POLUS: A SELF-EVOLVING MODEL-BASED APPROACH FOR AUTOMATING THE OBSERVE-ANALYZE-ACT LOOP

BY

SANDEEP UTTAMCHANDANI

B.E., University of Pune, 2000
M.S., University of Illinois at Urbana-Champaign, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

Systems today are managed by human administrators that are required to continuously *observe* the system, *analyze* its behavior, and *activate* corrective actions (generally referred to as the Observe-Analyze-Act loop). Automating the OAA loop within real-world systems is a non-trivial problem, but the growing economic incentive associated with making systems self-managing, and significant increase in the computation bandwidth have made OAA automation a promising area of research. The existing choices for OAA automation fall in the taxonomy of policy-based, feedback-based, empirical or learning-based, and model-based – none of the available solutions have achieved much traction in production systems, and automate only trivial management scenarios.

This thesis proposes POLUS: a methodology for OAA automation using a *a model-based approach with integrated learning and feedback*. Polus uses models of system behavior for deciding the corrective action to be invoked – it continuously refines models using monitor data, exhaustively searches for an optimal corrective action using constrained optimization, and executes the selected action using a variably aggressive feedback loop. The core architecture of POLUS closely resembles that of an Expert System: A *Knowledge-base* of models for components, workloads, actions, and a *Reasoning engine* that selects and executes a "feasible" action at run-time.

The objective of this thesis is to describe the POLUS methodology for OAA automation. The details of the methodology consist of: Representation of domain-specific details as models; creation and evolution of these models in an automated fashion; decision-making for the corrective action(s) to be invoked at run-time; handling divergent system behavior during action execution. POLUS is the first-of-a-kind in using a model-based approach for OAA automation; by applying the following operational principles, POLUS addresses challenges related to model inaccuracies in production systems, and the computational complexity of decision-making: 1) Models don't need to be perfectly accurate – they only need to be accurate enough to maintain the relative ordering during action selection; 2) The objective of action selection is not to find the most optimal one, but rather to avoid the worst ones; 3) Creation of models is not a one-time activity – it is a continuous process over the lifetime of the system.

The POLUS approach was built and evaluated as an OAA framework for a production storage system (having a limited set of corrective actions). The prototype implementation (referred to as CHAMELEON) is a resource arbitrator

that manages assignment of available storage resources to the host workloads. This mapping must ensure that a minimal number of workloads fail to meet their behavior goals (a *QoS violation*). CHAMELEON optimizes the overall system utility by automated invocation of the throttle and unthrottle corrective actions. In our experiments, CHAMELEON identified, analyzed, and corrected performance violations in 3-14 minutes which compares very favorably with the time a human administrator would have needed. Further, the self-evolving aspect of CHAMELEON facilitated deployment for large-scale storage systems that service variable workloads on an ever-changing mix of device types.

In summary, this thesis is the starting point for applying model-based OAA automation to production systems. We have demonstrated the feasibility of our approach in the context of action sets that have a relatively low resource overhead for invocation, and whose effects can be easily reversed. As future work, there are several interesting challenges in applying POLUS to systems with a wider cost-benefit spectrum of corrective actions; we enumerate these research challenges and their significance within the context of the existing design details of POLUS.

Dedicated to my parents Renu and Madhu,
my advisors Carolyn and Gul,
and my guru *John Palmer.*

Acknowledgments

I would like to thank my advisor Prof. Gul Agha for his intellectual guidance and constant encouragement, and for fostering a research environment with great intellectual freedom. He has been an excellent role model and has had a profound impact on not only my outlook towards research but also towards life in general.

I would also like to thank my committee members, Dr. Jai Menon, Prof. Dan Roth, Dr. Carolyn Talcott, and Prof. Yuanyuan Zhou for their invaluable guidance and feedback on my thesis research. Major portions of this thesis were motivated by frequent discussions with them, and some were completely worked out in collaboration with them. I thank Dr. Carolyn Talcott for giving me the much needed guidance, encouragement, and her valuable time during the early stages on my thesis in 2001. Also a special thanks to Dr. Jai Menon for finding brainstorming time in his busy schedule as an *IBM Fellow*, VP and CTO for storage and software at IBM.

I am grateful to colleagues at IBM Almaden Research Center for providing a very stimulating intellectual environment, inspiration, and cooperation during the four years of my PhD. I would like to thank Guillermo Alvarez, Ying Chen, Bala Iyer, Dharmendra Modha, John Palmer, David Pease, Archana Samtani, Gauri Shah, Kaladhar Voruganti, Honesty Young, Omer Zaki. A special thanks to John Palmer for actively brainstorming ideas on a regular basis, giving me his valuable feedback and expert insights. Also, Kaladhar Voruganti has been a wonderful friend, insightful guide, expert collaborator and motivating philosopher – he kept me going during my low-energy modes, and was available for countless technical discussions as well as advice. I am extremely grateful to Li Yin, my friend, collaborator and summer intern at IBM – she has been a wonderful person to collaborate with a large number of ideas.

There are several colleagues and friend at UIUC that were an important part of this thesis. A special thanks to Prasannaa Thati for being a close friend and extremely knowledgeable mentor. I would also like to thank Chetan Shivshankar for helping me getting settled during my frequent visits to UIUC. The Open Systems Lab members have provided a very cooperative environment, and I would like to thank Po-Hao Chang, Nadeem Jamali, Youngmin Kwon, Koushik Sen, Sudarshan Srinivasan, Predrag Tosic,Abhay Vardhan, and Reza Ziaei. I would also like to thank Shripad Thite for his help and cooperation.

Last but not the least, I am very grateful to my parents who have always

given me unconditional love and emotional support. No words can express my gratitude for their love and confidence in me.

Table of Contents

List of Tables

List of Figures

1 Introduction

System administration today is human-centric; administrators continuously monitor the system, analyze its behavior, and take corrective actions to ensure convergence towards certain threshold values for performance, availability, security (commonly referred to as *Service Level Objectives* or SLOs). This activity is referred to as the *observe-analyze-act* (OAA) loop, and required 24×7 to handle variations in workload characteristics, addition of new workloads, change in application priorities, and exception events such as component failures, load surges. Self-management has become a key value differentiator for hardware and software vendors [53]. There are several trends that make self-management a necessity: Growing number of users, protocols, administrative policies; increasing dependence on highly skilled administrators for tuning systems; increasing business impact of exception events such as component failures or load surges.

The OAA loop manifests itself within multiple operational layers of the production system. For example, in storage virtualization engines [47], the OAA loop continuously *tunes* and/or *modifies* the mapping of application-data to the available storage resources – tuning involves short-term actions such as prefetching, throttling, cache-allocation, while modification involves long-term actions such as migration, replication. Similarly, within databases (i.e. in the application layer), the OAA loop optimizes the query execution by either tuning low-cost knobs such as the bufferpool size or invoking high-cost actions such as creation of additional indexes [31]. These manifestations of the OAA loop have a different syntax but similar semantic structure:

- Observe: Extract domain-specific details from the monitoring data, flag abnormal behavior, and trigger the selection of corrective actions.
- Analyze: Select a "feasible" action based on the *cost-benefit* effect of the action in the current state, the *need* for correction in current and lookahead system states, and the spare resources available for invoking the action (i.e. the amount system can *afford*).
- Act: Invoke the selected action(s) incrementally, handle unexpected action invocation effects, and feedback to refine decision-making.

The notion of automated system management is not a new one. Expert systems have been used to automate various human-intensive tasks such as medical diagnosis [25], system configuration [21], etc. The idea of self-tuning software

was proposed by [12, 16], but has only become important in the last few years with the growing system complexity. In 1999, Jim Gray in his Turing award speech defined self-management as one of the "Dozen IT research goals" [50]. Currently, there are multiple self-management projects both within academia as well as industrial research labs: Autonomic Computing at IBM [34], Self-* storage at CMU [44], Recovery Oriented Computing (ROC) at Berkeley [82], AutoAdmin at Microsoft [31].

Automation of the OAA loop is a non-trivial problem; it is analogous to a chess-playing program [79] that **observes** the positions on the board, **analyzes** the best possible strategy based on current as well as lookahead states, and **executes** the individual moves ensuring they have an intended effect. But automation of the OAA loop is even more complex as it has the added complexity of: 1) formalisms for the corrective actions have a certain degree of non-determinism (since it is practically impossible to capture all the possible system and environment parameters that influence the action) – *in chess-playing, defining the effect of different moves is fairly straightforward*; 2) Accounting for the run-time changes in the system properties (e.g., a component failure will have a degraded performance) – *changes that won't happen in a chess game (e.g., the queen stops moving diagonally and is only allowed to move horizontal).*

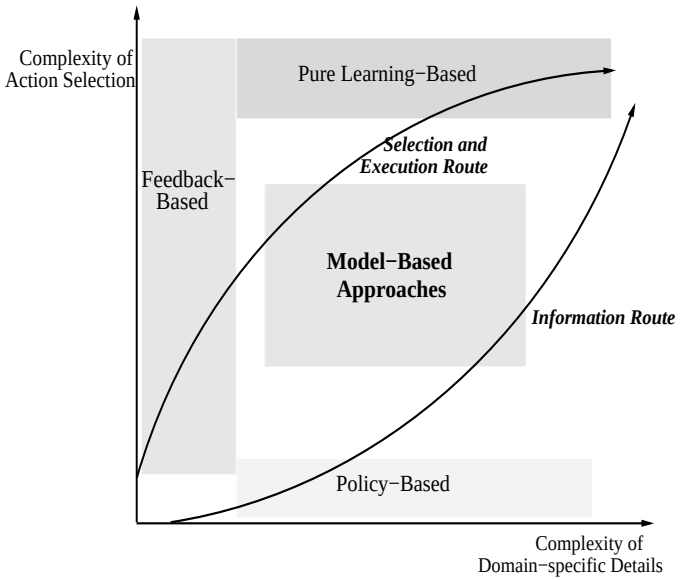


Figure 1.1: Taxonomy of existing approaches for OAA Automation

Existing approaches for automating the OAA loop fall in the taxonomy of: policy-based [36, 102], feedback-based [28], empirical or learning-based [69, 103], and model-based [6, 98, 99] approaches. Figure 1.1 classifies each of these approaches based on the complexity of specifying domain-specific details, and the mechanisms for selecting actions at run-time. Table 1.1 briefly describes the specification and action selection details for each of the approaches; it summa-

rizes the limitations of the approaches in the context of their deployment in production systems.

Approach	Domain-details	Action-selection	Limitations/Challenges
Policy-based	Consists of *rules* of the form of an ECA (Event-Condition-Action); the rules serve as "canned recipes" for system behavior in different states	Generally a sequential scan of the rules to determine the ones that are applicable for the current system-state	Suffers with the excessive complexity, and the brittleness of the rules to the underlying configuration
Feedback-based	Minimal knowledge; assumes little to no information about the system's response to corrective actions	Incrementally tries different permutations within the state-space; uses the instantaneous information as a basis for future actions	Works well for a small number of tunable parameters; infeasible for production systems with a large solution-space.
Empirical/ Learning-based	Starts with minimal or no domain knowledge; adds to the knowledge at run-time by maintaining a history of system behavior, state details, and the action invoked	Finds a system-state in the history of records that is closest to the current-state, and invokes a similar action.	Error-prone and infeasible in real-world systems with a large number of parameters
Model-based	Consist of mathematical functions for predicting different aspects of system behavior e.g. models for the response-time of the components, the load generated by the workloads	Selects the "optimal" corrective action at run-time by analyzing the cost-benefit of the available actions	The key challenges are representation of system details as models, mechanisms for creation and evolution of the models, formalisms for using these models at run-time to derive the answers, and accounting for inaccuracies of the models arising from missing parameters and unpredictable aspects of real-world systems

Table 1.1: Taxonomy for existing approaches for OAA automation

This thesis proposes a methodology for OAA automation using POLUS[1]: a model-based approach with the ability to refine models at run-time and vary the *aggressiveness* of action execution based on the model accuracy. The POLUS architecture consists of a *Knowledge-base* and a *Reasoning engine*. The Knowledge-base stores domain-specific details as models; the models are created and continuously refined by using a combination of designer defined feature-sets with Statistical Learning Techniques (SLT) [42]. The Reasoning engine selects and executes a "feasible" action; the action selection is formulated as a constrained optimization problem with administrator-defined objective functions such as optimize the overall system utility w.r.t the SLOs, etc.. The execution

[1]Name of the Greek God for Knowledge and Intelligence

of corrective actions is using a variably aggressive feedback loop that adapts with the accuracy of the models.

The outline for the rest of the chapter is as follows: We introduce the example of OAA automation within a production storage system, along-with the myths and realities of automation. We then describe a bird's eye-view of POLUS, key contributions of this work, and an outline for the rest of the thesis.

1.1 An Example of a OAA Automation in a Production Storage System

A typical consolidated storage system (as shown in Figure 1.2), has multiple clients storing and accessing petabytes' worth of data [80], serving the needs of various, independent, paying customers (e.g., a storage service provider) or divisions within the same organization (e.g., a corporate data center). Consolidation has proven to be an effective remedy for the low utilization that plague storage systems [53], for the expense of employing scarce system administrators, and for the dispersion of related data into unconnected islands of storage. In the *utility* model, each client is guaranteed a portion of the shared resources regardless of whether other clients over- or under-utilize their allocations.

A storage virtualization engine (such as SAN.FS [83], SAN Volume Controller [47]) simplifies the administrator's task of mapping the application-data to the available storage resources. At the time of deployment, an ill-formed application-to-resource mapping is derived based on limited information about the application's access characteristics and its requirements. At run-time, the system administrator is required to continuously monitor the system, analyze the current state, and possibly invoke corrective actions. This task of observe-analyze-act is required in a 24×7 fashion to handle variations in the workload characteristics, hot-spots in the utilization of system components, exception scenarios such as load surges or component failures. In the current state-of-art, a virtualization engine serves mainly as a monitoring dashboard with manual action invocation knobs for the administrator; ideally, the virtualization engine should automate the OAA loop, and invoke corrective actions that tune and/or modify the application-to-resource mapping.

There is a spectrum of corrective actions varying in their cost-benefit effects. One extreme of the spectrum has short-term tuning actions (such as throttling, prefetching, cache re-allocation) that are analogous to *pain-killers* that provide immediate relief, but do not cure the root-cause of the disease. On the other hand, long-term modification actions (such as migration, replication) are analogous to *antibiotics* that take some time to come into effect, but solve the root-cause of the problem by changing the data-to-resource mapping. The focus of this thesis is automated invocation of a single action i.e., analyzing the cost-benefit of action invocation, and deciding the values of the action invoca-

tion parameters; formalisms for selecting between multiple actions is beyond the scope of this work.

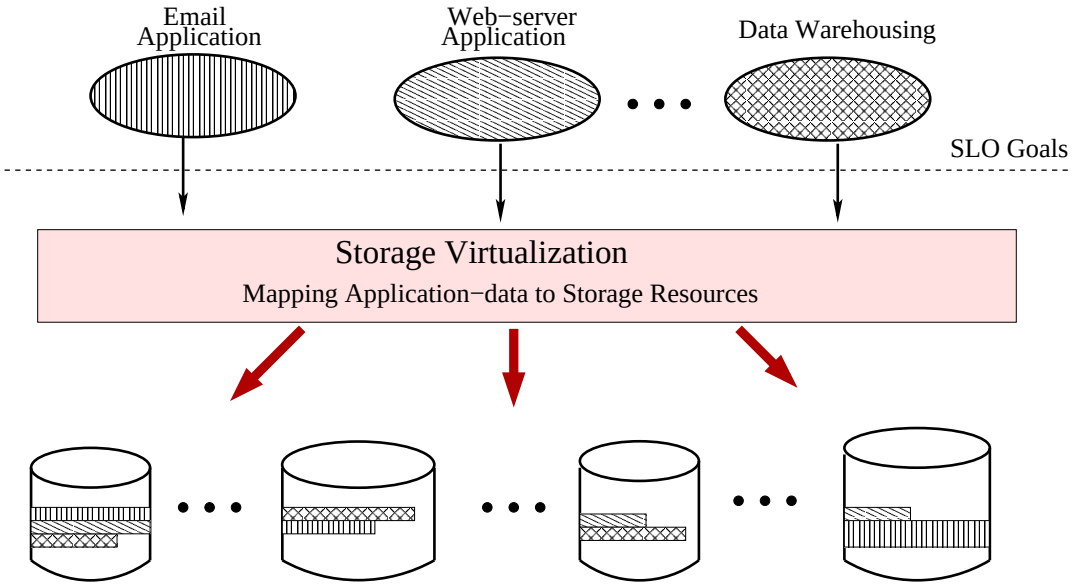


Figure 1.2: Mapping the data-sets of the workloads to the available resources

1.2 Myths and Realities of a Self-managing Framework

There are a number of criteria that an effective management software should meet. They include responding to changes in the environment (i.e. being self-evolving), starting off with basic information about the system and continuously adding to the domain details at run-time, exhaustive search of the relatively large solution-space of corrective actions, selectively passing control to the human administrator for system states in which the models have a low confidence value. This section illustrates these requirements as a series of myths and realities for system management within a production system.

Myth*: It is possible to have perfectly accurate domain-specific models*

Reality: In production systems, it is not possible to capture all the system and environment parameters that affect the output of the model. e.g., the effect of data migration is primarily dependent on the current load of the target component (where data is being moved); but there are additional second-order parameters such as the ON-OFF phase of the workload and its overlap with the current target component load plays an important role in deciding the effect of data migration. Also, complex effects such as that of an action on another

action [93] are difficult to account. It should be noted that the accuracy of the self-evolving model is not constant within the state-space, but varies depending on whether similar system states have been seen in the past.

Myth*: The objective of the management software is to find the most optimal corrective option*

Reality: The available corrective options in production systems is of the order of millions. This is because, the a corrective action when invoked with different parameter values has a different impact on system behavior e.g. the data migration action when invoked to migrate different data-sets to different target locations will be considered as different corrective options. The notion of finding the optimal corrective option is infeasible: First, inaccuracies in the models makes it difficult to distinguish one good action from the other – the best we can get is to distinguish between good and bad actions. Second, the computational complexity of searching a huge set of corrective options may be forbiddingly high making it necessary to have empirical techniques that prune portions of the solution-space. Thus POLUS uses the design principle that it is not important to find the most optimal action, but rather to avoid the worst ones.

Myth*: Automation removes the human administrator instanteously; also a management software outperforms human administrator*

Reality: At the time of deployment, the domain-specific details are bootstrapped by running calibration tests – these tests sparsely explore the state-space and serve as baseline values. At run-time, as similar system-states are visited repeatedly, the confidence-value of the models improves, and allows automated action invocation. Whenever the system operates in a relatively unexplored region of the state-space, the management software should flag for administrator input (unless the cost of invoking as well as reversing the action effect is small). Thus, the longer the system runs, the more useful the management software becomes; for complex systems, it might be months and even years before the management software takes over control of common run-time scenarios – the exception scenarios still need to handled by the administrator.

The objective of management software is not to outperform the administrator. This is analogous to Deep Blue playing chess with the grand master [2]. The strength of management software is the exhaustive exploration of the corrective options; while the strength of humans is a better perception of domain specific details (which in the case of automation is more critical).

Myth*: Domain-specific models can be trained for the entire state-space*

Reality: The operational state-space (i.e. permutations of system parameters values) for production systems is large – creating behavior models up-front for the entire state-space is infeasible and even impossible. Further, production system models for components, workloads, actions, have inherent non-linearity and there is no simple way to extrapolate them for the entire state-space.

