for each workload can be (1) provided by the administrators; (2) defined in terms of priority value and SLOs; or (3) defined by associating a dollar value to the level of service delivered, e.g., $1000/GB if the latency is less than 10ms, otherwise $100/GB.

- **System models**. System models are mathematical functions that describe the system behavior. SMART defines models for characterizing the capabilities of components in the storage system, the workload being presented to them, and the expected response to different action invocation options.

  - Component models. A component model predicts values of a delivery metric as a function of workload characteristics. SMART can in principle accommodate models for any system component. In Section 3.4.3, we will use a storage device as an example to illustrate how to construct component models using machine learning techniques. Building component models is an area of active research. Models based on simulation or emulation [33, 93] require a fairly detailed knowledge of the system's internals; analytical models [69, 74] require less, but device-specific information must still be considered to obtain accurate predictions. Black-box [6, 89] models are built by recording and correlating inputs and outputs to the system in diverse states, without regarding its internal structure. Since SMART needs to explore a large candidate space in a short time,

simulation-based approaches are not feasible due to the long prediction overhead. Analytical models and black box approaches both work with SMART. For the SMART prototype, we use a black-box approach to regress the models as a function of the workload characteristics. The models are first bootstrapped and continuously refined at run time. Details of model construction is given in Section 3.4.3.

- Workload models. The representation and creation of workload models has been an active area of research [14]. In SMART, workload models predict the load on each component as a function of the request rate of each workload. The rate of requests at component i originated from workload j may change continuously as workload j or other workloads change their access patterns. For example, a workload with good temporal locality will push other workloads off the cache.

- Action models. Action models predict the effect of corrective actions on workloads and components. For example, the throttling action alters the workload sending rate by limiting the number of requests served by storage devices and the migration action increases the load on the source and destination component due to the copy operation.

In general, for a given workload's I/O characteristics and action invocation parameters, SMART applies system models to estimate the performance of each workload, then estimate the system utility using the utility evaluator.

3.3.2 Utility Evaluator

The *Utility Evaluator* calculates the utility delivered by the storage system in a given system state. The calculation involves obtaining the access characteristics of each workload, and using the system models to interpolate their performance. Specifically, in our prototype, the Utility Evaluator uses the throughput and response-time for each workload to calculate the utility delivered by the storage system:

$$U_{sys} = \sum_{j=1}^{N} UF_j(Thru_j, Lat_j) \tag{3.2}$$

where N is the total number of workloads, UF_j is the utility function of workload j, with throughput $Thru_j$ and latency Lat_j.

In addition, for any given workload demands D_j, the maximum utility $UMax_{sys}$ is defined as the "ideal" utility if the requests for all workloads are satisfied. Utility loss UL_{sys} is the difference between the maximum utility and the current system utility. They can be calculated as follows:

$$\begin{aligned} UMax_{sys} &= \sum_{j=1}^{N} UF_j(D_j, SLO_{lat_j}) \\ UL_{sys} &= UMax_{sys} - U_{sys} \end{aligned} \quad (3.3)$$

where the SLO_{lat_j} is the latency requirement of workload j. In addition, the cumulative utility for a given time-window refers to the sum of the utility over the time-window.

3.3.3 Single Action Tools

These tools automate invocation of a single action. In principle, SMART can leverage existing single action decision tools to make decisions. Each of these tools typically includes the logic for deciding the action invocation parameter values, and an executor to enforce these actions.

As described earlier (Section 3.2), there exist many possible corrective actions. In principle, for any corrective action, once we have enough information about its invocation behaviors to build the action models, SMART can generate an integrated action plan, with the corrective action considered as action options. In the rest of this thesis, we use three commonly-used actions, throttling, migration and provisioning as examples to illustrate how SMART generates an integrated multi-action schedule. In Chapter 4 and Chapter 5, we will present the logic of invocation parameter selection for throttling and migration using system models.

3.3.4 Action Advisor

The Action Advisor generates the corrective action schedule. The steps involved are as follows (the algorithm is presented in details in Chapter 6):

- Generate and analyze the current state ($\mathcal{S}_0$) as well as look-ahead states ($\mathcal{S}_1, \mathcal{S}_2$, ...) according to the forecast future.

- Feed the system states along with the workload utility functions and performance models to the single action tools and collect their invocation options
- Analyze the cost-benefit of the action invocation options using the Utility Evaluator module and system models.
- Prune the solution space and generate a schedule of what actions to invoke, when to invoke, and how to invoke.

In this section, we described the architecture of the SMART framework and its key components. Its core piece is the Action Advisor, that communicates with the single action tools to collect the single action options and generates a multi-action plan using the input information. In the next section, we study in details one of the key elements of the input modules, storage performance models. We demonstrate how to construct accurate models using the black-box approach.

3.4 Experimental Exploration of Black-Box Based Component Models

In the previous section, we introduced the framework of SMART, a model-based action advisor. SMART suggests correction action plans based on system models, time-series forecasting, utility functions and other information. Among all inputs, system models provides the foundation for fast exploration of all possible action plans and their impact on the system. The effectiveness of the action plan generated depends on the accuracy of the system models. In this section, we use component models as an example to demonstrate how models can be constructed. Workload models and action models can be constructed with different measurable but similar techniques.

As we have discussed in Chapter 2, black-box approach is the most promising modeling technique for storage system management because it can adapt to system changes on the fly and place minimum requirements on device specific information. In this section, we use black-box approach to demonstrate how to construct component models. We evaluate the accuracy of an off-the-shelf black-box algorithm using a real-world storage system. Our goal is to understand the accuracy of

performance models in the real-world setup and to gain insights into how to improve the accuracy of performance models.

In the rest of this section, we will first introduce how to construct performance models using only those parameters that are easily measurable with standard tools such as *iostat*. Then we will present some experimental results collected in real-world setup and discuss our findings on improving the performance models.

3.4.1 Goal of Performance Modeling

The goal of performance modeling in storage systems is to have a prediction function for the observable I/O performance as a function of the total incoming load on the storage subsystem. Here, the storage subsystem can be as simple as a single disk, or as complicated as disks connected using switches and storage controllers. The total load is a summation of the load from individual workload streams (W), where each workload is characterized by the following parameters:

- $iops$ is the I/O request rate.
- $rwratio$ is the ratio of read requests to write requests.
- $rsratio$ is the percentage of I/O requests that access data sequentially versus at random.
- $reqsize$ is the average transfer size for the I/Os.
- $fprtsize$ is the size of the total data set accessed by this workload.

A storage subsystem has several performance metrics. Specifically, from the perspective of storage management, three parameters are the most important. They are enumerated below in ascending order of their modeling difficulty:

- **Detection of saturation state**: The goal is to predict if the storage system will be saturated by a set of workloads. A storage system is saturated when its operating point is beyond the knee of the typical hockey-stick curve for response time and throughput. Administrators tune the system to avoid saturation since it leads to growing queue size, increasing latency and reduced throughput. It is important to predict if an operating point is saturated.

- **Throughput**: Predict the overall throughput of the storage system as well as the per-workload throughput. The overall throughput is the sum of the throughput of all workloads in the system.

- **Latency**: Predict both the average latency and the per-workload latency. In this thesis, we use the average latency, but the techniques discussed are applicable to other values such as the 90th percentile latency value.

3.4.2 Procedure of Black-Box Based Modeling

Using the modeling terminology, we refer to the workload features ($W =< iops, rwratio, rsratio, reqsize, fprtsize >$) together with some additional measurable system parameters (if any) as *independent variable vector* and the performance metric, namely saturation state, throughput, or latency, as the *dependent variable*.

Figure 3.2. A regression tree example

We use the popular regression tree techniques for modeling. The idea of a regression tree is to estimate a regression function [1] with either a piecewise constant or piecewise function. The entire sample space starts from the root node and the tree is constructed by recursively partitioning the sample space. The internal nodes define the splitting points and each leaf node is associated with a constant or a function. This is used to predict the outcome value for all input data matching the criteria along the path to this point from the root node. There are several variants of the regression tree algorithm depending on the heuristics used to select the split variables. GUIDE is one of them.

[1] A regression function describes the relationship between dependent variable Y and the independent variable(s) (X).

It employs Chi-Square analysis of residuals and eliminates the bias in variable selection. We choose the GUIDE implementation as it is publicly available. Figure 3.2 gives an example of the regression tree returned by GUIDE.

Performance modeling has two phases: *bootstrapping* and *run-time*. In the bootstrapping phase, the system collects training data, each consists of the independent variable vector along with the corresponding value of the dependent variable. The training data is analyzed using GUIDE and a regression function is returned. In the run-time phase, the regression function takes the independent variable vector as input and outputs the corresponding dependent variable (i.e., performance prediction). Using GUIDE, the performance can be predicted by traversing the tree. For example, given the regression tree shown in Figure 3.2, for workload features $W =< 1000, 0.4, 0.8, 8, 50 >$, the $latency$ should be calculated using $Function3$. In addition, the models are evolved continuously over the life-time of the system. The models are bootstrapped using a small set of data, and continuously refined over newly observed data during the run-time phase.

3.4.3 Analyzing the Performance Models

In this section, we will create models for both storage systems with single and multiple workloads. Although in reality, it is often the case that the storage system is shared by multiple workloads, as we can see later in the section, models for single workload can help construct and understand the models for multiple workloads. After creating the base models, we will investigate techniques to improve the accuracy of the models later in the section.

3.4.3.1 Storage System With Single Workload

For a storage system with only one workload, each training data point consists of the workload features (W) and the observable performance. The model is formulated as follows.

$$Perf = Func(rwratio, rsratio, reqsize, fprtsize, iops) \tag{3.4}$$

Where the observable performance $Perf$ can be saturation state, throughput, or latency.

3.4.3.2 Multiple Workload Environment

For a storage system with more than one workload, workloads affect each other and alter the queuing delay in a very complicated manner. The aggregate performance such as the saturation state, the overall throughput and the average latency are important since they are good utilization indicators of the system. On the other hand, in real-world shared storage system, the SLOs are often defined on a per-workload basis. As a result, performance models for the throughput and latency of individual workload are critical as well. In the rest of this section, we will discuss models for the aggregate and per-workload performance, respectively.

A straight-forward way to model the performance in the multiple workload environment is to include the workload features W of all workloads as *independent variables* since all workloads will affect the performance:

$$\begin{aligned} Perf &= Func_{perf}(rwratio_1, rsratio_1, reqsize_1, fprtsize_1, iops_1, \\ &\quad ..., rwratio_n, rsratio_n, reqsize_n, fprtsize_n, iops_n) \end{aligned} \tag{3.5}$$

Here, the performance metric $Perf$ can be either the aggregate or per-workload performance, such as saturation state, the throughput and latency for each workload. In the rest of our discussion, models constructed using Equation 3.5 are referred to as the **fine** models.

The **fine** model is an intuitive extension of the single workload model. However, the feature space grows with the number of workloads in the system. It may require a very long bootstrapping time to collect enough training data for an accurate model. To solve this problem, we propose **coarse** models to trade the model accuracy for bootstrapping time. The key idea is to reduce the number of parameters by aggregating workload features together. For the aggregate performance metrics, the **coarse** models are constructed by using the single workload models (see Equation 3.4) with the weighted average features of all workloads as the input parameters, workload features W.

To create the per-workload performance models, the **coarse** approach will first construct the **fine** models for two workload scenario using Equation 3.5 with $n = 2$. Then, to predict the performance of workload i, the weighted average workload features of all workloads except workload i are estimated, referred to as W_{rest}. The performance of workload i can now be predicted by using the

two workload performance models, with W_{rest} and W_i as input parameters, shown as follows:

$$\begin{aligned} Individual_Perf_i &= Func_{two_workloads}(rwratio_i, ..., iops_i, \\ &\quad rwratio_{rest}, rsratio_{rest}, ..., iops_{rest}) \end{aligned} \tag{3.6}$$

In general, the **coarse** models are more scalable but less accurate due to fewer parameters and simpler models. Experimental results comparing the two models are given in Section 3.4.4.6.

3.4.3.3 Improving the Model Accuracy

In the models described earlier, the training data consisted of both the saturated as well as the non-saturated regions and the regression algorithm treated the data equally. However, these two regions behave very differently in storage system. The latency in the saturated region is dominated by the queuing delay and varies dramatically, which makes the modeling very difficult, if not impossible. This observation suggests that although it may be difficult to have a model accurate for both regions, it may be feasible to develop a good model for the non-saturated region only. Moreover, since in practice, the non-saturated region is where most control happens, it is where an accurate model is mostly needed. For the saturated region, the most critical control is to pull the system back to non-saturation. For this purpose, the saturation state model is sufficient—the storage management software or administrators can try various workloads configurations until the predicted saturation state becomes non-saturated.

In addition, the base models discussed earlier are constructed with only workload features as independent variables. These parameters have the advantage of easy to measure, but they provide limited information about the queuing behavior of the storage system, which plays a significant part of the latency. Other parameters such as the number of outstanding I/Os ($OSIO$) and the queue length may be difficult to measure, but they give more information on the queuing behavior. One intuitive way to improve the model accuracy is to include parameters related to queuing behavior. In our experimental section, we specifically use $OSIO$ as an example to study its impact on the model quality.

In this section, we described the procedure of constructing performance models for storage systems with single and multiple workloads. We also discussed two observations to improve the

model accuracy. In the nest section, we will present our evaluation results on the accuracy of various models using a real-world storage system.

3.4.4 Experimental Evaluation

In this section, we perform experimental study to evaluate the accuracy of various performance models. The experimental evaluation is divided into five parts. First, we evaluate the accuracy of various performance models for both single and multiple workload environments. Second, we analyze the impact of filtering data in the saturated region on the model accuracy. Third, we evaluate the impact of adding the number of outstanding I/Os as an independent variable. Fourth, we compare the model accuracy of the **coarse** models and the **fine** models. Fifth, we perform a sensitivity test of the model accuracy to the amount of training data.

In the rest of this section, we first describe the data collection procedure (Section 3.4.4.1) and the model evaluation metrics (Section 3.4.4.2). After that, we present our experimental results on the five tests.

3.4.4.1 Data Collection

The training and testing data used in our experiments are collected using a real storage system, with a host machine generating I/O streams on a fibre channel SAN. The host is an IBM X-series 440 server. It has a 2.4GHz 4-way Intel Pentium 4 CPU with 4GB RAM and runs on Redhat Server Linux 2.1 Kernel. The back-end storage is an eight drive RAID0 LUN created on a IBM FastT 900 storage controller with 512MB on-board NVRAM. The host and the storage controller are connected using a 2Gbps Fibre-Channel (FC) link. The storage system is accessed as a raw device so that we can control the workload features precisely.

We developed our own workload generator. It takes the $rwratio, rsratio, reqsize, fprtsize$ and $iops$ as input and generates the I/O requests to the storage system. The workload generator starts a number of *worker* threads. Each worker generates synchronized I/Os independently. For each test, which is one data point in the training set, we let the workload generator run for one minute to measure the throughput and latency in the stable state.

3.4.4.2 Model Evaluation Metrics

We evaluate the accuracy of a model using three metrics: the mean absolute error (MAE), defined as $|Y - \hat{Y}|$; the mean relative error (MRE) $|\frac{Y-\hat{Y}}{Y}|$ and the classic metric R in statistical learning, where, Y and $\hat{Y}$ are the observed and the predicted performance vector. R is defined as:

$$R = 1 - SS_{err}/SS_{total} \tag{3.7}$$

Where SS_{err} is the sum of square of regression errors, which is defined as $SS_{err} = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$ and reflects how well the regression function fits the data. SS_{total} is the sum of square of data errors, defined by $SS_{total} = \sum_{i=1}^{n} (y_i - \overline{y})^2$, and evaluates the randomness of the data itself. y_i, $\hat{y}_i$ and $\overline{y}$ are the ith observed value, predicted value and the mean of the observations, respectively.

3.4.4.3 Accuracy of Base Models

In this section, we study the base performance models, that only take workload features as the independent variables and all training data are treated equally. They are in contrast to later models, for which, the training data in the saturated region are filtered.

Single Workload Environment. In this test, we test 900 settings of $rwratio, rsratio, reqsize$, $fprtsize$. For each setting, the workload is generated with eight different sending rates ranging from 100 IOPs to 8000 IOPs. We have 7200 (7200 = 8*900) data points in total. Each data point consists of the workload features, the measured throughput and latency, and a saturation state label which is determined based on if the throughput is less than the sending rate. With these 7200 data points, we construct the saturation state model, the throughput model and the latency model using Equation 3.4, respectively.

For the saturation state model, out of the 7200 data points collected, 153 (2.13%) cases are mis-classified. The throughput model has a 0.89 R value, a 139.53 IOPS MAE with a 1671.7 IOPS mean throughput, and a 9.9% MRE. The latency model has a 0.882 R value, a 2.2 ms MAE out of the 12.6 ms mean latency, and a 30.3% MRE. Compared to the throughput model, the latency model has a much higher error rate, which indicates the latency is more difficult to predict. This is because the latency has a much larger variation compared to the throughput. When the system

enters the saturation region, the throughput either flattens out or reduces as the load grows, but the latency can be arbitrarily large due to the queuing effect.

Multiple Workloads Environment. For this test, in each run, we randomly pick n workloads from the 7200 settings used in previous test and start n instances of workload generator to send I/O requests. The throughput and latency for each workload are measured and the aggregate performances are calculated accordingly. The rest of our analysis is based on the 5000 training data points with three workloads running in the system. In addition, all results presented in this section are based on the **fine** model (Equation 3.5). The results of the **coarse** models are presented in Section 3.4.4.6. For the saturation state detection, out of the 5000 data points, 457 cases (9.15%) are wrongly classified. Compared to the single workload result, 2.13%, the higher mis-classification rate shows that the complicated workload interaction among multiple workloads make the performance less predictable.

Performance Metric	R **Value**	**MAE (Mean)**	**MRE**
Overall Throughput	0.677	226.5 (1614.3 IOPS)	20.8%
Average Latency	0.0028	100.2 (96.9 ms)	270%
Per-Workload Throughput	0.776	97.47 (556.9 IOPS)	23.0%
Per-Workload Latency	0.14	100.1 (99.7 ms)	257%

Table 3.1. Multiple workload environment

Table 3.1 shows the quality of models for aggregate throughput, average latency, per-workload throughput and latency. From the table, we have three observations as expected:

- The models for overall and per-workload performance have similar quality—20.8% and 23.0% MRE for throughput and 270% and 257% MRE for latency.
- The latency is more difficult to predict than the throughput. Specifically, with multiple workloads in the storage system, it is very difficult to construct a good latency model for all data points with the base model.
- The performance for multiple workload environment is less predictable than the single workload system. For example, the MRE of the throughput model grew from 10% to 20.8%. The higher error rate is a result of the complicated interaction among workloads.

3.4.4.4 Models With Saturation Data Filtered

In Section 3.4, we proposed the idea of filtering the saturated data to improve the model accuracy for the more-critical non-saturated region only. The goal of this test is to examine the effectiveness of this idea. In the rest of our discussion, we focus on the more realistic and more complicated configuration—storage system with multiple workloads.

Performance Metric	Data Set	R Value	MAE (Mean)	MRE
Per-Workload Throughput	ALL	0.776	97.47 (556.9 IOPS)	23.0 %
Per-Workload Throughput	NoSat	0.992	17.9 (490.1 IOPS)	4.13 %
Per-Workload Latency	All	0.14	100.1 (99.7 ms)	257%
Per-Workload Latency	NoSat	0.573	11 (29.9 ms)	76.8%

Table 3.2. All data vs. non-saturated data only

Table 3.2 compares the accuracy of models constructed using all data (ALL) and data in non-saturated region only ($NoSat$). For both throughput and latency, the $NoSat$ data set helps construct a better model. Specifically, the MRE of the throughput model improves from 23.0% to 4.13% and the latency model improves from 257% to 76.8%. The error rate of the latency model is still large, which indicates that additional mechanisms are needed to have a good latency model.

3.4.4.5 With Number of Outstanding IOs

In this section, we examine the effectiveness of considering the number of outstanding I/Os ($OSIO$) in the latency and throughput prediction. Here, the $OSIO$ is the number of outstanding I/Os in the whole system instead of for every workload. We choose the $OSIO$ as an indicator of the queue size because it is relatively easier to monitor.

Performance Metric	With OSIO	R Value	MAE (Mean)	MRE
Per-Workload Throughput	No	0.992	17.9 (490.1 IOPS)	4.13 %
Per-Workload Throughput	Yes	0.993	18.45 (490.1 IOPS)	4.54%
Per-Workload Latency	No	0.573	11 (29.9 ms)	76.8%
Per-Workload Latency	Yes	0.946	3.29 (29.9 ms)	19.3%

Table 3.3. Impact of OSIO

Table 3.3 compares the model accuracy with and without $OSIO$. The results are based on the

dataset $NoSat$ with multiple workloads in the system. For the per-workload throughput model, the quality of models with $OSIO$ remains roughly the same as without $OSIO$. This is because the workload generator limits the number of workers and each worker generates synchronized I/O requests. As a result, the impact of the queuing behavior is limited. However, for unsynchronized I/Os, where the queue size will grow with the sending rate, the $OSIO$ may become critical for throughput performance modeling. We leave the investigation on unsynchronized I/Os to our future work. The third and fourth rows of the table compare the accuracy of the per-workload latency models. The benefit of introducing $OSIO$ is obvious: the R value is increased from 0.573 to 0.946, the MRE is reduced from 76.8% to 19.3% and the MAE is reduced to 3.29 ms. We also examined the models the ALL data set and the results are very similar to the $NoSat$ data set. In summary, $OSIO$ is an important parameter for constructing a good latency model in the multiple workload environment.

3.4.4.6 Accuracy of Coarse Models

All of the previous results for multiple workloads are based on the **fine** model, where the features of each workload are counted as input parameters. The focus of this section is on the **coarse** models, for which, the aggregated workload features are used to approximate multiple workloads. The **coarse** models are proposed to reduce the number of parameters and the bootstrapping time to achieve better scalability. The goal of this test is to understand the cost of the workload features aggregation.

Performance Metric	**Model**	**R Value**	**MAE (Mean)**	**MRE**
Overall Throughput	coarse	0.886	146.4 (1470.4 IOPS)	10.4%
Overall Throughput	fine	0.99	37.9 (1470.4 IOPS)	2.68%
Average Latency	coarse	-0.0008	1.44 (27.9 ms)	52.7%
Average Latency	fine	0.984	1.67 (27.9 ms)	11.0 %

Table 3.4. Coarse models vs. fine models: aggregate performance

We first examine the accuracy of saturation state detection. The **coarse** model applies the saturation state model for single workload to classify the data. The average weighted workload features are used as the input. Out of the 5000 cases, 491 cases (9.82%) are mis-classified, which is similar

to the result of the **fine** model (9.13%). This shows that the **coarse** model works effectively for saturation state models.

We now compare the models of throughput and latency, for both aggregate and per-workload performance, using two modeling approaches (Table 3.4 and Table 3.5). The models are constructed after filtering the saturated data and with $OSIO$ as additional parameter. For all models, the **coarse** models have a higher error rate. However, the accuracy of throughput models are quite acceptable (MRE <10.4%). These results suggest that the **coarse** models can be a good candidate for bootstrapping the throughput model. But for latency models, the model error due to less information is significant (42.2% vs. 19.3% MRE).

Performance Metric	**Model**	R **Value**	**MAE (Mean)**	**MRE**
Per-Workload Throughput	coarse	0.988	19.03 (490.1 IOPS)	4.61%
Per-Workload Throughput	fine	0.993	18.45 (490.1 IOPS)	4.54%
Per-Workload Latency	coarse	0.467	9.05 (29.9 ms)	42.2 %
Per-Workload Latency	fine	0.946	3.29 (29.9 ms)	19.3 %

Table 3.5. Coarse models vs. fine models: per-workload performance

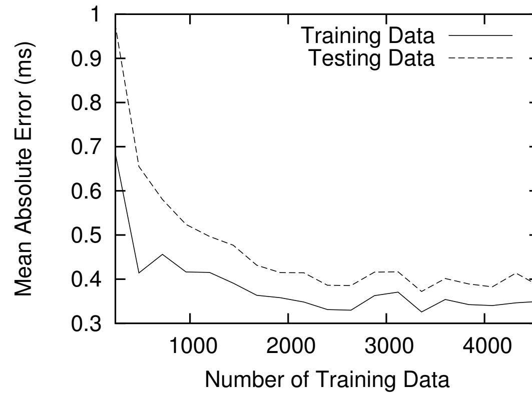
To balance the scalability and the accuracy, an intermediate solution is to construct **fine** models for m workload, in contrast to the default two workload models. m is a positive integer and reflects the trade-offs of scalability and accuracy. When more than m workload running in the storage system, the performance of workload i can be predicted by using i as one workload, grouping the remaining workloads into $m - 1$ sub-groups, and plugging in the $m - 1$ aggregate features into the m workload models.

3.4.4.7 Sensitivity Analysis of the Amount of Bootstrapping Data

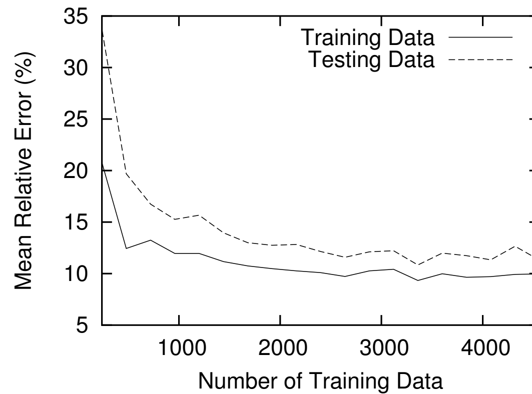
Previous tests answered the question on how good the models are. In this test, we aim to answer the question on how many training data are needed to construct a model with reasonable accuracy. The answer to this question can help us understand the bootstrapping overhead and configure the model refinement interval.

In this test, for each experiment, we randomly pick a proportion of data points from the entire data set as the training data for model construction. The rest data is then used as testing data. For

(a) R Value (Single)



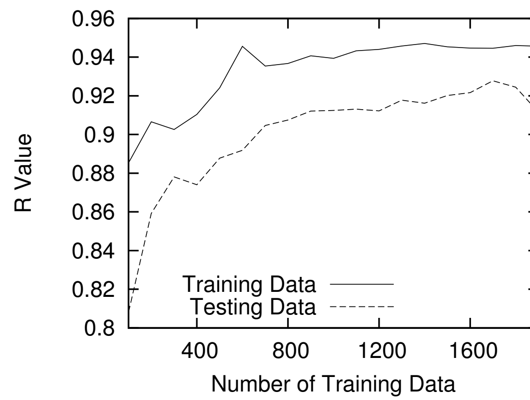
(b) Mean Absolute Errors (MAE)



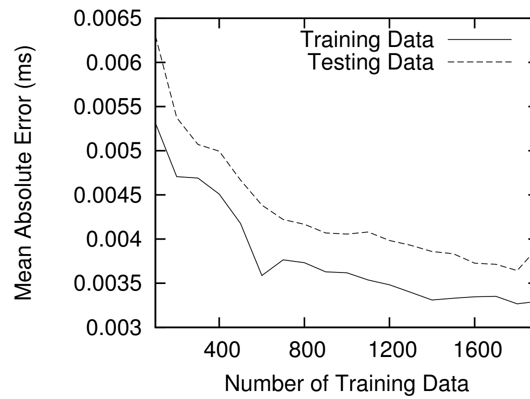
(c) Mean Relative Errors (MRE)

Figure 3.3. Impact of training size on latency models: single workload

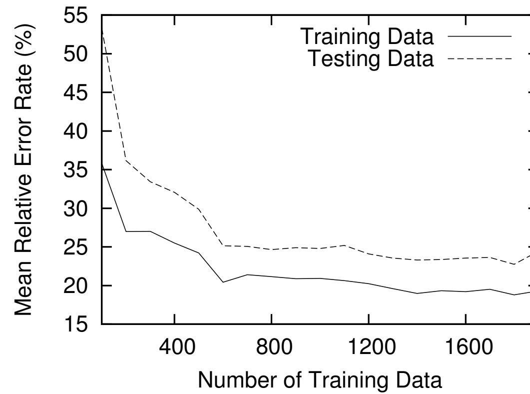
each performance model, we change the percentage of training data from 5% to 95% and record the R value, MAE and MRE on both the training and testing data. Results on all performance metrics



(a) R Value



(b) Mean Absolute Errors (MAE)



(c) Mean Relative Errors (MRE)

Figure 3.4. Impact of training size on latency models: multiple workloads

demonstrate similar trend with different knee points. We only present the results for the latency models because they are the most difficult to model. Figure 3.3 (a-c) shows the R value, MAE and

MRE of the latency models for single workload configuration. All three evaluation metrics require about 1000 non-saturated data points to reach to a stable model quality: 12% MRE on the training data and 15% on the testing data. Similarly, for multiple workloads configuration, 600 data points are necessary to construct a good latency model. Note that, each data point contains three per-workload performance values and can be used as three data points for per-workload performance model construction.

In general, the experimental results show that it is feasible to use regression tree based black-box approach to construct performance models with reasonable accuracy. For the storage system tested, we have an error rate of 19.3% for latency models and 4.5% for throughput models. In general, it is more difficult to predict the performance of a storage system with multiple workloads than that with single workload. In addition, we found that filtering the saturated data and including additional queue-size related parameter can improve the model accuracy. Last, the **coarse** models can be a candidate for fast bootstrapping. However, it has a significant lower accuracy than the **fine** models.

3.5 Conclusions

In recent years, storage system management has become more complicated due to increasing storage system consolidation. To reduce the total cost of ownership and to adapt to changes more quickly, the traditional manually controlled *observe-analyze-act* (OAA) loop must operate more automatically. In this chapter, we described the framework of SMART, a model-based automatic action advisor for storage systems. SMART aims to automate the "analyze" part of the OAA loop by generating optimal action plan for a given optimization time window. SMART takes system states, time-series forecasting, system models, utility functions and business level constraints as inputs. Its Action Advisor communicates with single corrective action tools to collect action options, and generates an action schedule based on further analysis. SMART is a model-based approach, where mathematical models are constructed to describe the system behaviors and used to explore the configuration space. As a result, the efficiency of SMART's decision is affected by the accuracy of system models. To understand the accuracy of system models in real storage systems, we used

component models as an example to demonstrate how machine learning techniques can help to build models without any expert knowledge and device-specific information. Our experimental results showed that using the off-the-shelf regression tree implementation GUIDE, by filtering the data in the saturated region and considering the number of outstanding I/Os, our latency models can achieve an error rate of 19.3% and our throughput models are around 4.5%.

In the rest of this thesis, we will first use throttling (see Chapter 4) and migration (see Chapter 5) as examples to illustrate how to develop single action tools that can determine the optimal invocation parameters using system models. We will then discuss details of how the Action Advisor can combine all information and generate an optimal multi-action schedule in Chapter 6.

Chapter 4

CHAMELEON: An Automatic Throttling Decision Tool for Storage System

In the previous chapter, we described the framework of SMART, a model-based automatic action advisor, which is designed to generate action plans by combining system models, time-series forecasting, business constraints and single action options. Before describing the design of the Action Advisor to generate an integrated multi-action plan in Chapter 6, we first discuss the mechanisms of single action tools. Specifically, in this chapter, we use throttling as an example to illustrate how to use system models to make the invocation decision automatically. Throttling is one of the commonly used corrective actions in modern storage systems. It improves the system utility by rate-limiting the low-utility workloads to leave more resource to the high utility ones. In this chapter, we present an automatic throttling framework, CHAMELEON.[1] In the SMART framework, CHAMELEON can act as the single action tool for throttling action. It takes the system models and system states as input and outputs the throttling options with invocation parameters to the Action Advisor. CHAMELEON makes the throttling decision by combining system models and constraint optimization. In our experiments, we replay traces from production environments in a real storage system. With our current system and the workload perturbations we impose, CHAMELEON can ana-

[1]Our throttling framework changes its decision making strategy among heuristic-based, model+heuristic based and model-based approaches according to the quality of system models. This is similar to the way Chameleon varies its color to adapt to the environment changes.

lyze and correct performance violations using the feedback loop in 3-14 minutes—which compares very favorably with the time a human administrator would have needed. In addition, CHAMELEON can also be used as a stand-alone tool to automatically make throttling decisions and correct SLO violations. In the rest of this chapter, we will first motivate the need for a throttling tool that can make intelligent throttling decisions to minimize the number of SLO violations and provide service isolation and performance differentiation in Section 4.1. Then we will introduce the architecture of CHAMELEON (Section 4.2) and describe its main components: the models (Section 4.3), the reasoning engine (Section 4.4), the base heuristics (Section 4.5) and the feedback-control of throttling execution (Section 4.6). After that, we describe our prototype and experimental results (Section 4.7). In Section 4.8, we describe how CHAMELEON works in the SMART framework. We summarize in Section 4.9. In the next chapter, we will discuss another commonly used corrective action, migration, to demonstrate how to discover more complicated action options by combining system models, time-series forecasting and risk management.