1.3 Bird's Eye-view of POLUS

POLUS consists of two core modules (as shown in Figure 1.3): The **Knowledge-base** and the **Reasoning engine**. The Knowledge-base stores models of domain specific details, refines the models at run-time using monitor data, and triggers the reasoning engine for selecting corrective actions. The reasoning engine selects a *feasible* corrective action, by matching the current system state with the cost-benefit attributes of the action; it executes the selected action(s) using a feedback loop.

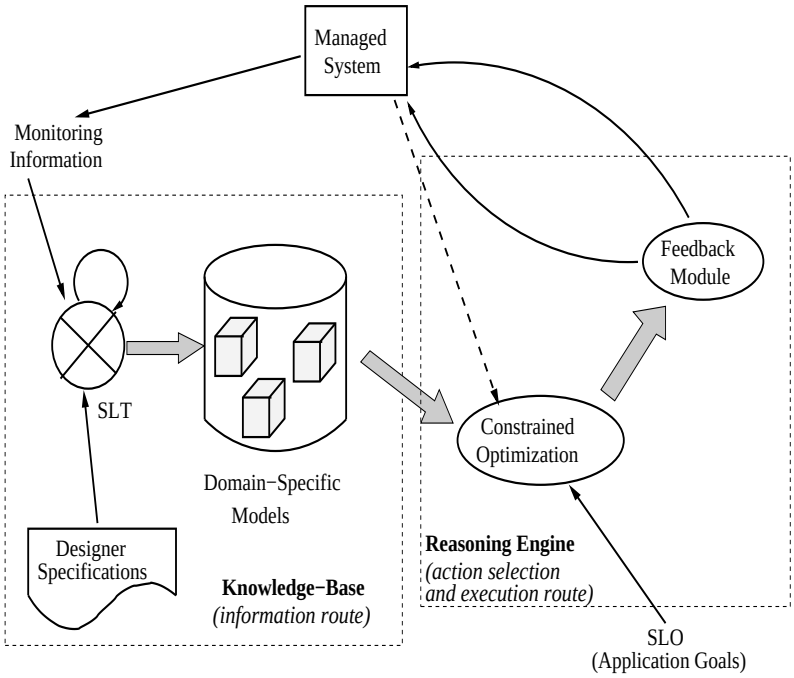


Figure 1.3: The POLUS Architecture for OAA Automation

1.3.1 Knowledge-base

Based on a decade of Expert Systems research, Edward Feigenbaum coined the Knowledge Principle [39] (also known as the second-law of artificial intelligence): *"The power of AI programs (i.e. expert systems) to perform at high*

levels of competence is primarily a function of the program's knowledge of its task domain, and not of the program's reasoning processes." The effectiveness of automated systems such as POLUS is dependent on the *richness* of domain-specific knowledge encoded within the management framework.

In POLUS, the domain-specific models are created by combining designer specified *feature-set* (i.e. the list of related parameters), with statistical learning [42] of the monitor data. In other words, the feature-set defines parameter x is related to y and z, while statistical interpolation derives the function between those parameters $\mathrm{x} = 3y^2 + 4z + 7$). The models are *self-evolving* in that statistical learning periodically recalculates the functions using the newly monitored data. In contrast to a pure black box technique, the advantage of using designer-specifications limits the the number of input parameters considered for correlation using SLT; this approach sacrifices accuracy for faster convergence, since the designer specifications generally enumerate parameters with a first-order effect.

1.3.2 Reasoning engine

Action selection consists of deciding the action along-with the values of its invocation parameters (we refer to the action and invocation-values pair as a *corrective option*). (e.g. the invocation parameters for the migration action are the data-set to be migrated, the target location, and the migration speed). Each corrective action changes the behavior of the system in a particular direction within the state-space (*benefit*); requires a certain amount of resources for being invoked (*cost*) The cost-benefit of invoking an action is dependent on the current state of the system, and the value of the invocation parameters. Thus the same action invoked with different invocation values or in a different state will have a different cost-benefit-time.

A production system can potentially have a large number of corrective options especially with different invocation values used with the same action. POLUS generates the top-k invocation options for each action; this is done using constrained optimization [75], where the invocation parameters of the action serve as the *variables*, the models serve as the *constraints*, and the application SLOs as the *objective functions*. In a system with m actions, there are total of km corrective options that will be considered by POLUS. For each of these option, POLUS calculates the cost, benefit, and time required for invoking the action using the domain-specific models, and filters the high-cost low-benefit options. The definition for cost and benefit are domain-specific.

The task of action selection finally translates to matching: 1) the cost of action invocation with the available resources in the current state; 2) the benefit of invoking the action with the system need either in the current or future states (based on growth trends); 3) the time required for taking the action with the time available before SLO violation (in the case of reactive triggers, this duration

is zero). POLUS uses a simple heuristic approach to randomly select an option the satisfies all the criteria – given the model inaccuracies, the candidate options are semantically equivalent at this stage in the decision-making.

The corrective option is executed incrementally using a feedback loop; the aggressiveness of action invocation (i.e. the step-size for incremental execution) is proportional to the confidence-value of the models. The feedback loop checks for divergence from predicted action effects, and re-triggers the decision-making for significant variations that occur during action execution e.g., significant change in the workload characteristics.

1.4 Outline of the thesis

The rest of the thesis is organized as follows. Chapter 2 starts-off by positioning the OAA automation framework within the high-level architecture of the overall system. It describes the taxonomy of existing approaches namely policy-based, feedback-based, empirical/learning-based, and model-based. The related work section is divided into two parts: 1) Representative examples for the taxonomy of existing system management approaches; 2) Details of AI projects and lessons learnt in non-procedural languages, expert systems, and machine learning – research that serves as building blocks for POLUS.

Chapters 3 and 4 cover the design details of the POLUS framework. Chapter 3 describes the Knowledge-base – representation of domain details as models, gray-box techniques for combining designer-specifications with information deduced by applying STL to the monitor data, algorithms for evolution of models at run-time, and issues related to bootstrapping and incomplete designer specifications. Chapter 4 describes the Reasoning engine – formalisms for deriving corrective options using constrained optimization, techniques for pruning the number of options considered during decision-making, algorithms for selecting a feasible option based on cost-benefit-time metrics. The chapter also covers details of feedback loop used for action execution.

Chapters 5 and 6 describe the application of POLUS for resource arbitration within a production storage system – the design details, implementation insights, and experimental evaluation of the deployment. Chapter 5 describes the instantiation of POLUS as a storage resource arbitrator. Chapter 6 describes the experimental validation using synthetic and real-world workload traces (namely SPC [35], HP's Cello96).

Chapter 7 summarizes the key features of POLUS, and how they address the existing skeptism for OAA automation in production systems. It chalks out directions for future research, envisioning two roadmaps: The Information route (*how accurate and self-evolving the models can be*), and the Action Selection Route *(how accurate the models need to be for selecting reasonable actions and how well can the system handle uncertainty).* Based on the insights gained from this PhD, we believe that the eventual reality of OAA automation lies at

the crossroads of domain-specific accuracy, and ability of reasoning engines to handle inaccuracies.

1.5 Contributions

Automation of the OAA loop is a non-trivial problem; the growing economic incentive for self-management, and significant growth in computing bandwidth have made OAA automation an active area of research. Previous research and commercial products in this domain have been largely focussed on using a policy-based approach, which did not get much traction in production systems because of the complexity in defining policies and brittleness of policies w.r.t. configuration changes. This thesis addresses an important and unsolved problem of OAA automating using a first-of-a-kind model-based framework with an innovative approach for integrated learning and feedback. In particular, this dissertation makes the following contributions:

- It presents a methodology for model-based automation of the observe-analyze-act loop. The description of methodology covers details for the representation of domain-specific details as non-monolithic models, creation and evolution of the models at run-time, and the reasoning process for selection and execution of the corrective actions.

- It provides details of applying the Polus methodology to a production storage system. It describes the design, implementation, and experimental evaluation of a storage resource arbitrator with guaranteed performance (i.e. throughput and latency bounds) to multiple application workloads sharing a common heterogeneous storage infrastructure. The arbitrator continuously monitors the component utilization and workload behavior, analyzes the system utility, and re-balances resource distribution in response to variations in workload access characteristics, performance goals, relative workload priorities, available component bandwidth.

- It addresses several challenges related to a model-based approach, and proposes novel techniques for meeting these challenges. These techniques include: an gray-box approach for creating domain knowledge using a combination of system-designer specifications and information extracted using machine learning techniques; a feedback-based execution of the corrective actions for handling varying confidence-value of the domain details; a constrained optimization formalism for parsing the solution-space. It demonstrates the feasibility of the following operational principles required in real-world OAA automation: 1) Models don't need to be perfectly accurate – they only need to be accurate enough to maintain the relative ordering during action selection; 2) The objective of action selection is not to find the most optimal one, but rather to avoid the worst ones; 3)

Creation of models is not a one-time activity – it is a continuous process over the lifetime of the system.

In summary, this thesis demonstrates that a model-based approach can be used as the basis of an effective, automation of the observe-analyze-act loop, and it provides practical advice and experimental evidence of successfully applying this technique within production system.

2 Automating the Observe-Analyze-Act Loop

The OAA loop refers to observing information from the system being managed, analyzing the current system state, and activating corrective actions. This chapter begins by positioning POLUS within the high-level architecture for automated system management. It then describes details of existing approaches for OAA automation. The rest of the chapter describes related work into two parts: First, a description of representative examples for the taxonomy of existing OAA frameworks namely policy-based, feedback-based, empirical or learning-based, and model-based frameworks; Second, a survey of AI research that serves as building blocks for POLUS namely lessons learnt in programming languages, approaches used for building Expert Systems, and applications of machine learning algorithms.

2.1 High-level System Architecture

The overall architecture is shown in Figure 2.1. The system being managed is referred to as the *managed system*. There can be a variety of methods for incorporating the automation logic within the managed system. For the sake of argument, we assume that there is a distinct automation module called the *OAA framework* that is responsible for extracting domain-details from monitoring data, analyzing the current system state, and executing the corrective actions within the system.

Multiple applications are using the managed system e.g., in an enterprise storage system, there are applications such as databases, e-mail, web-server, accounting software, middleware frameworks that use the storage system; each application requires certain behavior guarantees from the managed system, referred to as *Service Level Objectives* (for example, the database application may require the storage system responds to the I/O requests within a maximum of 5 msec.). The task of the OAA framework is to ensure these guarantees are met during ongoing variations in the request characteristics of the applications, failure/addition within the managed system, load surges for particular applications, etc. There are three key entities used in the formalism:

- Workloads (w): *Represents the requests generated by an application, running on the managed system. The parameters that characterize a workload differ from domain to domain.* For example, in storage automation, workloads are characterized by read_write ratio, random_sequential ratio,

request size, request arrival rate. Similarly, within database automation, the workload is characterized by type of query (update/data-read), the tables being accessed, sequential or random access of database tuples.

- Components (c): *Represents the properties of physical elements present within the managed system – properties such as the maximum number of requests that can be serviced, the error rate, the average down-time, etc.* Within a storage system, the components consist of controllers, disks, interconnect switches. Similarly, the components for a database are physical memory, CPU, disks, network. As a side-point, the sequence of components invoked while servicing the request is referred to as the *invocation path.*

- Actions (a): *Changes the observed system behavior by tuning and/or modification of the workload-to-resource mapping. The impact of invoking an action is not a constant, but rather dependent on the current system state as well as the invocation parameters used for invoking the action.* Examples of corrective actions within a storage system are data-migration, replication, prefetching.

The OAA framework takes the SLOs as input, and in light of the the current system state, makes the necessary action invocation. The interaction between the OAA framework and the managed system is via *sensors* and *actuators* – sensors gather information about the managed system, while the actuators effect the actions to be invoked on the managed system. The information collected by the sensors generally includes the workload characteristics, utilization of the components, comparison of the current behavior with the prescribed SLOs.

2.2 Taxonomy of Existing Approaches for Automated Management

Existing approaches for system automation can be broadly divided into: policy-based, empirical/learning-based, feedback-based, and model-based approaches. We describe details of each approach along-with their limitations in real-world systems.

2.2.1 Policy-based

System administrators encode policies as sets of event-condition-action (ECA) rules [106, 102, 59, 77]; these rules are fired when some precondition is satisfied (typically, one or more system observables change beyond a administrator defined threshold). The rule-based approach is based on the Hewitt's *pattern directed procedure invocation* paradigm [56, 55]; patterns are similar to event-condition definitions, while procedures are similar to the action definition.

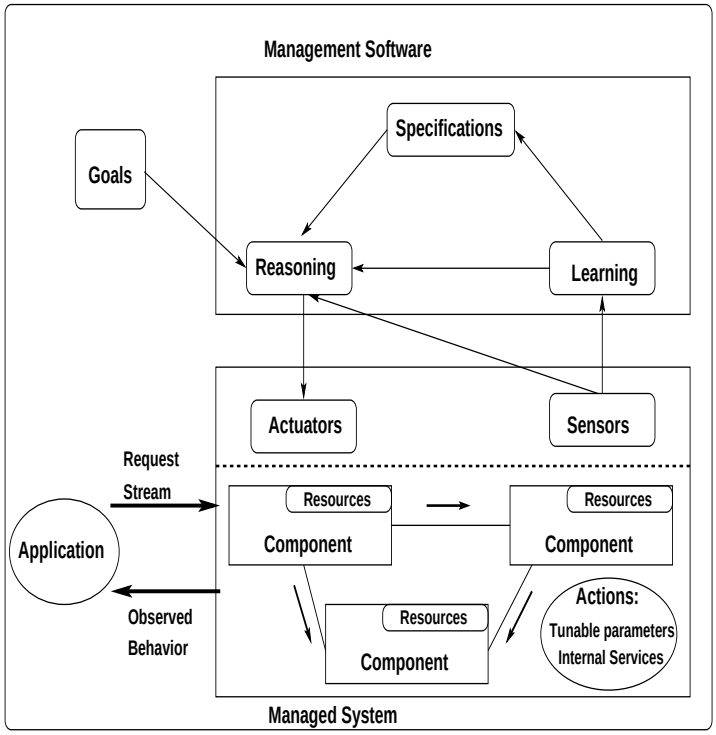


Figure 2.1: System model

Rules serve as "canned recipes" in determining the corrective actions to be invoked in different system states. At run-time, the management module simply invokes the rule that is applicable based on the event and system conditions. Writing management rules is done by experts with many years of prior experience as system architects and administrators. However, with the growing system complexity, even experts are encountering the following types of problems, making it difficult for them to design robust automation frameworks:

Complexity: The level of details, required to write rules is non-trivial (e.g. if *throughput_goal_violated* AND $Sequential/Random\_ratio > 1$, then increase data prefetching by 20%). It is difficult for the composers of ECA rules to: (1) choose which combination of system parameters to observe from a large set of possible observables; (2) determine appropriate threshold values after considering the interaction of a large set of system variables; and (3) select a specific corrective action from the large set of competing options. As the number of users, components, corrective actions, and service level agreements increases, it becomes computationally exhaustive, error prone, and probably impossible for system administrators to consider all the alternatives.

Brittleness: It is difficult for vendors to provide prepackaged ECA rules with their products – rules are brittle with respect to changing system configurations, user workloads, and department/business constraints – it

is non-trivial to update and verify the operational semantics of rules. It is difficult for the vendors to envision all of the potential run-time scenarios ahead of time and provide rules for them.

Even though rule-based system management is quite popular, and has been advocated as the silver bullet for system automation, it has not gathered much traction and has only been successful for automating trivial management scenarios.

2.2.2 Empirical-based/Learning-based

One way of determining the corrective actions is to empirically evaluate the performance of some or all of the possible corrective options. In other words, the OAA framework can select different corrective options according to some search criteria, measure the performance of the system for each of the options, and compare the resulting performance measurements to determine the optimal corrective option. The measurements needed for the comparisons can be obtained through the use of simulations, or based on a previously visited system state (similar to recording history of system-states using Case-Based Reasoning [69]).

In its simplest form, an empirical-based approach will try all possible corrective action settings to see which ones work best. However, for frameworks problems with a large number of possible settings, an exhaustive comparison may require an unreasonable amount of time, and some method of pruning the space of possible settings is typically employed. In production systems, the workloads are not known in advance or are likely to change over time, and it is not possible to easily reproduce the workloads ahead of time; in such cases, the OAA framework will need to perform empirical comparisons whenever a new workload arises. A key limitation of this approach is that it can be difficult to find the optimal corrective option in a reasonable amount of time, especially if the empirical comparisons must be performed dynamically as new workloads appear.

2.2.3 Feedback-based

In a feedback-driven approach, the OAA framework iteratively adjusts the knobs based on whether the values of one or more performance metrics converges or diverges to certain administrator-defined goals. For instance, if an OAA framework were attempting to maintain a particular response time for a database system, its knob adjustments would be based on how the system's current response time compares to some response-time goal. In feedback-based solutions, the magnitude of a given set of knob adjustments is determined using a spectrum of classical control theory techniques [43] to simplistic heuristics and estimation techniques. The OAA framework evaluates the instantaneous effect of knob

tuning to determine whether any additional adjustments are needed. This cycle of observations and adjustments is often referred to as a *feedback loop*, because the effects of one set of adjustments on the performance metrics are fed back into the tuner and used to determine the next set of adjustments.

Similar to empirical approaches, feedback techniques involve measuring the performance obtained for various corrective actions. However, empirical-comparison-based tuning methods compare the performance of various corrective options on a particular workload and use these comparisons to determine the optimal action to be invoked. Feedback-based methods, on the other hand, simply map the current values of the guiding performance metrics to the incremental invocation of the corrective action and repeat this process until no further adjustments are needed; such methods never explicitly compare the performance of the corrective options. Compared to empirical and model-based approaches, feedback techniques have the following advantages. First, they are easily adaptable to changes in the environment. Second, they are able to perform well without any training (and thus can handle previously unseen workloads), although experimentation may be needed to determine the critical values of the performance metrics or to devise the heuristics used to guide the knob adjustments. Thus, feedback mechanisms are also well-suited to tuning problems that involve rapidly fluctuating workloads – the feedback-based OAA framework can continuously make knob adjustments based on the changing values of its guiding performance metrics.

Although feedback mechanisms have been used effectively to create self-tuning systems, they have a number of disadvantages. First, it is often difficult to use feedback for systems with multiple actions (a.k.a. knobs). With a single knob, the OAA framework only needs to decide whether to increase or decrease the current knob setting, whereas adjustments to multiple knobs require an understanding of how the knobs interact and how they jointly affect the performance of the system. Second, a large number of knob adjustments may be needed before the optimal knob settings are reached, and this may lead to unreasonable runtime costs. Third, rapid adjustments of the knob values can lead to ping-pong effects such the system continuously vacillates between two bad states – this is especially true when the impact of the knob on the performance metrics is complicated enough such that there is no way of knowing when the optimal settings had been reached.

2.2.4 Model-based

In a model-based approach, the OAA framework uses models to predict the system's performance for any combination of workload characteristics, component utilization, and action invocation values. At run-time, these models to determine the knob settings that maximize convergence towards administrator defined goals for the current workload and invoke the actions accordingly –

the OAA framework internally analyzes several permutations of knobs values, formulated as a dynamic programming problem with an administrator goals serving as an objective function. Similar to the empirical-based comparisons, model-based approaches compare the performance of various corrective actions, but the comparisons are based on the model's predictions, not on measurements of the actual, simulated or recorded performance of the system. And like any OAA framework based on comparisons, a model-based approach may need to employ special search techniques when the space of possible corrective options is large enough that exhaustive comparisons are impractical.

There are a number of types of models that can be used to construct this type of an OAA framework, including statistical models based on regression that are used in Polus. Each type of model has an associated set of parameters, and the values of these parameters are typically learned from a collection of training examples – each of which consists of statistics capturing the workload characteristics, action invocation values, component utilization, and performance of the system over some interval of time. As the system runs, the model's parameters can be updated to reflect newly collected statistics, allowing the OAA framework to make better predictions and to adapt to changes in the environment.