4.1 Motivation

We have argued in Chapter 1 that, to maximize the system utility, we need to invoke corrective actions dynamically to change the application-resource mapping at run time. As described in Section 3.2, according to the invocation time and action impact, corrective actions can be classified into short-term actions, long-term actions and hardware provisioning. Throttling is one of the most commonly used short-term actions. It controls the rate of work admitted into the storage system, using mechanisms such as token-bucket algorithms [31] to regulate the resource competition and lead to a fair, predictable resource allocation. Compared to long-term and hardware provisioning actions, it has the advantages of low invocation cost and can correct the system's behavior in a very short time—sometimes instantaneously.

Because of its short invocation time and low invocation cost, throttling has been widely-used in storage systems. Researchers have proposed many mechanisms to control throttling automatically. Existing solutions have following limitations. Policy-based solutions, such as Sleds [17], lack the ability to adapt to system changes automatically. Feedback-based solutions, such as Façade [55]

and Triage [47], fail to provide service isolation and performance differentiation. Schedule-based solutions [45] cannot enforce a tight bound on the performance, which is essential for meeting the SLO constraints. We need an automatic decision tool that can minimize the number of SLO violations, and throttle the right workloads by the right degree. The right workloads are those that are supposedly responsible for the problem or those with lower priority. These workloads should be throttled just enough to correct the SLO violations.

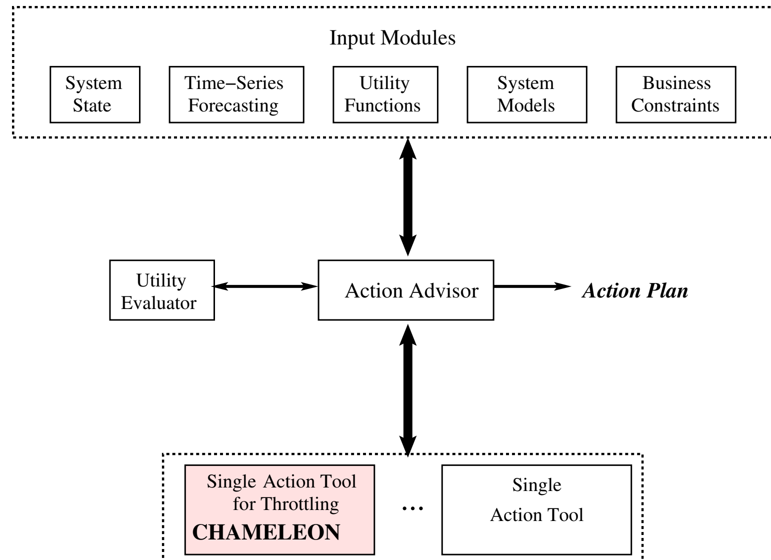


Figure 4.1. CHAMELEONin the SMART framework

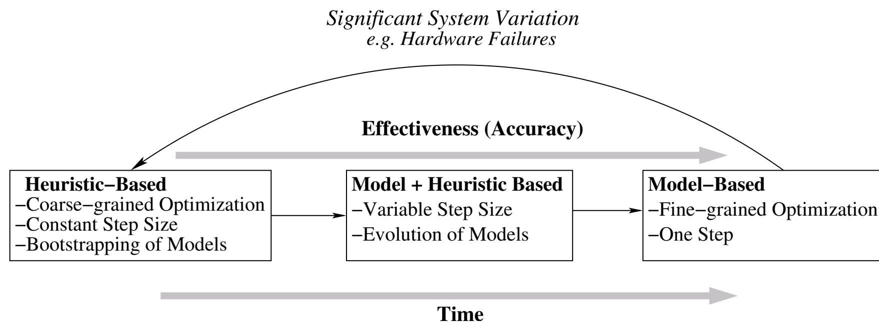


Figure 4.2. CHAMELEON moves along the line according to the quality of the predictions generated by the internally-built models at each point in time.

In this chapter, we present CHAMELEON, a model-based throttling tool for storage systems. CHAMELEON can work as the throttling decision tool in the SMART framework (Figure 4.1), as well as a stand-alone throttling tool. It defines its own logic of selecting throttling parameters and invocation strategy. Its goal is to make more accurate throttling decisions as it learns more about

the characteristics of the running system and the workloads. The key idea is to use system models to make optimal decisions, feedback loops to fine tune the action execution at run time, and heuristics as a fall-back strategy. As shown in Figure 4.2, CHAMELEON can operate at any point in a continuum between decisions based on relatively uninformed, deployment-independent heuristics, and blind obedience to models of the particular system. CHAMELEON's throttling decision consider the priorities of workloads, and therefore, can provide service isolation and performance differentiation. In addition, CHAMELEON's design choice of introducing the feedback-loop and heuristics also shows how a carefully designed actuator can compensate for the inaccuracy of the "analyze" decision.

In the rest of this chapter, we will first describe CHAMELEON as a stand-alone subsystem since we originally designed it as an early prototype of the more general SMART framework. We start from introducing the framework of CHAMELEON 4.2, then proceed to describe its main components: the models (Section 4.3), the reasoning engine (Section 4.4), the base heuristics (Section 4.5) and the feedback-controlled throttling (Section 4.6). After that, we describe our prototype and experimental results in Section 4.7. We evaluate CHAMELEON using a real system testbed. With our current system, for our test scenarios, CHAMELEON corrects the system behavior between 3 and 14 minutes. In Section 4.8, we describe how to integrate CHAMELEON into the general SMART framework by applying general optimization techniques in the reasoning engine. We summarize CHAMELEON in Section 4.9.

4.2 Overview

CHAMELEON is a framework for providing predictable performance to multiple clients accessing a common storage infrastructure using throttling actions. We begin with CHAMELEON's design as a stand-alone decision tool. We then describe how to generalize its design to service as a single action tool within the SMART framework in Section 4.8.

Each workload j has a priority value P_{W_i} and a known SLO_j associated with it. The goal of CHAMELEON is to maximize the number of workloads satisfying their SLOs. In addition, each workload uses a fixed set of components—its *invocation path* —such as controllers, logical vol-

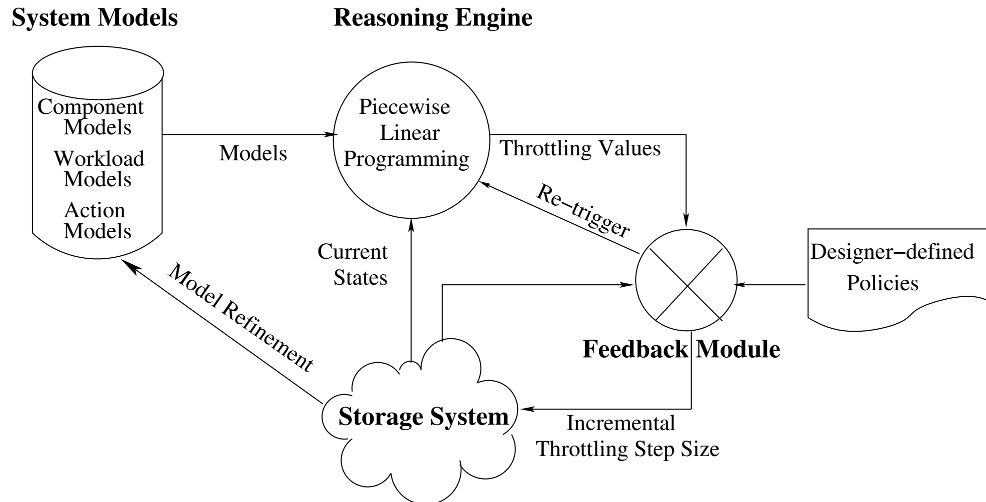


Figure 4.3. Architecture of CHAMELEON

umes, switches, and logical units (*LUNS*). CHAMELEON assumes SLOs are *conditional*—a workload will be guaranteed a specified upper bound on average I/O latency, as long as its I/O rate is below a specified limit. An SLO is *violated* if the rate is below the limit, but latency exceeds its upper bound. If workloads exceed their stated limits on throughput, the system is under no obligation of guaranteeing any latency. Obviously, such rogue workloads are prime candidates for throttling. But in some extreme cases, well-behaved workloads may also need to be restricted.

CHAMELEON automates the entire OAA loop. It can monitor, and optionally delay every I/O processed by the system. This can be implemented at each host, as in our prototype, or at logical volume managers, or at block-level virtualization appliances [34]. It then periodically evaluates the SLOs, i.e., the average latency and throughput value of each workload. When SLOs are violated, CHAMELEON performs its analysis and decides how much the workloads should be throttled. The throttling decision is enforced using a token-bucket protocol [31] on a per workload basis, where a I/O request can be admitted to the system only when tokens are present. As a result, by controlling the speed of tokens added to the bucket (referred to as token issue rate), the sending rate of each workload is limited.

To summarize the presentation thus far, the core of CHAMELEON consists of four parts, as shown in Figure 4.3:

- **System models**: by taking periodic performance samples on the running system,

CHAMELEON builds internal *black-box models* of system behavior without any human supervision. Models become more accurate as time goes by because CHAMELEON refines them automatically.

- **Reasoning engine**: CHAMELEON employs optimization techniques based on system models. It computes throttle values, and quantifies the statistical confidence of its own decisions.
- **Designer-defined policies**: As a fall-back mechanism, we maintain a set of fixed heuristics specified by the system designer for system-independent, coarse-grained resource arbitration. If the confidence value from the reasoning engine is below a certain threshold, such as during bootstrapping of the models, CHAMELEON falls back on the fixed policies to make decisions.
- **Informed feedback module**: To account for model errors and real system fluctuation, CHAMELEON employs a feedback loop to execute the throttling action. The general guiding principle is to take radical corrective actions as long as they are warranted by the available knowledge about the system.

In this section, we described the architecture of CHAMELEON and its key components. In the next section, we discuss how to construct the workload and action models for throttling using machine learning techniques.

4.3 System Models

CHAMELEON uses the same system models as SMART (see Chapter 3). It builds models in an automatic, unsupervised way and uses them to characterize the capabilities of components of the storage system, the workloads being presented to them, and their expected responses to different levels of throttling. Since we have studied the component models in details in the previous chapter, we will focus on the specific workload models and action models for throttling here.

4.3.1 Workload Models

Representation and creation of workload models has been an active research area [14]. In CHAMELEON, workload models predict the load on each component as a function of the request rate that each workload injects into the system. For example, we denote the predicted rate of requests at component i originated by workload j as w_{ij}(*workload_request_rate*$_j$). Regression techniques can be applied here to predict the w_{ij} as a function of workload_request_rate$_k$, where k includes all workloads running on component i. In addition, function w_{ij} changes continuously as workload j changes or other workloads change their access patterns. For example, a workload with good temporal locality will push other workloads out of the cache. To account for these effects, the function w_{ij} is refined continuously.

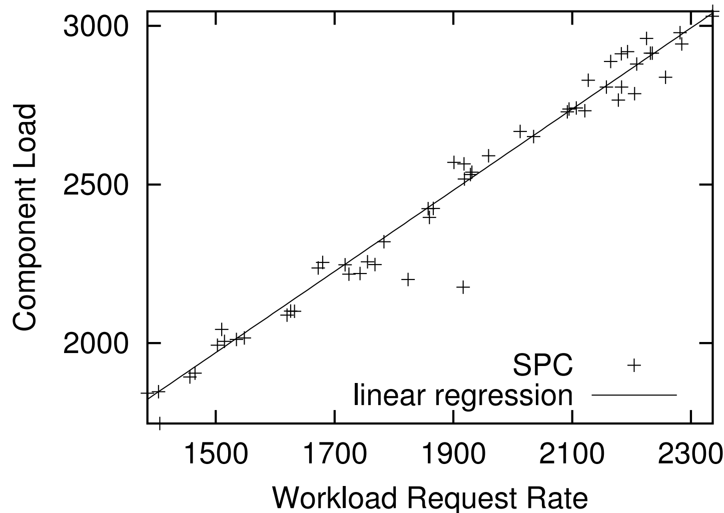


Figure 4.4. Workload model for SPC.

Figure 4.4 shows the workload models for the SPC web-search trace [27] accessing a 24-drive RAID 1 LUN on an IBM FAStT 900 storage controller with 512MB on-board NVRAM. The SPC trace was collected by running the SPC benchmark, which uses a highly efficient multi-threaded workload generator to emulate sophisticated enterprise class multi-user I/O applications [27]. We developed our own workload generator to replay the trace. It uses tokens to control the workload request rate and sequentially scans the trace to inject the corresponding I/O requests to the storage system (see Section 4.7 for details). The training data was collected by varying the workload request rate and recording the corresponding load seen by the storage controller using *iostat* [1]. In Figure

4.4, a workload request rate of 1500 IOPS in SPC translates to 2000 IOPS at the controller. The extra 500 IOs are a result of the disassembling operation introduced by the device driver's limitation on I/O request size.

In practical systems, reliable workload information can only be gathered from production runs. Therefore, we bootstrap workload models by collecting performance observations. CHAMELEON resorts to throttling heuristics in the meantime, until workload models become accurate enough.

4.3.2 Action Models

In CHAMELEON, action models predict the effect of corrective actions on workloads. The throttling action model computes each workload's average request rate (the input for workload models) as a function of the token issue rate, i.e., $a_j(token_issue_rate_j)$. Real workloads exhibit significant variations in their I/O request rates due to burstiness and ON/OFF behaviors [12]. We model a_j as a linear function: $a_j(token_issue_rate_j) = \theta \times token_issue_rate_j$ where θ is initially set to 1 for bootstrapping. This simple model assumes that the components in the workload's invocation path are not saturated.

In addition, function a_j will also deviate from our linear model because of performance-aware applications and higher-level dependencies between applications. We define performance-aware applications as those that modify their access patterns based on the I/O performance they experience.

In this section, we described the architecture and system models of CHAMELEON. We now proceed to describe the design of the reasoning engine of CHAMELEON. We will discuss how to predict the workloads performance for a given throttling decision by composing system models and how to find the optimal token issue rates using linear programming techniques.

4.4 Reasoning Engine

The reasoning engine computes the rate at which each workload should be allowed to issue I/Os to the storage system. We implemented it as a constraint solver using piecewise-linear programming [2]. The constraint solver analyzes all possible combinations of workload token rates

and selects the one that optimizes an administrator-defined *objective function*, e.g., “minimize the number of workloads violating their SLO”, or “ensure that highest priority workloads always meet their guarantees” (Section 4.4.2). The reasoning engine is not only invoked upon an SLO violation to decide throttle values, but also periodically to unthrottle the workloads if the load on the system is reduced (Section 4.4.3). In addition, the output of the constraint solver is assigned a *confidence value* based on the errors associated with the models (Section 4.4.4)

In the rest of this section, we first describe the intuition of the reasoning engine (Section 4.4.1) and give the formalization after that (Section 4.4.2). We then discuss the mechanisms of unthrottling (Section 4.4.3) and the calculation of confidence value (Section 4.4.4).

4.4.1 Intuition

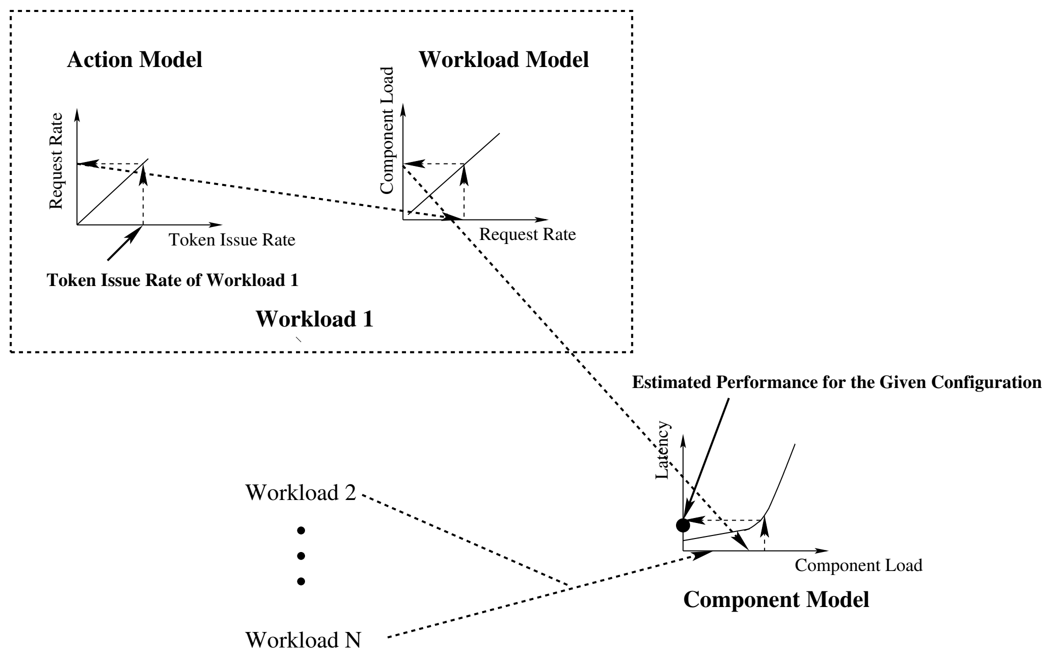


Figure 4.5. Overview of constraint optimization.

The reasoning engine relies on the component, workload, and action models as oracles on which to base its decision-making. Figure 4.5 illustrates a simplified version of how the constraint solver builds a candidate solution: (1) for a given configuration of token issue rates, it uses the action models to determine the change of the request rates for each workload, (2) it queries the work-

load model for each workload using the *under-performing* component to determine the change in the workload's I/O injection rate to the component. An underperforming component is the storage devices with workloads violating their SLOs, (3) it uses the component models to determine the throughput and latency for each workload, and (4) it records the value of the objective function for the given configuration. CHAMELEON repeat the same procedure for all combinations of components and token issue rates, and return the configuration leading to the optimal value of the objective function.

4.4.2 Formalization in CHAMELEON

We formulate the task of computing throttle values as a constraint optimization problem:

Variables: token issue rate t_i for workload i, $i = 1, 2, \ldots, n$.

Objective function: Many objective functions can be accommodated by the current CHAMELEON prototype. Moreover, it is possible to switch them on the fly. For our experiments, we choose an objective function that can

- Minimize the number of workloads violating their SLOs.
- Minimize the degree of throttling. Workloads should operate as close to the SLO throughput boundary as possible.
- Provide performance differentiation. Workloads with lower priority should have a higher probability of being throttled compared with the ones with higher priority.
- Provide service isolation. The throttling decision should consider the received performance of workloads. Those workloads that are responsible for the problem, such as those that are sending more I/Os than they are allowed, should be throttled with higher probability.

To correlate the probability of being throttled with the received performance, we characterize workloads into one of four regions (see Figure 4.6) according to their current request rates, latency and their SLO goals. Region names are mostly self-explanatory. *Lucky* workloads are getting a higher throughput while meeting the latency goal. *Exceeded* workloads get higher throughput at the expense of high latency. Each region is associated with a *quadrant priority* (represented as P_{quad_i}),

which represents the probability that workloads in that region will be selected as throttling candidates. In our formalization, we assume the P_{quad_i} is a constant. It is also possible to change it dynamically at run-time.

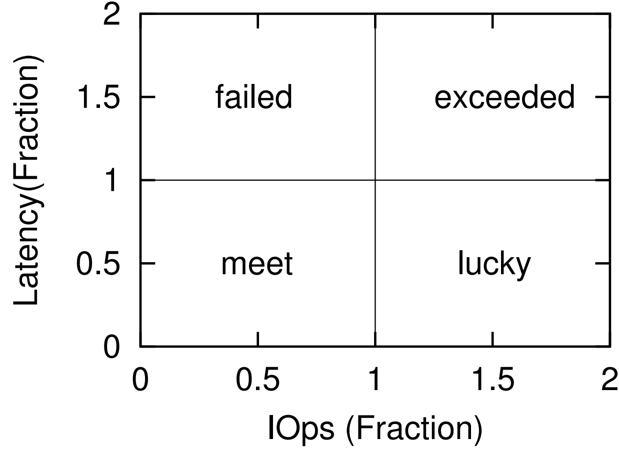


Figure 4.6. Workload classification. Region limits correspond to the 100% of the SLO values.

To meet all requirements, we define the objective function as follows:

$$\text{Minimize} \quad \sum_{i \notin \texttt{failed}} \left| P_{quad_i} P_{Wi} \frac{SLO_{Wi} - a_i(t_i)}{SLO_{W_i}} \right|$$

where P_{Wi} is the *workload priority* for workload i, P_{quad_i} is the *quadrant priority*, and $a_i(t_i)$ represents the action model for workload i.

This objective function can successfully capture all requirements. For example, the goal of minimizing the degree of throttling is enforced by introducing the item $\frac{SLO_{Wi} - a_i(t_i)}{SLO_{W_i}}$, the goal of performance differentiation is captured by the item P_{Wi}, and the goal of service isolation is achieved by introducing the P_{quad_i}.

Constraints: Constraints are represented as inequalities: the latency of a workload should be less than or equal to the value specified in the SLO. More precisely, we are only interested in solutions that satisfy $latency_{W_j} \leq SLO_{W_j}$ for all workloads W_j running in the system. We estimate the contribution of component i to the latency of W_j by composing our system models, i.e., $latency_{i,j} = c_{ij}(w_{ij}(a_j(t_j)), \cdots, w_{ik}(a_k(t_k)))$, where k is a workload running on component i. In this composition, $a_k(t_k)$ estimates the request rate of workload k given token issue rate t_k, which is then taken as input to the workload model w_{ik} to estimate the load at component i originated from

workload k. The same estimation is performed for all workloads running on component i. The component model c_{ij} estimates the performance of workload j on component i as a function of the load originated from all workloads running on component i.

The latency of workload j in the system is the sum of the latency along its invocation path (shown in Equation 4.1). The throughput of workload j can be estimated similarly, with the $throughput_j = min(throughput_{i,j})$ for all i on its invocation path.

$$latency_j = \sum_{i \in\ invocation\ path} latency_{i,j} \tag{4.1}$$

In this section, we have described how CHAMELEON makes the throttling decision by composing system models and applying constraint optimization technique. System model composition helps to determine the efficiency of a throttling decision without executing it. Constraint optimization technique allows us to explore the decision candidate space in a very short time. In addition, the design of our objective function can find an accurate throttling decision as well as providing service isolation and performance differentiation. Next, we will discuss how CHAMELEON unthrottles the workloads using the same formalization when the system load is reduced.

4.4.3 Workload Unthrottling

CHAMELEON invokes the reasoning engine periodically, to re-assess token issue rates. If the load on the system has decreased since the last invocation, some workloads may be unthrottled to re-distribute the unused resources based on workload priorities and average I/O rates. If a workload is consistently wasting its tokens because it has less significant needs, unused tokens will be considered for re-distribution. On the other hand, if the workload is using all its tokens, they won't be taken away from it, no matter how low its priority is. CHAMELEON makes unthrottling decisions using the same objective function with additional "lower-bound" constraints such as not allowing any I/O rate to become lower than its current average value.

4.4.4 Confidence on Decisions

The confidence value of the reasoning engine is based on the accuracy of the models. With system models constructed using black box approaches, the inaccuracies can stem from errors due to curve-fitting, and also from trying to use the models outside of the region(s) in which they were trained. The regression of system models is a multivariate regression problem, which predicts the dependent variable $\hat{y}$ as a function of multiple independent variables x and has the form of $\hat{y} = x^T\hat{\beta} + A + E$, where $\hat{\beta}$ is the coefficient vector, A is the Y intercept and E is an error term. There are multiple ways of capturing the inaccuracy for the multivariate regression models [44]. We choose the standard error $S_{\hat{y}}$ (see Equation 4.2) of the model prediction to capture both the errors from regression and from residuals.

$$S_{\hat{y}} = \sigma\sqrt{1 + x_*^T(X^TX)^{-1}x_*} \tag{4.2}$$

where σ is the standard error [2] of the regression model, x_* is the input vector whose y value to be predicted, and X is the observed input matrix used to construct the regression model. In CHAMELEON, we represent the confidence value CV of a model as $CV = 1 - S_{\hat{y}}$. Based on the value of CV, CHAMELEON switches its operation point among model-based or heuristic-based approaches, and tunes the aggressiveness of the throttling execution by changing the feedback step size (see Section 4.6 for details). An example transition strategy is to apply heuristics when the $CV < 0.6$, else follow the output of the reasoning engine.

4.5 Designer-Defined Policies

The system designer defines heuristics for coarse-grained throttling control. The system uses heuristics to make decisions whenever the predictions of the models cannot be relied upon—either during bootstrapping or after significant system changes such as hardware failures. The former is captured by the confidence value (CV) of the reasoning engine as described in Section 4.4.4. The latter is detected by monitoring the mean relative error of the predicted performance ($\hat{y}$) and

[2] The standard error is the estimated standard deviation of a statistic. It provides a simple measure of uncertainty in a value [78].

the observed value (y). The mean relative error is defined as $|\frac{y-\hat{y}}{y}|$. When the error is above some threshold, such as 40%, CHAMELEON switches to heuristics. For example, when hardware failures happens, the observed latency increases to six times of the predicted value, thus forcing the CHAMELEON to switch to heuristics.

These heuristics can be expressed in a variety of ways such as Event-Condition-Action (ECA) rules or hard-wired code. An example of the ECA rule is "when SLO is violated, if system utilization is greater than 85%, throttling workloads in the `lucky` region by 10%". In any case, fully specifying corrective actions at design time is an error-prone solution to a highly complex problem [85], especially if they are to cover a useful fraction of the solution space and to accommodate priorities. It is also very hard to determine accurate threshold values to differentiate different scenarios, in the absence of any solid quantitative information about the system being built. In CHAMELEON, the designer-defined heuristics are implemented as simple hard-wired code (see below).

1. Determine the list of components being used by the under-performing workload, referred to as $compList$.

2. For each component in the $compList$, add the non-underperforming workloads using it to the candidate list (the $candidateList$).

3. Sort the $candidateList$ first by current operating quadrant: *lucky* first, then *exceed*, then *meet*. Within each quadrant, sort by workload priority.

4. Traverse the $candidateList$ and throttle each workload, either uniformly or proportionally to its priority (e.g., the higher the priority, the less significant the throttling).

4.6 Informed Feedback Module

CHAMELEON's reasoning engine determines the token issue rate based on the estimated performance. The accuracy of the decision depends on the accuracy of system models used for performance estimation. In real world, system models always have errors from regression or from residuals. Blind obedience to decisions made based on such models may lead to over-throttling, under-utilizing the system, or under-throttling, leaving the SLOs violation unsolved. To compensate

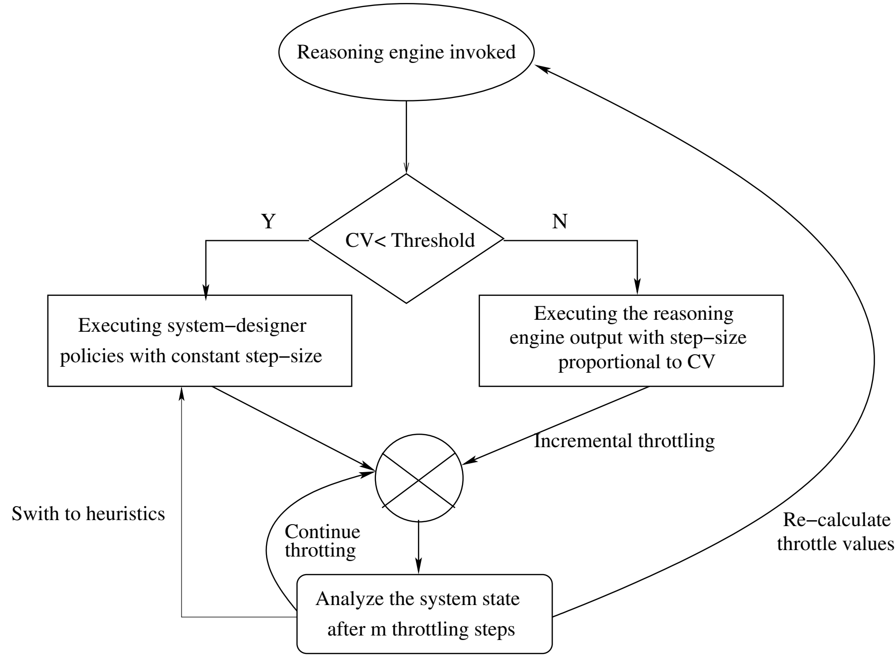


Figure 4.7. Operation of the feedback module.

for model errors and to account for system fluctuation, CHAMELEON incorporates a feedback module to execute the throttling strategy. The feedback module (see Figure 4.7) incrementally throttles workloads based on the decisions of either the reasoning engine or the system-designer heuristics. If CHAMELEON is following the reasoning engine, throttling is applied at incremental *steps* whose size is proportional to the confidence value of the reasoning engine; otherwise, the throttling step is a small constant value. In our prototype, when CHAMELEON follows the reasoning engine, the throttling is applied in following manner:

- Initialize the token issue rate to $\alpha * distance$, where $distance$ is the difference between the targeting token issue rate (t_j) and the current token issue rate, α is configured proportional to the confidence value of the constraint solver. Specially, in our prototype, $a \in \{0, 0.5\}$ and grows linearly with $CV = 1 - S_{\hat{y}}$.
- In each step, increase the token issue rate by $\frac{(1-\alpha)*distance}{b}$, where $(1 - \alpha) * distance$ is the remaining distance to the target after the initial step, b is the total number of steps planned, and is configured as a piece-wise constant value depending on the value of CV.

After every m throttling steps, the feedback module analyzes the state of the system and de-

cides among continue throttling, re-invoking the reasoning engine and switching to the designer heuristics. The feedback-module re-invokes the reasoning engine if any of the following condition is true:

- Latency increases for the under-performing workload. That is, it moves away from the *meet* region.
- A non-underperforming workload moves from the *meet* or *exceed* to *lucky*.
- Any workload undergoes a two times or greater variation, compared to the values at the beginning of throttling, in the request rate or any other access characteristic.
- There is a two times or greater difference between the predicted and observed response times for a component.

In addition, CHAMELEON switches to designer heuristics if any of the following happens: (1) the CV is below the pre-defined threshold, (2) the mean relative error of the predicted and the observed performance is above some threshold, or (3) the reasoning engine has been re-invoked consecutively for l times. Otherwise, the feedback module continues applying the same throttling decisions in incremental steps.

We have described how CHAMELEON combines system models and constraint optimization techniques to generate optimal throttling parameters, uses heuristic policies as the fall-back mechanism when the confidence value of the reasoning engine is low, and incorporates a feedback loop to compensate for the inaccuracy of models. By doing so, CHAMELEON takes the advantages of the model-based, feedback-based and the policy-based approaches. It makes more accurate throttling decisions as we know more about the running system and the workloads. In the next section, we present the evaluation results of CHAMELEON in a real system testbed using both synthetic workloads and real-world trace replay. Our focus is on understanding CHAMELEON's ability to select the right set of workloads, the effectiveness of the feedback loop and its responsiveness in correcting SLO violations.

4.7 Experimental Evaluation

The key capability of CHAMELEON is to regulate resource load so that SLOs are achieved. In our experiments, we use a variety of combinations of synthetic and real-world request streams to evaluate the effectiveness of CHAMELEON. As expected, synthetic workloads are easier to handle than their real-world counterparts that exhibit burstiness and highly variable access characteristics. In the rest of this section, we will first describe our testbed configurations (Section 4.7.1) and evaluation metrics (Section 4.7.2). We then present our results using synthetic workloads (Section 4.7.3) and real world trace replay (Section 4.7.4).

4.7.1 Testbed Configuration

The experimental setup consists of a host machine generating multiple I/O streams to a shared storage infrastructure. The host is an IBM X-series 440 server, which has a 2.4GHz 4-way Intel Pentium 4 CPU with 4GB RAM and runs the Redhat Server Linux 2.1 kernel. The back-end storage is a 24-drive RAID 1 LUN created on a IBM FAStT 900 storage controller with 512MB on-board NVRAM, and accessed as a raw device so that there is no I/O caching at the host. The host and the storage controller are connected using a 2Gbps FibreChannel link.

For synthetic workloads, we develop our own workload generator that takes the $rwratio, rsratio, reqsize, fprtsize$ and $iops$ (see Chapter 3) as input and generates the I/O requests to the storage system. The workload generator starts a number of $worker$ threads. Each worker generates synchronized I/Os independently.

For real world trace replay, we use the web-search SPC trace [27] and HP's *Cello96* trace [68] of a department file server. Both are block-level traces with timestamps recorded for each I/O request. In our experiments, we first pre-process the traces to count the number of I/O requests issued in every $10ms$ interval. At run-time, our workload generator creates a thread that is woken up every $10ms$ to allocate tokens to a worker thread according to the pre-processing results. By doing this, we can control the number of I/O requests to be issued in every $10ms$ interval. When a worker thread receives tokens, it sequentially scans the traces to inject the corresponding I/O requests to the storage system. Note that the rationale for batching I/O requests for every $10ms$ is that $10ms$

is the default system clock interrupts in our system, and therefore, is the minimum timeslice and scheduler quanta.

4.7.2 Evaluation Standard

In our evaluation, we examine the quality as well as the efficiency of CHAMELEON's decision. In terms of quality, a good throttling decision should (1) minimize the number of workloads in the *failed* region; (2) enforce a minimum degree of throttling. That is, the sending rate after throttling should be as close to the SLO throughput boundary as possible; and (3) throttle the right workloads. That is, the workload priority and quadrant priority should be correctly accounted for. A throttling decision is viewed as a good decision when it matches all three criteria. To evaluate the efficiency of CHAMELEON, we measure the time that CHAMELEON takes to correct the SLO violations, defined as the duration from SLO violation happens to SLOs are satisfied again.

4.7.3 Using Synthetic Workloads

The synthetic workload specifications used in this section are derived from a study in the Minerva project [5] (see Table 4.1). Our experiments in this section serve two objectives. First, they evaluate the correctness of the decisions made by the constraint solver. Throttling decisions should take into account each workload's priority, its current operating point compared to the SLO, and the percentage of load on the components generated by the workload. Second, these tests quantify the effect of model errors on the output values of the constraint solver and how incremental feedback helps the system converge to the optimal state.

Workload	Request size [KB]	Rd/wrt ratio	Seq/rnd ratio	Foot-print [GB]
W_1	27.6	0.98	0.577	30
W_2	2	0.66	0.01	60
W_3	14.8	0.641	0.021	50
W_4	20	0.642	0.026	60
W_5	2	1	1	0.01

Table 4.1. Synthetic workload streams.

We report experimental results by showing each workload's original and the post-throttling position in the classification chart as defined in Figure 4.6. In addition, to demon-

strate the degree of throttling for each workload, we give the throttling ratios, defined as $\frac{old_token_issue_rate_i - new_token_issue_rate}{old_token_issue_rate}$, in the figures (shown as t_i) rather than the absolute value of token rates.

Case 1: Effect of workload and quadrant priorities

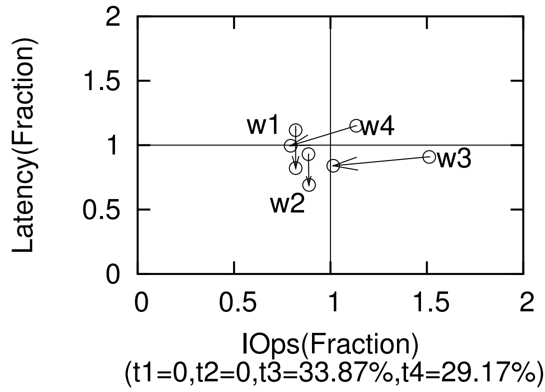
Figure 4.8 compares the direct output of the constraint solver with equal priority values (Figure 4.8(a)), different workload priorities ($W_1 = 8, W_2 = 8, W_3 = 2, W_4 = 8$) (Figure 4.8(b)) and the quadrant priorities (*failed* = 16, *meet* = 2, *exceed*=8 , *lucky*=8) (Figure 4.8(c)). In comparison to the equal priority scenario, W_3 (Figure 4.8(b)) and W_2 (Figure 4.8(c)) are throttled more when priorities are assigned for the workloads and quadrants respectively. This is because the constraint solver optimizes the objective function by throttling the lower priority workloads more aggressively before moving on to the higher priority ones.

Case 2: Usage of the component by the workload

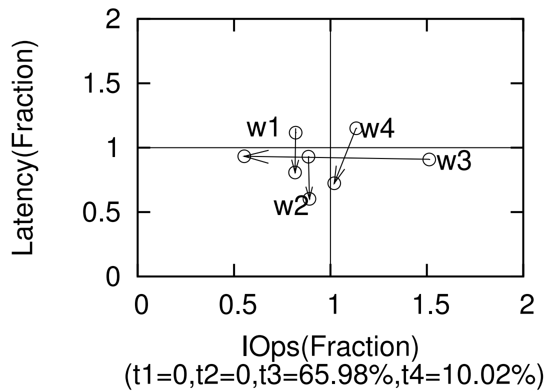
This experiment is a sanity check with workload W_5 operating primarily from the controller cache (2KB sequential read requests). Because W_5 does not consume disk bandwidth, the reasoning engine should not attempt to solve the SLO violation for W_1 by throttling W_5 even if W_5 has the lowest priority. As shown in Figure 4.9, the reasoning engine selects W_2 and W_3.

Case 3: Throttling executor combined with the Reasoning engine

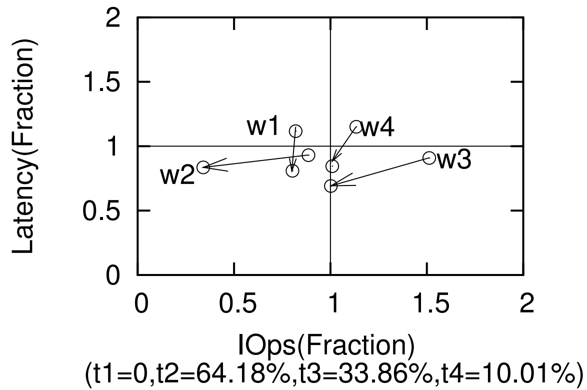
In CHAMELEON, the output of the reasoning engine is executed in incremental steps using the feedback loop. Instead of directly executing the throttling decision of the reasoning engine as in the case of no feedback, feedback controlled execution moves towards the decision of the reasoning engine in multiple steps. The step-size is dependent on the confidence value associated with the component models (see Section 4.6 for details). In this test, we evaluate the efficiency of the feedback controlled throttling executor by comparing its post throttling position with that of the no feedback loop case. We use the setting with different workloads priority and same quadrant priority ($W_1 = 8, W_2 = 8, W_3 = 2, W_4 = 8$) as an example. The workload status before executing



(a) Equal priority



(b) Effect of workload priority



(c) Effect of quadrant priority

Figure 4.8. Effect of priority values on the output of the constraint solver.

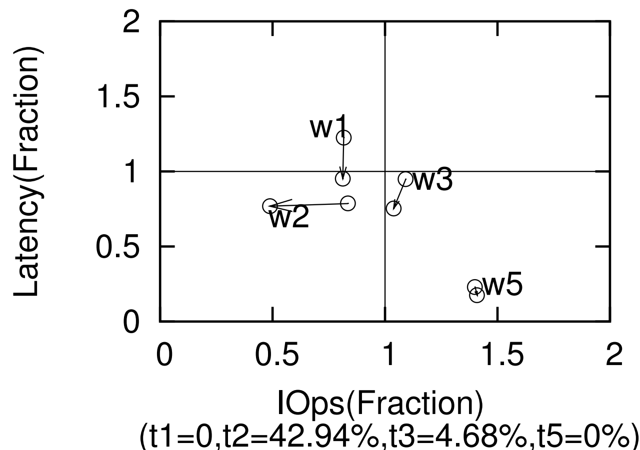


Figure 4.9. Sanity test for the reasoning engine (workload W_5 operating from the controller cache.)

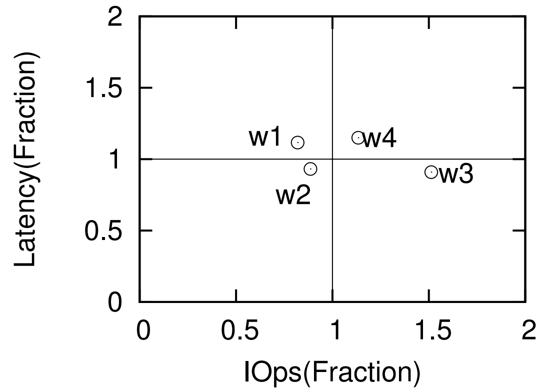
CHAMELEON's throttling decision is plotted in Figure 4.10(a). Figure 4.10(b) and Figure 4.10(c) plot the workload status without and with the feedback loop respectively. The result shows that with feedback, CHAMELEON stops right after W_1 meets its SLO goal. In this case, it results in a much smaller throttling ratio than without the feedback loop.

4.7.4 Replaying Real-World Traces

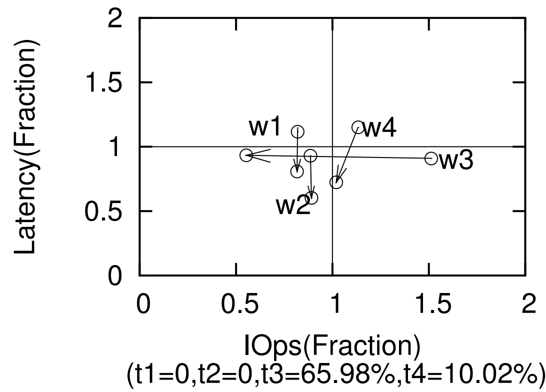
In these experiments, we replay the web-search *SPC* trace [27] and HP's *Cello96* trace [68]. Both are block-level traces with timestamps recorded for each I/O request. We use approximately six hours of SPC and one day of Cello96. To generate a reasonable I/O load for the storage infrastructure, SPC was replayed 40 times faster and Cello96 was replayed ten times faster.

In addition to the traces, we use a phased, synthetic workload, which is assigned with the highest priority. In the case without throttling, with three workloads, SPC, Cello96 and Synthetic, running on the system, one or more of them violate their SLOs. Figure 4.11 shows the throughput and latency values for this uncontrolled case. For all the figures in this subsection, there are four parts ordered vertically: the first plot represents the throughput for the SPC, Cello96, and the synthetic workload. The second, third, and fourth plots represent the latency for each workload, respectively.

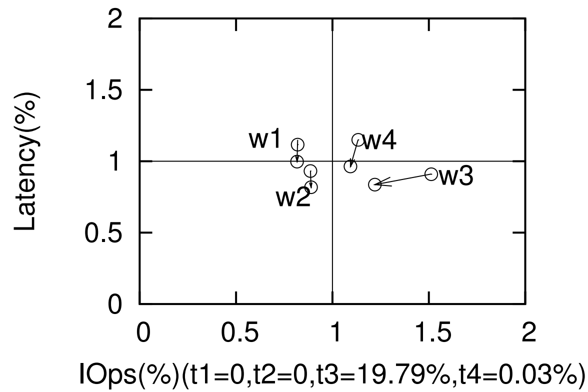
We use these experiments to evaluate the following properties:



(a) Initial workloads status



(b) Output of the reasoning engine



(c) Increment throttling using feedback loop

Figure 4.10. The final output of CHAMELEON using a combination of prediction and feedback-based approach

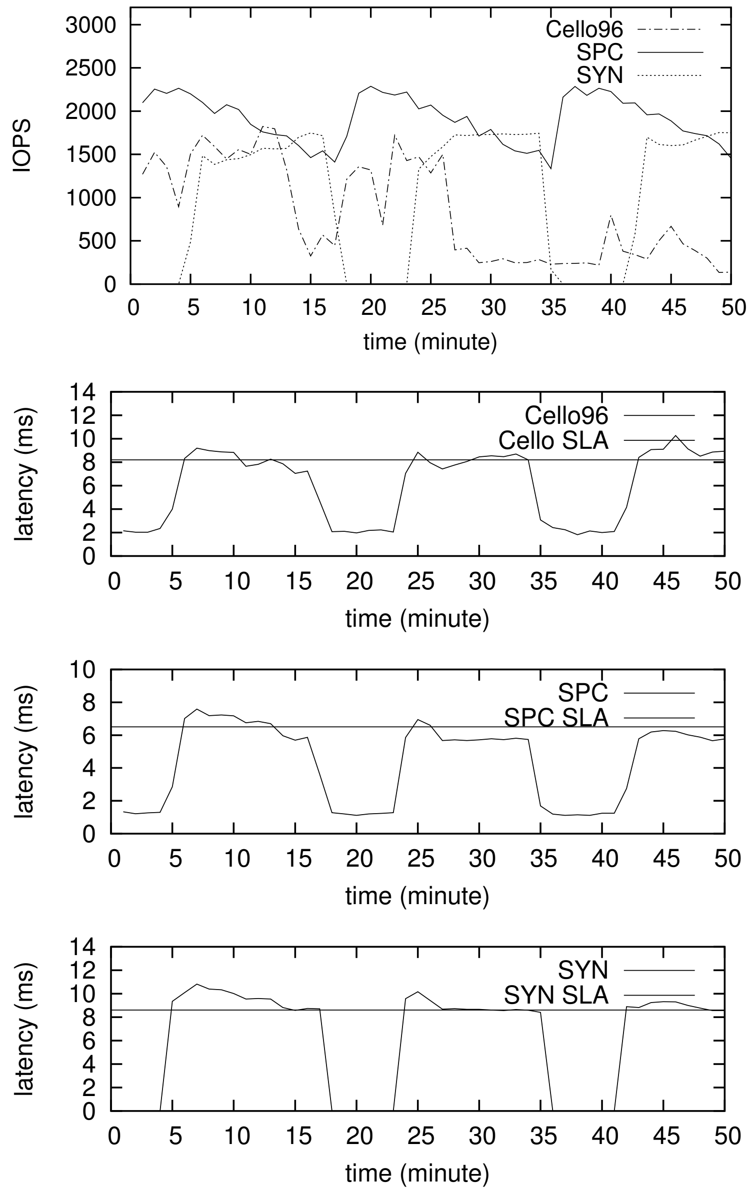


Figure 4.11. Uncontrolled throughput and latency values for real-world and synthetic workload traces.

- The throttling decisions made by CHAMELEON for converging the workloads towards their SLO;
- The reactiveness of the system with throttling and periodic unthrottling of workloads under reduced system load;
- The handling of unpredictable variations in the system that cause errors in the model pre-

dictions and force CHAMELEON to use the sub-optimal but conservative designer-defined policies.

In these experiments, the SLOs for each workload are: 1000 IOPS with 8.2 ms latency for Cello96, 1500 IOPS with 6.5 ms latency for SPC and 1600 IOPS with 8.6 ms latency for the synthetic workload unless otherwise specified.

Case 1: Solving SLO violations

The behavior of the system is shown in Figure 4.12. To explain the working of CHAMELEON, we divide the time-series into phases, shown as dotted vertical line in the figures as follows:

Phase 0 (0 - 5 min): Only the SPC and Cello96 traces are running on the system. The latency values of both these workloads are significantly below the SLO.

Phase 1 (5 - 13 min): The phased synthetic workload is introduced in the system. This causes an SLO violation for the Cello96 and synthetic traces. CHAMELEON triggers the throttling of the SPC and Cello96 workloads. Cello96 is also throttled because it is operating in the exceeded region, which means it is sending more than it should. Therefore, it is throttled even if its SLO latency goal is not met. The system uses a feedback approach to move along the direction of the decision of the constraint solver. In this experiment, the feedback system starts from 30% of the throttling value with a step size of 8% (see Section 4.6 for detailed algorithm). It takes the system six minutes to meet the SLO goal and stop the feedback.

Phase 2 (13 - 20 min): The system stabilizes after the throttling and all workloads can meet their SLOs.

Phase 3 (20 - 25 min): The synthetic workload enters the OFF phase. During this time, the load on the system is reduced, but the throughput of Cello96 and SPC remains the same.

Phase 4 (beyond 25 min): The system is stable, with all the workloads meeting their SLOs. As a side note, around 39 min the throughput of Cello96 decreases because it has a decreasing I/O sending rate.

Figure 4.12 shows the effectiveness of the throttling: all workloads can meet their SLO after

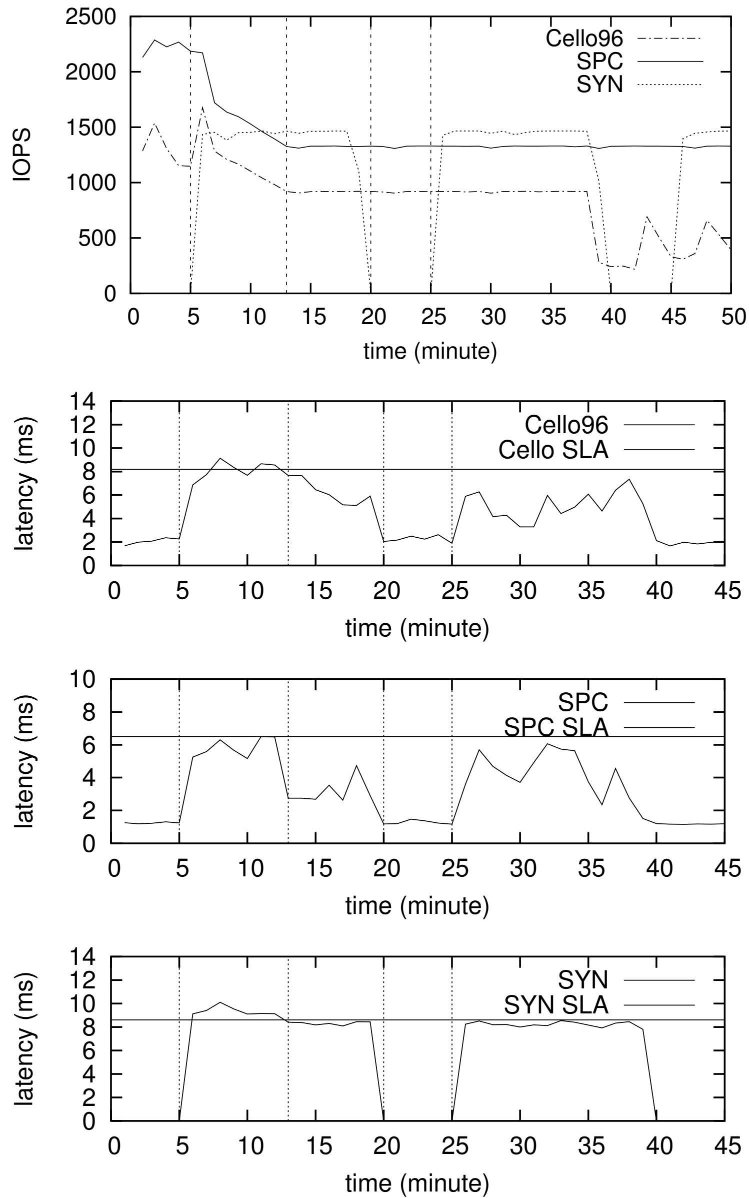


Figure 4.12. Throughput and latency values for real-world workload traces with throttling (without periodic unthrottling.)

throttling. However, because the lack of an unthrottling scheme, throttled workloads have no means to increase their throughput even when tokens are released by other workloads. Therefore, the system is underutilized.

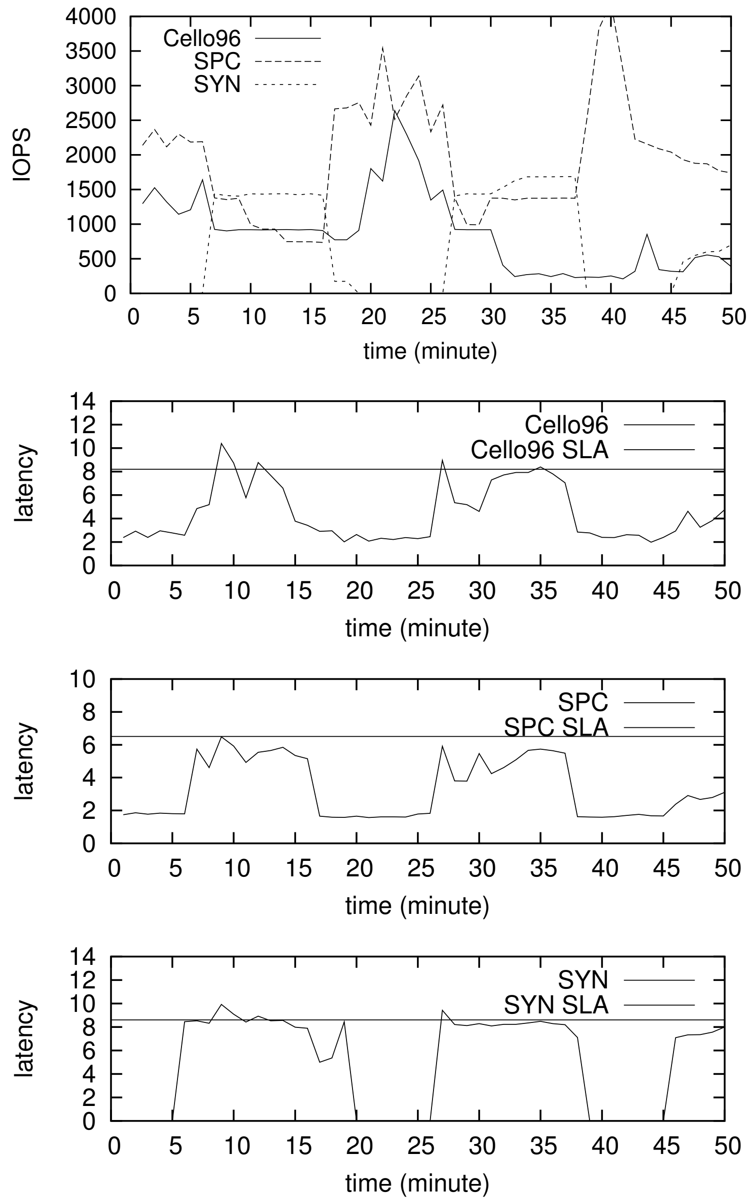


Figure 4.13. Throughput and latency values for real-world workload traces with throttling and periodic unthrottling.

Case 2: With throttling and unthrottling

The previous experiment demonstrates the effectiveness of throttling. Figure 4.13 shows throttling combined with unthrottling of workloads during the periods of reduced system load. Compared to Figure 4.12, the key differences are: (1) SPC and Cello96 increase their request rates when the system load is reduced (17 min to 27 min), improving overall system utilization; and (2) the system has a non-zero settling time when the synthetic workload is turned on (27 min to 29 min). In

summary, unthrottling allows for better system utilization, but requires a non-zero settling time for recovering the resources.

Case 3: Handling system changes at run-time

This experiment demonstrates how CHAMELEON handles system changes at run-time. These changes can be due to unpredictable system variations, such as hardware failures, or un-modeled properties of the system, such as changes in the workload access characteristics that change the workload models. Refining the models to reflect the changes will *not* be instantaneous. In the meantime, CHAMELEON should have the ability to detect decreases in the confidence value or prediction accuracy, and switch to a designer heuristics, or generate a warning for human administrator.

Figure 4.14 shows the reaction of the system when the access characteristics of the SPC and Cello96 workloads are synthetically changed such that the cache hit rate of Cello96 increases significantly and the SPC generates more random access. In reality, a similar scenario arises due to changes in the cache allocation to workload streams sharing the controller.

The SLOs used for this experiment are: 1000 IOPS with 7 ms latency for Cello96, 2000 IOPS with 8.8 ms latency for SPC and 1500 IOPS and 9 ms latency for the synthetic workload.

Phase 0 (at 3 min): The synthetic workload violates its latency SLO. In response, CHAMELEON decides to throttle the Cello96 workload using the original workload model. The output of the reasoning engine has a confidence value of 65%.

Phase 1 (3 min to 13 min): The feedback module continues to throttle for three consecutive steps; since the latency of the synthetic workload does not change, it re-invokes the reasoning engine. The output of the reasoning engine is similar to the previous invocation since the models haven‘t changed. This repeats for consecutive invocations of the reasoning engine. Finally the feedback module switches to use the designer-defined policies.

Phase 2 (13 min to 17 min): CHAMELEON uses the designer policy described in Section 4.5 and throttles all the non-violating workloads uniformly (*uniform pruning*). Both SPC and Cello96 are throttled in small steps, 5% of their SLO IOPS, till the latency SLO of the synthetic workload is satisfied.

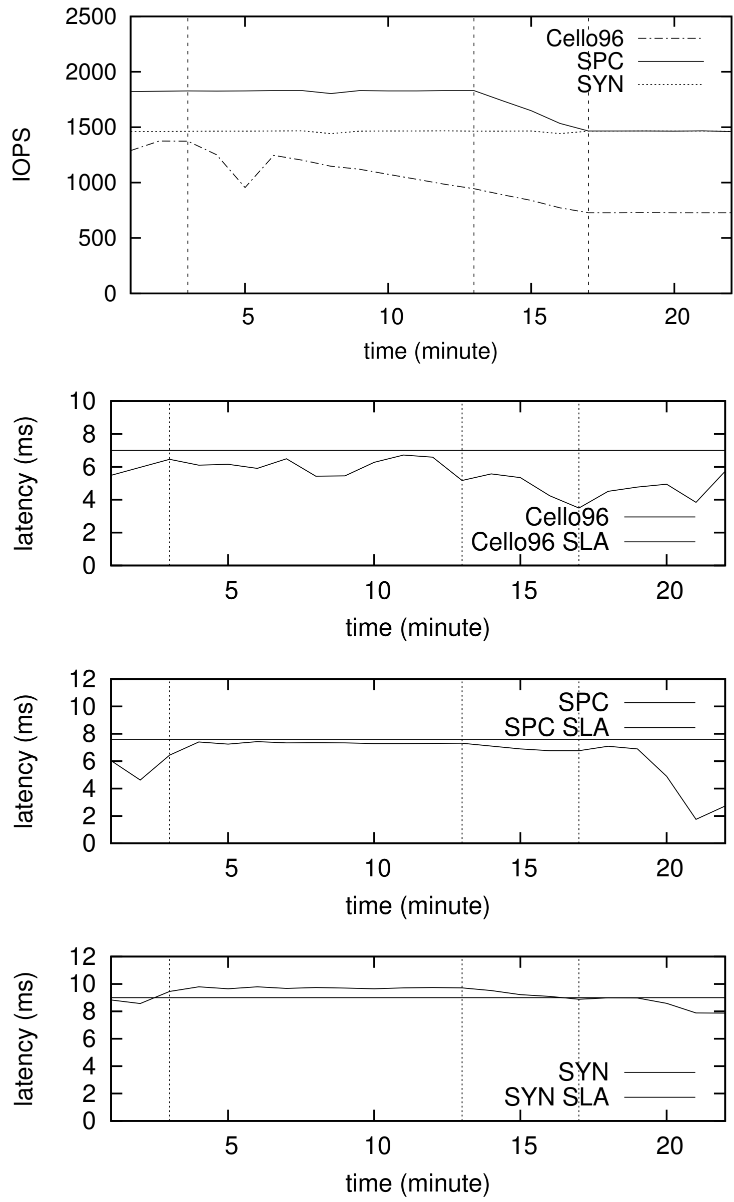


Figure 4.14. Handling a change in the confidence value of the models at run-time.

Phase 3 (beyond 17 min): All workloads are meeting their SLO goals and the system is stabilized.

4.7.5 Decision Overhead of the Reasoning Engine

The current implementation of CHAMELEON uses piece-wise linear programming for constraint solving. The decision overhead of the constraint solver is a function of the number of variables

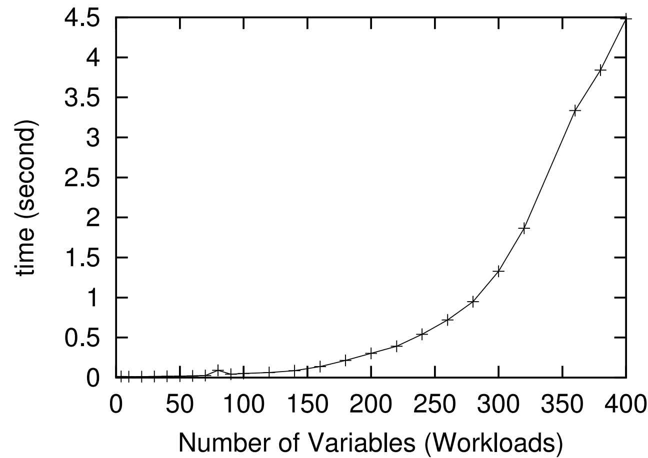


Figure 4.15. Decision overhead of the reasoning engine

involved. Figure 4.15 shows the amount of time CHAMELEON takes to generate the answer. This experiment is run on a P4 2.8 GHZ linux machine with 512MB memory.

In this section, we have presented our experimental evaluation of CHAMELEON using a real system testbed. The results show that CHAMELEON can automatically execute throttling procedures that move the system to its optimal state, as defined by the objective function. In all cases, the right workloads are throttled and the amount of the throttling is the minimum needed to meet the targets. With our current system and for the workload perturbations we imposed, we see reaction times between 3 and 14 minutes. While this is not instantaneous, it is almost certainly quicker than a human could react. Such reaction cause the operator to first notice the problem, then decide what to do and execute it. We also test the unthrottling decision and the results show that CHAMELEON successfully releases constraints on the workloads when the system load is reduced. In the next section, we will discuss how to generalize the design of the reasoning engine to integrate it with the SMART's framework.

4.8 Working with SMART

In the previous sections, we described the framework, reasoning techniques and execution control of CHAMELEON. In the more general SMART framework, CHAMELEON can act as the single throttling tool to determine the optimal throttling parameters and as a throttling actuator. Since

CHAMELEON's design methodology can be applied directly in SMART, in the rest of our discussion, we will focus on how to apply general optimization techniques to improve CHAMELEON's reasoning engine to match the objective of SMART—maximizing the system utility.

As shown in Figure 4.1, CHAMELEON's reasoning engine takes the current system states and system models as input and chooses the optimal token issue rates using the piece-wise linear programming technique. In SMART, the Action Advisor will feed the interested system states to CHAMELEON. The system models can be fed either by the Action Advisor or by input modules directly. These are described in Chapter 3. However, the piece-wise linear programming technique used by the reasoning engine cannot handle the objective function of maximizing the system utility. This is because it can have both linear and non-linear forms. As a result, the optimization algorithm of CHAMELEON's reasoning engine needs to be generalized to accommodate more general objective functions.

Choosing token issue rates to maximize the system utility can be formulated as a global optimization problem, where the problem is to find the global optimum of a given function in large search space. Many approximation algorithms have been proposed to reduce the complexity of global optimization problem [70], such as the hill climbing algorithm, tabu search [35] and genetic algorithms [60]. In our prototype of SMART to be described in Chapter 6, we use the simulated annealing (SA) algorithm, invented independently by S. Kirkpatrick, C. D. Gellatt and M. P. Vecchi in 1983 [50] and by V. Černý in 1985 [16]. The key idea of SA was to combine the hill climbing algorithm, that always goes uphill, with some reasonable degree of random walk to allow downhill moves and avoid local optimum. SA exploits an analogy between the annealing process in metallurgy, where a metal was heated and cooled in a controlled manner to reach to a minimum energy crystalline structure. In brief, each step of SA algorithm replaces the current solution, such as the token issue rates configuration, by a random "nearby" solution, such as allocating additional tokens to a randomly selected workload. If the move improves the objective value, it is always accepted. Otherwise, it is accepted with some probability, which decreases exponentially with the reduction of the objective value, as well as the "temperature". The "temperature" is a configurable parameters and deceases gradually during the process. It can be shown that, for any given finite problem,

if the temperature is decreased slowly enough, SA can find a global optimum with a probability approaching to one [70].

In summary, by replacing the piece-wise linear programming algorithm with a general global optimization formulation, CHAMELEON can work as the single action tool in SMART to find the throttling invocation parameters that can maximize the system utility.

4.9 Conclusions

An ideal solution for resource arbitration in shared storage systems would adapt to changing workloads, client requirements and system conditions. It would also relieve system administrators from the burden of having to specify when to step in and take corrective actions, and what actions to take. Thus administrators can concentrate on specifying the global objectives to maximize the storage utility and having the system take care of the details. No existing solution achieves this today. Prior approaches are either inflexible, or require administrators to supply up-front knowledge that is not available to them.

Our approach for identifying which client workloads should be throttled is based on constraint optimization. Constraints are derived from the running system by monitoring its delivered performance as a function of the demands placed on it during normal operation. The objective function being optimized can be defined by the administrator to reflect organizational goals. Given that the actions prescribed by our reasoning engine are only as good as the quality of the models used to compute them, CHAMELEON will switch to a conservative, decision-making process if sufficient knowledge is not available. We replay traces from production environments in a real storage system, and demonstrate that CHAMELEON makes accurate decisions for the workloads examined. With our current system and scenarios tested, CHAMELEON reacts to and solves the performance problems using the feedback loop in the range of 3-14 minutes.

CHAMELEON can work as the single action tool for throttling in the SMART framework as well as a stand-alone throttling decision tool. In Chapter 5, we will present SMARTMIG, a single action tool for another corrective action, migration. SMARTMIG is responsible of determining the migration invocation parameters, including *what* and *where* to migrate, *how* to migrate and *when* to

invoke the migration operation. It generates the migration plan by applying system models, time series analysis, constraint optimization and risk management analysis. In Chapter 6, we will describe our algorithm for generating an integrated, multi-action schedule that completes the analysis task of SMART.