In contrast to feedback-based approach, a model-based OAA framework can determine the optimal corrective option for the given system state after a single set of computations, avoiding the series of iterative knob adjustments that a feedback-based approach often requires. Additionally, a model-based framework can generalize from experience, using its models to predict the performance of previously unseen workloads and to thereby determine the optimal corrective option in those system states. However, there are potential drawbacks of model-based approaches to tuning. First, they typically require an initial bootstrapping of models; in such cases, the model's predictions are not accurate until sufficient training data has been collected. Moreover, the process of collecting training examples from an already deployed software system will often degrade the system's performance, because the need to see many different combinations of workload characteristics and knob settings means that some of the knob settings chosen during training will necessarily be non-optimal, and they may lead to significantly poorer performance than the system's default settings. Also, the models need to be evolved at run-time to accurately reflect the temporary or permanent changes in components and workloads characteristics. Second, it can be difficult to formulate behavior models for a complex software system. As the number of variables relevant to the tuning problem increases the challenge of developing an accurate model also increases. Although these potential drawbacks of model-based approaches are significant, they can be overcome (as described in chapters 3 and 4). Third, the action selection and execution should account for inaccuracies in model predictions, which in the worst case can lead to a wrong action being selected.

2.3 Related Work: Taxonomy of available approaches

2.3.1 Examples of Policy-based Approach

Most current commercial tools for automatic resource allocation (e.g., BMC Patrol [17]) are policy-based. Also, a significant percentage of automated QoS management in storage has been using the policy-based approach: Rome [110], Minerva [8], Hippodrome [10], and "attribute-managed storage" [49] projects from HP, SELF-* project [44] from Carnegie Mellon, Storage Tank [83], SledRunner [28] projects from IBM, Control Centre product line from EMC, Storage Central product line from Veritas. As mentioned earlier, rules are a clumsy, error-prone programming language; they front-load all the complexity into the work of creating them, in exchange for simplicity of execution at run time. Administrators are expected to create rules [59, 77] that 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 intended rule will fire if preconditions overlap. Since all this work has to be done in advance and with minimum quantitative information about the system, and simple policy changes may translate into modifications to a large number of rules, this approach is unlikely to significantly improve manageability and accuracy of response. Policy-based systems can only provide a coarse-grained optimization, as good as the human who wrote the rules. In contrast, POLUS relies on constraint-solving algorithms that explore the entire search space of throttle values for each workload. Instead of relying on hardwired thresholds, POLUS uses its dynamic models to make optimization decisions that admit iterative refinement.

Mark et. al., [13] propose to enhance the ECA rules, by adding a human-readable goal specification. The authors argue that the existing approach for defining rules, keeps the reason for writing the rule implicit, making it difficult to maintain and apply these rules. Their approach creates a mapping between the rule and user-requirements, making it easy for validation and usage. The approach can be applied to a limited domain and does not have well-defined semantic foundations.

Murthy et. al., [37] propose breaking the management policies into a hierarchy of simpler policies – the output of the higher level policy serves as an input for the lower-level policy, using the notion of "connector functions" that the authors have developed. This approach simplifies policy-definition but does not really solve the problem of specifying details and brittleness – the approach is still based on the policy-based model.

Zinky, et al., [112] propose a policy-based framework, called QuO, to implement QoS-enabled distributed object systems. The QoS adaptation is achieved by having multiple implementations for a given function-call on the server. Each

implementation is mapped to a set of system conditions and an expected QoS region. The QuO approach supports coarse-grained QoS requirements, and is useful for a limited set of system conditions.

2.3.2 Examples of Empirical/Learning-based Approach

The Vino project [90] constructs an operating system kernel that monitors its own performance and adapts to changing workloads. It determines the optimal policies (e.g., the optimal buffer-cache replacement policy) for handling a particular workload by comparison with results from simulation modules. Vino leverages the underlying extensible operating system replacing the simulation modules on a per-process basis [89]. Simulation modules maintain their own separate state and do not affect the global state of the system, and thus the kernel can continue to use the default policies while the simulations explore possible alternatives. This significantly reduces the potential negative effects of tuning on the performance of the system. However, most operating systems do not provide the extensibility needed to conduct this type of simulation.

The self-tuning methodology proposed by Feitelson and Naaman [41] exhaustively compares all possible operating system knob settings, using genetic algorithms to guide the search for good settings. In each round of comparisons, several candidate settings are tested, with settings that perform well in a given round of simulations having a higher probability to persist in the next round either in whole or in part. Various transformations combine and modify the current group of candidate settings according to their relative performance. The experimental validation of this work was using a parameterized scheduling algorithm, in isolation from the rest of the operating system – the simulation results ignore the complex tuning dependencies arising from other parts of the operating system.

Vuduc et al. [104] describe statistical models for determining the best implementation of a library subroutine for a particular platform and set of input parameters. Application-specific heuristics are often used to reduce the number of empirical comparisons to find the optimal implementation. To complement the use of such heuristics, the authors employ statistical methods to estimate the probability that the performance of the best implementation seen thus far differs from the performance of the optimal implementation by more than some user-specified value. When that probability falls below a second user-specified value, the search is halted.

The performance tuning methodology proposed by Reiner and Pinkerton [84] uses empirical comparisons to dynamically adapt the knob settings in an operating system. The comparisons are performed online during a special experimental phase. Each time the state of the system changes significantly, one or more of the candidate knob settings is chosen randomly to configure the system, and the resulting system performance is recorded. Over time, the system acquires

enough data to determine the optimal settings for each possible state of the system, and it concludes the experimental phase. Thereafter, the knobs are dynamically adjusted according to the optimal settings for each system state, and additional experiments are occasionally conducted to allow the system to adapt to changes in the environment. The methodology was implemented for tuning two scheduling parameters on a time-sharing system – the experimental results showed minor performance improvements; the authors did not attempt to assess the degree to which the the experimental phase degraded the system's performance.

The tuning approaches discussed in this section – like all approaches that base their tuning recommendations on empirical comparisons – are unable to generalize from experience. As a result, additional training is needed when new workloads arise – something that is not necessary in the model-based approach proposed in this thesis. Seltzer and Small's approach mitigates the impact of this additional training on the system's performance by using special simulation modules, but these modules are not available on most systems. As a result, approaches based on empirical comparisons can degrade the performance of the system when new workloads arise, unless they defer the additional training to periods when the system is idle or run the training offline. In any case, the need for additional training means that these systems are unable to make timely tuning recommendations for previously unseen workloads. The model-based approach presented in this thesis, on the other hand, can recommend optimal or near-optimal tunings for new workloads in a reasonable amount of time by generalizing from previously seen workloads.

2.3.3 Examples of Feedback-based Approach

The Comfort Automatic Tuning project [107] uses a feedback-driven approach to tune a database system's multiprogramming level (MPL) – a tunable parameter that limits the number of concurrent accesses to the database. The tuning is based on the lock contention in the system: when this metric exceeds a critical value, the MPL is reduced, and vice-versa. The critical value is determined experimentally – it is not a single value but rather a range of critical values that perform well on a wide range of workloads. The authors also present results showing that their approach allows the system to provide acceptable response times under extremely high loads.

In the SEDA framework for highly concurrent Internet applications [108], applications consist of a series of components called stages that are connected by queues of events, and feedback-driven tuners called resource controllers are used to dynamically adjust each stage's resource usage. For example, a stage's thread pool controller tunes the number of threads associated with the stage, adding a thread when the length of the stage queue rises above some threshold and removing a thread when it sits idle for longer than a second threshold.

SEDA's resource controllers operate at the application level, without needing to be aware of the resource management policies of the underlying operating system. It is unclear how sensitive these controllers are to the thresholds used to guide the knob adjustments, but the authors present results that demonstrate the ability of the resource controllers to effectively adapt to increasing load.

The Goal-oriented Buffer Management framework [24, 22, 23] uses feedback to tune knobs related to memory management and load control in a database system. The objective of the tuning is to meet the response-time goals of individual workload classes in a multi-class database workload, and the knobs are adjusted until either these goals are met or until the OAA framework determines that they cannot be met. The workload classes are tuned separately; depending on the nature of a given class's memory usage, either one or two knobs are adjusted using estimates and heuristics to guide the adjustments. Simulations show that both the one-knob and two-knob tuners are able to meet the goals of a variety of workloads, although it can take a long time to achieve the response-time goals of certain types of workloads.

Microsoft's AutoAdmin project adjusts the size of a database cache using a feedback-based technique [30]. The adjustments are based on the amount of free physical memory in the system: when the number of free memory pages drops below one threshold, the size of the database cache is reduced; when the number of free pages exceed a second threshold, the cache size is increased. The authors do not explain how the threshold values are chosen; as future work they outline plans to use feedback to adjust the number of pages read into the cache when the system performs read-ahead (i.e., when it proactively reads in pages that it anticipates will be accessed soon).

Feedback mechanisms have also been widely applied to resource management problems in operating systems, including CPU scheduling [33] and network congestion and flow control [60, 64]. To facilitate the use of feedback-based tuners in this domain, Goel et al. [48] have developed a toolkit of simple, modular feedback components that can be combined and reused. However, all of their example tuners adjust a single knob, and it is unclear whether their components can effectively handle software-tuning problems that involve the simultaneous adjustment of multiple knobs.

Although feedback-based methods can, in theory, avoid the need for training and model building required by model-based methods, the examples presented above demonstrate that experimentation and the development of heuristics are often required to construct an effective feedback-based tuner. Even the examples that do not mention the need for this type of preliminary work would need some means of determining the threshold values that guide the tuner's knob adjustments. The model-based approach on the other hand, is able to avoid the potentially lengthy series of iterative adjustments that a feedback-based approach may require.

2.3.4 Examples of Model-based Approach

The platform-dependent optimization framework proposed by Brewer [20] uses regression models to tune library subroutines. The models are created using linear regression but allows the independent variables to be non-linear (e.g., an independent variable can represent the product of two or more of the parameters of the subroutine being optimized). Although this approach works well for tuning subroutines, it is unclear whether it would be possible to produce accurate regression-based performance models of large-scale enterprise systems.

Matthews et al. [72] use a model-based approach to tune a modified version of the log-structured file system (LFS). Their models consist of simple formulas for estimating the cost or cost-benefit ratio of the possible knob settings; they are based on an understanding of the operations performed by the system and their associated costs. The model's only parameters are measurements of the costs of various operations on a particular disk. The experimental evaluation was using simulations of both LFS and the disk on which it resides – it is unclear how well such simple models would work on an actual system, or whether it would even be possible to predict the performance of more complex systems using such models.

The AutoAdmin project [5] uses model-based techniques for automating the selection of indices and materialized views to be created within the database – the objective is to speed up frequently occurring database queries. The decision-making is based on a cost estimate models for the database system's query optimizer. The AutoAdmin framework implements novel methods for selecting which indices and materialized views to consider and for efficiently searching through the space of possible combinations of indices and materialized views. The internal models of the query optimizer are not discussed, and thus it would be difficult to instantiate the proposed techniques to an arbitrary software system.

In the Odyssey platform for remote computing [81], applications adapt to changes in resource availability and user goals by varying the fidelity with which they operate (e.g., the frame rate used by a streaming video application). Narayanan et al. [78] augment Odyssey with a system that uses models to predict an application's resource usage as a function of the relevant input parameters and fidelity metrics, and to thereby recommend appropriate fidelity levels for a given operation. To avoid annoying the user, the initial training data is collected during a special offline mode in which a given operation is repeatedly run using randomly chosen fidelities and inputs, and the parameters of the models are refined as the system runs. For their initial prototype, Narayanan et al. employ linear regression to derive the models, and they use linear gradient descent [76] to update the model's coefficients over time. To determine the appropriate fidelities for a given set of inputs, the tuning system employs a gradient-descent solver.

Menasc et al. [74] use a tuner based on queueing network models [68] to optimize the quality of service (QoS) of an e-commerce site. When their system detects that a QoS guarantee has been violated, it employs a hill-climbing search guided by the model's predictions to find the knob settings that yield the locally maximal QoS. The authors present results showing that their tuner, which adjusts four knobs, is able to maintain reasonable QoS values in the face of increasing load. However, although queueing network models work well in this domain – in which the knobs being tuned are directly connected to queues of requests waiting to be processed by a Web server and an application server – it is unclear whether they could form the basis of a general software tuning methodology.

Vuduc et al. [104], after discussing how empirical comparisons can be used to determine the optimal implementation of a library subroutine for a given platform, note that the best implementation may depend on the input parameters. Therefore, they propose taking a set of several "good" implementations (possibly found using empirical comparisons) and using models derived from training data to determine which of these implementations is best for a given set of inputs. They experimentally compare the ability of three types of models – including the regression models proposed by Brewer and a statistical classification algorithm known as the support vector method [101] – to choose between three candidate algorithms for matrix multiplication. The latter method has the best performance of the three, but it is unclear how well it would scale to tuning problems with more than one knob or, more generally, to problems with larger numbers of possible knob settings.

2.4 Related work: Research in the domain of Artificial Intelligence

There are several interesting sub-problems in OAA automation that overlap with previous AI research. In particular, the Polus approach leverages the AI concepts and lessons learnt in the following three aspects:

- *Non-procedural program specifications*: From the standpoint of program specifications, existing rule-based techniques are analogous to procedural approaches that provide "canned" solutions for different system states. In contrast, the Polus derives the solution at run-time by combining domain-specific facts with reasoning techniques. Procedural specifications amalgate two pieces of semantic information: The **domain knowledge** (also called *concepts* in Kowalski's logic-programming terminology [67]), and **control knowledge** (also called *formalisms*). The concepts themselves are a combination of parameter relationship and their quantification. The Polus framework intelligently separates the three pieces of information namely *relationship between parameters, values of the relationships,*

and the control knowledge,

- *Expert System Frameworks*: An OAA framework is similar to an Expert System that takes as input the current system state along-with the SLOs, and generates as output the corrective action to be invoked. Early expert systems were built using the procedure/rule-based paradigm [25]. The problems of complexity and brittleness led to the development of non-procedural paradigms based on logic programming [109]. Based on the history of Expert Systems, the POLUS objective of non-rule based management is the next logically step in the evolution of expert systems for system management.

- *Machine Learning*: There is a plethora of monitoring data collected by existing management frameworks. Today, the monitoring data serves mainly as the "magnifying glass" for human administrators, and is presented within a monitoring dashboard consisting of thousands of gauge-values. POLUS uses statistical learning techniques to extract domain-specific details from the monitor data; this simplifies the task of defining domain-specific information, as well as allows management frameworks to be self-evolving with changes in system configuration, and application workloads.

2.4.1 Non-procedural Programming Languages

Programming languages can be divided into two categories: *Procedural* and *Declarative* [111]. Declarative approaches are further divided into *Logic-based* [45], *Semantic network based*, etc. Few approaches such as Minsky's *frame proposal* does not fall in either of these categories.

Existing rule-based approaches [59, 54] fall in the category of procedural representation schemes [55]. Procedural schemes can be classified based on two attributes: 1) activation mechanisms, and 2) the control structures. Comparing the rule-based approaches with Hewitt's Planner [56], the major differences are as follows:

- In terms of activation mechanisms, rule-based approaches are like Markov algorithms with a fixed ordering of rules with deterministic rule-matching. In contrast, the order in which the theorem patterns are matched is undetermined in Planner. Further, a theorem can call other theorems, while a rule-based system can only do this indirectly by raising appropriate events.

- In terms of control structure, there is no direct communication or control in rule-based systems (hence considered to be loosely coupled). Planner's control structure used backtracking in that when a theorem is executed and fails to achieve the predetermined goal, the side-effects of the unsuccessful theorem are erased and other theorems are tried. An extreme

proposal as far as control structures are concerned is Hewitt's Actor formalism [52, 3] which views all objects of a knowledge base as *actors* i.e. active agents.

In summary, the advantage of procedural schemes is that they allow the specification of direct interaction between facts, eliminating the need for wasteful searching [111]. On the other hand, a procedural knowledge base, like a program is difficult to write, understand and modify.

Logical representation schemes [51] represent facts as logical formulas using notions of constant, variable, function, predicate, logical connective and quantifier. Logic specifications initially had the drawback in representing procedural and heuristic knowledge, which led to MIT's "procedure-is-best" argument [111]. The procedural-logical fight ended with Kowalski's procedural interpretation of the Horn-clause linear resolution proof finder [66]. Kowalski proposed procedural semantics for logical formulas, in addition to the Tarskian semantics. The POLUS approach is similar to the logic based paradigm in representing the facts and formalisms using logical/relational formulas.

2.4.2 Expert System Architecture

The term expert systems refers to computer programs that apply substantial knowledge of specific areas of expertise to the problem solving process. Feigenbaum [40] empirically established that the means for intelligent actions in expert systems was primarily based on *knowledge* rather than *formalisms* (i.e. how the knowledge was used). Kowalski pointed-out [67] that formalisms are important because the use of poor formalism can interfere with the representation of knowledge and can restrict the uses to which knowledge can be put.

Expert systems have two components: Knowledge-base(to represent system facts) and the Reasoning engine (to use the system facts at run-time). Depending on the extent and depth of the explicit representation of knowledge, expert systems are classified as *low road, the middle road, and the high road* [21, 18]. Existing rule-based management systems are middle road systems that use canned problem-solving tactics rather than first principles (i.e. facts and formalisms). In contrast, the POLUS paradigm is a high road expert system that solves problems using first principles.

2.4.3 Machine Learning

Machine learning is used to interpolate the "black-box function" that relates the input and output values. Learning algorithms [85, 46, 65] are divided into two broad categories: *Supervised* and *re-enforcement* learning. POLUS uses a combination of supervised and re-enforcement algorithms to quantify the information defined by the user specifications. Further, the specifications are used as heuristics to prune the interpolation space. The model of pruning is similar to *Phenomenal data mining* proposed by McCarthy [73] that correlates the

phenomena (e.g. customer has diabetes) with the observations (e.g. the customer buys sugarless food). In POLUS, the user specifications serve as heuristics that define the phenomena (i.e. actions of the system) which are quantified by observations (i.e. monitoring the system activity).

The gray-box approach of combining designer-specifications with machine learning is new to the domain of system management, but there are a few manifestations of approach in other domains. For example the Snowball project [4] extracts information from the text; it starts off with initial sets of patterns, and recursively refines the patterns based on the input text. The details of the technique are tied to the domain of text extraction and analysis. Another similar concept is referred as *Lifelong Learning* [95] where the information is continuously evolved using the hypothesis from previous learning tasks.

3 Representation and Creation of Domain Knowledge

"The power of AI programs (i.e. expert systems) to perform at high levels of competence is primarily a function of the program's knowledge of its task domain, and not of the program's reasoning processes." This is referred to as the *Knowledge Principle* [39] (also known as the second law of artificial intelligence). This chapter describes details of the Knowledge-base used in POLUS. In particular, it covers *representation* of domain-specific details as mathematical model, *creation* of models using the *gray-box* approach of combining the designer-specified feature-set with statistical learning techniques, and *evolution* of these models at run-time. The chapter also discusses off-the-shelf techniques for reducing the model inaccuracies resulting from incomplete designer-defined feature-set. To make the discussion concrete, we use examples of storage management.

3.1 Motivation of the Gray-box approach for model creation

Management frameworks today have an extensive monitoring infrastructure [17, 58] – data that serves as a "magnifying glass" for the human administrator. There is a plethora of system behavior characteristics that can be extracted from the monitor data – information that is not being currently exploited for system management. It is important that the OAA framework leverages this information for evolving the Knowledge-base with the continuous changes in the system configuration and workload characteristics. Also, administrators are swamped by the large amount of monitored data presented to them; they would prefer the OAA framework to correlate the observable behavior with the raw measurable parameters – a notion commonly referred by administrators as "Give us information NOT data."