Chapter 5

SMARTMIG: An Automatic Proactive Data Migration Decision Tool

Growing consolidation of storage systems necessitates resource sharing among multiple competing applications. To maximize the system utility, corrective actions are invoked to change the application-to-resource mapping at run-time. Our work focuses on proposing framework and algorithms to generate the corrective action schedules automatically. We started with describing the framework of SMART, a model-based automatic action advisor in Chapter 3. In Chapter 4, we presented CHAMELEON, an automatic throttling decision tool that chooses the invocation parameters by combining system models and constraint optimization techniques. In this chapter, we will present SMARTMIG, a decision tool for another commonly used corrective action, *migration*. Different from throttling, which corrects the system behavior by reducing the loads to the system, migration improves the system performance by changing the physical location of data to redistribute the load within the system. Since migration involves data movement, it often has a long action execution time and its action effect is not easily reversible. As such, the decision strategy of migration is very different from that of throttling. Thus, it is necessary to discuss the design of the single migration action tool for SMART.

Migration as a corrective action requires deciding *what* data-set to migrate, *where* to migrate, *when* to migrate and *how* fast to migrate. SMARTMIG generates the complete migration plan over

three phases: the optimization phase chooses the optimal data-set and target component, the planning phase determines the migration starting time and the migration speed, and the risk management phase balances the benefit and risk of migration options. It applies system models, time-series analysis and risk management to account for both the temporary migration effect when migration is ongoing, and the permanent impact on the system after migration finishes. We develop a simulator to evaluate the efficiency of SMARTMIG's decision. Our results show that, for 83% of the scenarios tested, SMARTMIG can reduce the utility loss by more than 80% compared to the case of no action invocation. In the rest of this chapter, we will first motivate the need for a migration tool that can account for future system states and the risk of migration options, and can invoke migration at appropriate time in Section 5.1. We then give an overview of the decision making algorithm in Section 5.2, and describe the details of optimization, planning, and risk management phases in Section 5.3. We evaluate SMARTMIG in Section 5.4. Finally, we summarize this chapter in Section 5.5.

5.1 Motivation and Related Work

Automated invocation of data migration is an area of ongoing research. It requires first deciding *what* data-set to migrate, which we refer to as *migration candidates*. Next is *where* to migrate, which we call the *target*. We also must decide *how* to migrate, the migration speed. The final decision is *when* to migrate, the *migration starting time*. Existing efforts to automate migration focus on the new data placement configuration, that is, *what* to move and *where* to place them. Other solutions consider the migration speed. No previous approach considers all four aspects at the same time. Traditional approaches [24, 71, 90] of selecting the migration candidates are based on *data temperature*, defined as the quotient of the load and data size. Migration candidates are ranked in the descending order of temperature. The one with the highest temperature is migrated to the least loaded storage devices. These approaches fail to consider the component's ability to satisfy the workload requirements, and therefore, often lead to suboptimal system utility. Recently, Hippodrome [8] and its follow-on work [7] apply system models to select migration candidates and targets to meet the SLOs goals. They perform an iterative loop to refine the data placement automatically. However, their migration decisions are based on the "current" system state only without considering

future system states. This limits their ability to adapt to system changes proactively. In addition to these efforts in deciding the migration candidates and targets, researchers also propose algorithms to determine the migration speed, such as Aqueduct [54] and QoSMig [28]. As discussed in Chapter 2, these solutions rely on run-time signals to adjust the migration speed. As a result, they lack the ability to predict the possible impact of various migration options, which is critical for planning a migration operation beforehand.

In summary, existing solutions to make automatic the *what*, *where*, *how* decisions have following limitations. First, the decision-making is typically optimized for improving only the current system state rather than considering forecasted trends in system load and workload usage. This makes it difficult to avoid bad states by proactively invoking migration. Second, migration has been traditionally treated as a background task invoked when the system is lightly loaded. However, in utility-based computing, migration can be scheduled as a foreground task to correct resource bottlenecks for high-utility applications. This requires new techniques to determine *when* to schedule migration actions. Third, migration incurs the cost of moving data. Previous solutions fail to account for the penalty cost of incorrectly invoke the migration operation. An ideal migration planning tool should account for both the benefit and risk of a given migration operation in its decision making.

In the rest of this chapter, we present SMARTMIG, a proactive migration scheme for maximizing system utility. SMARTMIG uses a combination of system models, workload demand forecasting and risk management to decide *what*, *where*, *how* and *when* to migrate automatically. SMARTMIG determines the migration parameters in three phases. In the *optimization* phase, it makes the *what* and *where* decisions using constraint optimization with the objective to maximize the system utility for a given optimization time window. In addition, it outputs multiple options for later processing to improve the optimality of the final plan. In the *planning* phase, it determines the detailed *how* and *when* migration plan for the migration candidates and targets. In the *risk management* phase, it analyzes the level of risk involved versus the expected benefit, and selects the plan with best risk and utility efficiency. SMARTMIG requires the same set of input information as SMART (see Section 3.3). Therefore, it can operate as the single action tool for migration actions in the general SMART

framework. Its role is to generate migration options to the Action Advisor for action scheduling (see Chapter 6).

We will first give an overview of the decision making algorithm in Section 5.2. We discuss how to estimate the system utility for any time t in the optimization window and the rational for breaking the decision making procedure into three phases: optimization, planning and risk management. In Section 5.3, we describe the design details for each phase. Our experimental results are shown in Section 5.4. Finally, we summarize this chapter in Section 5.5.

5.2 Overview of SMARTMIG

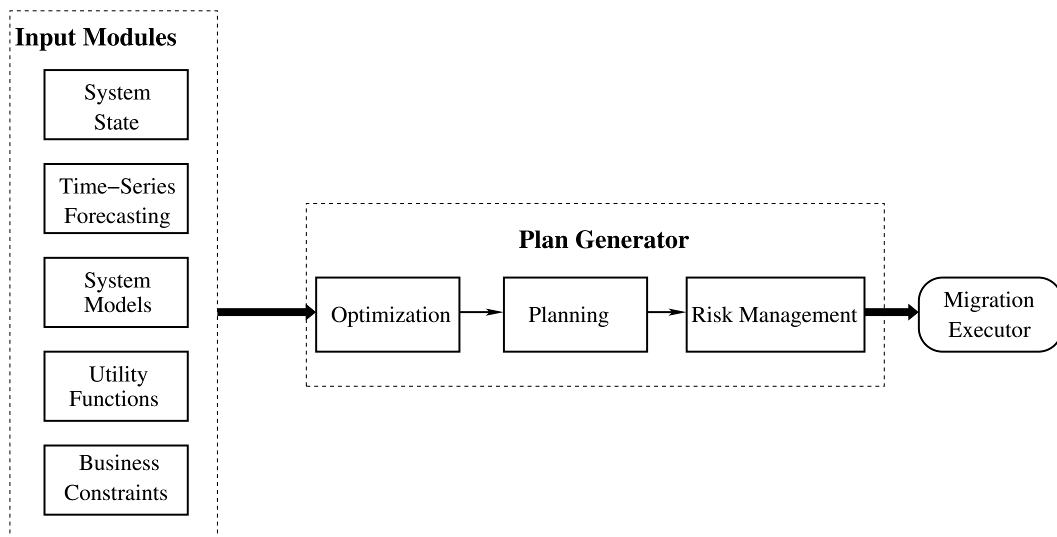


Figure 5.1. Architecture of SMARTMIG

In this section, we will give an overview of SMARTMIG and the mechanisms of generating the migration plan will be described in the next section.

Figure 5.1 gives the architecture of SMARTMIG. As with SMART, SMARTMIG takes system state, forecast workloads demands, system models, utility functions and business constraints as input. Details about the input modules are covered in Chapter 3. In particular, the migration operation has both permanent and transient impact on the system. By permanent, we mean the change of data physical placement. The transient impacts include the extra load on the source and the targets for copying the data. Given a migration plan, its permanent effect on workload performance can be

estimated as follows: first, it uses the workload models (see description in Section 4.3) to derive the load on the components; Then, it uses the component models (see Section 3.3) to estimate the performance using the new data placement configuration. For transient effects, we construct migration action models to capture the extra load introduced by the migration process as a function of the migration speed. We then apply the same model composition techniques discussed in CHAMELEON (see Section 4.4) to estimate the workload performance. With time-series forecasting, for any given time t, the system utility can be predicted by first plugging in the forecast workload demand as input to system models to estimate the received performance. Then the received performance is taken as input to utility functions to calculate the system utility for time t. In general, with system models, time series forecasting and utility functions, we can estimate the system utility at any time in the optimization time window, with or without the migration process.

SMARTMIG's design goal is to find a migration plan to maximize the system utility value for a given optimization window T— equivalent to minimizing the system utility loss, UL_{sys} (Equation 5.1).

$$\begin{aligned} UMax_{sys} &= \sum_{j=1}^{N} UF_j(D_j, SLO_{lat_j}) \\ UL_{sys} &= UMax_{sys} - U_{sys} \end{aligned} \tag{5.1}$$

The invocation of migration requires deciding *what*, *where*, *how* and *when* parameters. These parameters are not independent. The choice of one parameter may affect the optimality of another one. For example, the optimal migration speed depends on the (*what*, *where*) decision and the best migration starting time can only be determined if all other three parameters are known. The ideal optimal migration plan should consider the interaction among the four parameters and examine all possible combinations. However, full scan of the configuration space introduces high complexity— even its sub-problem of choosing the migration candidates and targets is a NP-complete problem [8]. To reduce the complexity, SMARTMIG trades the optimality for complexity. It breaks the decision procedure into three phases as follows:

- *Optimization Phase*: Finds the top-K answers for *what* data to migrate and *where* to place them.

- *Planning Phase*: Finds the best migration starting time and migration speed for each of the top-K $< what, where >$ pairs.

- *Risk Management Phase*: Evaluates the risk associated with each migration plan and selects the one leading to maximum utility and minimum risk.

In each phase, SMARTMIG uses system models, forecast workload demands and utility functions to estimate the decision impact on the system, and to explore the design space for the optimal plan. In the next section, we will describe the design details for each phase.

5.3 Migration Plan Generator

The decision making procedure of SMARTMIG consists of three phases: *optimization*, *planning* and *risk management*. This section covers the details for each phase.

5.3.1 The Optimization Phase: *What* and *Where*

The selection of migration candidates and targets represents the "permanent" effect of the migration. The goal is to find a new data placement configuration that maximizes system utility. This can be formulated as a constraint optimization problem, shown in Table 5.1.

Variable	$s_{ij} = 1$ if data j is placed on component i. Otherwise, $s_{ij} = 0$.
Objective	$Maximize \sum_{i=1,j}^{M} s_{ij} UF_j(Perf_{ij})$
Subject to	$\sum_{i=1}^{M} s_{ij} = 1$ for all j
	Where M is the number of components in the system, j is any workload running on the *underperforming* component, UF_j is the utility function of workload j (see Section 3.3), and $Perf_{ij}$ is the predicted performance of workload j on component i. The $Perf_{ij}$ is affected by all workloads running on component i. Therefore, it is dependent on the data placement configuration.

Table 5.1. Constraint optimization for *what* and *where*

Existing work on data placement often reduces the problem to the Knapsack problem [49], which is known to be NP-complete. In the Knapsack problem, for a given container with fixed

volume, and a set of items with different values and volumes, the goal is to choose a set of items to maximize the total value while satisfying the volume constraints. By mapping the component as container, with the resource as its volume, and the workloads as items, with the resource requirements as the volumes and the utility values as their values, we can directly apply the Knapsack approximation algorithms [20, 41, 49, 65]. In addition, we need to consider following effects. First, we focus on automatic performance optimization using metrics such as throughput and latency. The resource to be optimized is the bandwidth instead of the capacity. In the shared environment, the behavior of how a workload consumes the I/O bandwidth affects the resource requirements of other workloads sharing the same component. For example, the amount of load to the storage component is affected by the caching behavior, which is determined by all workloads competing for the cache. As a result, the "volume of an item", that is the amount of bandwidth required by a workload, is not a static value. It may change with different data placement configuration. Second, the utility value is a function of the received performance, which varies with different placement decisions. As such, the "value of an item" varies with different data placement configurations. Dynamic programming [70] can find the optimal migration candidates and targets, but has a high complexity due to the large search space. To reduce this complexity, SMARTMIG settles for an approximation solution using a greedy approach, shown in the flow chart in Figure 5.2.

In the flow chart, UL is the utility loss, t_k is the time when maximum utility loss happens in the given optimization window T. For a different T value, maximum utility loss may happen at a different time and therefore, the *what* and *where* decisions are optimized for different system settings. In each step of the greedy operation, for every ($workload_j$, $component_i$) pair, the utility gain UG_{ij} of moving workload j to component i is estimated. Here the utility gain UG_{ij} is defined as the utility difference of the new data placement from the old one. The pair leading to the maximum utility gain, shown as ($workload_n$, $component_m$) in the figure, is put into the migration candidate set. The greedy procedure repeats until the utility gain of moving any workload is marginal–less than some *threshold*. The threshold is a configurable parameter and reflects the trade-off between the convergence speed and the quality of the solution. The list of ($workload, component$) pairs in the candidate set are returned as one option.

In addition, to avoid the local optimal, SMARTMIG repeats the greedy procedure K times and

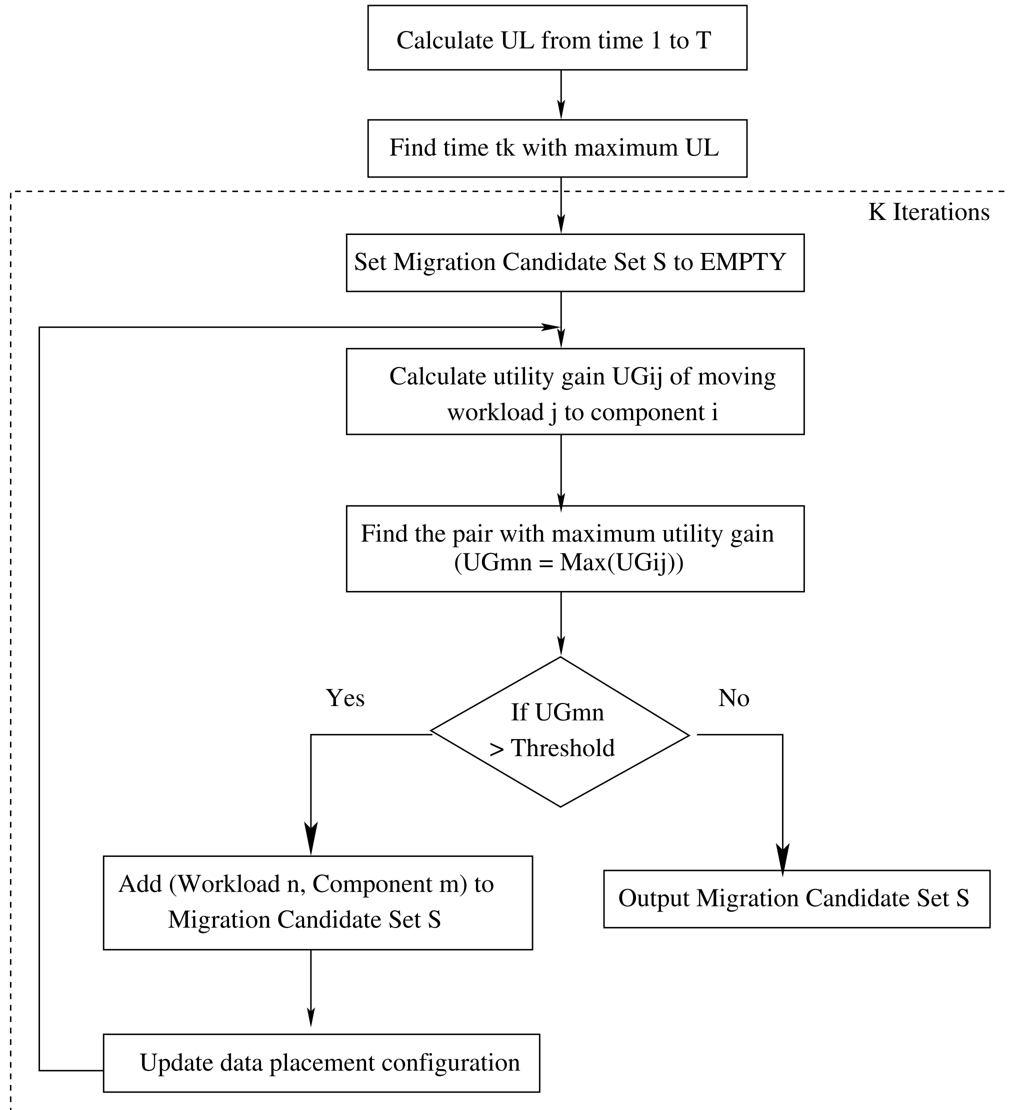


Figure 5.2. Flow chart of optimization phase

in each run, it removes the workload with the minimum utility gain to data size ratio in the candidate set from consideration in the following iterations.

In summary, SMARTMIG finds K migration candidates and targets options using the greedy algorithm. In the next section, we will discuss how to generate plans for each option. In particular, these plans choose the migration speed and the migration starting time.

5.3.2 The Planning Phase: *How* and *When*

The *optimization phase* makes the *what* and *where* decisions and sends the K options to the *planning phase* to determine the migration speed and the migration starting time. In the rest of our discussion, we will first describe how to determine the migration speed at any given time, and then discuss how to select the migration starting time.

How: Choosing Migration Speed

The migration process affects the system utility from two perspectives. On the one hand, it introduces extra utility loss by injecting a migration load to consume the already limited resources, such as I/O bandwidth. On the other hand, it can potentially improve the system utility after it is finished. As a result, it is very difficult to quantify the exact impact on the utility loss for different migration speeds. SMARTMIG chooses to correlate the migration speed with the amount of spare resource in the system, in this case, bandwidth. The intuition is to migrate more data when the system is lightly loaded and less when it is busy. It selects a migration speed proportional to the system's slack resources, particularly, available bandwidth. To enforce the migration speed, SMARTMIG treats the migration process as one workload and uses throttling mechanism to control its sending rate. In addition, SMARTMIG also controls the sending rate for all regular workloads to regulate the bandwidth competition and optimize the system utility. By correlating migration speed with system utilization status, and throttling low priority workloads, SMARTMIG minimizes the utility loss due to the migration procedure. At any time t, the *how* decision procedure is summarized as follows:

1. For both the source and the target components, estimates the component utilization: $Comp_Utilization = \frac{Total_Load}{Maximum_Load}$, where the $Total_Load$ is the predicted load on the component according to the workload demands forecast and the data placement configuration at time t.

2. For both the source and the target components, SMARTMIG uses Equation 5.2 to calculate the migration speed. The minimum value is chosen as the migration speed at time t.

$$MigSpeed = (1 - Comp_Utilization) * p * MAX_SPEED \qquad (5.2)$$

where p is a constant between 0 to 1 and has different values when the system is operating in different spectrum of utilization. The heavier the system is loaded, the smaller the p is. The MAX_SPEED is the maximum migration sending rate allowed by the system. It reflects how aggressively the migration can be performed.

3. SMARTMIG applies the throttling control using the token bucket algorithm to regulate the sending rate of the migration process as well as workloads running on the source or the target storage component. It applies the simulated annealing (SA) algorithm discussed in Section 4.8 to search for the optimal token issue rates leading to the maximum system utility. The basic idea of the SA algorithm is that, each step replaces the current token issue rates by a random "nearby" solution. For example, allocate additional tokens to a randomly selected workload. If the new solution improves the system utility, it is always accepted. Otherwise, it is accepted with some probability. The details are presented in Section 4.8.

Using this algorithm, we can decide the migration speed for time 0 to T, represented as $MigSpeed_i$ for $i = 0, \cdots, T$. Next, we will discuss how to determine the migration starting time t^*

When: Choosing the Migration Starting Time t^*

The best migration starting time is affected by the migration candidates, targets and the migration speed. For example, larger migration size often biases towards starting the migration when the system is lightly loaded; higher utility gain often encourages starting the migration earlier. In this section, we present an algorithm to find the best migration starting time for given *what*, *where* and *how* decisions.

First, let us consider the impact of migration starting time. For a migration operation starting from time t, the optimization time window is partitioned into three regions,

- *Before* region: interval [0, t). In this region, the system still stays with its old configuration and the migration operation has not been performed.
- *Ongoing* region: interval [t, t+m_t), where m_t is the migration execution time if the migration starts from time t. In this region, the migration process performs the copy operations and

the load is gradually directed to the new data location. The system behavior in this region is dominated by the migration speed—the *how* decision. The duration of m_t depends on the total size of data to be moved, $TotalMigSize$, and the migration speed:

$$m_t = min(\text{k such that } \sum_{j=t}^{k} MigSize_j \geq TotalMigSize) \tag{5.3}$$

where $MigSize_j = MigSpeed_j * Len_j$ is the amount of data moved with migration speed $MigSpeed_j$ in interval j with duration Len_j.

- *After* region: interval [t+m_t, T]. In this region, the migration process has finished and the system is operating under the new data placement. The system behavior in this region is determined by the migration candidates and targets.

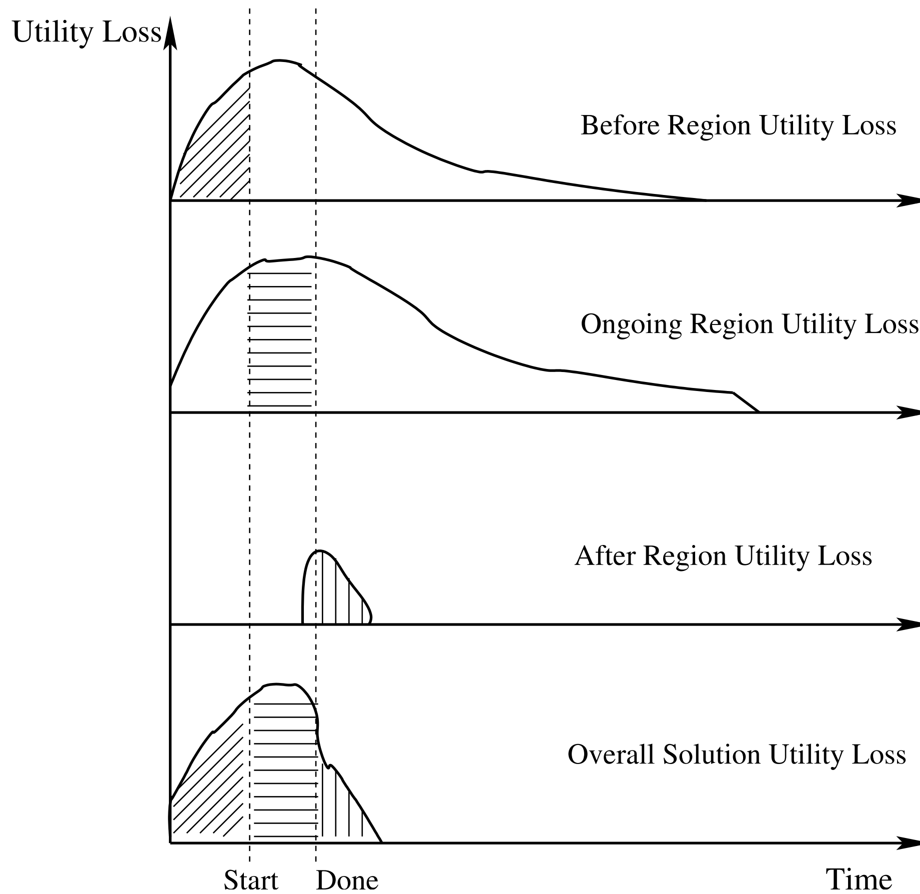


Figure 5.3. Overall solution utility loss

For a given migration starting time t, the accumulative utility loss for the entire optimization

time window is:

$$
\begin{aligned}
UL(T,t) &= UL_{Before} + UL_{Ongoing} + UL_{After} \\
&= \sum_{l=1}^{t-1} UL_{l_{sys}} + \sum_{l=t}^{t+m_t-1} UL^1_{l_{sys}} + \sum_{t+m_t}^{T} UL^2_{l_{sys}}
\end{aligned}
\quad (5.4)
$$

where UL, UL^1 and UL^2 are utility loss for the Before, Ongoing and After region respectively.

Using the *what*, *where* and *how*decisions, the UL, UL^1, UL^2 can be calculated as follows. At each time point l, the Before region utility loss, $UL_{l_{sys}}$, is calculated by first estimating the workloads performance under the old data location settings, then using utility functions to calculate the utility value of the estimated performance (see Section 3.3 for details). The estimation of the Ongoing region utility loss $UL^1_{l_{sys}}$ considers the migration speed $MigSpeed_l$. The After region utility loss $UL^2_{l_{sys}}$ is calculated under the new data placement configuration. The best starting time t^* is the time t leading to minimum $UL(T,t)$. Based on this, SMARTMIG finds the best starting time as follows:

- For every time t in the optimization window, SMARTMIG estimates the migration execution time (m_t) using Equation 5.3.

- For every time t, the algorithm estimates the following utility loss values: (1) if time t is under the old data placement configuration (the *Before Region Utility Loss* curve in Figure 5.3); (2) if the migration operation is underway at time t (the *Ongoing Region Utility Loss* curve); and (3) if the migration has finished by t (the *After Region Utility Loss* curve).

- SMARTMIG scans the optimization window. For each time t, it calculates the overall utility loss using Equation 5.4, with t as the start time. Figure 5.3 illustrates the calculation of the overall utility loss for a given migration starting time, shown as *start*, and the corresponding ending time, *end*. The overall utility loss is the shadow area in the last curve in the figure. It is the sum of the utility loss in the three regions, shown as the shadow areas in the top three curves.

- The t leading to minimum overall utility loss is selected as the migration starting time t^*.

In summary, the *planning phase* determines the migration speed along with the best starting time t^* for each of the K <*what*, *where*> options from the *optimization phase*. The K <*what*, *where*,

how, *when*> options with the corresponding overall system utility loss UL_k are then analyzed in the *risk management phase* to find the low risk, high benefit option.

5.3.3 Risk Management Phase

The goal of SMARTMIG is to assist the administrator in finding good migration options. The *optimization* and *planning phase* returns the top K options leading to the lowest utility loss. A time-series forecast may have errors and migration operations are not cost-free, because they will consume system resources to move data around. Thus, there is a risk associated with each migration option. The goal of the risk management phase is to evaluate the risks of the returned migration options. SMARTMIG can be applied to make suggestions to the administrators, or to control the migration operation without human intervention. In the former, the risk values are returned together with the utility benefit of migration options as references for the administrators. In the latter, they can be used to find the migration option with low risk and high benefit. In the rest of this section, we focus on the latter scenario. We will describe the design of risk management and how to balance the benefit and risk of a migration option.

The *risk* captures the probability that the utility improvement of the action invocation will be lost in the future system-states as a result of volatility of workloads. For example, the demand for W_1 was expected to be $10K$ IOPS after one month, but it turns out to be $5K$. Additionally, the formulation of the risk should take into account the amount of utility loss due to wrong decision. For example, moving data at 11am in a weekday morning *during high system utilization* has a higher risk compared with moving it at 9 pm on a weekend, *during low system utilization*. The utility loss due to a wrong decision is higher in the former case than the latter. Similarly, the impact of the wrong decision is dependent on the amount of data moved.

There are several techniques for measuring the risk. Actions for assigning storage resources among workloads are analogous to portfolio management in which funds are allocated to various company stocks. In economics and finance, the *Value at Risk*, or VaR [52], is commonly used to estimate the probability of portfolio losses based on the statistical analysis of historical price trends and volatilities in trend prediction. In the context of SMARTMIG, VaR represents the bound on

the standard deviation of the workload demands prediction. That is, with a 95% confidence, the standard deviation of the future prediction during the optimization window T will not exceed the VaR value, defined in Equation 5.5.

$$VaR(95\%\, confidence) = 1.65\sigma \times \sqrt{T} \tag{5.5}$$

where σ is the standard deviation of the time-series request-rate predictions.

The risk value $RF(k)$ of migration option k is calculated as follows:

$$RF(k) = (1 + \alpha_k) * VaR \tag{5.6}$$

where α reflects the risk factor of a migration option and is defined as follows:

$$\alpha_k \quad = \quad \frac{bytes_moved_k}{total_bytes_on_source} * Sys_Utilization_k \tag{5.7}$$

where $Sys_Utilization$ is the system utilization when the action is invoked and is estimated similarly as the $comp_Utilization$. Intuitively, a higher system utilization or larger migration data set will lead to a larger α_{M_k} and therefore, a larger $RF(M_k)$ value. This reflects the fact that the migration operation has a higher risk and is less preferred.

For each of the K options output by the *planning phase*, SMARTMIG calculates its risk value $RF(k)$ and scales its overall utility loss UL_k as follows:

$$UL_k^* = (1 + RF(k)) \times UL_k \tag{5.8}$$

SMARTMIG uses the scaling operation to balance the benefit (UL_k) and the risk (RF_k) of a migration option and selects the one with the minimum UL_k^*. Lastly, because migration always involves the cost of moving data, SMARTMIG compares the utility loss of the selected option with a pre-determined threshold and drops this option if its benefit cannot justify the risk. That is, its utility loss is greater than the threshold. Dropping the selected option indicates that migration will not be invoked.

In summary, SMARTMIG selects the migration plan through optimization, planning and risk management. The *optimization* phase returns K migration candidates and targets options. The

planning phase generates the detailed migration plans in terms of migration speed and migration starting time for them. The risk management phase performs risk analysis and sends the low risk, high utility migration option for execution. In the next section, we present our evaluation results in a simulator. Our focus is on understanding the working of SMARTMIG's three phases, SMARTMIG's efficiency in terms of the reduction in utility loss, and its computation time and sensitivity to system model errors.

5.4 Evaluation

The goal of our experiments is to evaluate SMARTMIG's ability to generate feasible and efficient migration plans. The experimental evaluation is divided into three parts. First, we test its ability to make decisions for different combinations of input parameter values using simulator (i.e., a sanity check); Second, we evaluate the run-time complexity by stressing SMARTMIG with different numbers of workloads and components (i.e., an efficiency test). Third, we evaluate the sensitivity of the migration plans to model-errors (i.e., a sensitivity test). In the rest of this section, we will first give details of the simulator, then present the evaluation results for sanity check, efficiency test and the sensitivity test.

5.4.1 Experiments Configuration

We implemented SMARTMIG using a simulator. The simulator takes as input the original system state, consisting of the initial data placement configuration, system models, utility functions and SLOs. It outputs the migration option with low risk and high benefit. The execution of workload demands and the selected migration option is simulated using our system models. We generate the testing scenarios using permutations of the following configuration parameters:

- *Initial data placement*: We intentionally create an un-balanced system. 60% of the workloads will go to one component and the remaining workloads are distributed to the rest of the components randomly. The rationale is to make the migration operation necessary.

- *Workload features:* The request rate and footprint size of each workload, as defined in Section 3.4, are generated using a Gaussian mixture distribution [56], whose density function is the sum of multiple parameterized functions. In particular, with 60% of probability, the request rate of a workload follows a normal distribution with a lower mean, 100 IOPS. For the remaining 40% of workloads, the request rates follow a normal distribution with a mean of 600 IOPS. The reason for using the Gaussian mixture distribution is to mimic the real-world system behavior: a small number of applications contribute to the majority of the system load and access the majority of data.

- *Workload trending*: To mimic workload changes in real systems, we changed the request rates of the workloads over time. In our experiments, 30% of workloads were increased, another 30% were decreased and the other 40% were flat. In particular, the increasing step size is generated using a random distribution with a mean of 1/10 of the original load and the decreasing step size is randomly distributed with a mean of 1/20 of the original load.

- *Utility functions*: The utility function of meeting the SLO requirement for each workload is assumed to be a linear function and the coefficients are random numbers uniformly distributed between 5 and 50. The utility function of violating the SLO requirement is assumed to be zero for all workloads. The SLO goals are also randomly generated with considerations of the workload demands and performance.

- *Others*: The optimization window is 14 days, the standard devision of the time-series forecast is 10% and the time unit is on a per hour basis unless otherwise specified.

We vary the number of workloads in the system from 10 to 100 and generate ten scenarios automatically for each of them. Out of the 100 scenarios, 14 are dropped because they don't have any utility loss. The remaining 86 scenarios experience a utility loss to various degrees, ranging from 0.7% to 55% of the maximum system utility. We do not generate scenarios with a higher utility loss because migration may not be the correct solution in that case. An example of such a case is when new hardware should be requested to correct performance problems. Figure 5.4 plots the Cumulative Distribution Function (CDF) of the percentage of utility loss (Equation 5.9) for all

86 scenarios.

$$utility_loss_percentage = \frac{Max_Utility - No_Action_Utility}{Max_Utility} \tag{5.9}$$

As shown in the figure, more than 55% of the scenarios experienced more than 10% utility loss,

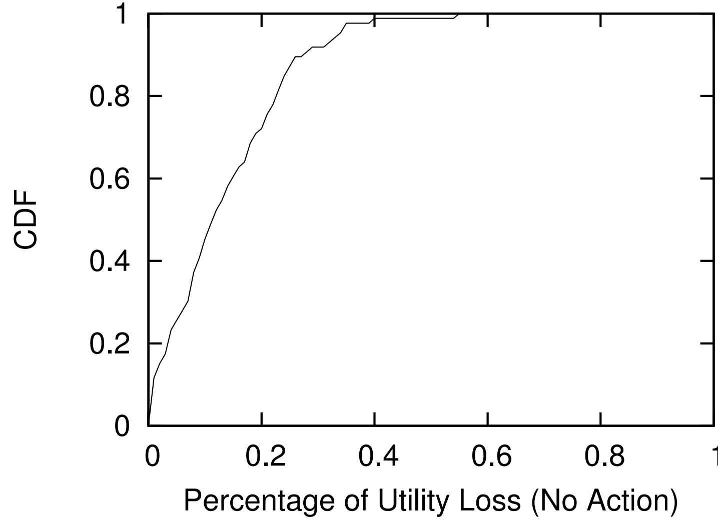


Figure 5.4. CDF of the percentage of utility loss with no action invocation for all 86 scenarios

and around 30% had a utility loss higher than 20%.