There are two extremes for creating domain knowledge:

- *White-box* approaches where the system-designer defines detailed formulas [45, 51] or rules [36, 71, 102] to describe the characteristics of the system. These techniques are limited by excessive complexity involved in writing the domain-specific details, and brittleness of these details to ongoing changes in the system.

- *Black-box* approaches where the system starts-off with minimal background

about the domain details and refines them using statistical learning techniques [69, 103]. This approach is error-prone, and requires an infeasible number of iterations for converging in real-world multi-parameter systems.

The domain knowledge in POLUS is represented as mathematical functions (referred to as *models*); these models are created using a *gray box approach* of combining simplistic system-designer specifications with the information gathered using statistical learning techniques. For each of these models, the designer specifications enumerate feature-set, while regression techniques such as Neural Networks [87], Support Vector Machines [26] are used to deduce the exact mathematical function that correlates these parameters. These functions are then continuously refined at run-time by periodically applying regression to the newly monitored data. The advantages of the gray-box approach are simplistic designer-defined specifications, non-brittleness of the knowledge-base, and faster convergence of the deduced functions by limiting the number of parameters considered for regression.

3.2 Representation of Domain-specific details

Domain-specific details for real-world systems are complex – it involves a large number of measurable parameters associated with individual components, workloads, application behavior, and workflow dependencies. This section describes how POLUS represents measurable parameters as non-monolithic mathematical models. The nature of these models is dependent on the SLO attributes supported by the OAA framework – the scope of this thesis are the performance attributes (i.e. goals for the observable throughput and latency values).

The domain-specific parameters are represented as models for components, workloads, and actions. These models capture the relevant system information required for deciding corrective actions at run-time (the details of action selection are covered in the next chapter).

- *Component model* (c): Represents the properties of the component as a function of incoming load e.g., the response-time of the storage controller as a function of the IO request characteristics.

- *Workload model* (w): Represents the workflow dependencies of the application requests on the individual components in its invocation path e.g. 1000 transactions/sec at the application-level translates to 1400 packets/sec at the network switch, and 2200 IOPS at the storage controller.

- *Action model* (a): Represents the *cost-benefit* effects of action invocation e.g., data migration action has a benefit of balancing load across the components, at the cost of movement of data.

3.2.1 Component Model

Component models predict the response time as a function of the request characteristics. In the case of storage systems, the request characteristics consists of parameters such as req_size,req_rate, rw_ratio – the response time for workload Wi will be of the form:
$R_{Wi} = c(req\_size, req\_rate, rw\_ratio, random/sequential, cache\_hit\_rate...)$
Figure 5.4 represents a projection of c for a FAStT 900 storage controller with 24 drives connected in RAID 0 configuration. Response time (R) is a summation of service time ($Rservice$) and wait time ($Rwait$). Service-time represents the time required to execute the request (generally dependent on the characteristics of the request), while the wait-time represents the queueing delay resulting from one or more workload streams sending requests to the components.

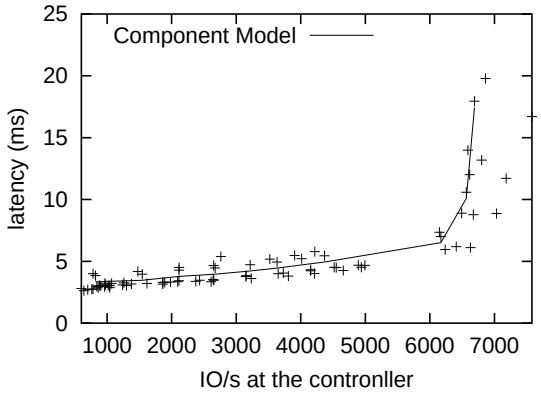


Figure 3.1: Component model.

$Rservice$ is bootstrapped and refined at run-time using regression algorithms (as described in the next section). $Rwait$ represents the time spent in the queue due to interleaving with requests from the same workload or other workloads using the component. There are several formalisms to capture the queueing delay, generally in a domain-independent fashion. A simplistic approach to estimate $Rwait$ is using a multi-class queueing model [61]; the resultant response time R is approximated as:

$$R_{Wi} = \frac{Rservice_{Wi}}{1 - U}$$

such that *utilization* U of the component is:

$$Utilization\,(U) = \sum_{i=1}^{n} req_rate_{Wi} Rservice_{Wi}$$

The formalisms for queueing delay can be made further accurate by accounting from the load variations (also referred to as *ON-OFF phases*) of the workloads [19].

3.2.2 Workload models

Representation and creation of workload models has been an active area of research [27] – there are multiple domain-specific parameters used for characterizing the workload stream (similar to the load characteristics at the component). In POLUS, the workload models predict the req_rate at the invocation-path components, as a function of the application-level request rate e.g., 1000 transactions for the application i translates to 1500 IO requests at the storage controller j (illustrated in Figure 5.5). Component_load$_{i,j} = w_{i,j}(workload_request_rate_j)$

It should For the sake of simplicity, and based on empirical observation, the other workload characteristics besides $request\_rate$ (such as req_size, rw_ratio) are generally constant and don't need to be modelled.
Component_load$_{i,j} = w_{i,j}(workload_request_rate_j)$

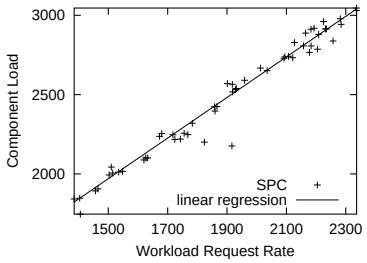


Figure 3.2: Workload model for SPC.

The current formulation of workload models keeps track of changes in req_rate as a function of application-level activity – there are are other parameters such as req_size, rw_ratio, $random/sequential$ that change over time. (besides req_rate) that change over time. To avoid having a complex function with these parameters, POLUS maintains $w_{i,j}$ as a *moving average* that gets recomputed by regression every n sampling periods – this implicitly factors in the effect of other parameters besides req_rate.

In addition to the dependency model, POLUS uses time-series algorithms to predict the variance of $w_{i,j}$ as a function of time – the standard technique for time-series forecasting is using ARIMA [96] (Auto-Regressive Integrated Moving Average). In this approach, "auto-regressive" terms counts the lag of the differenced series appearing in the forecasting equation, while the "moving average" terms account for the lag of the forecast errors.

The time-series predictions are maintained as long-term monthly or seasonly trends, and short-term daily or weekly patterns in the workload characteristics. Interpolations based on trends and patterns have been previously applied for optimizing load-balancing [86], data prefetching [96], etc. Long-term trends help in distinguishing random spikes from long-term change in the request characteristics e.g., a linear increase of 5% per month in the e-mail workloads. Similarly,

short-term patterns are helpful in understanding the daily load fluctuations – it was observed by Chase et al. [29] that typical internet service loads vary by factors of three or more through the day and through the week. Figure 3.3 shows request rates for the www.ibm.com site over a typical week starting on a Monday and ending on a Sunday. The trace shows a consistent pattern of load shifts by day, with a weekday 4PM EST peak of roughly 260% of daily minimum load at 6AM EST, and a traffic drop over the weekend.

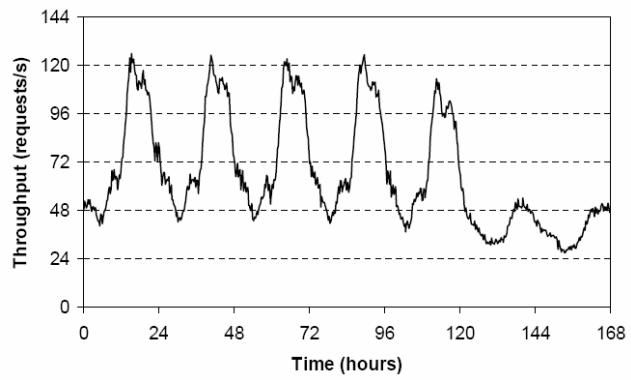


Figure 3.3: Load pattern on www.ibm.com for a one week duration

3.2.3 Action Model

An action model represents the cost a_{cost} and benefit $a_{benefit}$ effect of invoking an action – the effects are primarily dependent on the current utilization of the components, the workload characteristics, and action invocation values. Actions differ along the spectrum of cost-benefit values. Actions like throttling and data prefetching have a negligible invocation cost a_{cost}, while actions like migration and replication have a significant a_{cost} from reading source data and writing it to the target. The overhead for migration is a one-time cost, while that for replication is dependent on the number of IO writes and the number of replicas.

The cost a_{cost} and benefit $a_{benefit}$ are represented as one of the following:

1) *Change in the component model (c_new)* This represents the tuning or re-configuration of the component causing changes in its observed behavior e.g., changing the RAID level of the drive array, increasing the cache distribution among the workload streams.

2) *Change in the workload model (w_new)* This represents change in the workload dependency on the components in its invocation path e.g., data migration action changes the percentage of IO requests serviced by different disk arrays, replication action helps load balance read requests among multiple storage controllers.

3) *Change in the workload access characteristics w_access* This represents a change in the characteristics such as $req\_rate, rw\_ratio, random/sequential$ e.g., the throttling action is represented as a change in the workload request

rate. In summary, a_{cost} and $a_{benefit}$ are both of the form:

$$c\_new|w\_new|w\_access = a(invocation\_param, \alpha)$$

where α represents the set of relevant workload, component, and system setup parameters that affect the action.

The knowledge-base maintains models for c_new, w_new, w_access – the system designer feeds initial bootstrapping functions, which are refined over time with the newly monitored data. On one hand, functions for w_access are relatively straightforward to specify, and non-brittle w.r.t. the underlying infrastructure. For example, migration of data moves the load from source to the target such that the $w\_new_{source} = 0$ and $w\_new_{target} += wld_{moved}$. On the other hand, the functions for w_access and w_new are generally brittle w.r.t. the deployment infrastructure and need to be continuously refined at run-time. For example, the throttling action changes the req_rate of the workload – the initial bootstrapping can be a linear approximation between throttle rate and the workload req_rate, but production-system applications such as databases, web-servers, are sensitive to back-pressure and scale back their req_rate due to finite internal concurrency or change in access plans, making w_access a relatively complex non-linear model.

3.3 Creation of the Knowledge-base models

POLUS uses a gray-box approach where the system designer or domain-expert enumerates the feature set (i.e. the input parameters for the models), while data regression techniques are used to deduce the actual function. The intuition of the gray-box approach is that the list of correlated parameters is non-brittle w.r.t the underlying physical infrastructure, while the co-efficents of the parameter functions are brittle and need to constantly evolve with time. The alternatives to a gray-box approach are a white-box and black-box approach; in a white-box approach, the system designer defines the comprehensive model, which in production systems is non-trivial, error-prone, and difficult to maintain; in contrast, a black-box approach starts off with no information, and uses all measurable parameters of the system to deduce the model – this approach suffers from lack of convergence and large number of data points required for creating the model. The block diagram of the gray-box approach is shown in Figure 3.4) – it sacrifices accuracy for faster convergence and continuous refinement, since the designer feature-set generally consists of parameters with a first-order effect.

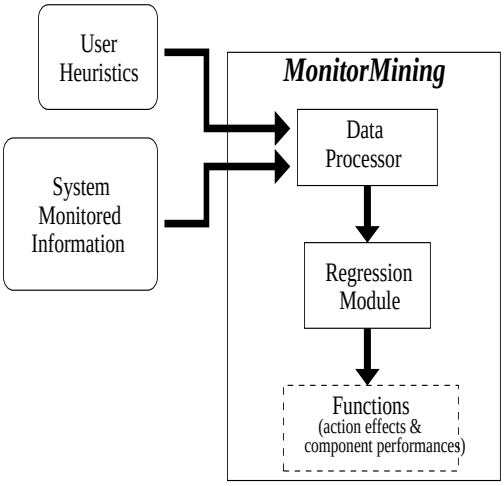


Figure 3.4: The overall procedure of deriving Knowledge-base models.

3.3.1 Designer-defined Specifications

The system designer enumerates the feature set for the action, component, and workload models e.g. *Parameter X is related to the target Parameter Y.* Additionally, the specifications can have an optional hint for the type of relationship e.g. *There is a quadratic relationship between Parameter X and Parameter Y.* The general form of these specifications is:

```
<model-type>
 <function-name>
    <output-parameter>
    <input-parameters>
 <\function-name>
<\model-type>
```

Figure 3.5 gives an example of designer specifications for the migration action – defined in terms of attributes for the workload streams at the current location (i.e. source) and new location (i.e. target).

3.3.2 Extracting domain-details using Statistical Learning Techniques

Using the designer specifications, POLUS analyzes the performance log for deriving the mathematical functions. The schema for the performance logs is as shown Figure 3.6 – it records the indiviual workload characteristics, the resulting load at the indiviual components, the time-stamp and values for action invocation parameters.

At run-time, the parameters specified by the designer feature-set are extracted from the performance logs and given to the regression algorithms. PO-

```
<Action: Migration>
 <Cost-function>
  <output_parameter ="request-rate" @ source>
     <input_parameters>
       <parameter name="migration_speed" func="linear" />
       <parameter name="data_size" />
     </input_parameters>
  <output_parameter ="request-size" @ source>
     <input_parameters>
       <parameter name="disks_per_lun">
       < parameter name = "stripe_size" />
     </input_parameters>
  <output_parameter ="read/write_ratio" @ source>
    <input_parameters>
       <parameter name="workload_characteristics" />
     </input_parameters>
  <output_parameter ="random/sequential_ratio" @ source>
    <input_parameters>
       <parameter name="workload_characteristics" />
     </input_parameters>
 <\Cost-function>

 <Benefit-function>
  <output_parameter ="request-rate","request-size", "read/write_ratio",
  "random/sequential_ratio" @ source>
     <input_parameters>
       <CONSTANT />
     </input_parameters>
 <\Benefit-function>
<\Action>
```

Figure 3.5: Specifications for the migration action(for simplicity we only enumerate the source component parameters).

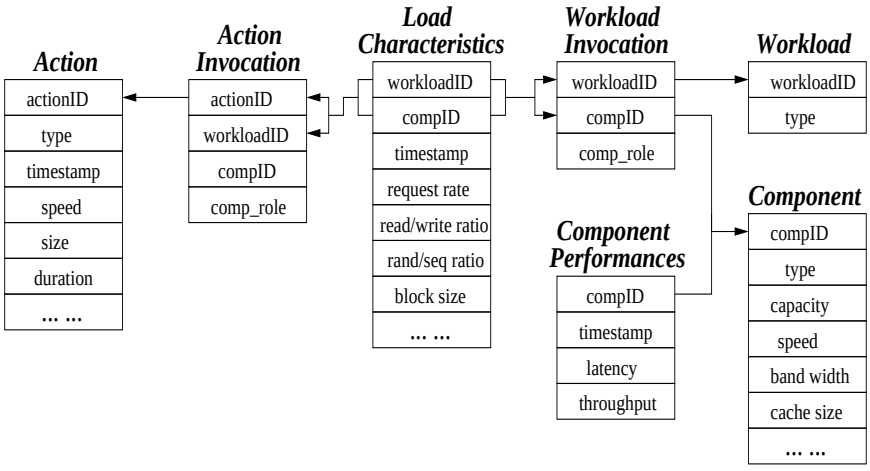


Figure 3.6: The schema of the performance logs.

LUS implements two approaches for regression – Support Vector Regression (SVR) [26] that is relatively easy to implement, and the traditional Neural Network [87] with back-propagation.

- The key idea of SVR is to find the balance point between the training error and the complexity of the function; in other words, it avoids finding complex functions with low error only on training data but high error using real-time data. SVR is able to identify linear functions, polynomial functions, and functions of arbitrary shapes as directed by user – it is usually inefficient for large datasets. In using SVR, POLUS uses a brute force approach to determine the function type (in case they are not specified by the designer). It applies different function forms to the data and chooses one with the "best-fit." The list of candidate functions used are: (1) linear (x), (2) quadratic ($x^2 + ax$), (3) power (x^a), (4) reciprocal ($\frac{1}{x}$), (5) logarithm ($\ln(x)$), (6) exponential (a^x), and (7) simple combinations of two of them, such as reciprocal linear ($\frac{1}{x+a}$).

- Neural networks can find functions of arbitrary shapes by adapting its network structure with the data. It is efficient and can perform reinforcement learning to adapt to changing environments.The structure of a neural network is shown in Figure 3.7. A neural network contains an input layer, one or more hidden layers, and an output layer.

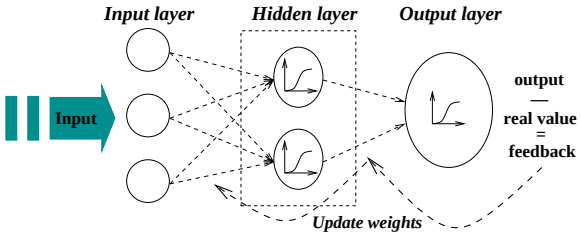


Figure 3.7: Adaptive learning of neural networks.

In summary, neural networks and support vector machines can both identify functions of arbitrary shapes. But they usually have better performances when the data can be well represented by some simple standard models such as linear, quadratic, etc. The time complexity for Neural networks should be linear to the data size (but usually it will iterate many rounds for optimization). The time complexity for support vector machines is quadratic w.r.t. number of data points.

3.4 Bootstrapping and Evolution of models

The initial baseline values for the action, workload, and component models are generated as follows:

- Component models: The initial values are generated either from the component's performance specifications provided either by the vendor, or by running calibration tests and measuring the component's behavior for different permutations of workload characteristics. The calibration tests generate I/O requests with different permutation of <request_size, read_write_size, random_sequential_ratio, queue_depth>. For each of the IO permutations, the iops, wait-time, and service-time counters are collected from the component.

- Action models: The effect of an action is mainly dependent on the implementation details of the actions rather than the deployment specific details. As such, the baseline values for the action models can be prepackaged by running in-house experiments to invoke the action for different workload characteristics and invocation parameter values.

- Workload models: The initial values of the workload models is based on libraries of workload characteristics for different applications such as e-mail, web-server, online-transactions, etc.

The models are continuously updated using the newly monitored data; this improves the accuracy of the regression functions (increasing the number of data-points that have been seen in the past), and also accounts for changes in the system (especially the workload models). Evolving models using neural networks is based on the difference between the predicted value and the actual monitored value; this difference is used for *back propagation* i.e. change the link weights between units of different layers. POLUS uses two approaches to evolve the models: 1) A computationally efficient approach is to invoke regression after every m additional data-points are collected from the system; this approach is used for the component and action models as they are relatively static compared to the workload models 2) Another approach is to update the model after every prediction; in this the difference between the predicted value and the actual value is used as an error-feedback to adjust the coefficient values in the model using re-enforcement based neural networks. The experimental section compares results of both these approaches.

3.5 Handling incomplete specifications

The system designer may not always provide a feature-set with all the parameters that have a first-order effect. Missing first-order parameters lead to inaccuracy of the models and appear as larger differences between the predicted value and the actual value. A data mining approach called *Iceberg Cubing* [14] is used for this purpose. The approach can be formally stated as: *Given a set of records with K parameters $x_1, \ldots, x_K$ and a target value y, find out all groups of at least m records that have identical or similar values on at least $K - \delta$*

parameters ($\delta = 1$ or 2). We say two values v_1, v_2 of parameter x_k are similar to each other if $v_1 - v_2 \leq \epsilon \cdot range(x_k)$.

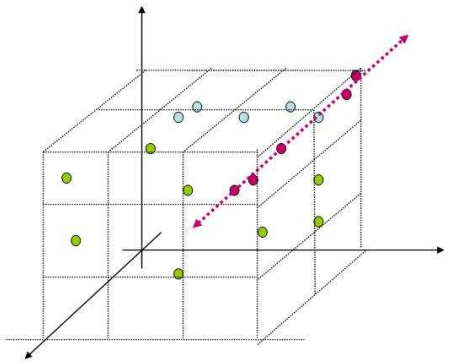


Figure 3.8: Data points and data cube.

To illustrate this, consider the designer-specifications are shown in figure 3.9. In these specifications, num_threads is not specified as a relevant parameter. POLUS uses Bottom-up computation (BUC) as an Iceberg Cubing algorithm, and its internal working is described as follows.

```
<component name="disk">
  <output_parameter ="IOPS">
    <input_parameters>
      <parameter name="RW_ratio" />
      <parameter name="SR_ratio" />
      <parameter name="block_size" func="linear" />
    </input_parameters>
  </output_parameter>
 </component>
```

Figure 3.9: Incomplete component specifications.

100 records are randomly selected and plotted in Figure 3.10. It is hard to judge whether num_thread and IOPS (output parameter) are related, when the effects of three other parameters are present.

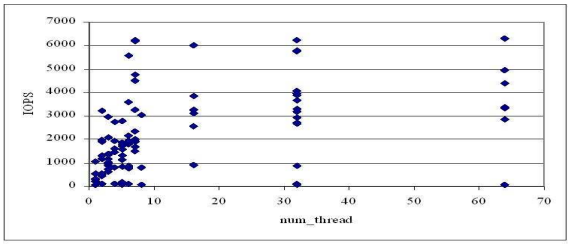
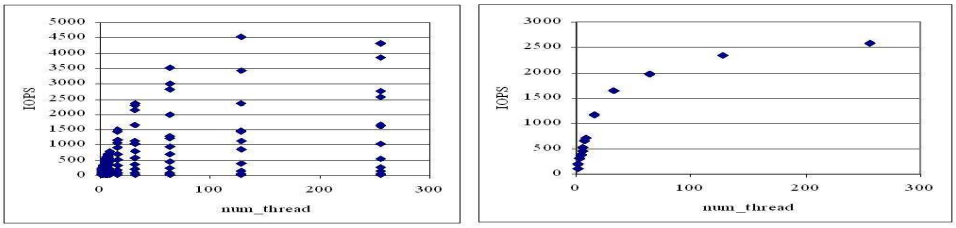


Figure 3.10: Plot of IOPS vs. num_thread.

As such, in order to identify the relationship between num_thread and IOPS, BUC finds all the records with a certain RW_ratio and SR_ratio (but different

block_size), and plot them in Figure 3.11 (a). From this plot it is clear that num_thread and IOPS are related, but it is still hard to find how they are related. In Figure 3.11 (b) BUC plots records with identical values on all parameters except num_thread, and it becomes obvious that IOPS is a sub-linear function of num_thread; regression techniques can be used to the exact function.



a) with different block sizes

b) with a certain block size

Figure 3.11: Plot of IOPS vs. num_thread by fixing the values of other parameters such as RW_ratio, SR_ratio.

In summary, the gray-box approach as proposed in this thesis is new to the domain of system management; there are a few interesting manifestations of the gray-box approach in other domains. For example the Snowball project [4] for information retrieval extracts information from the text; it starts off with initial sets of patterns, and recursively refines the patterns based on the input text. The details of the technique are tied to the domain of text extraction and analysis. Another similar concept is referred as *Lifelong Learning* [95] where the information is continuously evolved using hypothesis from previous learning tasks.

4 Reasoning for Corrective Action Selection

The knowledge-base defines models to predict observable behavior in different system states; at run-time, these models are used to select "feasible" corrective action(s) that converge the system behavior towards the administrator-defined SLO goals. This chapter describes the methodology for triggering action-selection, deciding invocation values of the action, and execution of corrective action(s). The outline of the chapter is as follows: it starts-off with a description of real-world requirements for action selection and execution. It then describes the methodology for action selection in a step-by-step fashion: enumerating the available corrective choices, analyzing the choices within the solution-space, techniques for pruning the choices and selecting a feasible action plan. The selected action(s) is then executed using a variably aggressive feedback loop – the step-size of the feedback loop is proportional to the confidence-value of the knowledge-base models. The chapter concludes with a discussion of techniques for triggering the reasoning engine.

4.1 Real-world Challenges for Automated Action Selection and Execution

The task of action selection and execution is non-trivial: First, corrective actions differ significantly in their invocation overheads, benefit to the system state, and time for completion. Second, the same action when invoked with different invocation parameter values exhibits different cost-benefit-time characteristics – the procedure for action selection must consider a significant subset of these choices, before making a decision. Third, the domain-specific models have inaccuracies in their predictions; these inaccuracies along-with unexpected run-time changes need to be handled during action execution. The rest of the section covers each of these challenges in detail.

Corrective actions drive the behavior of the system towards the desired SLOs. Actions differ in their operational semantics, and can be categorized along the spectrum of *short-term* actions that have a low invocation overhead and a short-lived benefit, to *long-term actions* that have a high invocation overheads and a relatively permanent benefit. In other words, short-term actions are analogous to *painkillers* – they do not rectify the root-cause of SLO violations, but rather perform light-weight tuning of the system parameters to solve the current SLO violation e.g., data-prefetching, throttling. Similarly, long-term actions are

analogous to *antibiotics* – they solve the root-cause of the SLO violation, and have a non-negligible invocation overhead e.g. data-migration, replication, etc. Selecting from the spectrum of short-term to long-term actions should take into account the *need* for action invocation (for a transient load spike, a long-term action should not be selected), and whether the system can *afford* the resource overhead of invoking the action.

Within storage systems, the OAA framework maintains the mapping of application workloads to the resources, and invokes corrective actions for modifying the mapping in response to changes in workload characteristics, exception events such as load surges, component failures, and growth trends in workload request rates. Let W represent the set of workloads in the system, and C represent the heterogeneous storage components of the system that differ in performance, capacity, reliability. There is a many-to-many mapping between W and C such that individual workloads w are mapped to one or more components c, and vice versa. The short-term and long-term corrective actions adapt the change workload-to-component mapping $R : W \rightarrow C$:

In a a given system state, the effect of action invocation is dependent on values assigned to the invocation parameters. An action can have a different cost-benefit effects for different permutations of values assigned to the invocation parameters e.g., the effect of migration action will be different for data-sets d_1 and d_2, being migrated to either component c_1 or c_2 – the permutations $(d_1, c_1), (d_1, c_2)(d_2, c_1)(d_2, c_2)$ are all treated as different corrective options. Thus there are a large of number of corrective options, proportional to the number of corrective actions, the number of invocation parameters for each of these actions, and the number of possible values that can be assigned to the invocation parameters. The OAA framework should be exhaustive in considering these choices as well as be computationally efficient in pruning the choices.

Finally, real-world system models have inaccuracies – it is not possible to take into account all the system parameters and the environment variables that are a part of the model. Also, the accuracy of these models is not constant; it depends on whether similar states have been seen in the past (in which case the accuracy of the models will be intuitively higher), and whether there are exception scenarios such as hardware failures, workload changes, etc. In the later case, the models will have a low accuracy, and might not reflect the changes in the system until several additional iterations of monitor data. The inaccuracies of models is reflected in the decision-making, resulting in imprecise invocation values for the actions, or a divergent system behavior from wrong a action selection. Thus the execution of actions in the OAA framework should have the ability to account for the inaccuracies of the models, as well as react to divergent system behavior resulting either from wrong action invocation or system changes that happen after the action is selected.

4.2 Methodology for Action Selection

Actions are invoked in response to violation of SLOs – the current implementation of Polus supports performance SLOs on a per-workload basis. A workload will be guaranteed a specified upper bound on average I/O latency, as long as its I/O rate (i.e., the throughput) 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 restricted performance; but in some extreme cases, well-behaved workloads may also need to be restricted. It should be noted that for a workload operating below its throughput limit, there is no straightforward approach to distinguish the scenario where the application does not have more requests to issue, from a scenario where an application wants to issue more requests, but the system is not able to support those – thus latency and not throughput is used as a trigger for SLO violation.

To illustrate the action selection methodology, we use two storage management actions as representative examples – throttling is representative of a short-term actions while data migration is representative of a long-term action.

4.2.1 Chaining models to predict action invocation effect

In order to invoke a corrective action, the OAA framework predicts the effect of action invocation on the system behavior. As mentioned earlier, the operation semantics of action invocation are captured as a change in component model, workload model, or workload access characteristics. This section describes of chaining the knowledge-base models to derive the impact of action invocation on the system behavior.

The metric for observed behavior in Polus is system performance i.e. the total number number of requests served by the system *(throughput T_i)*, and the response-time for each request *(latency L_i)* in workload W_i – a violation of SLO goals in production systems is generally triggered by latency violation. The sequence of steps in predicting the cost-benefit of an action on the observed system behavior is as follows:

1. Calculate the action model for the given invocation parameter values: Depending on the nature of the action, its cost and benefit are represented as knowledge-base models for *c_new*, *w_new*, or *w_access* – for the given invocation parameter values, the OAA framework calculates the value of these models. For example, in throttling, w_access_i represents the new workload request-rate for W_i as a function of the throttle value parameter t_i i.e. $w\_access_i = A_{throttle}(t_i)$ If the impact of the action is represented as *c_new*, step 2 is ignored.

2. Output of the action model piped into the workload model For each workload in the system, their access characteristics and component dependencies are re-calculated using w_new and w_access. For example, the new workload model for the throttled workload W_i is: Component_load$_{i,j}$ = $w_{i,j}(w\_access_i)$ where $w_{i,j}$ represents the load on component j originating from workload W_i.

3. Output of the workload model piped into component model This step calculates the response time of the individual components in the invocation path as a function of the total load on the component (i.e. summation of the workloads using the component).

$$total\_load_{compj} = \sum_k Component\_load_{k,j}$$

If the action impact was represented as c_new, the new component model is used for the calculating the response time.
$R_{Wi} = c(total\_load_{compj})$

4. Calculating the cost-benefit effect on system behavior

Cost represents the transient or permanent overhead of invoking the action. For example, the migration action has a transient overhead in copying data from the source to the target, while the replication action has the permanent overhead of updating all the replicas on every write operation. Cost is defined as the percentage increase in the workload latency multiplied by the workload priorities P_{W_i}.

$$\sum_{i=1}^{N} \frac{(R\_new_i - R\_current_i)}{SLO_{W_i}} SLO_{W_i}$$

Benefit of the action is defined by the change in the workload's distance from its respective SLO goal.

$$Distance = \sum_{i=1}^{N} \frac{(violation\_req\_rate - current\_rate)}{SLO_{W_i} P_{W_i}}$$

4.2.2 Generating the solution-space

Given that each action can be potentially invoked with infinite combination of invocation values, this step creates a finite set of choices that will be considered during action selection. We use constrained optimization techniques (such

as linear programming) to generate the $top-k$ invocation values for each action. Thus, in an OAA framework with m corrective actions, the solution-space consists of the order of km corrective options that will be considered.

To solve the problem using constrained optimization [], the formalism involves defining *variables, constraints, and objective functions.* The variables are the invocation parameters; the constraints are inequality equations based on the component, workload, and action models. The constraints also define either the current or predicted system usage (depending on whether the Reasoning engine is invoked proactively or reactively).The objective function is generally similar to the benefit metric used while evaluating the options. Internally the constraint solver uses dynamic programming to exhaustively explore different combinations of invocation parameters and generates the one that optimizes the value of the objective function. The top-K invocation values for each action are generated using different objective functions that vary along the spectrum of lowest-cost highest-benefit.

We illustrate the generation of the solution space by using throttling and migration action as examples.

Throttling action

The invocation parameter for throttling is the token issue rate for individual workloads, represented as t_i. The objective function is to maximize the number of workloads meeting their SLOs with a consideration of workload's application priority (P_{W_i}) and quadrant priority (P_{quad_i}) [1] to control the probability that workload is selected as a throttling candidate. The constraints are each workload's latency should be no bigger than the latency specified in the SLO.

Table 4.1 shows the linear programming formulation for deciding the optimal throttling values.

Variables:	t_i: the token issue rate of workload i.
Objective Function	$Minimize \sum_{i \notin \texttt{failed}} \left\| P_{quad_i} P_{W_i} \frac{SLO_{W_i} - A_{throttle_i}(t_i)}{SLO_{W_i}} \right\|$ where SLO_{W_i} is workload i's SLO IO rate and $A_{throttle_i}$ is workload i's throttling action model.
Constraints:	$latency_i \leq SLO_{latency_{W_i}}$ where $SLO_{latency_{W_i}}$ is the specified latency in SLO $t_i \geq 0$

Table 4.1: Formulation for generating throttle value options

[1] Workloads are pigeonholed into one of the four regions according to their current request rate, latency and SLO goals: `meet, failed, lucky` and `exceeded` and quadrant priorities are assigned accordingly. For more information, please refer to [99]

Migration

The invocation parameters for migration are: the data-set to migrate, the target component, and the speed of migration. The data-set and the target governs the permanent benefit of migration, while the migration speed controls the transient cost. We use separate constraint solving problems for the permanent benefit and the transient cost.

Deciding the migration speed

The invocation of data migration will transiently introduce additional load on the source and target components; this load can be treated as a additional workload stream in the system. Thus the selection of migration speed can be formulated as the throttling problem i.e. deriving the minimum throttle value of the additional migration workload at the source and target components. Ideally, the migration speed should as fast as possible, and should not cause normal workloads to violate their SLO. As such, the constraint solving also considers temporarily throttling low priority workloads (such as data backup) to have more bandwidth available for migration. The constraint solving for the migration speed is shown in Table 4.2.

Variables	t_i represents the token issue rate for normal workload i and m is the migration speed.
Objective function	$Minimize \sum_{i=1}^{N} \lvert P_{W_i} P_{quadrant_i} \frac{SLO_{W_i} - A_{throttle_i}(t_i)}{SLO_{W_i}}$ $+ P_{mig} \frac{max\_speed_{mig} - A_{mig}(m)}{max\_speed_{mig}} \rvert$ where the first part is exactly the same as in throttling. P_{mig} is the priority of migration process and is used to control the sets of workloads that can be sacrificed to accommodate migration operation. The max_speed_{mig} is the pre-specified maximum speed that the system is willing to support.
Constraints	$latency_i \leq SLO_{latency_{W_i}}$ $t_i \geq 0$ where $latency_i$ is calculated using equation **??** and the $total\_request\_rate$ in the equation is calculated by summing over component load from all workloads and migration workload if it exists.

Table 4.2: Migration Speed Selection – using the throttling formulation, with migration treated as an additional workload

Selecting the data-set and the target

The objectives for data-set and target selection are: (1) minimize the load variation across the possible targets; (2) minimize the size of data that is moved

(to avoid wasting bandwidth). Additionally, the selection should ensure that the target component can accommodate the migrated dataset without violating existing workload's SLO.

The target selection is formalized as a constrained solving problem as shown in Table 4.3. For each of the dataset, the constraint solving is used to select a target that can accommodate the data-set *(constraint)*, and minimizes the load variance across the candidate targets *(objective function)*. In case the constraint solver returns no feasible solution, the data-set is removed from the candidate list because no target can accommodate it without violating constraints.

Variables:	One for each candidate target: $s_j = 1$ means target component j is chosen as the new target and $s_j = 0$ means not selected.
Objective function	$Min \sum_j s_j((L_s - l^* - L_{mean})^2 + (L_j + l^* - L_{mean})^2$ where l^* is the load on the migrated data-set, L_s is the look-ahead load on the source before migration.
Constraints	$latency_{workload\ k\ on\ component\ j} \leq SLO_k$ $latency_{workload\ k\ on\ source} \leq SLO_k$ $latency^* \leq SLO^*$ $s_j = 0, 1$ $\sum_{j \neq source}^{M} s_j = 1$ Where the constraints guarantee that the existing workloads on the target, those on the source, and the migrated data-set can still meet their SLO.

Table 4.3: Migration Target Selection

4.2.3 Deciding the plan for invoking corrective action(s)

The previous steps focussed on deciding the invocation parameters of a single action. In a production system, the management software has to select between corrective options from multiple actions such as throttling, migration, replication, hardware provisioning. The current implementation of POLUS uses a heuristic approach for deciding the plan of action invocation – a detailed formulism for deciding the schedule of corrective actions is a complex problem and is a part of our ongoing research.

Selecting a corrective option is based on matching the system state with the cost-benefit attributes of the action – the outcome could be one of the following:

- Invoke a short-term action right-now
- Invoke a short-term action right-now and flag the a long-term action to be invoked at a later stage when resources are available
- Invoke a long-term action right-now

- No feasible solution – flag administrator for possibly adding hardware to the system

The action selection takes into account the system *needs* (in the current and lookahead states), and the spare resources it can *afford* for action invocation.

- *Is it a random spike?* This ensures that CHAMELEON does not make long-term changes in response random spikes and glitches; these are handled by invoking short-term actions. It represents the *need* for invoking the action.

- *How much overhead can the system afford?* This is based on the expected load pattern on the system for the next few hours and stipulates the maximum possible resources available for invoking actions (i.e. the maximum cost). CHAMELEON will select the highest benefit option available for the lowest cost.

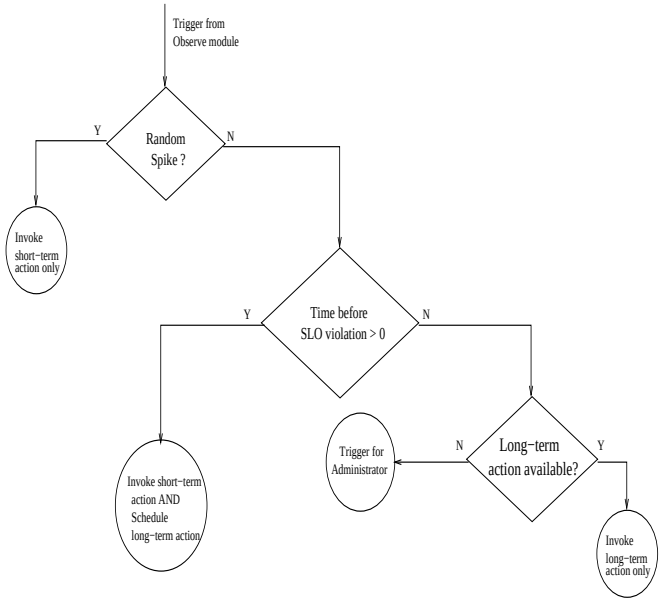


Figure 4.1: Flowchart for selecting plan for invocation of corrective actions

To illustrate the working of action selection, we describe three different invocation scenarios.

Scenario 1: Reactive Trigger for SLO violation

The SLO violation could be either because the the resources are completely saturated (especially after a failure) or there is a hot-spot in the resource utilization. As a first-step, CHAMELEON decides whether the violation is the result of a random spike. In that case, it invokes a short-term action, and does not flag for any future action.

In case, the violation is a result of a workload trend or events such as a hardware failure, CHAMELEON analyzes the load pattern of the system (*(if it is 9am on a weekday, the system should not invoke a long-term action because there will be a peak at 11 am), whereas if it is 8pm, the system is possibly already under a low load to afford a long-term action*). If none of the long-term actions can be accommodated in the current-state, CHAMELEON schedules a long-term action by making an entry in the log, and storing the results of the current analysis along-with the values of the system load. If none of the long-term options are available (generally indicates that the system is completely saturated); CHAMELEON sets a alert for the administrator.