The percentage of utility loss captures the degree of the utility loss that the storage system experiences due to resource limitation and competition. Ideally, it should be zero when the system is configured appropriately. A good migration algorithm should be able to find the migration option leading to a zero utility loss. In our later sections (Section 5.4.3 and 5.4.4), we evaluate the quality of SMARTMIG's decisions based on the definition of the percentage of utility loss.

In the next section, we will present our sanity check results that show the working of the three phases of SMARTMIG.

5.4.2 Sanity Check

Working of Three Phases

In this experiment, our goal is to examine the output of the three phases. We randomly pick one scenario from the 86 cases. The scenario has twenty workloads distributed on four components. The

initial data placement caused a 7.8% utility loss. SMARTMIG is called for a 14 days optimization window. The top five solutions returned are summarized in Table 5.2.

#	Migration Candidates and Targets	Size (GB)	Utility Loss	Start Time (hour)	Scaled Utility Loss
1	(1: 0→2) (5: 0→2)	19	10408	5	11629
2	(3: 0→3) (4: 0→1) (5: 0→2)	152	22552	4	43715
3	(3: 0→3) (5: 0→2) (13: 0→3)	28	10408	1	12208
4	(5: 0→2) (13:0→3) (14: 0→2)	118	22552	5	38981
5	(5: 0→2) (13:0→3) (17: 0→2)	20	10408	4	11694

Table 5.2. Solutions returned by SMARTMIG with a 14 day optimization window

The table shows that the *optimization phase* finds five sets of (*what*, *where*) decisions with different data size and utility loss. For each of them, the *planning phase* selects different starting times. The table cannot show the migration speed because this changes with time. In the *risk management* phase, the utility losses for these migration options are scaled by different degrees to balance the risk and the benefit. Finally, Option 1 is selected because it leads to minimum scaled utility loss—0.04%.

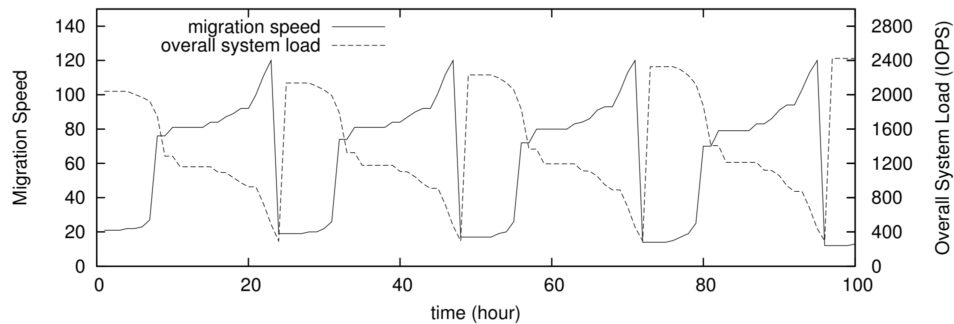


Figure 5.5. Migration speed vs. component load

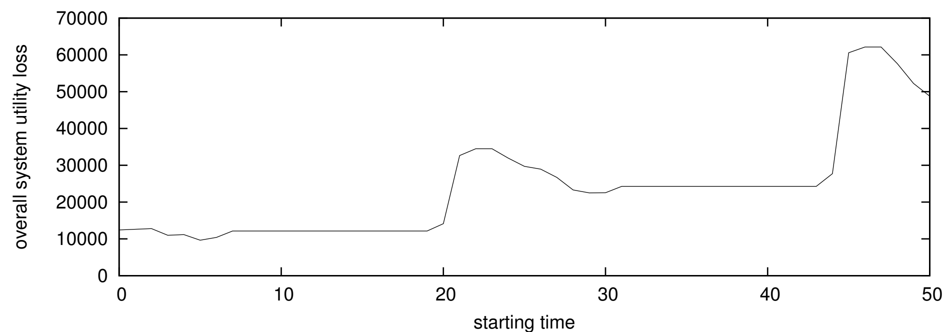


Figure 5.6. Overall utility loss for various migration start times

Figure 5.5 shows the details of the selected migration speed. The X axis in the figure is the time index, the left Y axis is the migration speed and the right Y axis is the overall system load. The figure shows that the migration speed changes according to the load in the component. In addition, the starting time selection is shown in Figure 5.6. We only plot the first fifty hours for better visibility. The X axis is the starting time and the Y axis shows the corresponding overall system utility loss if the migration starts at time x. As shown in the figure, the minimum overall system utility is achieved at Time 5, when the system is lightly-loaded for the first time.

Impact of the Optimization Window T

SMARTMIG is designed to find the best migration option for a given optimization window T specified by the administrators. For different values of T, SMARTMIG returns different migration parameters that leads to better system utility for it. Using the same settings as in the previous test, we change the optimization window from 14 days to seven days. SMARTMIG returns different *what* and *where* answers. The corresponding five options are shown in Table 5.3.

#	Migration Candidates and Targets	Size (GB)	Utility Loss	Start Time (hour)	Scaled Utility Loss
1	(1: 0→2)	14	10354	5	11308
2	(6: 0→1)	12	81677	0	87850
3	(3: 0→2) (5: 0→2)	17	10408	6	11501
4	(4: 0→2) (5:0→2)	140	37461	5	42282
5	(5: 0→2) (13:0→3)	16	10354	5	11436

Table 5.3. Solutions returned by SMARTMIG with a 7 day optimization window

Based on the scaled utility loss, we select Option 1 and migrate Workload 1 from Component 0 to Component 2. This is a different decision than the previous test.

Impact of Utility Configuration

SMARTMIG aims to optimize the system utility. The utility function affects the solution SMARTMIG will return. In this test, we use the same settings as in Section 5.4.2 and randomly change the utility functions of three workloads (7, 13, 15) to lower values. The results returned by

SMARTMIG are given in Table 5.4. Option 4 is selected, which is different from the one selected in Section 5.4.2.

#	Migration Candidates and Targets	Size (GB)	Utility Loss	Start Time (hour)	Scaled Utility Loss
1	(1: 0→2) (3: 0→2) (17: 0→2)	30	2488	4	2949
2	(3: 0→2) (17: 0→2)	16	21569	6	23700
3	(4: 0→2) (5: 0→3) (17: 0→2)	144	8371	30	15813
4	(5: 0→2) (13: 0→3) (17: 0→2)	20	2488	5	2796
5	(5: 0→2) (17:0→2)	9	15347	6	16200

Table 5.4. Five solutions returned by SMARTMIG with different utility configuration

In summary, these sanity check experiments demonstrate SMARTMIG's ability to optimize the migration decisions for different input parameters and system configurations.

5.4.3 Efficiency Tests

In this section, we will first evaluate SMARTMIG's efficiency in terms of improving the system utility, thus eliminating the system utility loss. We then measure the decision overhead of SMARTMIG.

Percentage of the Utility Loss Elimination

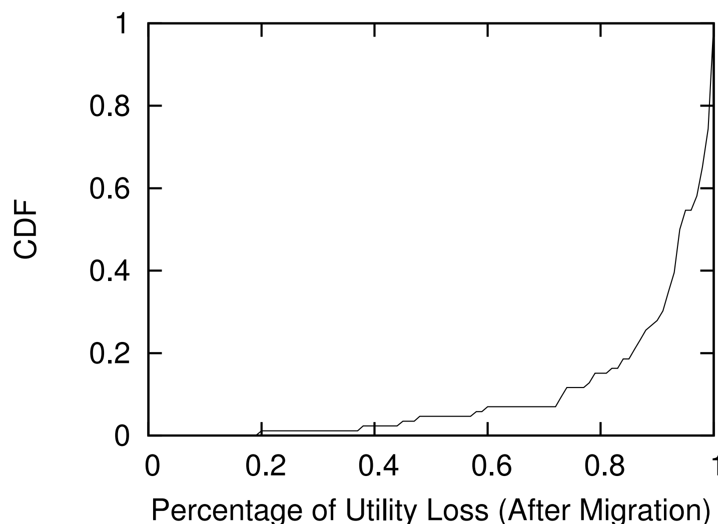


Figure 5.7. CDF of the percentage of the saved utility loss

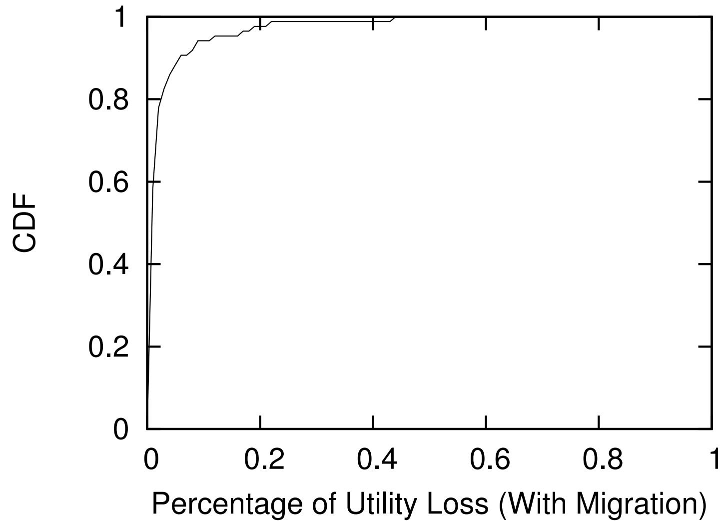


Figure 5.8. CDF of the percentage of the utility loss with SMARTMIG

For each of the 86 scenarios, SMARTMIG selects the migration option with the minimum scaled utility loss. To understand how much of the system utility loss is saved by invoking the migration operation chosen by SMARTMIG, we measure the percentage of the saved utility loss (Equation 5.10) and plot the CDF curve (Figure 5.7).

$$percentage_of_saved_utility_loss = \frac{No_Action_Utility_Loss - Utility_Loss_With_Migration}{No_Action_Utility_Loss} \quad (5.10)$$

The figure shows that for more than 83% of the scenarios tested, SMARTMIG successfully eliminates 80% of the utility loss, and for another 12% of cases, it reduces the utility loss by 60%. The percentage of utility loss (Equation 5.9) of invoking SMARTMIG's decision is plotted in Figure 5.8. We can see that for more than 90% of the cases, the system exhibits less than 0.7% utiltiy loss, which is a significant improvement compared to the case of no migration, where only about 30% of the cases exhibits less than 0.7% utility loss (see Figure 5.4).

Computational Overhead of SMARTMIG

Figure 6.27 plots the computational overhead in computing time of SMARTMIG. We run the simulator on a Linux machine with a 2.66GHZ Pentium 4 CPU and 512MB of memory. We vary the number of workloads in the system and record the time SMARTMIG takes to generate a migration plan. Ten cases are generated for each workload number setting and the average time is plotted

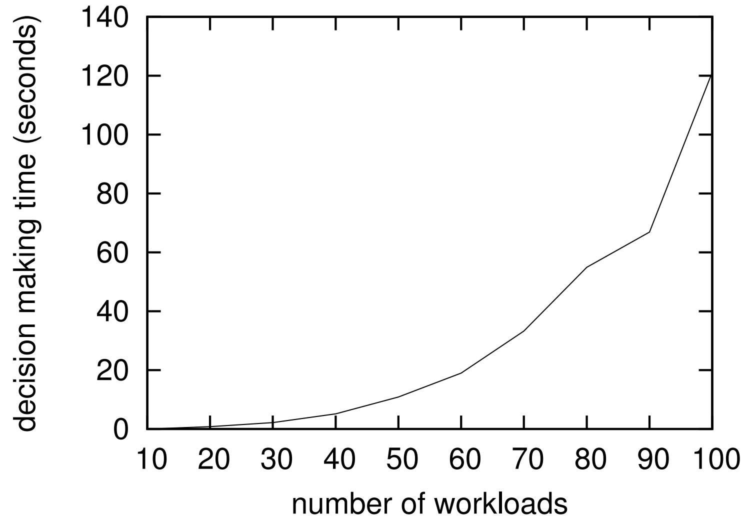


Figure 5.9. Computation overhead of SMARTMIG

in the figure. The curve grows exponentially with the number of workloads. The majority of the overhead is due to the decision on the migration speed, where SMARTMIG greedily searches for the optimal bandwidth allocation for each workload. This exponential growth can be reduced by relaxing the sending rate control of the workloads, but with the cost of a higher utility loss due to the un-regulated bandwidth competition.

5.4.4 Sensitivity Test of Performance Model Errors

Our sanity check experiments are based on the assumption that perfect component models are available. However, in reality, this is never true. To measure the sensitivity of the migration decisions to the component model errors, we perform the following experiment:

- First, we generate the synthetic component models to simulate the models constructed in the real systems. For any given system state, These models are used to calculate the "predicted performance" and the "predicted utility loss", which are used by SMARTMIG to make migration decisions.
- The real system performance is simulated by introducing a random error to the "predicted performance". The random error is generated using a normal distribution. For example, with

a 10 ms "predicted latency" and 0.2 as the random error, the "real latency" is simulated using 12 ms (10*(1+0.2)). The "real utility loss" is then calculated based on it.

This design allows us to control the model error rates and measure their impact on SMARTMIG's decision. Figure 5.10 plots the accuracy of the predicted utility loss, defined as $\frac{Predict_Utility_Loss}{Real_Utility_Loss}$.

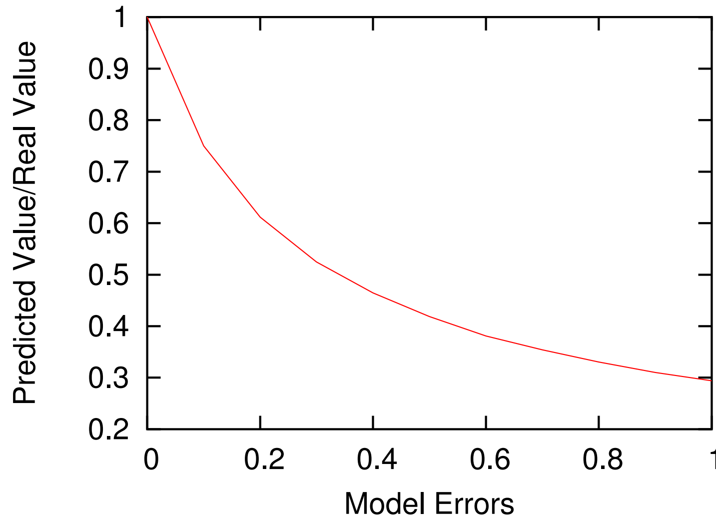


Figure 5.10. Impact of model errors on the accuracy of the predicted utility loss

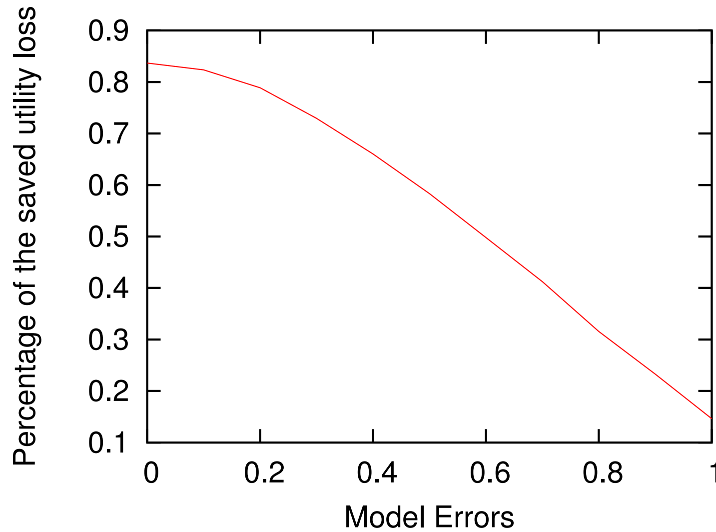


Figure 5.11. Impact of model errors on the percentage of the saved utility loss

As shown in the figure, the accuracy drops very quickly with the growing error, indicating the predicted value matches less with the real value. We also measure the impact of the model errors

on the percentage of the saved utility loss (Equation 5.10) to understand the errors' impact on the effectiveness of SMARTMIG (Figure 5.11). From the figure, we can see the with an error rate of 30%, SMARTMIG can eliminate 73% of the utility loss on average. When the error rate goes up to 60%, SMARTMIG can still reduce the utility loss by 48%.

These experiments show that an accurate model can improve the effectiveness of SMARTMIG's decision. But even with a high model error, SMARTMIG can still provide useful suggestions to the system.

In summary, our experimental results show that SMARTMIG can generate effective migration plans that account for various configuration settings, such as utility functions and optimization time windows. For our test scenarios, SMARTMIG eliminates 80% of the utility loss compared to the no action option. Our computational overhead analysis shows that SMARTMIG can make decision in the order of minutes. Our sensitivity tests show that SMARTMIG's efficiency are improved with better models. However, SMARTMIG can provide useful suggestions even when the model error is high.

5.5 Summary

In this chapter, we described SMARTMIG, a proactive data migration decision scheme. It acts as the single action tool for migration in SMART's framework. SMARTMIG finds the optimal migration plan using three phases: optimization, planning and risk management. It combines system models, time-series analysis, risk management and constraint optimization to generate migration plans. Its migration plan can choose the migration starting time, which enables foreground migration operation. In addition, it considers both the current and forecast system states and selects the migration plan leading to maximum system utility adjusted based on the risk analysis. Experimental results show that SMARTMIG's decision can adapt to the changes of configuration parameters. For the tested scenarios, SMARTMIG eliminates 80% of the utility loss compared to the no action option. Our sensitivity tests show its high tolerance to model errors. In the next chapter, we will present the design of SMART's core piece, the Action Advisor. In particular, we will discuss how to generate a

multi-action schedule to improve the system utility by combining single action options determined by single action tools such as CHAMELEON and SMARTMIG.

Chapter 6

SMART: An Integrated Multi-Action Advisor for Storage Systems

Storage system management has become more complicated due to the trend of accelerating storage system consolidation, where many applications share storage devices through the virtualization layer. To adapt to system changes and meet the SLOs of applications, administrators continuously *observe* the storage system, *analyze* the problem, and *invoke* corrective actions to improve its utility. The human controlled *observe-analyze-act* loop is slow and expensive. The trend is to automate the OAA loop to reduce the cost of ownership and to more rapidly adapt to system changes. The focus of this thesis is to automate the analyze part of the loop—generate the corrective action schedules automatically. We have described the framework of SMART, a model-based automatic action advisor in Chapter 3. In Chapter 4 and Chapter 5, we presented model-based single action tools, CHAMELEON for throttling and SMARTMIG for migration to determine the action invocation parameters automatically. However, in existing solutions, such tools are not properly integrated into a single management framework. As a result, administrators still need to decide which action to take and when to invoke it. It is challenging to find the best decision among multiple actions. It requires considerations of system behaviors, action impacts, future system states and business impact. In addition, there will be situations where a combination of actions would provide better cost benefit tradeoffs, which makes the decision more difficult. Our SMART framework aims to assist adminis-

trators by generating effective action schedules to specify *when* to do *what* actions. In this chapter, we will present the decision algorithm of the Action Advisor, the intelligence of SMART. It considers multiple types of corrective actions in an integrated manner and generates a combined corrective action schedule. In addition, our algorithm can handle the expected predictable system changes as well as unexpected events, where predicted and observed states vary. Specifically, Section 6.3.1 presents the design of a recursive greedy algorithm to plan for the expected system changes, while Section 6.3.3 develops a defensive strategy to handle the system exceptions. Our experimental results show that our decision algorithms can account for various system configurations and generate flexible action schedules to improve the system utility. In more than 90% of the tested scenarios, our algorithm reduces the utility loss by more than 94%, with a decision time on the order of minutes. And in the other 10%, the utility loss is reduced by 70%. In the rest of this chapter, we will first motivate the need for a multi-action schedule in Section 6.1. Then we will briefly revisit the framework of SMART in Section 6.2 (see Chapter 3 for more details). After that, we will describe how to generate action schedules for the *normal* mode using our recursive greedy algorithm in Section 6.3.1, and our on-line decision algorithm for the *unexpected* mode in Section 6.3.3. In Section 6.4, we will present our experimental results for the sanity check, the efficiency test, the computational overhead and the sensitivity test with respect to the input information errors.

6.1 Motivation

With an increase in the number of applications, the amount of managed storage, and the number of policies, storage system management becomes more challenging. To adapt to system changes and improve system utility, administrators invoke corrective actions at run-time to change the resource-to-application mapping. In recent efforts towards self-managed storage systems to reduce the total cost of ownership, many research prototypes [8, 28, 54, 55] and products [25, 26] automatically generate the invocation parameters for single action. However, these tools suffer from the following two key drawbacks:

- **Single Action Based:** Existing efforts have focused on using a single type of corrective action, such as workload throttling, data migration, or adding more resources, to correct SLO

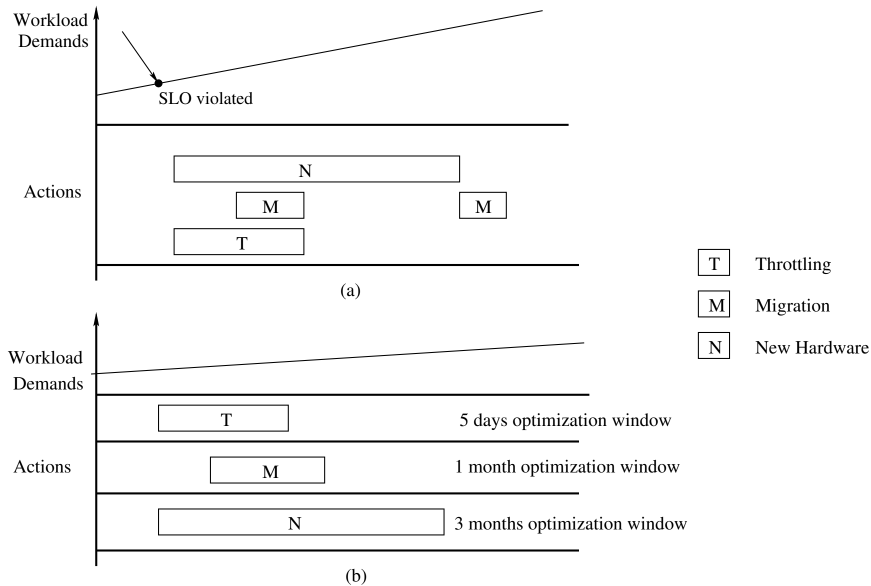


Figure 6.1. Problem with single action

violations. These tools lack the ability to consider alternative actions to provide a better solutions. When the SLO for a workload is violated, many situations require a combination of actions to achieve the maximum system utility improvement. For example, in Figure 6.1 (a), when the SLO is violated, the best action in the long run is to provision a new disk array. However, it takes time to order and install the new hardware. In this case, instead of waiting for the new hardware and losing utility continuously, a better solution is to start throttling immediately and migrate some load to a less contended storage device before the new hardware arrives. In addition, if necessary, the workload can be migrated to the new storage device after it arrives. The decision tool must be able to advise the administrators to take a single action or a combination of actions at the right time.

Existing storage management systems from various vendors provide good domain specific decision tools such as network planning, storage controller planning and migration planning. However, these tools work independently without considering other tool. For example, capacity planning tools, which are typically considered as long term actions, are not well integrated with the throttling planning. The latter is an instantaneous action. The lack of the proper integration between the decision tools puts the responsibility of their integration on a system administrator. As the system scales, this becomes more difficult, often resulting in solutions

that are either grossly over-provisioned with excess resource, or under-provisioned causing frequent SLO violations. As a result, we need a tool that can assist the administrators in integrating multiple actions to improve the system utility.

- **Single time-window based**: Another drawback of the existing tools is that they take a "one-size fits all" approach to solve the system's problem. For example, the throttling tool will always invoke the throttling no matter if the system wants to be optimized for one hour or for one year. However, the solution that is the most cost effective for one week might be different from the best solution for one year. For example, as shown in Figure 6.1 (b), the optimal solutions in terms of the system utility vary for different time windows: If the optimization time window is five days, throttling is the optimal solution. If it is one month, data migration is the most cost effective solution. When the window is opened to three months, adding new hardware is the best way to cost effectively improve system utility. Overall, time is an important aspect and is often overlooked in the decision-making process by the administrators and existing tools. Existing automatic decision tools do not allow administrators to consider plans for different optimization time windows. As a result, the administrators may not choose the best action at the right time.

In this chapter, we present an action schedule algorithm for the Action Advisor in SMART's framework. The key contributions of our algorithm are:

1. **Integrated multi-action planning:** It combines seemingly disparate storage management actions such as workload throttling, data migration, and resource provisioning into an integrated framework.

2. **Multi-granularity temporal planning:** It allows for the specification of optimization time windows. For example, one could specify that they want the most cost effective solution for one day or one year.

3. **Action selection for unexpected workload variations:** It can determine whether the surge in the I/O requests is due to an unknown workload spike or a known trend and select corrective actions accordingly.

The rest of this chapter is organized as follows, Section 6.2 briefly revisits the framework of SMART and describes how CHAMELEON and SMARTMIG can be plugged in to SMART. Section 6.3 describes the action scheduling algorithm. Section 6.4 presents the experimental results, and Section 6.5 summarizes this chapter.

6.2 Framework Revisited

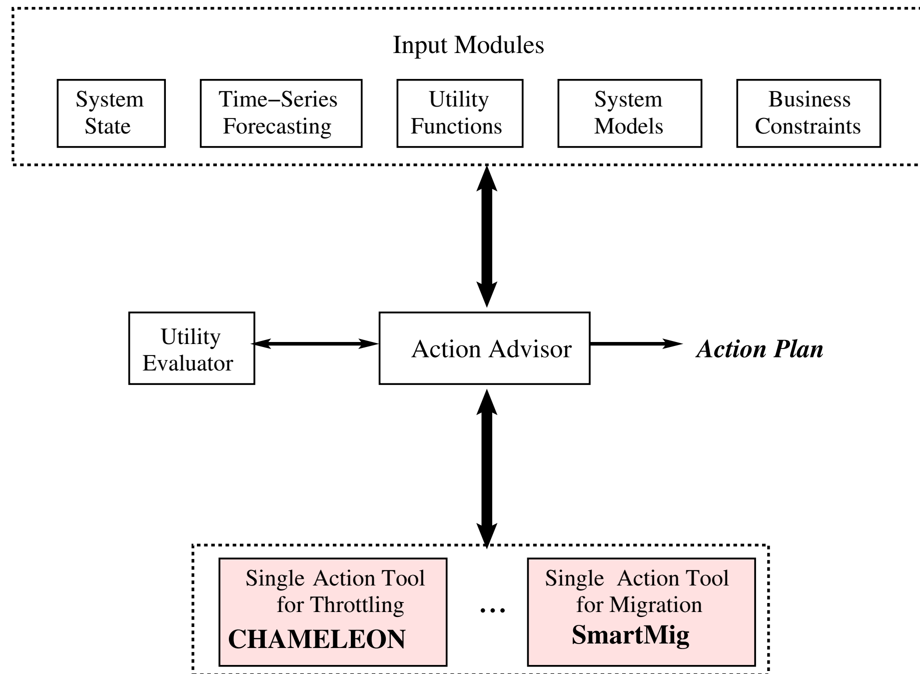


Figure 6.2. Architecture of SMART: a model-based action advisor

In Chapter 3, we described the framework of SMART. The key components of SMART include *input modules*, the *utility evaluator*, *single action tools* and the *action advisor*, shown in Figure 6.2:

- The **input modules** consist of the system state $\mathcal{S}$ collected using monitoring sensors, time-series prediction of workload demands, utility functions, system models for storage devices, and business-level constraints such as budget constraints and the optimization window.

- The **utility evaluator** estimates the system utility (Equation 3.2), the maximum system utility and the utility loss (Equation 3.3) for given workloads' performance. For the current system state, the workloads' performance can be measured directly. For future system states, they

can be interpolated by composing the action, workload and component models as described in Section 5.2.

- The **single action tools** find the optimal corrective action options in a given system state, and return the optimal invocation parameters to the Action Advisor. For example, CHAMELEON is a throttling tool. It takes a system state ($\mathcal{S} =< \mathcal{C}, \mathcal{W}, \mathcal{M} >$) as input, applies the general optimization technique to determine the token issue rates t_i for each workload i, and sends them to the Action Advisor. Similarly, SMARTMIG is a migration tool, which takes a set of system states, including both the current $\mathcal{S}_0$ and the future states $\mathcal{S}_1, \mathcal{S}_2, \cdots$, as input. It then performs the optimization, planning and risk management analysis, and outputs the <*what*, *where*, *how*, *when*> decision (Section 5.3). For both CHAMELEON and SMARTMIG, the required input information, such as system models, time-series forecast and utility functions, can be obtained from the input modules.

- The **Action Advisor** is the core of the SMART framework. It collects all single action options, analyzes their impacts on the system utility and generates an action plan that can optimize the system utility given the optimization window and the business constraints.

SMART can be deployed in file systems, storage resource management software and storage virtualization boxes. The key requirements are sensors to monitor the system behavior and collect data for establishing models and time-series forecast, and actuators to invoke the scheduled corrective actions. The decision algorithm can be located anywhere in the system as long as it can access the input modules and communicate with the actuators. In our prototype, we implemented SMART on the IBM's General Parallel File System (GPFS) [73], a scalable, distributed shared-disk file system.

In this chapter, we focus on the design of the Action Advisor, the intelligence behind SMART. The Action Advisor can be invoked both reactively as a response to an SLO violation, or proactively, driven by periodically examining the future system states using the time-series forecasts. When it is invoked, it will execute the following steps:

- First, it analyzes the current state $\mathcal{S}_0$ as well as the look-ahead states ($\mathcal{S}_1, \mathcal{S}_2, \cdots$) using the forecast workload demands. For each system state, it will first interpolate the performance of each workload using the system models, with the forecast workload characteristics as input

parameters. After that, it will calculate the corresponding system utility using the utility evaluator.

- Then it feeds the system states to the single action tools and collects their invocation options. The system states fed to the single action tools are determined by the interface of each action tool. For example, the Action Advisor generates individual queries to the CHAMELEON box for every current and future system state. But for SMARTMIG, it feeds all the current and future states in the optimization window in a single query and lets the SMARTMIG choose the best migration plan for the given optimization window.

- Next it analyzes the effect on utility for each action option using the utility evaluator and system models. This is similar to the first step.

- Last, it generates a schedule of *what* actions to invoke, *when* to invoke them and *how* they should be invoked. Depending on the operation mode, which can be either normal or unexpected, Action Advisor uses different decision algorithms to generate the schedule.

In summary, SMART is a framework for generating corrective action plans automatically. It takes the system models, time-series forecast, utility functions, business constraints as input, interacts with single action tools to collect the action options, and uses the Action Advisor to generate a multi-action schedule to maximize the system utility. In the next section, we will present our recursive greedy algorithm and the risk management analysis for finding the action schedule when the system states are predictable. After that, we will present our defensive action strategy based on the well-known *ski-rental* algorithm to handle the unexpected system changes.

6.3 Decision Algorithms of the Action Advisor

The Action Advisor is the core of SMART. It determines an action schedule consisting of one or more actions (*what*) with action invocation time (*when*) and parameters (*how*). The goal of the Action Advisor is to pick a combination of actions that will improve the overall system utility or, equivalently, reduce the system utility loss. First, we intuitively motivate its design to be followed by the details of how the Action Advisor performs its task.

The Action Advisor operates in two different modes depending on whether the corrective actions are being invoked proactively in response to the forecast workload growth, or reactively in response to unexpected variations in system states. The former is the *normal mode*, while the later is the *unexpected mode*.

In the normal mode, SMART uses a *recursive greedy algorithm with lookback and lookforward* to select the actions within the optimization window. The key idea is to use the greedy approach to select a single action option which is optimal for the entire optimization window. After that, we will partition the optimization window into the lookback and lookforward window. The former is the time window before the previously selected action finishes and the latter is the window after that. Within each sub-window, the same *greedy-lookback-lookforward* procedure is recursively performed to continuously find new actions that can be combined with the previously selected actions to improve the system utility further.

In the *unexpected mode*, SMART selects actions defensively. It tries to avoid invoking expensive actions since the workload variation could go away soon after the action in invoked, making the action invocation overhead wasted. However this needs to be balanced with the potential risk of the high workload persisting and thus incurring continuous utility losses. We formulate this analysis as a decision-making problem with an unknown future and apply the "ski-rental" online algorithm [48] to select actions.