Scenario 2: Proactive Trigger based on workload trends

This is similar to reactive trigger, with the exception that the action invocation is not required to be instantaneous. For these scenarios, CHAMELEON will always select a long-term option (if one is available). CHAMELEON can be made conservative in selecting long-term actions – this is accomplished by setting a low value for the bound that allows invocation of long-term actions (e.g. the load should be $x\%$ below average at the time of invoking the long-term action).

Scenario 3: Opportunity Window

This Analysis module is triggered when the system is lightly loaded (especially at night) – an opportunity to invoke long-term actions, in case they are necessitated. CHAMELEON traverses the log of delayed actions; for each of the delayed actions, it first verifies the workload models, component models, and workload trends for changes that might have taken place. In case of changes, it re-calculates the action to invoked.

4.3 Techniques for handling model inaccuracies during action execution

Models are continiously evolving with additional monitor data collected from the system – their accuracy is not constant. Further, their accuracy is dependent on whether: a) the system has seen similar states in the past, in which case the accuracy of the models will be intuitively higher; b) During exception scenarios such as hardware failures, workload changes, etc., the models will have a low accuracy, requiring several iterations to reflect the changes in the system. Besides imprecise knowledge, inaccuracies can stem from errors due to curve-fitting, and also from trying to use the models outside of the region(s) where they were trained (residuals). There are multiple statistical formulas to represent the confidence values [61] . POLUS uses the following formula to capture both the errors due to regression and the residuals.

$$S_p = S\sqrt{1 + \frac{1}{n} + \frac{(x_p - \overline{x})^2}{\sum x^2 - n\overline{x}^2}} \tag{4.1}$$

where S is the standard error, n is the number of points used for regression, and $\overline{x}$ is the mean values of the predictor variables used for regression. (S_p) represents the standard deviation of the predicted value in predicting the value of y_p using input variable x_p. In POLUS, the confidence value (CV) of a model is represented as the inverse of its S_p.

POLUS uses a feedback loop for action execution. The general guiding principle is to take radical corrective action as long as that is warranted by the available knowledge about the system. If the confidence value from the constraint solver is below a certain threshold (e.g., during bootstrapping of the models), POLUS invokes the administrator to make decisions. Otherwise, the feedback module applies decisions incrementally: the step size is a function of the confidence value. After each iteration, the feedback module has the option of continuing to apply prior decisions incrementally, or querying the reasoning engine to re-evaluate throttling decisions (e.g., when it observes abrupt changes in the system's behavior). Chapter 5 gives additional details of the feedback loop in the context of the storage management example.

4.4 Discussion: Triggering Corrective Actions

Existing approaches for triggering the reasoning engine are reactive – actions are invoked in response to violation of SLO goals. This approach is relatively simple to implement, but has the following limitations: 1) Long-term actions such as migration, replication, hardware provisioning, require a finite amount of time to complete – they cannot be invoked after the SLO violation has already occurred; 2)There are a certain amount of spare resources required for action invocation – resources that may not be available in an overloaded system e.g. re-distribution of resources by migrating data to a faster storage device is not possible in the current state when it is overloaded and violating its goals.

This section describes initial ideas related to proactive triggering for action invocation – invoking corrective actions before the SLOs are violated. Techniques for proactive triggers are based on detecting abnormal parameter values for the system parameters – abnormalities that will eventually cause the violation of SLOs. There are two key challenges in implementing proactive triggers: First, defining the notion of "abnormality" is non-trivial. Second, the impact of a parameter on system behavior needs to analyzed in the context of other parameter values – an abnormal value for a single parameter does not mean that the system is in a bad-state e.g. a high CPU utilization does not by itself imply an abnormality; the diagnosis should take into account domain-specific

details such as category of the parameter (CPU, I/O, Network), and the workflow dependencies between the components.

Existing approaches for abnormality detection and diagnosis are as follows:

- *Coarse-grained Thresholds Values*: This approach is the simplest to implement, and has a widespread use in commercial management frameworks. In this approach, abnormalities are detected by fixing threshold values for different system parameters e.g. a disk response time beyond a 100msec threshold will be considered an abnormality. The problem of these coarse grained values is that they are useful for mainly capturing failure induced abnormalities, rather than abnormalities in general.

- *Static Calibration Models*: This approach involves running calibration tests for creating models of system behavior, that are then used at run-time to determine abnormal behavior. The limitation of these approaches is that they are based on a static definition of "abnormality" that may not be accurate in a majority of system states. This is especially true in storage systems with: 1) Inherent non-linearity of storage components; 2) Changes in the workload characteristics govern the expected behavior; 3) Effects of workload interleaving, and cache allocation among different workload streams play a significant role in the component behavior - these effects are constantly changing at run-time.

As a part of POLUS, we are currently building PTRIGGER: framework for abnormality detection and diagnosis that: a) Proactively triggering the reasoning engine, before the system is actually in a bad-state; b) Self-evolves its definition of system "abnormality" by evolving the mapping of workloads to observable behavior; b) Is proactive in triggering the reasoning engine, before the system is actually in a bad-state. PTRIGGER monitors the system and internally stores the information as data-clusters (using the well-studied Sequential Leader Algorithm). These data-clusters serve as the mapping function for system behavior (e.g. response time) with the current state (e.g. workload characteristics). At run-time, the abnormality detection module analyzes the distance of the current monitored value w.r.t the nearest data-cluster for past values. This comparison is used to determine whether the parameter value can be flagged as abnormal.

5 POLUS Application: Putting it together

The focus of this chapter is to give an end-to-end picture for OAA automation using POLUS – it describes concepts, related work, design, and implementation of a resource-arbitrator for a production storage system (referred to as CHAMELEON). The POLUS approach as described in this thesis can be applied to a variety of generic system management problems – problems currently being solved using complex and brittle policy-based commerical tools [17, 58]; we conclude the chapter by enumerating a few additional examples.

5.1 CHAMELEON: A Self-evolving Resource Arbitration Module for Storage Systems

A typical consolidated storage system, in which multiple clients store and access petabytes' worth of data [80], serves the needs of various, independent, paying customers (e.g., a storage service provider) or divisions within the same organization (e.g., a corporate data center). Consolidation has proven to be an effective remedy for the low utilizations that plague storage systems [53], for the expense of employing scarce system administrators, and for the dispersion of related data into unconnected islands of storage. In the *utility* model, each client is guaranteed a portion of the shared resources regardless of whether other clients over- or under-utilize their allocations. Purchasing costs play a dwindling role relative to *managing* costs in current enterprise systems [53].

An OAA framework addresses the problem of allocating resources in a fully automated, cost-efficient way so that most clients experience predictable performance in their accesses to a shared, large-scale storage utility. Although performance is just one of the dimensions of Quality of Service (*QoS*), it is the most critical and less understood. Static provisioning approaches to providing performance isolation and guaranteed performance are far less than optimal, given the high variability (e.g., burstiness) of I/O workloads and the incomplete characterizations of storage device capabilities[28]. Furthermore, static resource allocations do not contemplate hardware failures, load surges, and workload variations; system administrators must currently deal with those by hand, as part of a slow and error-prone observe-act-analyze loop. Prevalent access protocols (e.g., SCSI and FibreChannel) and resource scheduling policies are largely best-effort; unregulated competition is unlikely to result in a fair, predictable resource allocation.

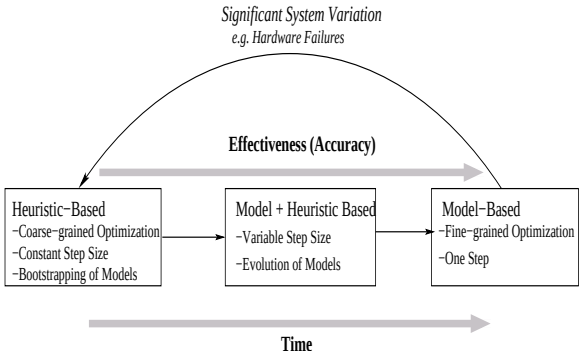


Figure 5.1: CHAMELEON moves along the line according to the quality of the predictions generated by the internally-built models at each point in time.

Previous work on this problem includes management policies encoded as sets of rules [59, 102], heuristic-based scheduling of individual I/Os [28, 62, 70, 57], decisions based purely on feedback loops[63, 32] and on the predictions of models for system components[9, 6, 11]. The resulting solutions are either not adaptive at all (as in the case of rules), or dependent on hard-to-obtain models, or ignorant of the system's performance characteristics as observed during its lifetime.

The POLUS approach demonstrates a novel technique for making automatic throttling decisions, based on a combination of performance models, constrained optimization, incremental feedback, and policies. CHAMELEONis a framework in which clients whose Service Level Agreement (*SLOs*) are not being met get access to additional resources freed up by *throttling* (i.e., rate-limiting) [28, 63] competing clients. Our goal is to take more accurate corrective actions as we learn more about the characteristics of the running system, and of the workloads being presented to it. As shown in Figure 5.1, CHAMELEON operates at any point in a continuum between decisions made based on relatively uninformed, deployment-independent heuristics, and on blind obeyance to models of the particular system being managed (Figure 5.1). This is done by having the throttling step size be a function of the statistical confidence of the models built automatically by CHAMELEON; when that confidence falls below a threshold, we revert to generic base heuristics.

5.1.1 Resource Arbitrator System Model

CHAMELEON is a framework for providing predictable performance to multiple clients accessing a common storage infrastructure, as shown in Figure 5.2. Multiple hosts connect to storage devices in the *back end* in such a way that CHAMELEON can monitor every I/O processed by the system, thus gathering information on the access patterns, throughput and latency. Each workload has a known SLO associated with it, and uses a fixed set of physical components (such as controllers, disks, switches), and logical components (such as logical volumes)

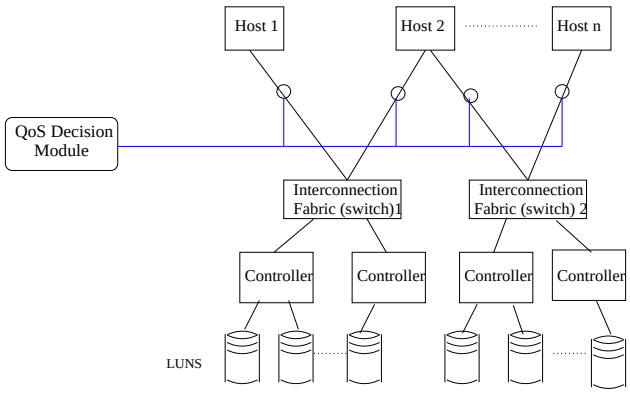


Figure 5.2: System model: CHAMELEON has access to performance data captured between the hosts and the storage back-end, and can effect throttling decisions at the same points. Such instrumentation points can be co-located with logical volume managers or block-level virtualization appliances [38].

that are together referred to as its *invocation path*. CHAMELEON detects workloads whose SLOs are not being met, and solves the violations by identifying and throttling workloads whose resource consumption should be curtailed. It also periodically checks for unused bandwidth, and selectively unthrottles some workloads.

Our SLOs are *conditional*: they specify maximum average I/O latencies over short sampling periods, as long as workloads request up to a maximum number of bytes and I/Os (throughput) during said periods. If workloads inject load into the system at more than the rate prescribed in their SLOs, the system is under no obligation of guaranteeing any bound on latency. Obviously, such rogue workloads are prime choices for resource restriction; but in some extreme cases, well-behaved workloads may also need to be restricted. Throttling is effected at the client hosts by the leaky bucket protocol [97] where each workload is given tokens every 10 ms., and I/O rates are averaged over a sliding window of 1200 s. for comparison with the SLO.

5.1.2 The Existing Rule-based approach for building resource arbitrators

In our version of the problem, a resource arbitrator manages the assignment of available storage resources to the host workloads. This mapping must ensure that a minimal number of workload fail to meet its SLO (a *QoS violation*). The resource arbitrator is invoked each time a QoS violation is detected. Upon invocation, the resource arbitrator attempts to bring the system to a state where a minimal number of SLOs are violated, by identifying workloads whose resource consumption should be *throttled*.

Choosing which workloads to throttle is a fairly complex task for many reasons. First, workload access patterns change constantly (e.g., as a result of

burstiness). The amount of resources freed up by throttling a given workload to a given degree is a dynamic function. Second, a workload's behavior may be related, at the application level, to that of other workloads or even human users. For instance, throttling accesses to a database log will affect other database-related workloads. The resource arbitrator needs to consider these dependencies when they exist. Third, each workload uses a fixed set of physical components referred to as its *invocation path*. The resource arbitrator should make sure that the workloads being throttled share either invocation path elements or application-level dependencies with the workloads that are experiencing QoS violations—otherwise their performance would be independent of one another, so throttling them would not help remedy the problem. Fourth, failures occur at unpredictable times. Even if data remains accessible due to built-in redundancy (e.g., RAID) performance will typically suffer because of the decrease in the overall amount of available resources. resource arbitrators need to adapt to these events within a fairly short time interval, reapportioning resources so that the system continues to satisfy the SLOs. Fifth, the resource arbitrator should be potentially able to throttle any subset of the workloads in the system (although doing so optimally is NP-hard); this results in an exponential number of possible decisions.

Many existing implementations of storage resource arbitrators are based on flavors of policy-based management [59, 77] where system behavior is described as a set of rules that are invoked when certain system conditions are met. Most rules are variations on the theme of Event-Condition-Action (*ECA*), with the semantics that the action will be executed if both a given type of event occurs (e.g., a violation of a given QoS metric) and a condition is satisfied. Let us consider writing a few example rules to define the behavior of a hypothetical resource arbitrator[1]. The set of rules can be divided into two categories:

Rules for selecting candidate workloads:
Workloads are throttled in increments of step-size.

Condition: If workload exceeds (1.6 SLO) $\wedge$ invocation-path(w) $\triangle$ invocation-path(w_{under_provi}
Action: Mark workload as candidate and *step-size* = 15%

Condition: If workload is between (1.25--1.6 SLO) $\wedge$ invocation-path(w)
$\triangle$ invocation-path($w_{under\_provisioned}$)
Action: Mark workload as candidate and *step-size* = 10%

Condition: If workload exceeds (1.15 SLO) $\wedge$ invocation-path(w) $\triangle$
invocation-path($w_{under\_provisioned}$)
Action: Mark workload as candidate and *step-size* = 3%

Rules for deciding which candidate workloads should be throttled:
The correlation between the workloads is represented as a *correlation probabil-*

[1]Predicate $A \triangle B$ is true iff sets A and B have a nonempty intersection.

ity cp, associated to each workload, that throttling this workload will indirectly throttle any other workload in the system.

Condition: If num_candidate_workloads $>$ 1 $\wedge$ $average\_cp_{candidate} < 0.4$ $\wedge$ variance_cp < 0.1
Action: Filter workloads with $cp > 0.8$; *then* Throttle all workloads

Condition: If num_candidate_workloads $>$ 1 $\wedge$ $average\_cp_{candidate} < 0.4$ $\wedge$ variance_cp > 0.3
Action: For workloads with $cp < 0.2$, throttle 85% of over_usage; For workloads with cp between $0.2 - 0.6$, throttle 45% of over_usage

Condition: If num_candidate_workloads $>$ 1 $\wedge$ $average\_cp_{candidate} > 0.6$ $\wedge$ variance_cp > 0.3
Action: Select workload with Minimum(cp)

The example highlights the main limitations of rule-based approaches for building resource arbitrators namely complexity, limitations of the expressive power, brittleness, and lack of adaptivity.

5.1.3 Other Approaches for building resource arbitrators

Feedback-based approaches use a narrow window of the most recent performance samples to make allocation decisions based on the difference between the current and desired system states. They are not well-suited for decision-making with multiple variables [94], and can keep thrashing between local optima. from Façade [70] controls the queue length at a single storage device. If there is a QoS violation, Façade decreases the target queue length, doing the equivalent of throttling the combination of all workloads down to the current request rate of the device; unlike CHAMELEON, it does not make complex decisions about which subset of the workloads should be left alone. Under overload conditions Façade will reduce its target queue length all the way down to 1, thus disallowing internal optimizations in the storage device and getting poor performance; CHAMELEON will provide differentiated service by throttling the low-priority workloads. Triage [63] keeps track of which performance band the system is operating in; it shares Façade's lack of selectivity, as a single QoS violation may bring the whole system down to a lower band (which is equivalent to throttling every workload). Sleds [28] can selectively throttle just the workloads supposedly responsible for the QoS violations, and has a decentralized architecture that scales better than Façade's. However, the policies for deciding which workload to throttle are hard-wired and will not adapt to changing conditions. Hippodrome [11] fine-tunes the initial data placement iteratively. Given the high cost of each data migration, it can take a long time to converge and may get stuck

in local minima as it relies on a variation of hill-climbing.

Scheduling-based approaches establish relative priorities between workloads and individual I/Os. Jin et.al. [62] compare different scheduling algorithms for performance isolation and resource-usage efficiency; their experimental results show that scheduling is effective but cannot ensure tight bounds on the SLO constraints (which is especially required for high-priority workloads). Stonehenge [57] uses a learning-based bandwidth allocation mechanism to map SLOs to virtual device shares dynamically; although it allows more general SLOs than CHAMELEON, it can only arbitrate accesses to the storage device, not to any other bottleneck component in the system. In general, scheduling approaches are designed to *optimize for the common case*, and may not be effective in handling exception scenarios such as hardware failures.

Model-based approaches depend on accurate models of the storage system in order to make decisions. Minerva [6] assumes that models are given—but system administrators very rarely have that level of information about the devices they use. Polus [98] proposes to build those models on the fly; CHAMELEON is an intermediate step towards the full Polus vision. The main challenge in this category is to acquire robust, accurate models—far from trivial for practical systems. In general, model construction has proved to be a difficult problem for both storage devices [91, 100] and workloads [7]. The subtle interplay between the rich set of characteristics I/O workloads exhibit and the optimizations that storage devices implement for them means that constructing analytical models is a complex, labor-intensive task. Some existing tools for storage system (re)configuration make automatic data placement decisions based on heuristics.

5.1.4 Bird's Eye-view of CHAMELEON

The core of CHAMELEON consists of four parts, as shown in Figure 5.3:

- **Knowledge base**: by taking periodic performance samples on the running system, CHAMELEON builds internal representations of system behavior without any human supervision; these we encapsulate using *black-box models*. Our black-box models are mathematical functions that quantify the capabilities of each component in the system, the demands placed by each workload on each component, and the reaction of each workload to different levels of throttling. Models get better as time goes by, for CHAMELEON refines them automatically; they may bootstrap from a *tabula rasa*, or from convenient oversimplifications (e.g., an M/M/1 queueing system) for faster convergence.