The Action Advisor uses a primitive mechanism to transition between the normal and unexpected modes. It continuously compares the observed workload demands against the predicted ones using any chosen predictor model, such as ARIMA [84]. If the difference is large, for example, greater than twice the standard deviation of the prediction, it moves into the conservative unexpected workload mode. While in that mode, it continuously refines the future predictor functions by including the newly observed values. When the predicted and observed values are close enough for a sufficiently long period, it transits to the normal mode.

In the rest of this section, we first present action selections for the normal mode (Section 6.3.1), followed by the discussion on the risk management analysis to account for the future uncertainty

and the action invocation risk (Section 6.3.2). After that, we will present the defensive strategy for the unexpected mode (Section 6.3.3).

6.3.1 Normal Mode: Greedy Pruning with Lookback and Lookforward

The Action Advisor takes as input the current system state, the workload prediction, the utility functions, the available budget B for new hardware and the length of the optimization window, T. The goal is to find a set of actions that maximize the cumulative system utility in the optimization window. This process is formulated as a tree-construction algorithm (see Figure 6.3).

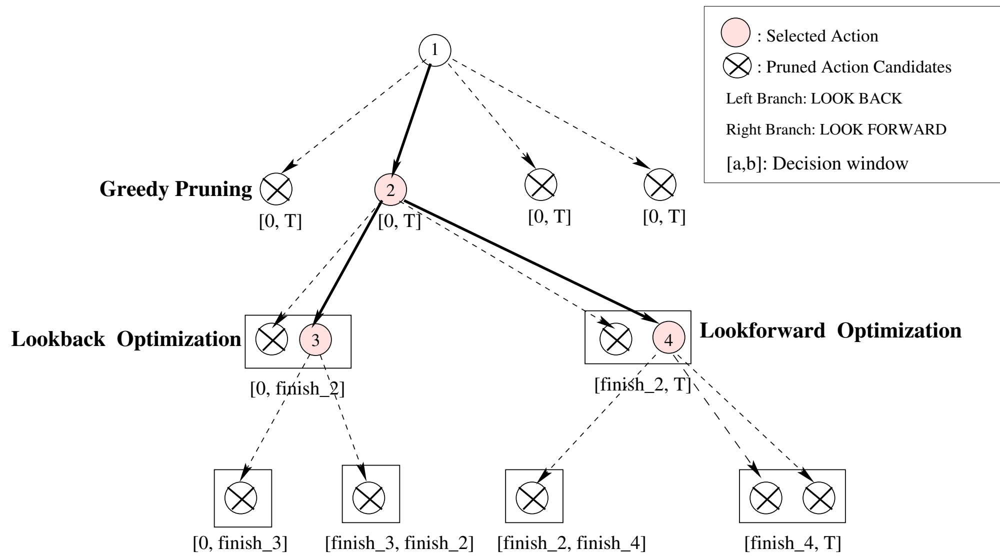


Figure 6.3. Tree based action schedule generation

In the tree-based representation, the root corresponds to the entire optimization window, $[0, T]$. The branches originating from the root represent the candidate actions returned by the single action tools. For m possible action options there will be m branches. The resulting node i for each action has the following information:

- The start ($start_i$) and end time (end_i) of the decision window, which specifies the time duration for which the action is optimized. For nodes originating from the root, the value is $[0, T]$, where T is the optimization window from the business constraints. For other nodes in the tree, the decision window is smaller as a result of the lookback and lookforward partition.

- The selected action and its invocation parameters.
- The action invocation time and finish time $[invoke_i, finish_i]$.
- The initial state $\mathcal{S}_i$ and resulting state $\mathcal{S}_{i+1}$. For example, the migration will change the workload-component mapping $\mathcal{M}$ and the hardware provisioning will change the component sets $\mathcal{C}$.
- The predicted cumulative utility loss UL_i, defined as the sum of system utility loss from $start_i$ to end_i if action i is invoked.

Greedy Pruning: Using the basic greedy approach, the Action Advisor selects a first-level node in the tree that has the lowest utility loss UL_i and prunes the other $m-1$ branches. The pruned actions are represented by the circles crossed out in Figure 6.3. In addition, a threshold is introduced to ensure that the action gives sufficient improvement in terms of the system utility. The selected action will be scheduled only if the improvement exceeds the threshold. The threshold is a configurable parameter. It controls how aggressive the Action Advisor prunes the action options—a higher value leads to more aggressive pruning. In our experiments, we set the threshold as 10%, indicating that an action option will be inserted to the action schedule only if it can reduce the system utility loss by more than 10%. The pseudocode for this greedy pruning procedure is given in Figure 6.4.

Lookback and Lookforward Optimization: The greedy pruning procedure selects the best action option for the given optimization window. However, in real-world systems, it may be required to invoke more than one actions concurrently. For example, if data migration is selected, it might be required to additionally throttle the lower priority workloads until all data are migrated. The Action Advisor uses the *lookback and lookforward optimization* to improve the action plan. The lookback and lookforward time windows are partitioned based on the finish time ($finish_i$) of the selected action. Lookback seeks action options in the time window $[start_i, finish_i]$ before the selected action finishes. Lookforward examines possible actions in the window $[finish_i, end_i]$ after the selected action finishes (see Figure 6.5).

The time $finish_i$ is chosen as the splitting point for two reasons. First, the system state may be permanently changed after the action finishes. For example, hardware arrival changes the component set and the migration operation changes the workload-to-component mapping. The change of

```
Function GreedyPrune (NodeSet) {
    MinNode = FindMinUL(NodeSet);
    Foreach (i in NodeSet) {
        if (i != MinNode) {
            delete(i, NodeSet);
        }
    }
    If ((UL(Parent(MinNode)) -UL(MinNode)}) < Threshold){
        delete(MinNode, NodeSet);
    }
}
```

Figure 6.4. Pseudocode for the greedy pruning procedure

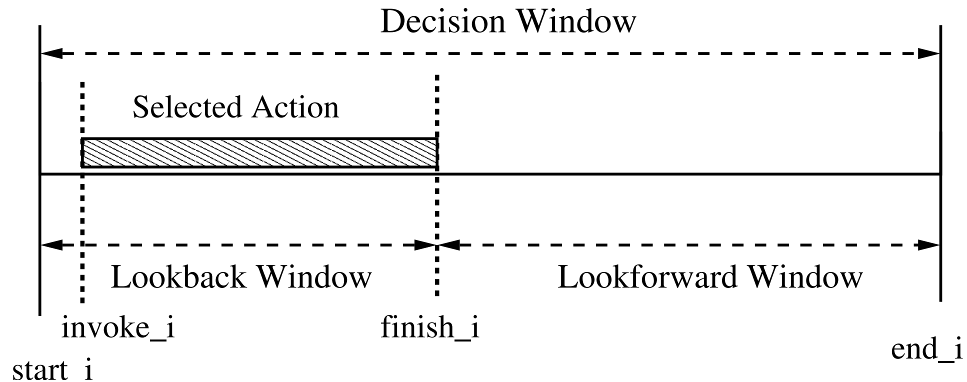


Figure 6.5. Lookback and lookforward optimization

system state may cause the cost-benefit of action options changed. For example, the original optimal migration operation is less preferable due to the arrival of new hardware. The second reason is that any action scheduled before the selected action finishes needs to satisfy the no-conflict constraint. We will describe this later.

Basically, the lookback phase explores possible actions that can be performed while the selected action is executing or has a wait time due to time constraints or resource limitation. For example, Figure 6.1 represents the case in which before new hardware arrives, redistributing the load using migration can result in a higher system utility than the old data placement configuration. In this

scenario, the tree scheduler may select "add hardware" as the best action. It then looks back to check if another action could give further improvement before the hardware arrives. The lookforward phase explores actions that can be invoked after the current one. This addresses cases where a combination of actions is the best solution and no single action can give the necessary benefit. For example, after hardware arrives, it often makes sense to schedule a migration operation to re-balance the system load. In addition, within the lookback and lookforward time windows, the *greedy-lookback-lookforward* procedure is recursively performed to construct an action schedule until the `GreedyPrune` finds no action option.

In the tree-construction, the action candidates for the lookback and lookforward windows are represented as the left and right children. These are marked as the circles in the rectangles in Figure 6.3. The pseudocode for lookforward and lookback optimization is given in function `Lookback` and `Lookforward` respectively (see Figure 6.6).

When considering actions to be scheduled before another action finishes in the lookback phase, the actions should not conflict with the existing selected actions. We say two actions conflict if one of the following conditions is true:

- They depend on the same resource. For example, two migration options are conflict if their source and target components overlap.
- Action j overlaps with an action k already in the schedule, and action j violates the precondition for action k. For example, migration action 1 of moving data A from LUN_1 to LUN_2 will invalidate action 2 of moving data A from LUN_1 to LUN_3 because the pre-condition of action 2 that data A is on LUN_1 is no longer true.

In summary, the Action Advisor generates a schedule of corrective actions using a recursive approach. The pseudocode is given in Figure 6.7. The final action schedule is obtained by sorting the un-pruned nodes in the tree in the order of their action invocation time ($invoke_i$). The un-pruned nodes are represented as solid circles in Figure 6.3.

```
Function Lookback(i) {
          Foreach (ActionTools) {
                Find action option in [start_i, finish_i];
                    If (!(Conflict(Existing actions)) {
                            Add to Left_Children(i);


                }
          }
          GreedyPrune(Left_Children(i));
          If (Left_Children(i)!=NULL) {
                     Lookback(Left_Children(i));
                     Lookforward(Left_Children(i));
          }
}

Function Lookforward(i) {
        Foreach (ActionTools) {
                    Find action option in [finish_i, end_i];
                    Add to Right_Children(i);
          }
          GreedyPrune(Right_Children(i))
          If (Right_Children(i)!=NULL) {
                     Lookback(Right_Children(i));
                     Lookforward(Right_Children(i));
          }
}
```

Figure 6.6. Pseudocode of lookback and lookforward optimization

6.3.2 Risk Management

In our previous discussion, we selected actions based on the cumulative utility loss UL_i. The accuracy of UL_i depends on the accuracy of future workload forecasting, performance prediction

```
Function TreeSchedule() {
  Foreach (ActionTools) {
    Find action option in [0, T];
    Add to Children(root);
  }
  GreedyPrune(Children(root));
  If (Children(root) !=NULL) {
     Lookback(Children(root));
     Lookforward(Children(root));
  }
}
```

Figure 6.7. Pseudocode of the recursive greedy procedure

and the estimation of the action's impact on the system utility. Inaccurate estimation of UL_i may result in decisions leading to less that expected overall utility. To account for the impact of inaccurate information, we perform the risk management analysis on UL_i for each action option. This is similar to the risk management in SMARTMIG, described in Section 5.3.3. The *risk* captures both the probability that the utility gain of an action will be lost as a result of volatility in the prediction of the future workload, and the amount of utility that could be lost due to a wrong action decision.

We borrow the concept of *Value at Risk* (VaR) from the domain of portfolio management [52]. In the context of SMART, the VaR gives the bound on the standard deviation of the workload demands prediction during the given optimization window, with a 95% confidence value. It measures the volatility in the workload future prediction and is calculated using the following equation:

$$VaR(95\% \, confidence) = 1.65\sigma \times \sqrt{T} \tag{6.1}$$

where σ is the standard deviation of the time-series request-rate predictions and T is the optimization window.

In addition, we also define the risk coefficient α_i for every action option i based on its opera-

tional semantics as follows :

$$\begin{aligned}
\alpha_{thr} &= 0 \\
\alpha_{mig} &= \frac{bytes_moved}{total_bytes_on_source} * Sys_Utilization \\
\alpha_{hw} &= \frac{hardware_cost}{total_budget} * (1 - Sys_Utilization)
\end{aligned}$$

Where $Sys_Utilization$ is the system utilization when the action is invoked. The risk coefficient α_i reflects the invocation risk associated with every action option. Intuitively, a higher system utilization or a larger migration data set increases the risk of invoking the migration operation, and a high hardware purchase cost or a lightly loaded system makes the provisioning more risky.

Based on the VaR value and the risk coefficients, the risk value $RF(A_i)$ for each action option i can be calculated using the same equation (Equation 5.6) as in SMARTMIG:

$$RF(i) = (1 + \alpha_i) * VaR$$

Then the Action Advisor will calculate the risk value $RF(i)$ and scale the utility loss UL_i for each action option i to balance the benefit (UL_i) and the risk $RF(i)$. The greedy pruning described earlier is performed based on the scaled UL_i^*:

$$UL_i^* = (1 + RF(A_i)) \times UL_i \tag{6.2}$$

In summary, when the system has good knowledge on the future workload demands, the Action Advisor generates the action schedule using the recursive greedy algorithm with lookback and lookforward optimization. In addition, to account for future uncertainty and action invocation risk, the Action Advisor performs risk management analysis and calculates the scaled utility loss (UL_i^*) for each action option, which is then used for greedy pruning.

6.3.3 Unexpected Mode: Defensive Action Selection

In the previous section, we described the decision algorithm for generating the action schedules in the normal mode. However, in the real world, there are always situations, such as workload

characteristics change, workloads come and leave, or load surges, that make an accurate prediction infeasible. Optimizing the action selection for the unexpected system states is challenging since it is difficult to predict the duration for which the workload variations will persist. The Action Advisor uses a strategy similar to the one used in online decision making scenarios such as the "ski rental: to rent or to buy" [48]. In that scenario, the choice of whether to buy or rent a ski equipment has to be made without the knowledge of how many times one might go skiing in the future. In the absence of the knowledge about the future, the commonly used strategy is "to keep renting until the amount paid in renting equals the cost of buying, and then buy." This strategy is always within a factor of two of the optimal cost, regardless of how many times one goes skiing [48].

The Action Advisor follows a similar online strategy. It selects the least costly action until the cumulative utility loss for staying with that action exceeds the cost of invoking the next expensive action. When SMART is in the unexpected mode, the Action Advisor first finds all action candidates under the assumption that the system state and workload demands will remain the same. For each candidate A_i, the cost at Time 0, when SMART is switched to the unexpected mode, is initialized as the extra utility loss and hardware cost if any paid for the action invocation. This is shown in Equation 6.3:

$$Cost(A_i, 0) = \sum_{t=0}^{leadtime(A_i)} (U_{sys}(no_action, t) - U_{sys}(A_i_ongoing, t)) + HW_Cost(A_i) \quad (6.3)$$

In this equation, $U_{sys}(no_action, t)$ is the system utility value at time t if no corrective action had been taken and $U_{sys}(A_i_ongoing, t)$ is the system utility at time t if A_i is underway. For example, for throttling, $Cost(A_i, 0)$ will be zero because the lead time is zero. For migration, the $Cost(A_i, 0)$ is the total utility loss over $leadtime(A_i)$ due to allocating bandwidth to move data around.

Action Advisor selects the action with the minimum cost and invokes it immediately. Over time, the cost of each action candidate, including both the selected one and the unchosen ones, is updated continuously to reflect the utility loss experienced if A_i had been invoked ($UL(A_i, t)$). Equation (6.4) gives the value of $Cost(A_i, t)$ at time t:

$$Cost(A_i, t) = Cost(A_i, t-1) + UL(A_i, t) \quad (6.4)$$

The cost is continuously updated and the Action Advisor will invoke another action when its cost is lower than the previously invoked action. Figure 6.8 illustrates the defensive algorithm described above. In the figure, the X axis is the time and Y axis is the action cost. The Time 0 is the

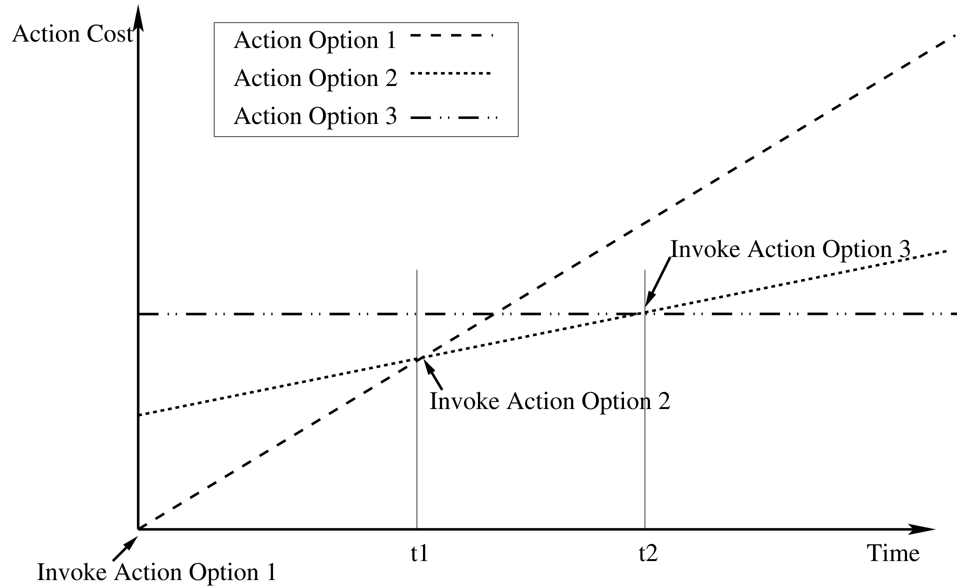


Figure 6.8. Example of the unexpected mode

start time of the unexpected mode. In the example, the Action Advisor finds three action options and invokes Option 1 at Time 0 immediately because Option 1 has the minimum action cost. While the Action Advisor operates in the unexpected mode, the cost for all action options are updated continuously using Equation 6.4. At Time t_1, the cost of Option 2 is lower than Option 1. That is, $Cost(A_2, t_1) < Cost(A_1, t_1)$. As a result, the Action Advisor invokes Option 2 immediately. Similarly, at Time t_2, Option 3 is invoked because it becomes the lowest cost option.

This procedure for cost updating and action invocation continues until one of the following situations happens:

- The system goes back to a good state, defined as no utility loss, for a period of time. In this case, the Action Advisor will stop the action selection procedure because the exception has gone.
- The system collects enough new observations and transitions back to the normal mode. This

occurs when the distance between the predicted values and the observed values are within the threshold.

In summary, the Action Advisor is designed to have two operation modes, *normal* and *unexpected*. In the *normal* mode, the Action Advisor leverages the forecast workload demands and aggressively selects actions that can benefit the predicted future. It recursively performs the greedy-lookback-lookforward optimization to improve the action schedule until the utility gain of adding another action is marginal, that is, below a given threshold. To account for the future uncertainty and the action invocation cost, the Action Advisor also performs risk management analysis. This is to scale the utility loss for each action option such that the benefit and risk can both be counted. When the predicted values do not match the observations, the Action Advisor switches to the *unexpected* mode, where a defensive action strategy is applied. The key idea is to always invoke the action option with the minimum action cost. In both the normal and the unexpected model, the Action Advisor continuously collects new observations from the system to improve the prediction accuracy. In the next section, we will present our experimental results in both the GPFS system and a simulator. Our focus is on understanding SMART's efficiency in terms of the reduction in utility loss, and its computation time and sensitivity to input errors.

6.4 Experimental Evaluation

SMART generates an action schedule to improve the system utility. To evaluate the quality of its decision, we implement SMART in both a real file system GPFS [73] and a simulator. The real system implementation allows us to verify if SMART can be applied in practice. The simulator provides us a more controlled and scalable environment, thus allowing us to perform repeatable experiments to gain insights on the overhead and sensitivity to input information errors.

SMART is designed to automatically generate the action plan to maximize the system utility for a given optimization window. To evaluate the quality of its decision, we need to answer following questions:

- First, we need to examine whether SMART's decision can adapt to configuration changes.

SMART is designed to find the action plan that is optimal for given configuration parameters such as the utility functions and the optimization window. When these parameters are different, SMART should be able to account for these differences and generate an appropriate plan accordingly. To examine SMART's ability to account for the configuration parameters and adapt to their changes, we design the sanity check tests, where we vary the configuration parameters and observe the changes on the generated action plan. By doing this, we can examine whether SMART meets its design goal of generating the optimal action plan for a given configuration.

- Second, we must examine whether SMART works as expected. SMART can be used to control the corrective action invocation automatically without human intervention. Alternatively, it can be used to assist the administrators to find the optimal action plan. For both cases, the question is, if we follow the SMART's decision, will the system utility be improved? We design the feasibility experiments, in which, we execute SMART's action plan in the GPFS prototype and evaluate the system utility it achieves. Specifically, we select two representative cases in the sanity check tests. We then compare the observed system utility in the real execution with the predicted value calculated based on system models and workload forecast. By doing so, we examine whether SMART can work as expected in real system, and thus improve the system utility.

- After examining how SMART adapts to configuration changes and its behavior in real system, we next examine the quality of SMART's decision with or without input errors. We perform the sensitivity test to examine the quality of the decisions with accurate component models and future prediction. We then introduce errors to the models and predictions to evaluate their impact on the quality of SMART's decisions. By doing this, we gain insights into how input errors may affect SMART's decision.

- Lastly, to examine whether SMART can find the optimal action plan rapidly, we perform the decision overhead experiment to measure the decision-making time of SMART as a function of the number of workloads.

In the rest of this section, we first define our evaluation metrics. After that, we describe our

GPFS prototype implementation and the testbed configurations. Then, we present the experimental results of four tests, sanity check, feasibility test, sensitivity test and decision overhead test.

6.4.1 Evaluation Metric

An ideal yardstick to evaluate the quality of its decisions is by comparing SMART with existing automated algorithms or with decisions made by an administrator. However, we are not aware of any existing work that considers multiple actions. Neither is comparison with the administrator's decision feasible because it is difficult to quantify how representative is a given individual's decision. Thus, we take an alternative approach. We compare SMART's quality with theoretical bounds. Since SMART's design goal is to find corrective actions that maximize the system utility, we evaluate the quality of an action plan by comparing the system utility of SMART's decision with the maximum theoretical system utility (see Equation 3.3) and the system utility without any action. The former is the upper bound and the latter is the lower bound. In our tests, we use two evaluation metrics. One is the utility loss of SMART, defined as the distance of SMART's system utility from the maximum system utility. It considers whether SMART can correct the system behavior and how far it is from the best system utility value. The second metric is the distance of SMART's utility loss from that of no corrective action. It measures the gain of using SMART. These two metrics are the basic ones we use in our tests. In some experiments, we define other evaluation metrics, such as percentage of utility loss, based on these two values. We give the detailed definitions when they are used.

6.4.2 GPFS Prototype Implementation

The SMART prototype is implemented on the General Parallel File System (GPFS): a commercial high-performance distributed file-system [73]. GPFS manages the underlying storage systems as *pools* that differ in their characteristics of capacity, performance and availability. Storage systems can be accessed by any clients nodes running on separate physical machines transparently.

The prototype implementation involves *sensors* for monitoring the workloads states, *actuators* for executing corrective actions and the Action Advisor for performing the decision making.

Sensors: They collect information about the run-time state of workloads. The monitor daemon in each GPFS client node tracks the access characteristics of the workload and writes it to a file, that can be analyzed periodically in a centralized fashion. Workloads are the unit of tracking and control in the prototype implementation. A workload is manually specified as a collection of PIDs assigned by the OS using a registration daemon. The monitoring daemon does bookkeeping at the GPFS read/write function call after the VFS translation.

Action actuators: Although the long term goal of the prototype is to support all corrective actions, as a proof of concept, we implement action actuators for three most commonly used corrective actions: throttling, migration and adding new hardwares. An example of the latter is adding pools in GPFS.

- The I/O throttling is enforced at the GPFS client nodes using a token-bucket algorithm [31]. Decision-making for throttling each workload is made in a centralized fashion, with the token-issue rate and bucket size being written to a control file that is then periodically (20 ms) checked by the node throttling daemon.
- Similarly, the control file for the migration daemon consists of entries of the form <file name, source pool, the destination pool> and the migration speed is controlled by throttling the migration process. The migration daemon runs in one of the client nodes. It periodically checks for updates in the control file and invokes the GPFS built-in function `mmchattr` to migrate files.
- Because the addition of hardware normally requires human intervention, we mimic the effect of adding new hardware by reserving hardware devices and forbidding the access to them until SMART decides to add them into the system. In addition, the reserved storage devices are configured with different lead times to mimic the overhead of placing orders and performing installations.

Action Advisor Integration: The SMART action advisor is implemented using techniques described in Section 6.3. The time-series forecasting is done off-line, where the monitored access characteristics for each workload are periodically fed to the ARIMA module [84] for refining the

future forecast. Similarly, performance prediction is done by bootstrapping the system for the initial models and refining the models as more data are collected. Once the Action Advisor is invoked, it communicates with single action decision-making tools and generates an action schedule. The selected actions are hold by the action advisor until action invocation time, as determined by SMART. At that time, the control files for corresponding action actuators are updated according to SMART's decision. The single action tools for throttling and migration are based on the CHAMELEON and the SMARTMIG frameworks we have already described. SMART reaches its provisioning decision by estimating the overall system utility for each provisioning option. This in turn considers the utility loss before the hardware arrives, the loss introduced by the load balancing operation after the new hardware comes into place, and the financial cost of buying and maintaining the hardware.

6.4.3 Testbed Configurations

Workload	Request Size [KB]	Read/ Write Ratio	Random/ Sequential Ratio	Footprint Size [GB]	ON/OFF Phase [Hour]	ON/OFF [Iops]
W_{Trend}	16	0.7	0.8	60	12/12	150/100
W_{Backup}	16	1	1	600	8/16	250/0
W_{Phase}	8	0.8	0.9	6	14/10	150/100

Table 6.1. Access characteristics of workloads

The configuration in our testbed is as follows:

- ***Workloads:*** We use the same trace replay technique and synthetic workload generators as in the CHAMELEON testbed. A detailed description is in Section 4.7. By mixing real-world trace replay with synthetic workloads, we introduce both real system fluctuation and long-term workload trend. We use four workloads. One is a two month trace replay, spanning from Nov. 1, 1999 to Dec. 30, 1999, of HP's Cello99 traces [68]. The other three workloads are synthetic workload traces with the following characteristics: (1) W_{Trend} is used to simulate a workload with a growth trend—the ON phase load increases by 100 IOPS every day while the OFF phase load increases by 50 IOPS; (2) W_{Phase} is a workload with periodic ON-OFF phases. Many real applications like departmental file servers exhibit this ON-OFF behavior as a result of users' ON/OFF working schedule; and (3) W_{Backup} simulates a backup load

with its ON-phase as an inverse of the *phased* workload. This is to mimic the real world scenario in which backup applications often operate when regular applications (the W_{Phased}) are lightly loaded. The access characteristics of these workloads are summarized in Table 6.1. The footprint size of real applications are often on the order of TB. We intentionally create workloads with footprint size in the range of GB to reduce the overhead of data movement. SMART can naturally handle workloads with large footprint size because the footprint size is a parameter for both system models and the risk management analysis. Other I/O characteristics are chosen to generate a reasonable load to the testbed.

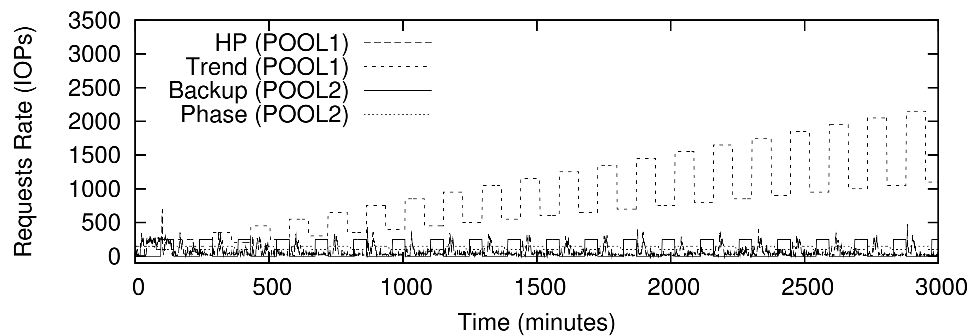
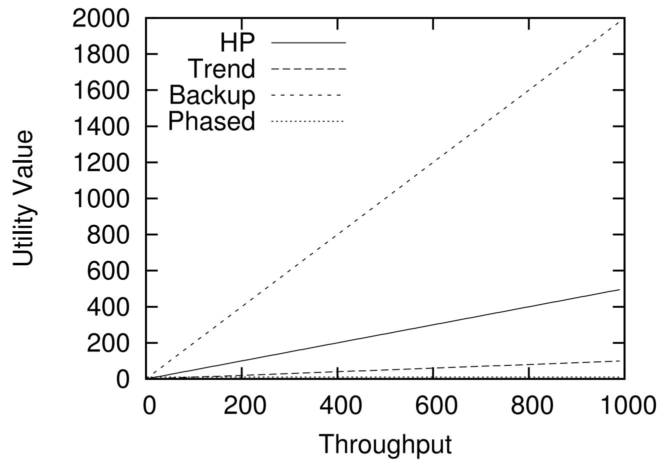


Figure 6.9. IO rate of workloads as a function of time

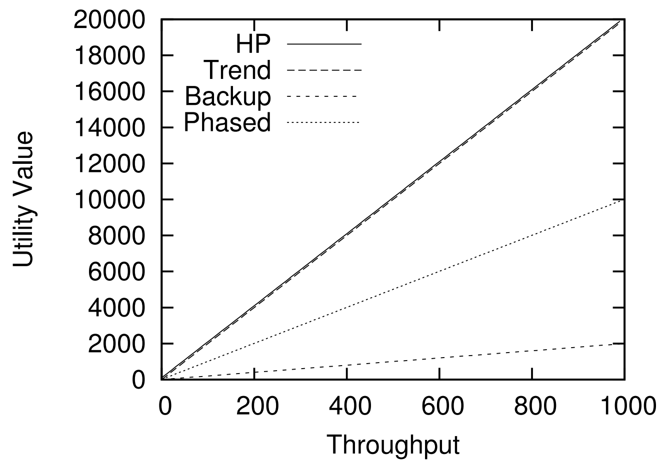
Figure 6.9 shows the I/O rate of these workloads as a function of time. In addition, the utility functions of workloads are defined based on the throughput and the SLO latency. Its value is a function of the received throughput and latency. Depending on whether the received latency can meet the $SLO_{latency}$ or not, different utility functions are used. Figure 6.10 plots the default utility functions for violating and meeting the SLO latency goals.

- ***Components:*** Due to hardware resource limitation, we configure our test bed with three logical volumes: POOL1 and POOL2 are both RAID 5 arrays with 16 drives each, and POOL3 is a RAID 0 with 8 drives. POOL3 is originally off-line, and is accessible only when SMART selects hardware provisioning. The initial workload-to-component mapping is: [HP: POOL1], [Trend: POOL1], [Phased: POOL2] and [Backup: POOL2].

- ***Miscellaneous settings***: The optimization window is set to one month because the synthetic trending workload (W_{Trend}) is growing with a much faster speed, 100 IOPS growth every day, than in the real world. The default budget constraint is $20,000. The one day standard



(a) Utility functions for violating the SLO latency goals



(b) Utility functions for meeting the SLO latency goals

Figure 6.10. Utility functions of workloads

deviation of the future prediction for risk analysis is configured as 10% unless otherwise specified.

For these initial system settings, the system utility loss at different time intervals without any corrective action is shown in Figure 6.11.

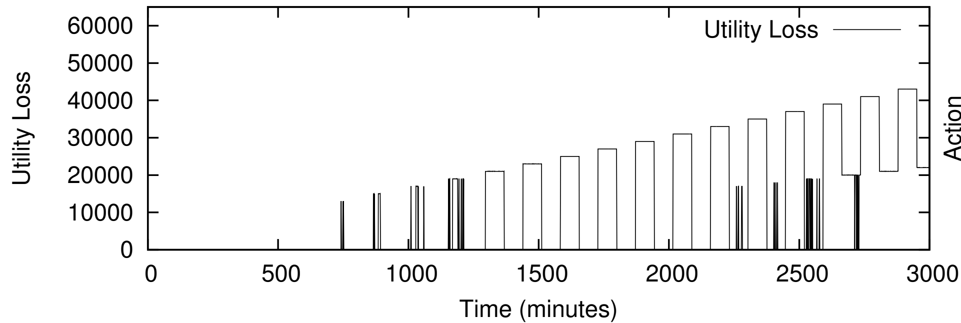


Figure 6.11. Utility loss if no corrective action is invoked

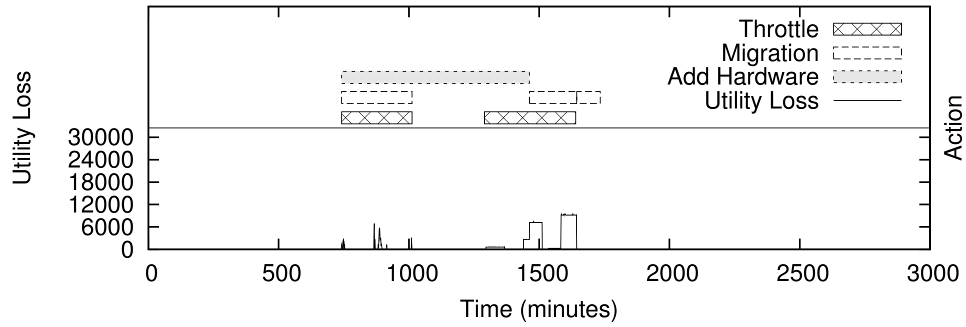
6.4.4 Sanity Check

As a sanity check, we give the Action Advisor the initial system settings as input, and we vary the configuration parameters to examine their impacts on SMART's action schedules. The key question is whether SMART will generate a different schedule that is appropriate for the configuration and improves the system utility.

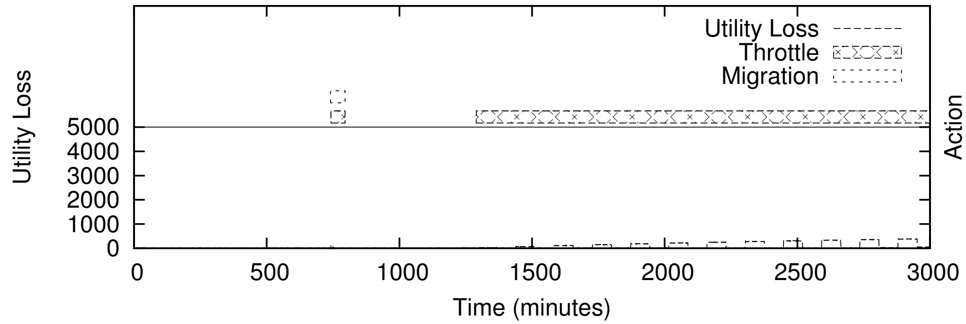
We vary the following configuration parameters: the utility function values (Test 1), the length of optimization window (Test 2), the budget constraints (Test 3) and the risk factor (Test 4). In Test 5, we explore how SMART handles unexpected mode. For each of these tests, we present the corrective action schedule generated by SMART, and the corresponding *predicted* utility loss as a function of time. In the figures, the action schedule is represented by using different patterned boxes for different actions: the empty box for migration, the cross patterned box for throttling and the shadowed box for provisioning. The length of the boxes represent the invocation duration of the action.

Test 1: Impact of Utility Functions

SMART selects actions that maximize the overall system utility value, which is driven by the utility functions for individual workloads using the storage system. In this test, we vary W_{Trend}'s utility function when meeting the SLO latency goal from the default $20 * Thru$ to $540 * log(Thru + 1)$. As shown in Figure 6.12(a), the default utility assignment for W_{Trend} causes a fast growing overall utility loss. Therefore, SMART chooses to add new hardware. However, for the low value



(a) High Utility Value Assignment $UF_{trend} = 20 * Thru$



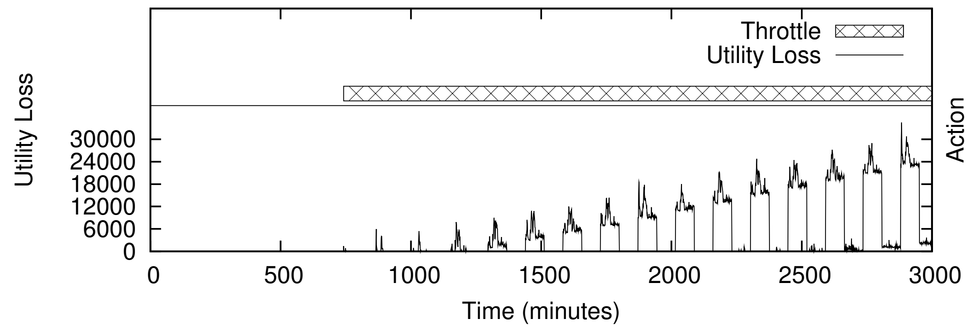
(b) Low Utility Value Assignment $UF_{trend} = 540 * log(Thru + 1)$

Figure 6.12. Action invocation for different utility functions assigned to W_{trend}

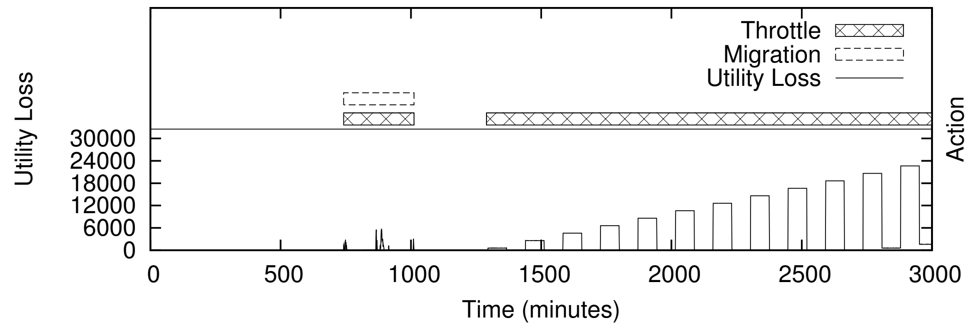
utility assignment, the latency violation caused by the increasing load causes a much slower growth in the utility loss. As a result, the cost of adding new hardware cannot be justified for the current optimization window. Thus SMART decides to settle for throttling and migration. As the utility loss slowly approaches to the point where the cost of adding new hardware can be justified, SMART will suggest adding new hardware.

Test 2: Impact of Optimization Windows

SMART selects action options that are optimal for a given optimization window. In this test, we vary the optimization window to two days, one week and one month, the latter is the default value. Compared to the schedule for one month in Figure 6.12(a), Figure 6.13 shows that SMART



(a) Two days optimization window



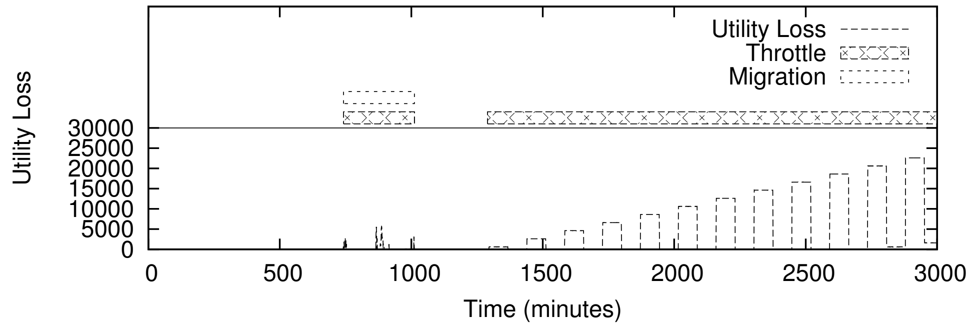
(b) One week optimization window

Figure 6.13. Action invocation for different optimization windows

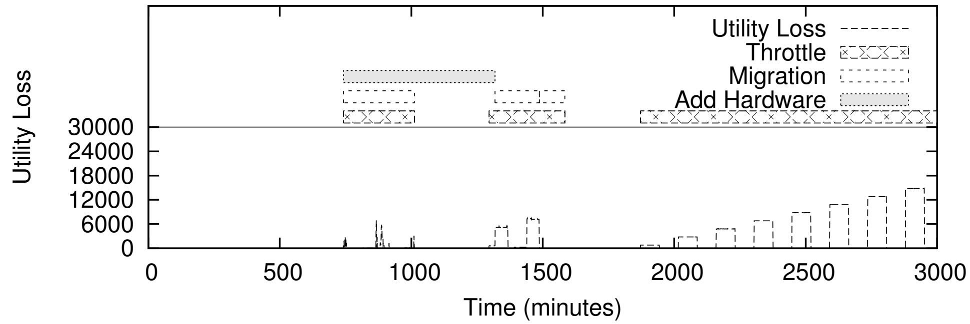
correctively chooses varying action schedules for different optimization windows. In brief, for a short optimization window (Figure 6.13(a) and 6.13(b)), SMART correctly selects action options with lower cost. For a longer optimization window (Figure 6.12(a)), it suggests corrective options with higher costs, but which yield better system utility in the long run.

Test 3: Impact of Budget Constraints

Test 3 demonstrates how SMART responds to various budget constraints. As shown in Figure 6.14, SMART settles for throttling and migration if no budget is available for buying new hardware. With $5000 budget, SMART opts for adding a hardware. However, compared to the hardware



(a) No budget available



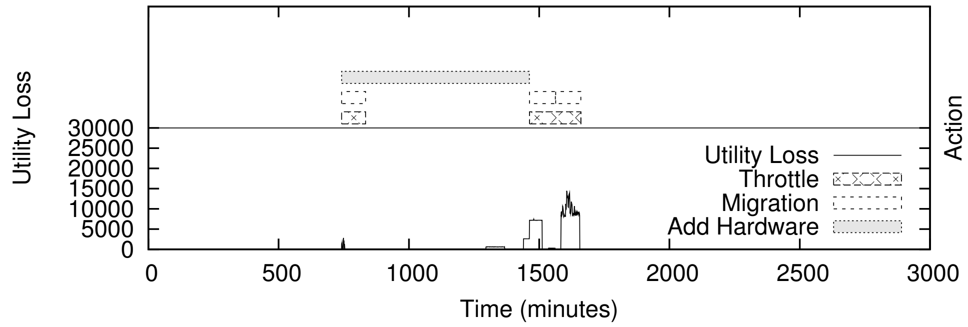
(b) Low budget available ($5000)

Figure 6.14. Action invocation for different budget constraints

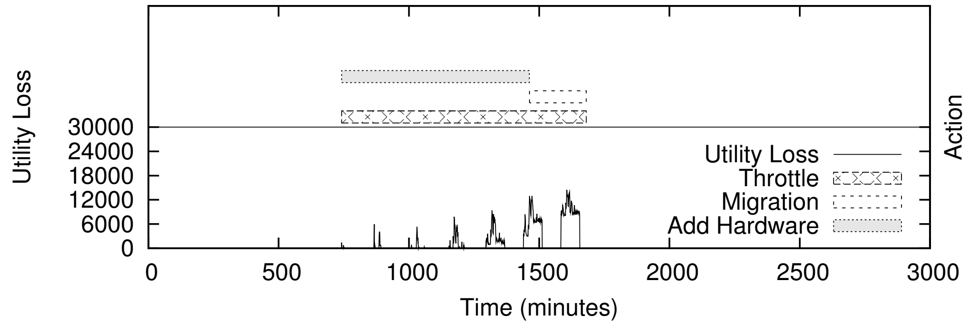
selected for the default $20,000 budget (shown in Figure 6.12(a)), the hardware selected does not solve the problem completely, so SMART also adds a traffic regulation using throttling.

Test 4: Impact of Risk Management

SMART uses risk management to balance between the risk of invoking an inappropriate action and its benefit on the utility value. For this experiment, the size of the dataset selected for migration is varied from 20GB to 1TB. This changes the risk value (see Equation 6.3.2) associated with the action options. SMART will select the high-risk option only if its benefit is proportionally higher. As shown in Figure 6.15, SMART changes the ranking of the corrective options and selects a different action invocation schedule for the two cases.



(a) Migration option: 20GB data movement

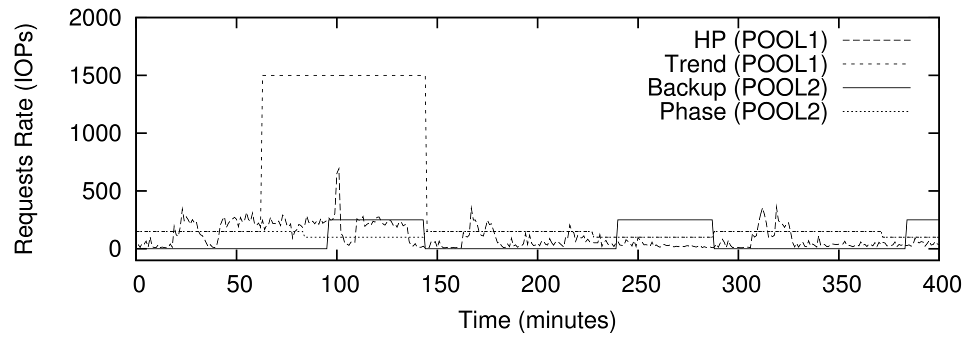


(b) Migration option: 1TB data movement

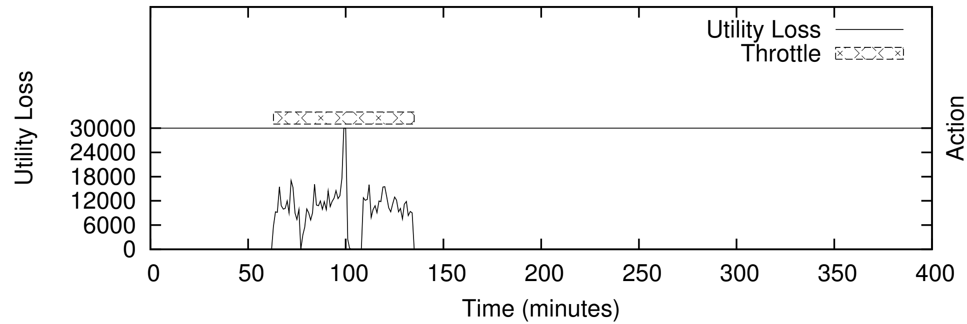
Figure 6.15. Action invocation for different risk factor

Test 5: Handling of the Unexpected Case

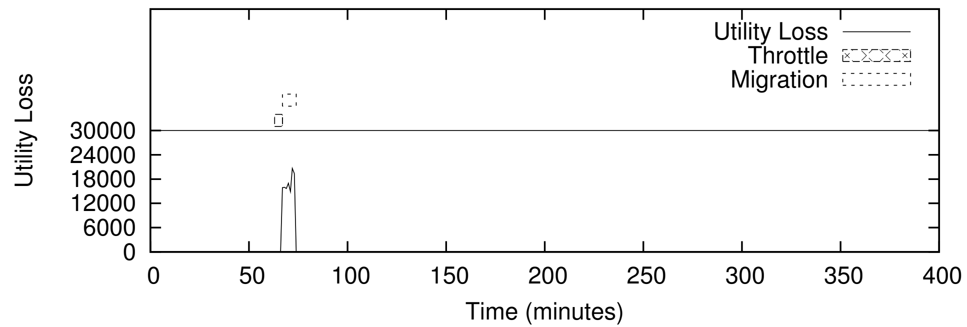
This test explores SMART's ability to handle unexpected workload demands. Figure 6.16(a) shows the sending rate of the workload demands. From the 60th minute to the 145th minute, W_{Trend} sends at 1500 IOPS instead of the normal 250 IOPS. The difference between the predicted value, 250 IOPS, and the observed value, 1500 IOPS, exceeds the threshold and SMART switches to the unexpected mode. We test two cases with different migration options. For Case 1, the available migration option involves moving one TB of data. For Case 2, the migration option only requires moving eight GB of data. As shown in Figure 6.16, for both cases, SMART invokes throttling directly. But for Case 1 (Figure 6.16(b)), the migration operation is never invoked. This is because the spike duration is not long enough to reach a point where the migration invocation cost is less



(a) Workloads demands



(b) Result for a “short” spike



(c) Result for a “long” spike

Figure 6.16. Action invocation for unexpected case

than the utility loss of staying with throttling. For Case 2, at the 5th minute, the utility loss due to settling for throttling exceeds the cost of moving eight GB of data. Thus, SMART invokes migration immediately.

6.4.5 Feasibility Test Using the GPFS Prototype

In these tests, SMART runs in the actual GPFS deployment. The tests serve two purposes: (1) verify if the action schedule generated by SMART can actually help reduce the system utility loss; and (2) examine if the utility loss predicted by SMART matches with the observed utility loss. We run two tests to demonstrate the normal and the unexpected mode operation.

Test 1: Normal Mode

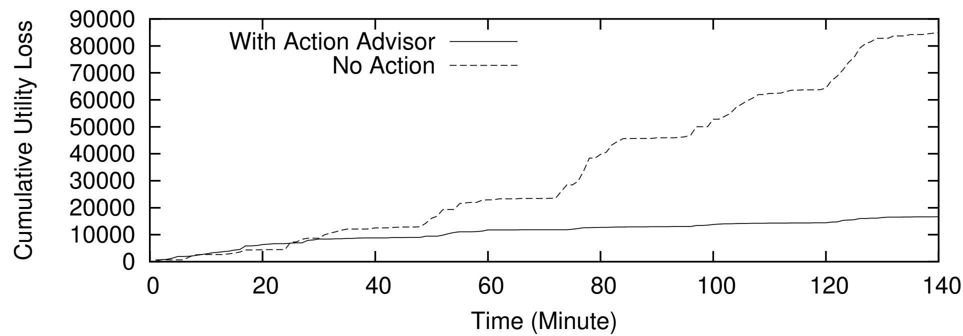


Figure 6.17. The cumulative utility loss comparison between no action and SMART's actions

In this experiment, to reduce the experiment running time, the I/O requests are replayed/generated 60 times faster. In other words, every minute in the figure corresponds to one hour in the trace. The settings for this experiment are the same as those described in Section 6.4.3 but with two exceptions: the footprint size and the lead time for adding hardware. The footprint size is shrunk by a factor of 60. The lead time of adding hardware is set to 55 minutes. This maps to the real value of 55 hours.

In the first step, SMART decides that the best option is to add a new storage pool. Because it takes 55 minutes to come into effect, SMART looks back to seek for solutions that can reduce the utility loss for the time window [0, 55]. It chooses to migrate the HP workload from POOL1 to POOL2 and throttle all workloads until the new pool arrives. After POOL3 joins, the load balance operation decides to migrate the $Trend$ workload to POOL3 and HP back to POOL1. The final workload-to-component mapping is: HP on POOL1, $Backup$ and $Phased$ on POOL2 and $Trend$ on POOL3.

As shown in Figure 6.17, compared to the case of no action, SMART's action schedule eliminates about 80% of the utility loss. Also the utility loss grows at a much slower rate. Before the 20th minute, the utility loss of no action is slightly lower than with SMART because SMART's schedule pays extra utility loss for invoking the migration operation.

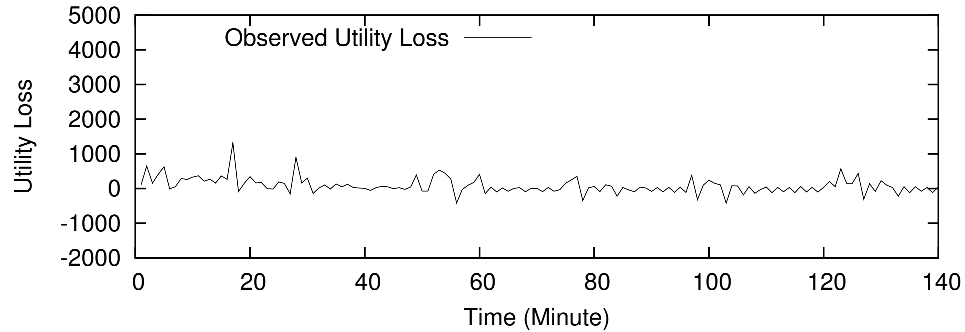


Figure 6.18. The observed utility loss

From Figure 6.18, we can see that sometimes, the utility loss is a negative value. This is because the maximum utility is estimated based on the *planned* workload demands, while the observed utility value is calculated based on the *observed* throughput. Due to the lack of precise control in task scheduling, the workload generator cannot generate I/O requests precisely as specified. For example, the workload generator for the HP traces plans to send requests at 57.46 IOPS at the 33rd minute, but has a real throughput of 58.59 IOPS. In this case, the observed utility value is higher than the estimated maximum value, thus, resulting in a negative utility loss. For a similar reason, the observed utility loss fluctuates frequently around zero.

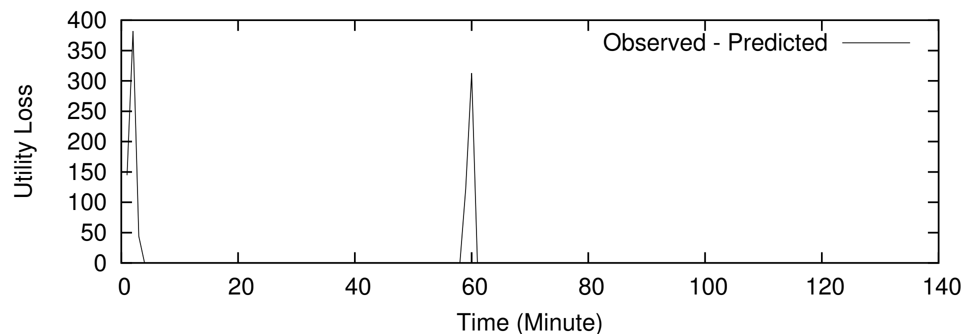


Figure 6.19. Difference in utility loss (after filtering): observed value-predicted

SMART schedules actions and predicts the utility loss to be close to zero. However, the observed

utility loss, shown in Figure 6.18, has many non-zero values. To understand the cause of this, we filter out the amount of observed utility loss due to imprecise workload generation described above, and plot the remaining utility loss in Figure 6.19. We can see that the predicted and observed values usually match except for two spikes at the second and the 58th minute. Digging into the log of the run-time performance, we found several high latency spikes on the migrated workload during the migration procedure. These latency spikes are more than 60 ms, which is much higher than the normal delay of 10 ms. This is because the migration process locks 256 KB blocks for consistency purposes. Hence, if the workload tries to access these blocks, it will be delayed until the lock is released. The performance models fail to capture this scenario, so we observe a mismatch between the predicted and observed utility values. However, these are transient behaviors in the system and do not affect the overall quality of SMART's decision.

Test 2: Unexpected Mode

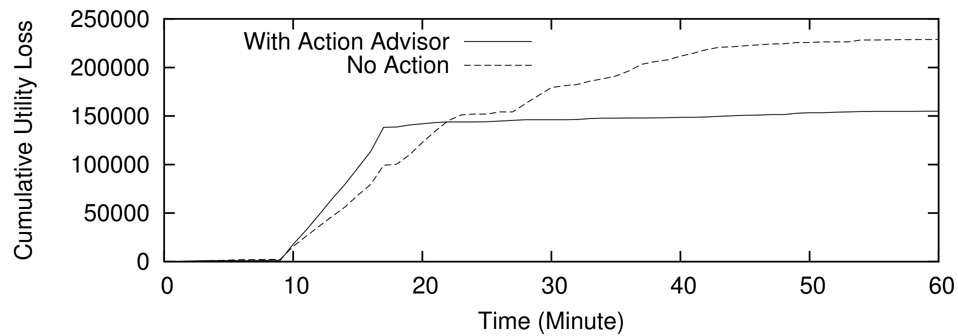


Figure 6.20. Unexpected mode: the cumulative utility loss with migration invoked and with no action

Similar to the sanity check test of unexpected case handling in Test 5, we intentionally create a load surge from the 10th to 50th minute. SMART invokes the throttling operation immediately and waits for about 3 minutes till the invocation cost of the migration option is lower than the utility loss of throttling. The migration operation executed from the 13th to 17th minute and the system experiences no utility loss once it is done. Similar to the previous test, the temporary smaller utility loss for the case of no action (shown in Figure 6.20) is due to the extra utility loss for the data movement. We skip the other figures, as they offer no new observations for predicted and observed utility values.

6.4.6 Sensitivity Test

In the sensitivity test, we examine SMART's sensitivity to errors of performance models and future prediction in a variety of configurations. Because the simulator provides a more controlled and scalable environment and allows us to test various system settings, we use simulations to perform the sensitivity test. Our simulator is based on SMARTMIG, described in Section 5.4, with the Action Advisor algorithm replacing the migration decision algorithm.

In this test, we vary the number of workloads in the system from 10 to 100. For each workload setting, we automatically generate 50 scenarios in a similar fashion as in the SMARTMIG's evaluation. We randomly generate the initial data placement, workload trending, workload features and utility functions for each scenario, as described in Section 5.4. We then invoke the Action Advisor to compute the action schedule.

For a three month optimization window, the 500 scenarios experience the utility loss in various degrees, ranging from 0.02% to 83% of the maximum system utility. The Cumulative Distribution Function (CDF) of the percentage of utility loss without corrective actions is shown in Figure 6.21, where the percentage of utility loss is defined in Equation 6.5. As shown in the CDF curve, for more than 90% of cases, the system experiences a utility loss higher than 20%.

$$percentage\ of\ utility\ loss = \frac{utility_loss}{maximum_utility} \tag{6.5}$$

Test 1: Without Model Errors and Prediction Errors

For this test, we assume both the performance prediction and the future prediction are accurate. The CDF of the percentage of utility loss with SMART's actions is plotted in Figure 6.21. As shown in the figure, for all scenarios, the utility loss is less than 20%. This is a significant improvement compared to the case of without SMART, which results in 90% of cases with a utility loss higher than 20%.

We also measure the percentage of the saved utility loss, defined in Equation 6.6. It reflects the amount of utility loss eliminated by SMART's action plan. When it is approaching to 1, the action

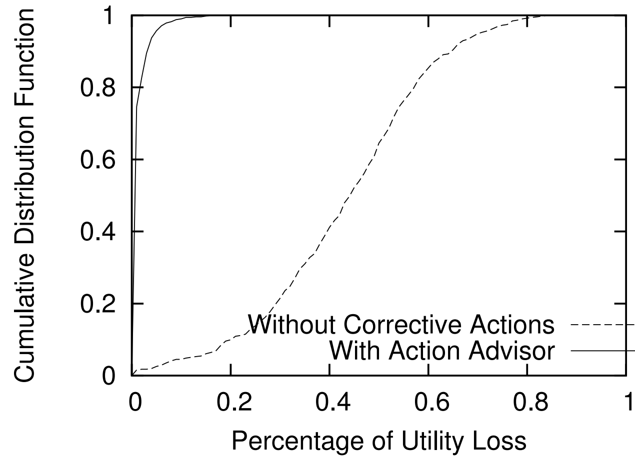


Figure 6.21. CDF of the percentage of utility loss

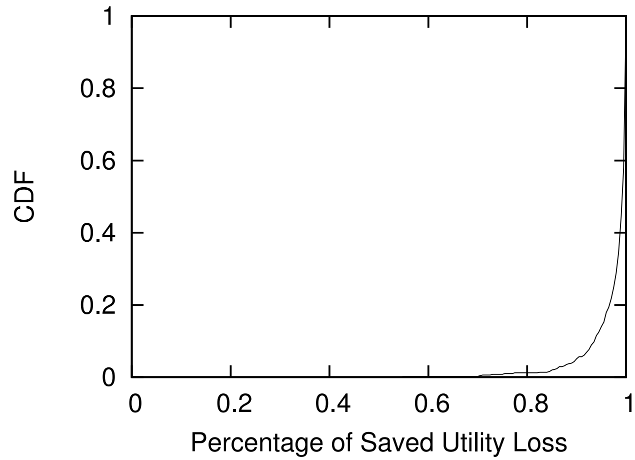


Figure 6.22. CDF of the percentage of the saved utility loss

plan is more effective.

$$percentage_of_saved_utility_loss = \frac{utility_loss_without_actions - observed_utility_loss}{utility_loss_without_actions} \quad (6.6)$$

Figure 6.22 plots the percentage of the saved utility loss. It shows that for more than 90% of tested scenarios, SMART elimiates more than 94% of utility loss. For the remaining 10% of cases, it reduces the utility loss by 70%.

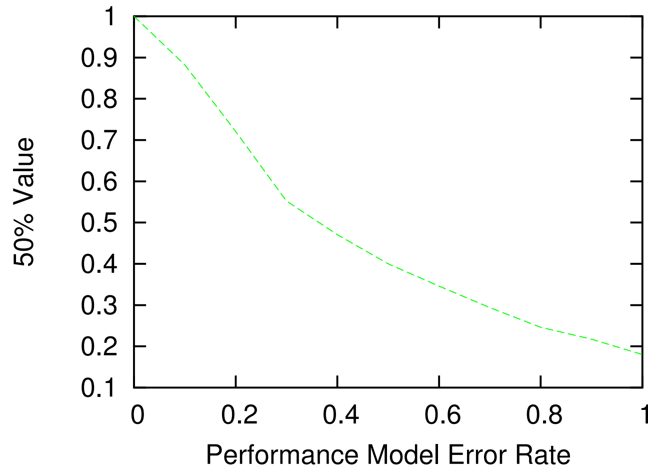


Figure 6.23. Impact of model errors on the accuracy of the predicted utility loss

Test 2: With Performance Model Errors

Our previous analysis is based on the assumption that accurate component models are available for extrapolating the performance for a given workload. However, this is never true in real-world systems. To understand how performance prediction errors affect the quality of SMART's decision, we perform the following experiments:

- First we generate the synthetic system models to simulate the real system models constructed using modeling techniques, such as black-box regression and analytical models. For any given system state, we use these models to calculate the predicted performance and the predicted utility loss, that are used by SMART to generate action schedules.

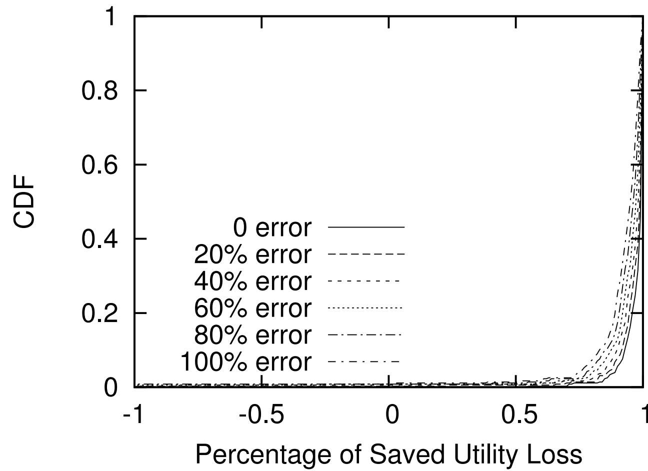
- The observed real system performance is simulated by introducing a random error to the predicted performance. The random error is generated using a normal distribution with the specified error rate as the standard deviation. For example, if the predicted latency is 10 ms and the random error generated is 0.2, the observed latency is simulated as 12 ms (12=10*(1+0.2)). The observed utility loss is then calculated based on it.

This design allows us to control the model error rates and measure their impact on SMART's schedule. We vary the error rate from 0 to 100%. Figure 6.23 plots the accuracy of the predicted utility loss (see Equation 6.7), which reflects the degree of mismatch between the predicted and

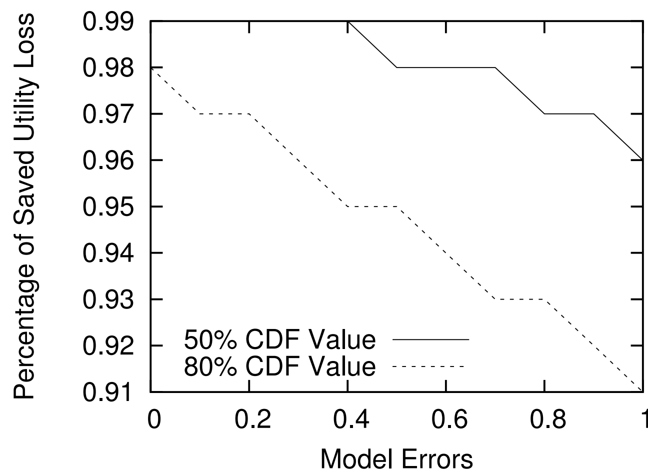
observed utility loss caused by performance prediction errors.

$$accuracy_of_the_predicted_utility_loss = \frac{predicted_utility_loss}{observed_utility_loss} \tag{6.7}$$

As shown in the figure, the accuracy drops very quickly with the growth of the model error. We obtain 70% accuracy for a 20% model error and 30% accuracy for a 60% model error.



(a) CDFs of the percentage of the saved utility loss



(b) Percentage of the saved utility loss

Figure 6.24. Impact of performance model errors

Figure 6.24(a) plots the CDF of the percentage of the saved utility loss for various model error rates. We can see that as the model error increases, the saved utility loss of SMART's action schedule

drops. But the reduction is small. To take a closer look at the impact of the model error rate, we plot the 50% and 80% values from the CDF curve as a function of the model error rate. This is shown in Figure 6.24(b). The figure shows that the model error rate has a minor impact on the percentage of the saved utility loss—less than 10%. The step-wise decreasing behavior is because that, when the model error rate grows, SMART will return the same action schedules if the error does not affect the preference order of the action options.

In summary, these results show that SMART's quality is affected by the performance model errors, but it is not very sensitive to them.