- **Model-based optimization**: CHAMELEON decides the workload throttle values using constrained optimization techniques such as piecewise linear programming. Our *constraints* compare load and performance as predicted by the models against the SLOs. Many possible administrator-defined

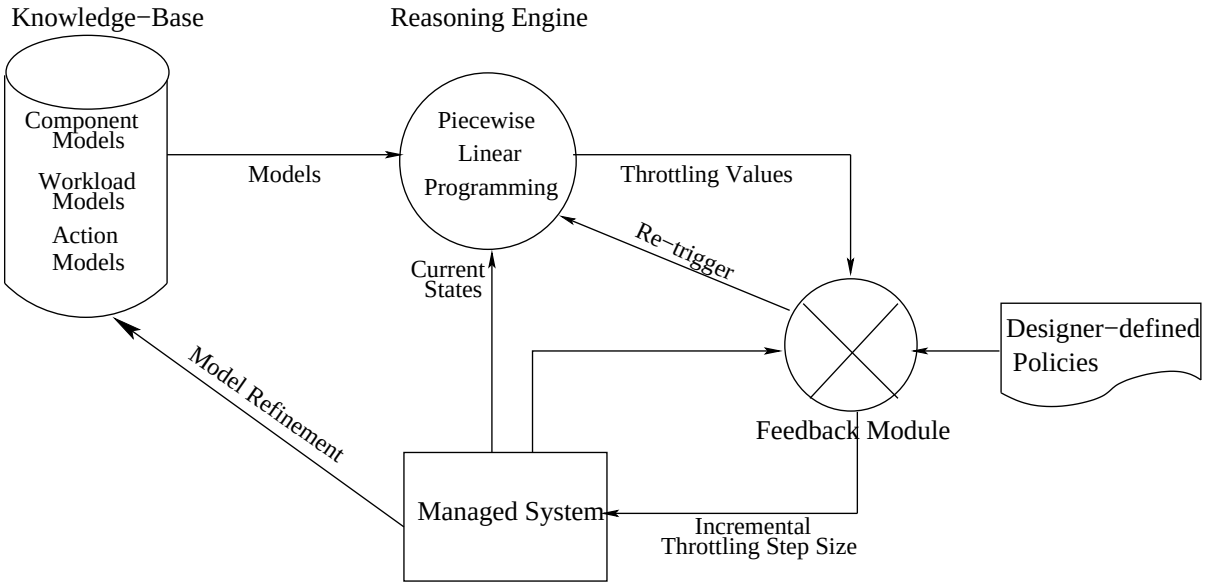


Figure 5.3: Architecture of CHAMELEON.

objective functions can be used to reflect the business goals of the storage utility, e.g., "minimize the number of SLO violations", or "ensure that highest priority workloads always meet their guarantees". Based on the errors associated with the models, the output of the constraint solver is assigned a *confidence value*.

- **System-designer policies**: As a fallback mechanism, we maintain a set of fixed heuristics specified by the system designer for system-independent, coarse-grain resource arbitration. Examples include "throttle all workloads sharing any component with the workload in trouble", or "greedily throttle lower-priority workloads as long as high-priority SLO violations exist".

- **Informed feedback loop**: The general guiding principle is to take radical corrective action as long as that is warranted by the available knowledge about the system. If the confidence value from the solver is below a certain threshold (e.g., during bootstrapping of the models), CHAMELEON falls back on the fixed policies to make decisions. Otherwise, the feedback module applies throttling decisions incrementally: the step size is a function of the confidence value. After each iteration, the feedback module has the option of continuing to apply prior decisions incrementally, or querying the reasoning engine to re-evaluate throttling decisions (e.g., when it observes abrupt changes in the system's behavior).

5.1.5 Prototype details: CHAMELEON

This section describes details for each of the four modules of CHAMELEON.

Knowledge base

CHAMELEON builds models in an automatic, unsupervised way. It uses them to characterize the workload being presented to the storage system, and the expected response of system components to different load types and intensities.

Models based on simulation or emulation require a fairly detailed knowledge of the system's internals; analytical models require less, but device-specific optimizations must still be taken into account to obtain accurate predictions [100]. *Black-box* models are built by recording and correlating inputs and outputs to the system in diverse states, without regarding its internal structure. We chose them because of properties not provided by the other modeling approaches: black-box models can evolve with changes in the component behavior, workload characteristics, and action effects, and they make very few assumptions about the phenomena being modeled. Because of this, black-box models are an ideal building block for an adaptive, deployment-independent management framework that doesn't depend on preexisting model libraries.

At the same time, the black-box models used in CHAMELEON are less accurate than their analytical counterparts; our adaptive feedback loop compensates for that. The focus of this paper is to demonstrate how several building blocks can work together in a hybrid management paradigm; we do not intend to construct good models, but to show that simple modeling techniques are adequate for the problem. CHAMELEON's models are constructed using Support Vector Machines (*SVM*) [26], a machine-learning technique for regression. This is similar to the CART [105] techniques for modeling storage device performance, where the response of the system is measured in different system states and represented as a best-fit curve function. Table-based models [9], where system states are exhaustively recorded in a table and used for interpolation, are not a viable solution as they represent the model as a very large lookup table instead of the analytic expressions that our constraint solver takes as input.

Black-box models depend on collecting extensive amounts of performance samples. Some of those metrics can be monitored from client hosts, while others are tallied by each component—and collected via proprietary interfaces for data collection, or via standard protocols such as SMI-S [92].

A key challenge is bootstrapping, i.e., how to make decisions when models have not yet been refined. There are several solutions for this: run a battery of tests in non-production mode to generate baseline models, or run in a monitor-only mode until models are sufficiently refined, or use a pre-packaged library. We follow different approaches for different model types; but in all cases models are incrementally refined from performance observations, while the level of confidence in their predictions increases. We proceed to discuss how models are represented internally, bootstrapped and refined from performance observations.

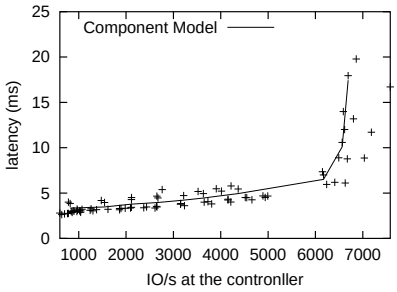


Figure 5.4: Component model.

Component models

A component model predicts values of a delivery metric as a function of workload characteristics. CHAMELEON can in principle accommodate models for any system component. In particular, the model for a storage device takes the form:
Response_time = $c(req\_size, req\_rate, rw\_ratio,$
$random/sequential, cache\_hit\_rate)$
Function c is inherently non-linear, but can be approximated as piecewise linear with a few regions. We obtain this representation using SVM, as shown in Figure 5.4. Another source of error is the effect of multiple workloads sending interleaved requests to the same component. We approximate this nontrivial computation by estimating the wait time for each individual stream as per a multi-class queueing model [61]; more precise solutions [19] incorporate different workload characteristics. The effects of caching at multiple levels (e.g., hosts, virtualization engines, disk array controllers, disks) also introduce additional errors.

We took the liberty of bootstrapping component models by running offline calibration tests against the component in question: a single, unchanging, synthetic I/O stream at a time, as part of a coarse traversal of c's parameter space.

Workload models

Representation and creation of workload models has been an active area of research [27]. 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, to predict the rate of requests at component i originated by workload j:
$\text{Component\_load}_{i,j} = w_{i,j}(workload\_request\_rate_j)$
In real scenarios, function $w_{i,j}$ changes continuously as workload j changes or other workloads change their access patterns (e.g., a workload with good temporal locality will push other workloads off the cache). To account for these effects, we represent function $w_{i,j}$ as a *moving average* [96] that gets recomputed by SVM every n sampling periods.

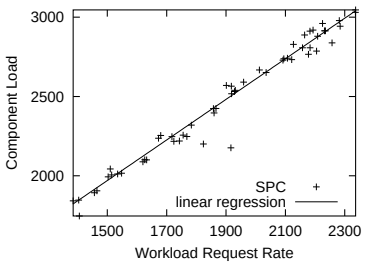


Figure 5.5: Workload model for SPC.

Figure 5.5 shows the workload models for the SPC web-search trace [35] running on a 24-drive RAID 1 LUN defined on an IBM FAStT 900 storage controller. From the graph, a workload request rate of 1500 iops in SPC translates to 2000 iops at the controller.

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

Action models

In general, action models predict the effect of corrective actions on workload requirements. The throttling action model computes each workload's average request rate as a function of the token issue rate, i.e.
Workload_request_rate = $a(token\_issue\_rate)$
Real workloads exhibit significant variations in their I/O request rates due to burstiness and to ON/OFF behaviors [19]. We model a as a linear function: $a(token\_issue\_rate) = \theta \times token\_issue\_rate$ where $\theta = 1$ initially for bootstrapping. This simple model assumes that the components in the workload's invocation path are not saturated.

To handle bursty workloads more realistically, we could have θ be a function of the request rates observed in the latest n sampling periods. This would maintain the Probability Distribution Function (PDF) of the request rate for each workload, and compute θ as a moving average of a given percentile.

Function a will, in general, also deviate from our linear model because of performance-aware applications (that modify their access patterns depending on the I/O performance they experience) and of higher-level dependencies between applications that magnify the impact of throttling.

Reasoning engine

The reasoning engine computes the rate at which each workload stream should be allowed to issue I/Os to the storage system. It is implemented as a constraint solver (using piecewise-linear programming [1]) that analyzes all possi-

ble 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".

It should be noted that the reasoning engine is not just invoked upon an SLO violation to decide throttle values, but also periodically to unthrottle the workloads if the load on the system is reduced.

Intuition

The reasoning engine relies on the component, workload, and action models as oracles on which to base its decision-making. Figure 5.6 illustrates a simplified version of how the constraint solver builds a candidate solution: 1) for each component used by the *underperforming* workload (i.e., the one not meeting its SLO), use the component's model to determine the change in request rate at the component required to achieve the needed decrease in component latency; 2) query the model for each workload using that components, to determine which change in the workload's I/O injection rate is needed to relieve the component's load; 3) using the action model, determine the change in the token issue rate needed for the sought change in injection rate; 4) record the value of the objective function for the candidate solution. Then repeat recursively for all combinations of component, victim workload, and token issue rates. The reasoning engine is actually more general: it considers *all* solutions, including the ones in which the desired effect is achieved by the combined results of throttling more than one workload.

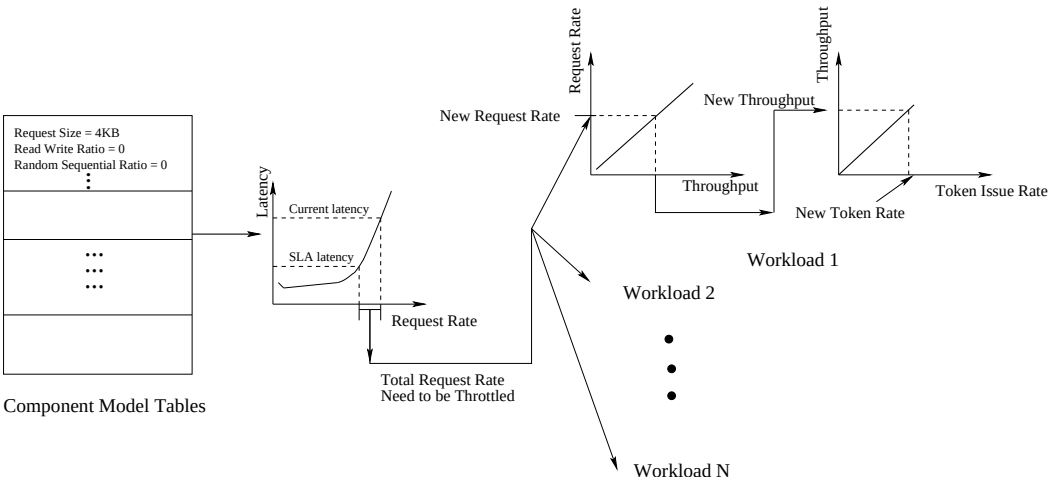


Figure 5.6: Overview of constrained optimization.

Formalization in CHAMELEON

To formalize the problem for constraint solving, we need to formulate the task of deciding throttle values in terms of variables, objective function, and constraints.

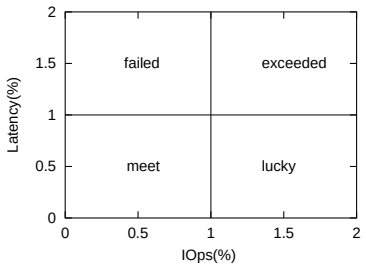


Figure 5.7: Workload classification. Region limits correspond to the 100% of the SLO values.

Variables

One per workload, representing its token issue rate:$t_1, t_2, \ldots$

Objective function

Workloads are pigeonholed into one of the four regions (Figure 5.7) according to their current request rate, latency, and SLO goals: `meet`, `failed`, `lucky`, and `exceeded`. Region names are self-explanatory—`lucky` denotes workloads that are getting a higher throughput while meeting the latency goal, and `exceeded` denotes higher throughput while violating the latency goal.

Many objective functions can be accommodated by the current CHAMELEON prototype (e.g., all linear functions); moreover, it is possible to switch them on the fly. For our experiments, we used

$$Minimize \sum_{i \notin \texttt{failed}} \left| P_{quadrant_i} P_{Wi} \frac{SLO_{Wi} - a_i(t_i)}{SLO_{W_i}} \right| \tag{5.1}$$

where P_{Wi} are the *workload priorities*, $P_{quadrant_i}$ are the *quadrant priorities* (i.e., the probability that workloads in each region will be selected as throttling candidates), and $a_i(t_i)$ represents the action model for Wi. Table 5.1 provides some insight into this particular choice.

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 $latency_{Wi} \leq SLO_{Wi}$ for all workloads W_i running in the system.

The value of $latency_{Wi}$ is estimated using the following chain of parameters: t ⇒ application_request_rate ⇒ component_request_rate ⇒ component_latency. Equivalently, $latency_{Wi} = c(w(a(t)))$.

For example, with only a single workload W_1 running in the system with its I/O requests are being served by a storage controller followed by physical disks

Intuition	How it is captured
The lower a workload's priority, the higher its probability of being throttled	The solver minimizes the objective function; violating the SLO of a higher priority workload will reflect as a higher value for $P_{Wi}\frac{SLO_{Wi}-a_i(t_i)}{SLO_{W_i}}$
Workloads in the `lucky` or `exceeded` region have a higher probability of being throttled	This is ensured by the $P_{quadrant_i}$ variable in the objective function; the `lucky` and `exceeded` have a higher value compared to the other regions (for example $P_{meet} = 1, P_{exceed} = 8, P_{lucky} = 32$). It is also possible to define $P_{quadrant_i}$ as a function.
Operating workloads closer to the SLO boundary	This is ensured by the difference of the current throughtput and SLO-value in the objective function; it is possible to assign a value function for workloads operating beyond their SLO using a bimodal objective function (currently that value is zero).

Table 5.1: Internals of the objective function.

is represented as the following constraint:

$$c_{controller}(w_{1,controller}(a_1(t))) + c_{disks}(w_{1,disks}(a_1(t))) \leq SLO_1$$

In general, multiple workloads will share the components. As such, a more realistic example is:

$$c_{controller}(total\_req_{controller}) \; + \; c_{disks}(total\_req_{disks}) \leq SLO_{W1}$$

$$total\_req_{controller} = w_{1,controller}(a_1(t_1)) + w_{5,controller}(a_5(t_5))$$

where w_1, w_5 are the workloads sharing the storage controller.

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 will be unthrottled.

In CHAMELEON, the goal of unthrottling is to re-distribute the unused bandwidth of the storage system based on the priority values and the average I/O rates. If a workload is consistently wasting tokens issued for it (because it has less significant needs), the additional tokens will be considered for redistribution; 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. Unthrottling decisions are constructed using the same objective function, but with additional "lower-bound" constraints such as not allowing each I/O rate to become lower than its current average value.

Designer-defined Policies

The system designer defines heuristics as a coarse-grained control mechanism, for deciding the workloads to be throttled; this is required in scenarios where the predictions of the models cannot be relied upon (either during bootstrapping or after significant system changes such as hardware failures). For example, "a component is saturated if its utilization is greater than 85%", or "start throttling workloads in the `lucky` region", or "if the workload-priority variance is less than 10%, uniformly throttle all workloads sharing the component". These heuristics can be expressed in a variety of ways such as Event-Condition-Action (ECA) rules or hard-wired code.

Coming up with useful throttling heuristics is a highly complex problem [98], especially if they are to consider a useful fraction of the solution space and to accommodate priorities; this is an error-prone process that is sure to result in a brittle set of policies (especially with respect the threshold values changes with the underlying physical configuration).

In CHAMELEON, the designer-defined heuristics are implemented as simple hard-wired code which is a modified version of the throttling algorithm used in Sleds [28]:

1 Determine the components being used by the underperforming workload and generate a *compList*.

2 For each component in the *compList*, determine the non-underperforming workloads using the component and add them to the *candidateList*.

3 Within the candidateList, order the workloads into groups based on their current operating quadrant: `lucky`, then `exceed`, then `meet`. Within each group, order the workloads based on their priority values.

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

Informed Feedback

Figure 5.8 shows the working of the feedback module. The feedback module incrementally throttles workloads based on the decisions of either the reasoning engine or the system-designer heuristics (when the confidence value of the reasoning engine is below a certain threshold value, say 30%). The step size for the incremental throttling is proportional to the confidence value of the constraint solver or is a small constant value while using designer heuristics.

After every m incremental throttling steps, the feedback module analyzes the state of the system. If any of the following conditions is true, it re-invokes the reasoning engine (otherwise it continues applying the same throttling decisions in incremental steps):

- Latency increases for the underperforming workload (i.e., it moves away from the meet region).
- A non-underperforming workload moves from the meet or exceed region to the lucky region.
- Any workload undergoes a 2X or greater variation in the request-rate or any other access characteristic (compared to the values at the beginning of throttling).
- There is a 2X or greater difference between predicted and observed response times for a component.

After the reasoning engine is invoked consecutively for l times and has a lower confidence value in each invocation, the feedback module discards the throttle values and switches to the designer heuristics. There can be multiple reasons for the above-mentioned conditions to be true: component failures, application-level correlations between workloads, unpredictable variations in the component behavior.

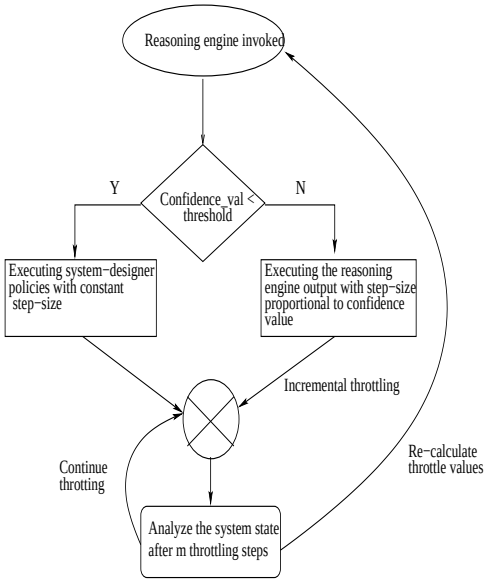


Figure 5.8: Working of the feedback module

5.1.6 Summary of the CHAMELEON Prototype

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 action, and what actions to take—thus allowing them to concentrate on specifying the global objectives that maximize the storage utility's business benefit, and having the system take care of the details. No

existing solution satisfies these criteria; prior approaches are either inflexible, or require administrators to supply up-front knowledge that is not available to them.

Our approach to identifying which client workloads should be throttled is based on constrained 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. CHAMELEON's approach to model building results in a solution that requires no prior knowledge about the quantitative characteristics of workloads and devices—and that can make good decisions even in the presence of realistic scenarios, like those involving workloads with relative priorities. The objective function being optimized can be defined, and changed, by the administrator as a function of 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 insufficient knowledge is available.

We replayed traces from production environments on a real storage system, and found that CHAMELEON makes very accurate decisions for the workloads examined. CHAMELEON always made the optimal throttling decisions, given the available knowledge. The times to react to and solve performance problems were in the 3-14 min. range, which is quite encouraging.

As areas for future work, first, we can improve the quality of model representations and the processes used to build them: component models could account for phased workloads and accurate interleaving, and workload models could incorporate additional workload characteristics such as temporal locality (and even incorporate some degree of prediction using techniques related to ARIMA [96]). Second, the reasoning engine could be based on a more general type of optimization, e.g., use non-linear programming for the constraint solver as supported by OPT++ [75]. Finally, we could account for a variety of additional real-world aspects: preventing over-fitting in models, avoiding oscillations or ping-pong effects, or even generating explanations for the administrator for the throttling decisions made by CHAMELEON.