Test 3: With Time-Series Prediction Errors

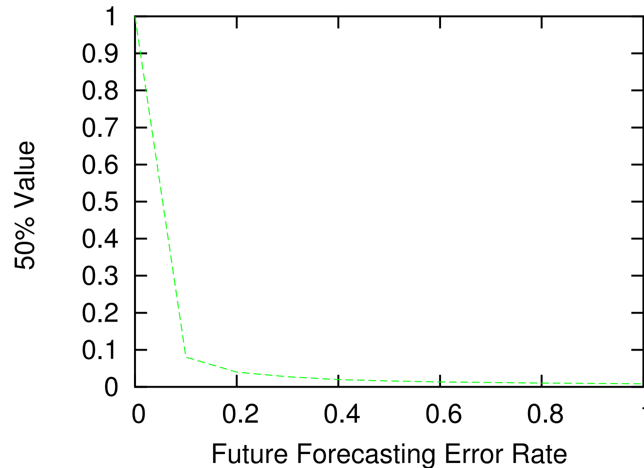


Figure 6.25. Impact of future forecasting errors on the accuracy of the predicted utility loss

We introduce the future forecasting error in a fashion similar to the experiment with the performance model errors. The workload demands forecasting is generated first and is used by SMART for decision making. The real workload demands are generated by scaling the predicted workload demands with a random factor. The random factor follows a normal distribution, with the future forecasting error as the standard deviation. The *predicted* utility loss is calculated based on the forecast demands and the *observed* utility loss is calculated based on the real workload demands. For this set of tests, we restrict the Action Advisor to operate in the *normal mode* because, otherwise,

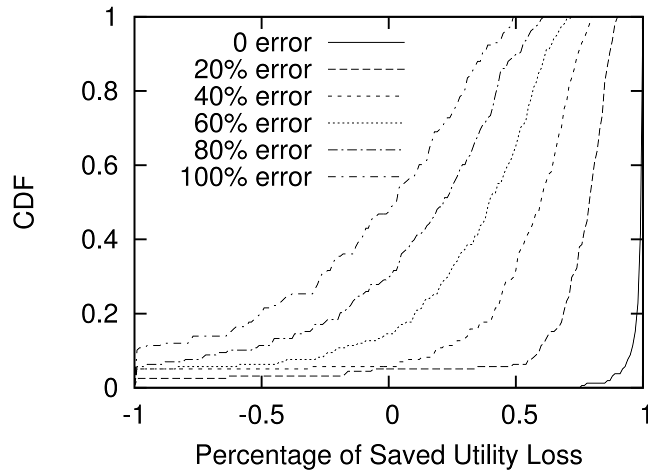
the Action Advisor will automatically switch to the *unexpected* mode and the future forecasting is ignored.

Figure 6.25 shows the accuracy of the predicted utility loss (see Equation 6.7). Compared to Figure 6.23, the prediction accuracy drops much faster. With a 10% future forecasting error, the predicted utility loss is only 8% of the observed utility loss. This is because, for most actions like throttling and migration, the performance values composed using the system models are used to compare against some boundary, such as SLO constraints. As long as the model errors will not make the action option flip from meeting the SLO to violating it, the impact is marginal. However, the future forecasting errors can change the preference of action options. For example, migration is preferred over throttling when there is a growing workload demands. As a result, the future forecasting errors have a bigger impact on the accuracy of the predicted utility loss.

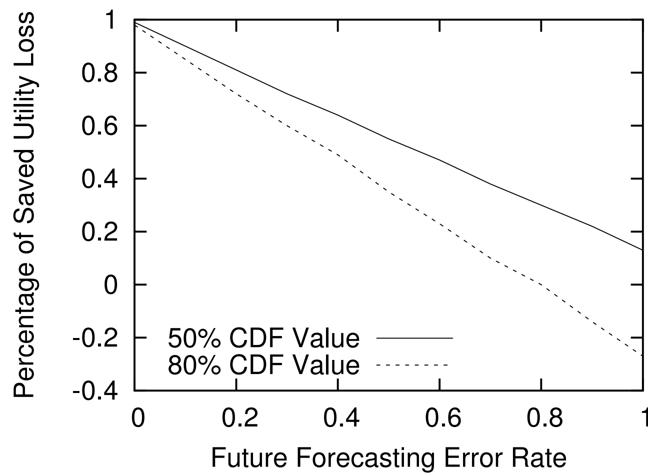
Figure 6.26(a) plots the CDF of the percentage of the saved utility loss after invoking SMART's action schedule. Compared to Figure 6.24(a), the percentage of the saved utility loss drops much faster. Figure 6.26(b) plots the 50% and 80% CDF values. The 50% CDF curve shows that, for about 50% of cases, with a 20% future forecasting error, SMART's action schedule can reduce the utility loss by up to 83%. But when the prediction errors grows to 60%, SMART can only eliminate less than 45% of utility loss. From the 80% CDF curve, we can see that, for about 20% of cases, SMART can only eliminate less than 15% of the utility loss when the future forecasting errors is 60%. When the errors grows, SMART may result in a system utility worse than the case of no action. These results confirm our design choice: when the difference between the future prediction and observed values are high, we should apply a defensive strategy.

6.4.7 Decision Overhead of SMART

Figure 6.27 plots the decision overhead of SMART. We ran the simulator on a Linux machine with Pentium 4 3.0 GHZ CPU and 512MB memories. We vary the number of workloads in the system and record the invocation and return time of SMART. The average decision time for the 500 scenarios are plotted in Figure 6.27(a). The figure shows an exponentially growing trend with the number of workloads in the system. The decision time with 100 workloads approaches to 14



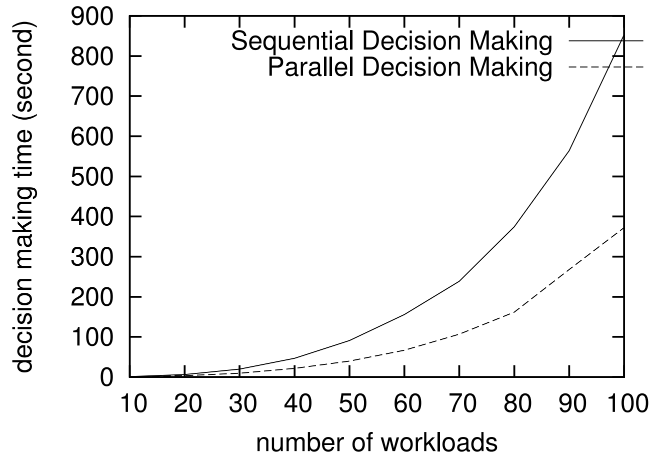
(a) CDFs of the percentage of the saved utility loss



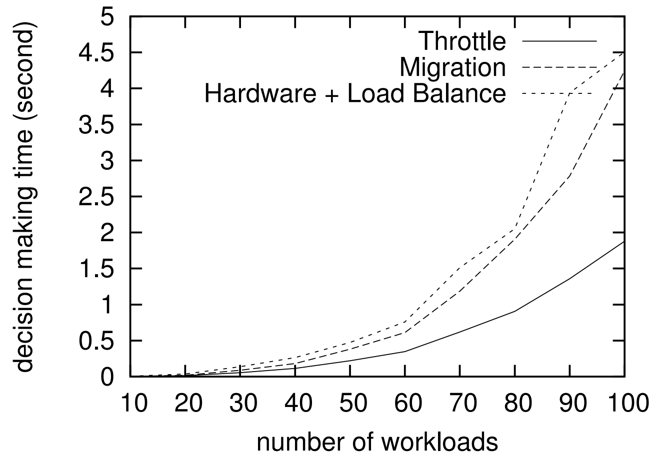
(b) Percentage of the saved utility loss

Figure 6.26. Impact of future forecasting errors

minutes. One reason for this long decision time is that the simulator makes decisions sequentially: it first makes the throttling decisions, then the migration decisions, and the provisioning decisions. In a real system, the Action Advisor can communicate with single action tools in parallel, thus, reduce the decision overhead. To estimate the overhead of parallel decision making, we measure the decision time for each individual action and use the the maximum of the three to approximate the decision time using parallel queries. The results are shown in Figure 6.27(a). With parallel queries,



(a) Decision time for the sequential and parallel decision making



(b) Decision time for individual action

Figure 6.27. Computational overhead of SMART

SMART takes six minutes to generate an action schedule with 100 workloads in the system. In addition, because for each system state, the Action Advisor needs to evaluate the system behavior for every action option, the decision overhead is also related to the number of future states. We measure the decision time with one system state for each type of actions and plot the results in Figure 6.27(b). The figure shows that for every system state, the decision for all actions can be made on the order of seconds.

In summary, we tested the SMART framework using both the GPFS prototype and the simulator. Our sanity check results show that SMART can account for various configuration parameters and

generate appropriate action schedules. Our feasibility test using the GPFS prototype verifies that SMART can improve the utility loss in the real system and the predicted utility values match well with the observed values. Our sensitivity tests show that with accurate system models and future prediction, for more than 90% senarios tested, SMART can elimiate more than 94% of utility loss. For the remaining 10%, it reduces the utility loss by 70%. In addition, our results show that SMART is more sensitive to the future forecasting errors than to the model errors. Our strategy of defensive action scheduling starts acting when the future prediction is inaccurate. The decision overhead analysis shows that SMART can generate an action plan on the order of minutes.

6.5 Conclusion

To reduce the total cost of ownership and adapt to system changes more responsively, the industrial trend is to introduce systems that can manage their storage more automatically. This thesis focuses on automating the choice of the corrective actions. Our framework SMART aims to generate an action schedule about *when* to invoke *what* actions in an integrated manner. Previous chapters illustrated the model-based decision making processes for two corrective actions, throttling and migration. In this chapter, we described the intelligent optimization method in SMART, which we call the Action Advisor. The Action Advisor considers a variety of information including the forecast system state, the estimated action cost-benefit effect and the business constraints. From these inputs, it generates a combination of corrective actions that can reduce the system utility loss. We design it to create action plans for different optimization windows as well as react to both expected and unexpected system changes. It uses a recursive greedy algorithm with lookback and lookforward to generate plans for expected system changes, and a defensive strategy based on the well-known ski-rental algorithm to handle unexpected cases. We have implemented SMART's prototype in a distributed file system, GPFS. Our experiments show that the system utility is improved as predicted. With accurate information, for the scenarios we tested, SMART's action decision can reduce the utility loss by 94% for more than 90% of cases and by 70% for the remaining 10% scenarios. Most of the utility loss experienced by SMART is due to transient operations like data movement, or limited resource such as before hardware arrives. We also evaluate the quality of SMART's decision with

input errors. We find that SMART's decision is more sensitive to inaccurate future prediction than to system model errors. These results suggest us to apply SMART more carefully when we have limited knowledge about the system or the workloads. SMART's decision making time grows exponentially with the number of workloads in the system. In our testbed, with less than 100 workloads, the decision time is on the order of minutes.

Chapter 7

Conclusions and Future Work

In this chapter, we summarize our contributions in Section 7.1. We then finish this thesis with a discussion of directions in future work in Section 7.2.

7.1 Thesis Summary

In this thesis, we focus on automating the *Analyze* part in the *Observe-Analyze-Act* process of storage system management. We tackle the problem of generating the corrective plan leading to maximum system utility automatically and responsively. We face four fundamental challenges: how to react rapidly, how to invoke action proactively, how to compare alternative plans fairly, and how to handle unexpected system changes. To generate an action plan rapidly, we construct system models to describe the system behavior and apply constraint optimization techniques to explore the search space quickly. To enable proactive action invocation, we predict the future workload demands using time-series analysis and perform risk management to balance the benefit and risk of an action plan. To compare corrective actions without bias, we utilize the concepts of system utility and the optimization window, and look for actions that can maximize the system utility for a given optimization window. To handle unexpected system changes, we borrow the concept of the well-known ski-rental algorithm and design a defensive strategy to make action decisions when we have limited knowledge about the system and the workloads.

In this thesis, we develop a new framework, SMART, which is a model-based action advisor. It generates action plans consisting of *what* corrective actions to invoke, *when* to invoke and *how* to invoke. It takes into account the future workload demands, component and workload models, cost and benefit analysis of action options and business constraints. We develop the following specific tools within the more general framework.

- **CHAMELEON: An Automatic Throttling Decision Tool**. Working as a single action tool for throttling in the SMART framework, CHAMELEON automatically identifies which workloads should be throttled and to what degree. It takes the targeting system state and system models as input and generates the throttling options. The system state describes the workload characteristics, component ability and the application-to-resource mapping. The system models are used to predict the system behavior for a given throttling decision, which specifies the token issue rates for all workloads. Constraint optimization techniques are then applied to scan the candidate space for the optimal settings. In addition, to accommodate model errors and workload fluctuation, it uses a feedback-loop to control the throttling execution. It also defines simple heuristics as the fall-back strategy when the system knowledge is insufficient. To evaluate the efficiency of CHAMELEON, we replay traces from production environments in a real storage system. Our experimental results show that CHAMELEON makes accurate decisions for all the scenarios examined. It always makes the optimal throttling decisions based on the available knowledge. With our system and workloads we imposed, CHAMELEON can analyze and correct the SLO violations using the feedback loop in 3-14 minutes—which compares very favorable with the time a human administrator would have needed. CHAMELEON demonstrates that model-based approaches can explore the design space and select an optimal setting very quickly.

- **SMARTMIG: A Proactive Risk Modulated Migration Tool**. SMARTMIG acts as the single action tool for migration in the SMART framework. It can find the optimal migration plan, including *what* data to migrate, *where* to place them, *when* to migrate and *how* fast to migrate. It combines system models, time-series analysis and constraint optimization techniques to find the plan leading to the maximal system utility for a given optimization window. To reduce the complexity, it breaks the decision process into three phases: optimization, planning and risk

management. The optimization phase chooses the migration candidates (*what*) and targets (*where*) options. The planning phase determines the detailed migration plan in terms of the migration speed (*how*) and the migration starting time (*when*). The risk management accounts for the future uncertainty and action invocation overhead. Compared with previous solutions, SMARTMIG's migration plan considers both current and future system states, as well as the risk of migration options. We evaluate SMARTMIG in a simulator. Our experimental results show that SMARTMIG can adapt to the changes of configuration parameters and generate an optimal migration plan. For our testing scenarios, SMARTMIG successfully reduces the system utility loss by 80% for more than 83% of cases. Our sensitivity test shows SMARTMIG's high tolerance against model errors. In summary, SMARTMIG demonstrates that, by combining system models, time-series analysis and constraint optimization techniques, we can discover the options for complicated actions rapidly.

- **SMART: An Integrated Multi-Action Advisor**. The intelligence behind SMART is its Action Advisor. It is responsible for generating the action plan for both expected and unexpected system changes. Depending on the accuracy of the future prediction, the Action Advisor operates in either the *normal* mode or the *unexpected* mode. In the former, it applies a recursive greedy algorithm to find the set of actions that can improve the system utility. It also performs the risk management to balance the benefit an risk of an action option. In the unexpected mode, it applies a defensive strategy based on the well-known ski-rental algorithm. The key idea is to always invoke the action with the lowest cost. The Action Advisor can transition between two modes when the system changes so much that the future prediction is no longer correct, or when it collects enough new observations on the system states to have an accurate future prediction. We implemented SMART's prototype in a distributed file system, GPFS. Our experiments show that SMART can generate action plan that is optimal for the given system configurations. For the scenarios tested, SMART can reduce the utility loss by 94% for 90% cases and by 70% for the rest. Our sensitivity test shows that SMART requires accurate future prediction to work efficiently, but it is less sensitive to system model errors. These results suggest us to apply SMART more carefully when we have limited knowledge about the system or workloads.

In summary, we develop a framework, SMART, to automate the analysis of corrective action plans for storage system management. Our study shows that model-based approaches enable us to quickly scan the candidate space and find the optimal action plan. At the same time, it is critical to consider realistic factors such as model and future prediction errors, action invocation overhead, business constraints and unexpected system changes in the design. In the next section, we will discuss several possible follow-on works for our thesis based on lessons learned.

7.2 Future Work

To fully automate the "analyze" part of the OAA loop, several challenges remain to be solved. Here are several possible follow-on work for our thesis.

- **Improving the decision making strategy**. The goal is to make our algorithm less sensitive to input errors and generate the plan faster. Our discussion in Chapter 5 and Chapter 6, shows that the input errors can affect the quality of our decision. To reduce the impact of input errors, on the one hand, we need to deploy better modeling technique as well as the future forecasting techniques to improve the accuracy of input information. On the other hand, we need to improve the decision making strategy to make it more robust to input errors. Currently, the Action Advisor operates in a binary mode, normal or unexpected. As a result, our decision algorithm either trusts the input or ignore them completely. One possible direction is to increase the number of modes and adjust the aggressiveness of the decision strategy by controlling the type of actions considered according to the accuracy of the input information.
- **Reducing the decision overhead**. The current design has a decision overhead in the range of six minutes with 100 workloads in the system. Although it is faster than the time required by a human administrator, we need to reduce the overhead further. On the one hand, we need to continuously look for better optimization techniques. The simulated annealing algorithm used by throttling and the greedy approximation algorithm for migration are both time consuming operations. We need to exploring other constraint optimization techniques and hypothesis to reduce the decision overhead. On the other hand, we can reduce the decision overhead by developing mechanisms to adjust the frequency of SMART invocation. Querying single action

tools for each system state contributes the majority of the decision overhead. One possible direction to reduce the overhead is to discovery the opportunities to reuse SMART's decision when possible. For example, if a system state is similar to a previously seen one, SMART can skip querying the single action tools.

- **Integrating with existing tools**. SMART is designed as a general framework. We intentionally separate single action tools from the action advisor such that the action tools can be upgraded easily in the future. The separation also allows SMART to work with single action tools other than CHAMELEON and SMARTMIG. In principle, SMART can leverage any existing tools. To do so, we need to design appropriate interfaces between the Action Advisor and single action tools. One direction is to create a standardized API for all single action tools and the Action Advisor. The ongoing SMI-S [79] (Storage Management Initiative) is one of the existing standardization efforts for services such as migration and hardware provisioning. Our efforts can become part of it.

In the long run, the following directions are interesting and important for further investigation.

- **Automatic problem determination**. SMART is invoked when the storage system experiences problems such as SLO violations and utility loss. In this thesis, we assume that the root cause of the problem is known and the bottleneck component has been located. By bottleneck component, we refer to the devices, such as storage controller, disks or switches, that are overloaded. Under this assumption, SMART considers action candidates that can solve the problem. In a distributed storage system with a complicated infrastructure, determining the root cause of the problem and locating the bottleneck component is challenging. Existing storage management softwares, such as TPC [26] and ControlCenter [25] collects extensive monitoring information, which can be used to diagnose the problem. However, it remains a challenge how to extract useful informations from the huge amount of data, and how to correlate the observations and determine the root cause of the problem automatically.

- **Benchmarking the automatic management software**. Many research prototypes and commercial products have been developed to automate the storage system management. They often have different objectives, adopt different strategies and are evaluated in different systems.

Designing a benchmark tool that can quantify the performance of alternative management softwares and find the best strategy for a given system setup is an interesting and important direction. In addition, existing tools often evaluate the effect of management decisions in single dimension, such as performance, availability or security. In many cases, a management decision will alter the system behavior in multiple dimensions. For example, a replication decision affects the system performance, as well as the availability. A benchmark tool that can uniformly evaluate and compare the complete impacts of management decisions is desirable.

- **Constructing utility functions**. SMART uses utility functions to capture the users' degree of satisfaction. It assumes that utility functions can be defined based on SLOs or other pricing information. In an environment like corporation data center, such information may not always be available. The challenge is how to figure out what the user wants. One possible solution is to define the utility function using a complaint-based approach. The idea is to change the workload's performance by varying the resource allocated to it and adjust the utility function based on users response. For example, reducing the utility value for a given performance if the user complaints.

Bibliography

[1] Beyond Linux from scratch. http://www.linuxfromscratch.org/blfs/view/svn/general/sysstat.html.

[2] GLPK (GNU) linear programming kit. http://www.gnu.org/software/glpk/glpk.html.

[3] The network simulator ns-2. http://www.isi.edu/nsnam/ns/.

[4] NIST NET - a Linux-based network emulation tool. *Computer Communication Review*, June 2003.

[5] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001.

[6] Eric Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-4, HP Laboratories, July 2001.

[7] Eric Anderson, Joseph Hall, Jason D. Hartline, Michael Hobbs, Anna R. Karlin, Jared Saia, Ram Swaminathan, and John Wilkes. An experimental study of data migration algorithms. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*, pages 145–158, 2001.

[8] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of Conference on File and Storage Technologies (FAST'02)*, pages 175–188, Monterey, CA, January 2002.

[9] Eric Anderson, Mahesh Kallahalla, Susan Spence, Ram Swaminathan, and Qian Wang. Ergastulum: quickly finding near-optimal storage system designs. Technical Report HPL-SSP-2001-05, HP Laboratories, June 2002.

[10] David A.Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD'88)*, pages 109–116, Chicago, IL, June 1988.

[11] Michael E. Azoff and Eitan M. Azoff. *Neural Network Time Series Forecasting of Financial Markets*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[12] Elizabeth Borowsky, Richard Golding, Patricia Jacobson, Arif Merchant, Louis Schreier, Mirjana Spasojevic, and John Wilkes. Capacity planning with phased workloads. In *Proceedings of the first International Workshop on Software and Performance (WOSP'98')*, pages 199–207, October 1998.

[13] Eric A. Brewer. High-level optimization via automated statistical modeling. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'95')*, pages 80–91, Santa Barbara, CA, USA, July 1995. ACM Press.

[14] Maria Calzarossa and Giuseppe Serazzi. Workload characterization: a survey. *Proceedings of the IEEE*, 81(8):1136–1150, 1993.

[15] Scott D. Carson and Sanjeev Setia. Analysis of the periodic update write policy for disk cache. *IEEE Transactions on Software Engineering*, 18(1):44–54, January 1992.

[16] V. Cerny. Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. *journal of optimization theory and applications*, 45:41–51, 1985.

[17] David Chambliss, Guillermo A. Alvarez, Prashant Pandey, Divyesh Jadav, Jian Xu, Ram Menon, and Tzongyu P. Lee. Performance virtualization for large-scale storage systems. In *Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS'03)*, pages 109–118, October 2003.

[18] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. *SIGMETRICS Performance Evaluation Review*, 31(1):300–301, 2003.

[19] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 103–116, October 2001.

[20] Chandra Chekuri and Sanjeev Khanna. A PTAS for the multiple knapsack problem. In *Proceedings of the eleventh annual ACM-SIAM Symposium on Discrete Algorithms*, pages 213–222, January 2000.

[21] Mike Chen, Anthony Accardi, Emre Kiciman, Dave Patterson, Armando Fox, and Eric Brewer. path-based failure and evolution management. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI'04)*, pages 309–322, San Francisco, CA, USA, March 2004.

[22] Mike Chen, Alice Zheng, Jim Lloyd, Michael Jordan, and Eric Brewer. Failure diagnosis using decision trees. In *Proceedings of the First International Conference on Autonomic Computer (ICAC'04)*, pages 36–43, Los Alamitos, CA, USA, May 2004.

[23] Thomas Clark and Tom Clark. *IP SANS: An Introduction to iSCSI, iFCP, and FCIP Protocols for Storage Area Networks*. Addison-Wesley Professional, 2001.

[24] George P. Copeland, William Alexander, Ellen E. Boughter, and Tom W.Keller. Data placement in Budda. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD'88)*, pages 99–108, June 1988.

[25] EMC corporation. EMC ControlCenter family of storage resource management (SRM). http://www.emc.com/products/storage_management/controlcenter.jsp.

[26] IBM Corporation. IBM TotalStorage. http://www-1.ibm.com/servers/storage.

[27] Storage Performance Council. SPC I/O traces. http://www.storageperformance.org/.

[28] Koustuv Dasgupta, Sugata Ghosal, Rohit Jain, Upendra Sharma, and Akshat Verma. QoSMig: Adaptive rate-controlled migration of bulk data in storage systems. In *Proceedings of IEEE International Conference on Data Engineering 2005 (ICDE'05)*, pages 816–827, April 2005.

[29] Yixin Diao, Joseph L. Hellerstein, and Sujay Parekh. Self-managing systems: A control theory foundation. In *Proceedings of 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 441–448, April 2005.

[30] Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin, and Amin M. Vahdat. Model-based resource provisioning in a web service utility. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'03)*, March 2003.

[31] Paul Ferguson and Geoff Huston. *Quality of Service: Delivering QoS on the Internet and in Corporate Networks*. John Wiley & Sons, Inc., January 1998.

[32] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. Self-* Storage: Brick-based storage with automated administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, August 2003.

[33] Gregory R. Ganger, Bruce Worthington, and Yale Patt. The DiskSim simulation environment version 1.0 reference manual. Technical Report CSE-TR-358-98, University of Michigan, February 1998.

[34] Joseph S. Glider, Carlos F. Fuente, and William J. Scales. The software architecture of a SAN storage control system. *IBM System Journal*, 42(2):232–249, 2003.

[35] Fred W. Glover, Manuel Laguna, and Rafael Marti. *Scatter search, Advances in evolutionary computing: theory and applications*. Springer-Verlag New York, Inc., New York, NY, 2003.

[36] Gartner Group. Total Cost of Storage Ownership—A User-oriented Approach. *Research note, Gartner Group*, February 2000.

[37] Michael Hawkins. Total Cost of Ownership: The driver for IT infrastructure management. http://www.informit.com, June 2001.

[38] Francisco Hidrobo and Toni Cortes. Towards a zero-knowledge model for disk drives. In *Proceedings of the Fifth Annual International Workshop on Action Middleware Services (AMS'03)*, pages 122–130, June 2003.

[39] Andy Hospodor. Mechanical access time calculation. *Advances in Information Storage Systems*, 6:313–336, 1995.

[40] Lan Huang, Gang Peng, and Tzi cker Chiueh. Multi-dimensional storage virtualization. *SIGMETRICS Performance Evaluation Review*, 32(1):14–24, 2004.

[41] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the Knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, October 1975.

[42] IETF Policy Framework Working Group. IETF Policy Charter. http://www.ietf.org/html.charters/policy-charter.html.

[43] Sun Microsystems Inc. NFS: Network File System Protocol Specification, 1989.

[44] Raj Jain. *The Art of Computer System Performance Analysis*. John Wiley & Sons, Inc., February 2001.

[45] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. *SIGMETRICS Performance Evaluation Review*, 32(1):37–48, 2004.

[46] John F. Nash Jr. The bargaining problem. *Econometrica*, 18:155, 1950.

[47] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance isolation and differentiation for storage systems. In *Proceedings of the twelve th International Workshop on Quality of Service*, June 2004.

[48] Richard M. Karp. On-line algorithms versus off-line algorithms: How much is it worth to know the future? *Algorithms, Software, Architecture, Information Processing 92*, 1:416–429, 1992.

[49] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, August 2005.

[50] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[51] Edward K. Lee and Randy H. Katz. An analytic performance model of disk arrays. *SIGMETRICS Performance Evaluation Review*, 21(1):98–109, 1993.

[52] Thomas J. Linsmeier and Neil D. Pearson. Risk measurement: An introduction to value at risk. In *Finance 960906, Economics Working Paper Archive EconWPA*, September 1996.

[53] Wei-Yin Loh. Regression trees with unbiased variable selection and interaction detection. *Statistica Sinica*, 12:361–386, 2002.

[54] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. Aqueduct: online data migration with performance guarantees. In *Proceedings of Conference on File and Storage Technologies (FAST'02)*, pages 175–188, January 2002.

[55] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: virtual storage devices with performance guarantees. In *Proceedings of the second Conference on File and Storage Technologies (FAST'03)*, pages 131–144, San Francisco, CA, April 2003.

[56] Geoffrey McLachlan and David Peel. *Finite Mixture Models*. Wiley-Interscience, October 2000.

[57] Daniel A. Menasce, Daniel Barbara, and Ronald Dodge. Preserving QoS of e-commerce sites through self-tuning: a performance model approach. In *Proceedings of the 3rd ACM Conference on Electronic Commerce (EC'01)*, pages 224–234, Tampa, Florida, USA, October 2001.

[58] Arif Merchant and Philip S. Yu. An analytical model of reconstruction time in mirrored disks. *Performance Evaluation*, 20(1-3):115–129, May 1994.

[59] Arif Merchant and Philip S. Yu. Analytic modeling of clustered RAID with mapping based on nearly random permutation. *IEEE Transactions on Computers*, 45(3):367–373, March 1996.

[60] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[61] Robert J.T. Morris and Brian J. Truskowski. The evolution of storage systems. *IBM Systems Journal*, 42:205–217, 2003.

[62] Walter C. Oney. Queueing analysis of the scan policy for moving-head disks. *Journal of the ACM*, 22(3):397–412, July 1975.

[63] Kihong Park, Gitae Kim, and Mark E. Crovella. On the effect of traffic self-similarity on network performance. In *Proceedings of SPIE International Conference on Performance and Control of Network Systems*, Dollas, TX, November 1997.

[64] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Transaction on Networking*, 3(3):226–244, June 1995.

[65] David Pisinger. A minimal algorithm for the 0-1 Knapsack problem. *Journal of Operations Research*, 45:758–767, 1997.

[66] Kristal Pollack and Sandeep Uttamchandani. Genesis: A scalable self-evolving root-cause analysis framework for storage systems. In *Proceedings of the 26th International conference on distributed computing systems (ICDCS'06)*, page 33, Los Alamitos, CA, USA, July 2006.

[67] John G. Proakis and Dimitris G. Mandalokis. *Digital Signal Processing Principles, Algorithms and Applications*. Prentice Hall, 1996.

[68] Chris Ruemmler and John Wilkes. A trace-driven analysis of disk working set sizes. Technical Report HPL-OSR-93-23, HP Laboratories, Palo Alto, CA, USA, April 1993.

[69] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[70] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[71] Peter Scheduermann, Gerhard Weikum, and Peter Zabback. Adaptive load balancing in disk arrays. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO'93)*, pages 345–360, October 1993.

[72] Jiri Schindler and Gregory Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, December 1999.

[73] Frank Schmuck and Roger Haskin. GPFS: A shared disk file system for large computing clusters. In *Proceedings of Conference on File and Storage Technologies (FAST'02)*, pages 231–244, Monterey, CA, January 2002.

[74] Elizabeth Shriver, Arif Merchant, and John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. In *Proceedings of the 1998 ACM SIGMETRICS joint International Conference on Measurement and Modeling of Computer Systems*, pages 182–191, Madison, Wisconsin, USA, June 1998. ACM Press.

[75] Wolfgang Singer. NAS and iSCSI technology overview. http://www.snia.org/education/tutorials/2006/spring/storage/NAS_and_iSCSI_Technology_Overview.pdf.

[76] BMC Software. PATROL for storage networking. http://www.bmc.com/products/proddocview/0,2832,19052_19429_3786404_9689,00.html.

[77] Jon A. Solworth and Cyril U. Orji. Write-only disk caches. *SIGMOD Record*, 19(2):123–132, 1990.

[78] Charles J. Stone. *A Course in Probability and Statistics*. Duxbury Press, 1996.

[79] Storage Networking Industry Association. SMI Specification version 1.0. http://www.snia.org, 2003.

[80] John D. Strunk and Gregory R. Ganger. A human organization analogy for Self-* systems. In *Proceedings of the first workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[81] David G. Sullivan and Margo Seltzer. Isolation with flexibility: A resource management framework for central servers. In *Proceedings of 2000 USENIX Technical Conference (USENIX'00)*, pages 337–350, San Diego, CA, June 2000.

[82] David G. Sullivan, Margo I. Seltzer, and Avi Pfeffer. Using probabilistic reasoning to automate software tuning. *SIGMETRICS Performance Evaluation Review*, 32(1):404–405, 2004.

[83] Toby J. Teorey and Tad B. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, 15(3):177–184, 1972.

[84] Nancy Tran and Daniel A. Reed. ARIMA time series modeling and forecasting for adaptive I/O prefetching. In *Proceedings of the 15th International Conference on Supercomputing*, pages 473–485, Sorrento, Italy, 2001. ACM Press.

[85] Sandeep Uttamchandani, Kaladhar Voruganti, Sudrashan Srinivasan, John Palmer, and David Pease. Polus: Growing storage QoS management beyond a 4-year old kid. In *Proceedings of 3rd File and Storage Technologies (FAST'04)*, pages 31–44, San Francisco, CA, March 2004.

[86] Mustafa Uysal, Guillermo A. Alvarez, and Arif Merchant. A modular, analytical throughput model for modern disk arrays. In *Proceedings of the ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'01)*, pages 183–192, August 2001.

[87] Dinesh C. Verma. Simplifying network administration using policy based management. *IEEE Network Magazine*, 16(2):20–26, March 2002.

[88] Dinesh C. Verma and Seraphin Calo. Goal Oriented Policy Determination. In *Proceedings of the 1st Workshop on Algorithms and Architectures for Self-Managing System*, pages 1–6, June 2003.

[89] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with CART models. *SIGMETRICS Performance Evaluation Review*, 32(1):412–413, 2004.

[90] Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. Self-tuning database technology and information services: From wishful thinking to viable engineering. In *Proceedings of the 28th Conference on Very Large Data Base Conference (VLDB'02)*, August 2002.

[91] John Wikes. Data services - from data to containers. *Keynote address at File and Storage Technologies (FAST'03)*, March 2003.

[92] Neil C. Wilhelm. An anomaly in disk scheduling: a comparison of FCFS and SSTF seek scheduling using an empirical model for disk accesses. *Communication of the ACM*, 19(1):13–17, 1976.

[93] John Wilkes. The pantheon storage-system simulator. Technical Report HPL-SSP-95-14, HP Laboratories, December 1995.

[94] Philip S. Yu and Arif Merchant. Analytic modeling and comparisons of striping strategies for replicated disk arrays. *IEEE Transactions on Computers*, 44(3):419–433, 1995.