5.2 Example Applications

POLUS is a model-based approach with the ability to refine models in an online fashion, and vary the aggressiveness of action execution based on the accuracy of these models. POLUS is an alternative to complex and brittle policies, uninformed feedback, and non-converging learning-based techniques. This section a few additional applications of POLUS, that are being currently implemented as a prototype.

5.2.1 Automated Management of a Distributed File-system

Capacity planning, application/storage performance management, backup/restore operations, configuration management, security, and availability analysis are some of the key storage management responsibilities of a system administrator. Typically, storage administrators write scripts that automate many of these storage management tasks. As the number of business service level agreements, department policies, QoS goals, storage devices, protocols, applications, and users increases, it becomes difficult for system administrators to ensure performance, provisioning, availability and security goals by using ad hoc script writing approaches . Storage vendors are trying to add sophisticated systems management functionality into databases, file systems, storage controllers, storage resource managers, storage area network managers, capacity planning managers and other storage management software. The major focus of these products is to reduce management complexity by allowing a system administrator to specify high level QoS goals with respect to expected performance,availability, provisioning, and security, and to automatically transform these high level QoS goals into low level system actions.

Currently, this transformation process is built using the policy-based paradigm, where policies are specified as collection of rules that are in the ECA format (Event-Condition-Action) [59]. Rules define how the system behaves for different possible system states and goal values. At run-time, the management module simply invokes the rule that is applicable based on the event and system condition. Even though goal based storage management approach has been advocated as the silver bullet that can help to reduce the management complexity for system administrators, this approach has not gained much traction because current policy management frameworks are providing support for only simple and trivial storage management scenarios. The POLUS framework takes away the complexity of writing policy mapping code from human experts and moves it to a combination of a knowledge-base and reasoning engine.

5.2.2 Workflow-aware and Resource-aware Scheduling in Information Retreival Engines

Information Retreival Engines [**?**] are responsible for parsing data collected from sources such as the websites, blogs, chat logs, news sites, and extracting the user-specified semantic information. These engines run on one or more computer clusters with hundred of nodes – they internally consist of multiple data-classifiers (a.k.a. miners) that analyze data for different key-value permutations. Scheduling the tasks of the engine to run efficently on a cluster of hetergoneous machines is a non-trivial problem. The traditional scheduling techniques (such as those based on shortest-job-first, first-come-first-serve, round-robin) are not

useful in this domain because of the following reasons:

- Classifiers have application-level dependencies (a.k.a. workflow dependencies) such that the output of one classifier is an input for the other. As such, the engine needs to coordinate/synchronize their results, as it makes no sense for one classifier to analyze the 1000th web-page while the dependent classifier is still on its 10th web-page (producing information faster than it can be consumed).

- Each classifier exhibits different resource usage patterns. For example, few classifiers might be very memory intensive causing large amounts of data to be paged-in, while a few other classifiers might be IO intensive. From the scheduling perspective, two classifiers are considered to be equivalent if they both exhibit similar resource usage patterns.

In summary, the objective is to build a scalable, self-evolving framework for scheduling classifiers that maximizes the hardware utilization, taking into account the resource-usage patterns of the classifiers as well as their workflow dependencies.

Using a POLUS-based approach, the scheduler framework will be designed as follows:

- A regression-based approach for creating self-evolving models for the resource usage pattern of the individual classifiers

- The task of scheduling is formalized as a constrained optimization problem that exhaustively searches the solution space for the most optimal answer; the optimizations account for both the workflow dependencies as well as the resource-usage patterns – these are defined as constraints in the solver

- Adapts the aggressiveness of the scheduling decisions based on the confidence values of the workflows and resource-patterns of the classifiers

6 Experimental Evalutation

This chapter describes the experimental evaluation of POLUS – an assessment of the proposed approach to select feasible corrective actions in different system scenarios. The experiments were performed using two different POLUS prototypes. The first protoype is of CHAMELEON: a self-evolving resource arbitrator for storage systems (details in chapter 5) – the evaluation consists of running CHAMELEON on a real-storage system, with real-world and synthetic worloads as the load generators. The second prototype is an implementation of POLUS using ABLE (Agent Building and Learning Environment) [15]; this prototype was the initial proof-of-concept, and used for managing a SAN file-system simulator – the experimental evaluation compared the complexity of specifications, and brittleness of the domain details to the popular rule-based approach.

The experimental results were quite encouraging. By striking a balance between sub-optimal policies, un-informed feedback control, and model-based predictions, CHAMELEON was able to react to workload changes in a nimble manner, resulting in a marginal number of QoS violations. In our experiments on a real storage system using real-world workload traces, CHAMELEON managed to find the set of throttling decisions that yielded the maximum value of the optimization function, while minimizing the amount of throttling required to meet the targets and while satisfying the QoS requirements of most clients – it identified, analyzed, and corrected performance violations in 3-14 minutes, which compares very favorably with the time a human administrator would have needed. Since it does not depend on prior knowledge about devices or workloads, our approach can be easily deployed on large-scale storage systems that service variable workloads on an ever-changing mix of device types.

The outline of the chapter is as follows. It starts-off with a description of the experimental test-bed for CHAMELEON, evaluation results using a range of synthetic workloads as well as real-world workload traces (namely SPC and CEllo96), a discussion of the observed results. The rest of the chapter gives details of the ABLE prototpe for POLUS, details of the file-system simulator, the evaluation metrics for comparing POLUS with a rule-based approach, and the results of the comparision.

6.1 Experimental Evaluation of CHAMELEON

The experimental setup consists of a host machine generating multiple workload streams that are being served by a storage infrastructure. The host is an IBM x-series 440 server (2.4GHz 4-way with 4GB memory running Redhat Server 2.1 kernel); the back-end storage is a 24 drive RAID 1 LUN created on a IBM FAStT 900 storage controller (dual HBA) with 512MB of on-board NVRAM. The host and the storage controller are connected using a 2Gbps FibreChannel (FC) link.

The IO generation is controlled by a token-based leaky bucket protocol i.e. a token is required to issue an IO request to the storage system. The number of tokens issued to each workload stream is controlled by CHAMELEON that is running on the host machine as a separate process. The RAID 1 logical volume is mapped at the host as a `raw` device; as such there is no IO caching at the host-level.

The key capability of CHAMELEON is to regulate resource load so that SLAs are achieved. The experimental results use numerous combinations of synthetic and real-world request streams to evaluate the effectiveness of CHAMELEON; synthetic workloads are easier to handle compared to their real-world counterparts that exhibit a bursty and highly variable access characteristics. In addition to the effectiveness, we evaluate the computation complexity of the constraint solver as a function of the number of workloads.

6.1.1 Using synthetic workloads

The synthetic workload specifications used in this section were derived from Minerva's performance study [6]. Since the workloads are relatively static and controlled, the models for action and workload have a small error rate. In this experiment, the component model has $r = 0.72$, the workload and action models with $r > 0.9$. (r is the correlation coefficient [61] and represents the accuracy of the models; the closer r is to 1, the more accurate the models are.)

These tests serve two objectives. First, they evaluate the correctness of the decisions made by the constraint solver i.e. the throttling decisions should take into account the workload priorities, current operating point compared to the SLA, and the percentage of load on the components generated by the workload. Second, the tests depict the effect of model errors on the output values of the constraint solver and how incremental feedback helps the system converge to an optimal state.

Test 1: Workloads with equal priorities

Figure 6.1 shows the latency and throughput values for the four workloads W_1, W_2, W_3, and W_4 running on the system. Initially the system runs in uncontrolled phase (till t=60 sec); in this phase workload W_1 is violating its SLA

Workload	Request Size (KB)	Read Write Ratio	Sequential Random Ratio	Footprint Size (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

Table 6.1: Synthetic workload streams

while other workloads such as W_3 and W_4 are above their SLA. CHAMELEON calculates the new token issue rate for each workload and executes them incrementally with a step-size proportional to the confidence value (step-size in this case is 12%). The settling time between each incremental step is 60 sec. At time t = 300 sec, the system converges to a state where no workload is violating its SLA.

The final state of the system can be represented in the SLA quadrant(figure 6.2), where the arrows represent the new operating point after throttling. Figure 6.2 compares the system state that would have been achieved if the output throttle values [1] of the reasoning engine were directly executed; the throttle values cause over-throttling with workload W_1 operating much further in the meet region and no workload in the exceed region. Over-throttling is caused by model errors (in this case component model) where the predicted latency used by the reasoning engine was higher compared to the actual observed value.

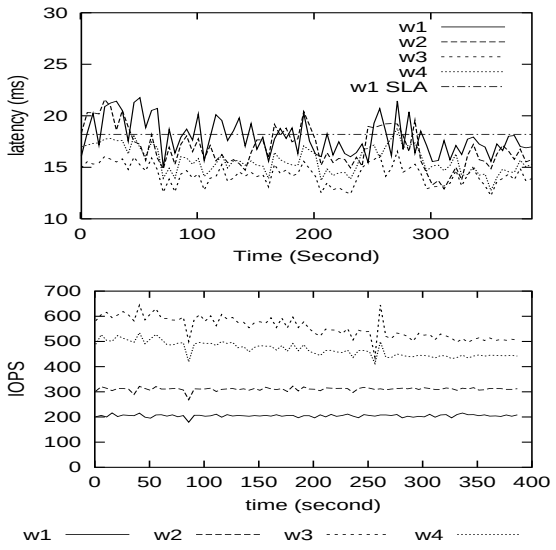
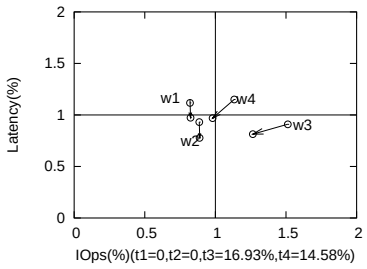
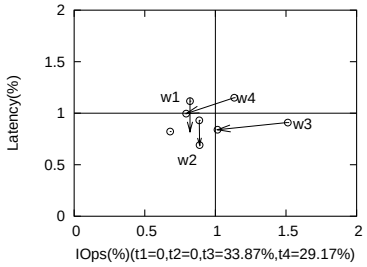


Figure 6.1: Throughput and latency values for synthetic workloads with equal priorities. Throttling of workloads starts at t=60 sec.

[1] The throttle values shown at the bottom of SLA quadrant figures represents the percentage decrease in the token issue rate



Final system-state with incremental throttling

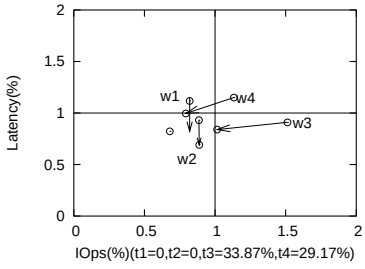


Final system-state if the output of the reasoning engine is directly executed

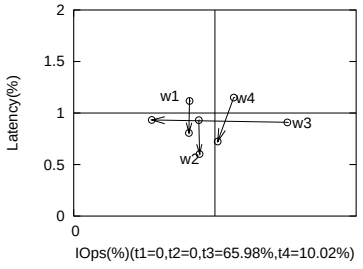
Figure 6.2: Effect of model errors on output of the constraint solver.

Test2: Effect of workload and quadrant priorities

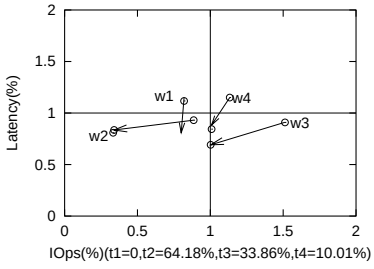
Figure 6.3 compares the direct output of the constraint solver with priority values for the workloads ($W_1 = 8, W_2 = 16, W_3 = 2, W_4 = 8$) and the SLA quadrants. Compared to the no priority case, workload W_3 is throttled more aggressively. This is because the constraint solver internally uses a greedy algorithm, throttling the lowest priority workload before moving to the higher ones.



Equal priority



Workload priorities



Quadrant priorities

Figure 6.3: Effect of priority values on the output of the constraint solver.

Test 3: Usage of the component by the workload

This test is a sanity check with workload W_5 operating primarily from the controller cache (not using the disk bandwidth). To solve the SLA violation for workload W_1, the reasoning engine shouldn't select W_5 for throttling even

if W_5 has the lowest priority. The throttling decision made by CHAMELEON is as shown in figure 6.4 which indicates the reasoning engine selects W_2 and W_3.

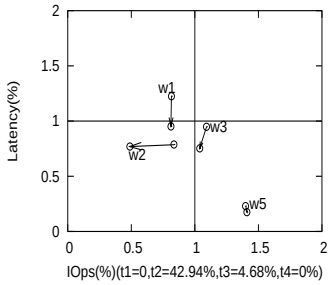


Figure 6.4: Sanity test for the reasoning engine (workload W_5 operating from controller cache

6.1.2 Using real-world workload trace replay

In these experiments, we replay the real-world workload for SPC web-search [35] and HP's Cello96 [2] traces. Cello96 was collected from a departmental fileserver over the period from 9 September to 29 November 1996. Both these traces are block-level with timestamps recorded for each IO request. The trace duration for SPC is around 6 hours while that for Cello96 is 1 day long. To generate a reasonable IO load for the storage infrastructure, the SPC traces were replayed 40 times faster and the cello traces were replayed 10 times faster.

For these tests, in addition to the SPC and Cello96 traces, a phased synthetic workload was used; this workload is assigned the highest priority. In an uncontrolled case i.e. without throttling, with three workloads running on the system, one or more of them violate their SLA. Figure 6.5 shows the throughput and latency values for uncontrolled case. The horizontal line in the delay figures represents the SLAs for each workload. As we can see from the figure, when the synthetic workload is turned on, the SLA on latency were violated.

The aim of the tests is to evaluate the following:

- The throttling decisions made by CHAMELEON for converging the workloads towards their SLA.
- 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 predictions, forcing CHAMELEON to use the sub-optimal but conservative designer-defined policies.

For these experiments, the models were reasonably accurate (component $r = 0.68$, workload $r = 0.7$, and action $r = 0.6$).In addition, the SLAs for each

[2] http://tesla.hpl.hp.com/public software

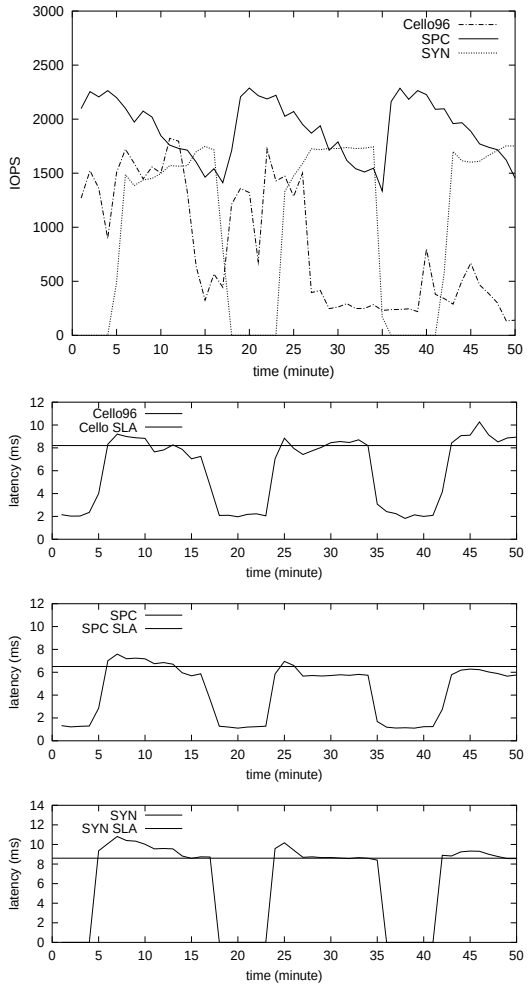


Figure 6.5: Uncontrolled throughput and latency values for real-world workload traces

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

Case 1: Solving SLA violation using throttling

The behavior of the system is shown in figure 6.6. To explain the working of CHAMELEON, we divide the time-series into phases described as follows:

Phase 0 (t=0 to t=5 min): Only the SPC and Cello96 traces are running on the system; the latency values of both these workloads is significantly below the SLA.

Phase 1 (t= 5 min to t= 13 min): The phased synthetic workload is introduced in the system. This causes an SLA violation for the Cello96 and synthetic traces. CHAMELEON triggers the throttling of the SPC and Cello96 work-

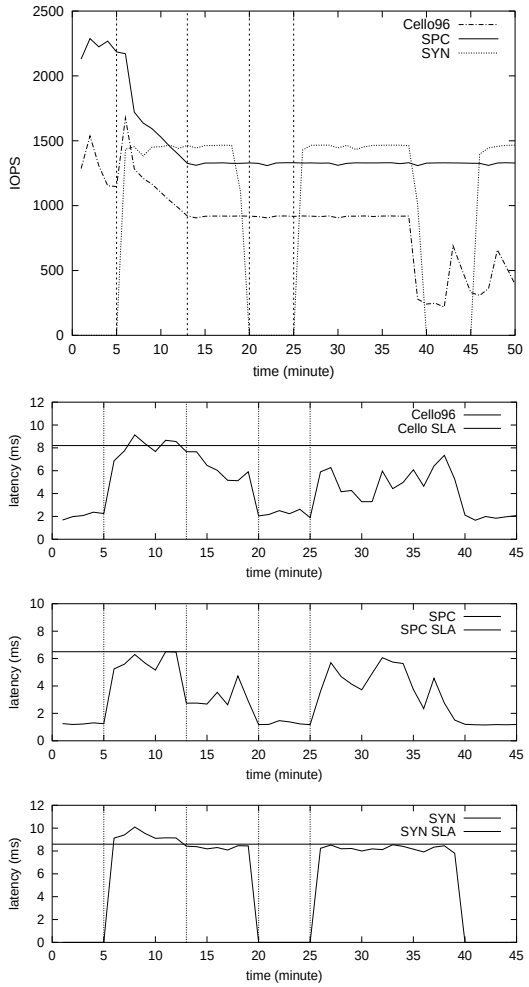


Figure 6.6: Throughput and latency values for real-world workload traces with throttling (without periodic unthrottling)

loads (Cello96 is also throttled because it is operating in the exceeded region, means it is sending more than it should. Therefore, it is throttled even its SLA latency goal is not met). The system uses a feedback approach to move along the direction of the output of the constraint solver. In this experiment, the feedback system starts from 30% of the throttling value and uses step size is 8%. (30% and 8% are decided according to the confidence value of the models). It took the system 6 minutes to meet the SLA goal and the feedback stops.

Phase 2 (t=13 min to t= 20 min): The system stabilizes after the throttling and all workloads can meet their SLAs.

Phase 3 (t=20 min to t= 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 t= 25 min): The system is stable, with all the workload meeting their SLA. A side-point is that at around t=39 min, the throughput of Cello96 decreases further; this is because of the inherent trace characteristics.

Figure 6.6 shows the effectiveness of the throttling: all workloads can meet their SLA after throttling. However, because the lack of unthrottling scheme, throttled workloads have no means to increase their throughput even when tokens are released by other workloads. Therefore, the system is underutilized.

Case 2: Side-by-side Throttling and Unthrottling of workloads

Figure 6.7 shows throttling of workloads with periodic unthrottling (every 60 sec) during reduced system loads. In comparison to Figure 6.6, there are four interesting observations:

- First, the behavior of system during the off phase of the synthetic workload (t=17 min to t=27 min). In this duration, the system load is reduced that triggers unthrottling of the SPC and Cello96 workloads. Unthrottling is based on workload priorities and the average IO demand of the individual workload streams. The SPC and Cello96 grab more tokens and are sending out accumulated requests due to limited tokens when the synthetic workload is on.

- Second, the settling time required to throttle the SPC and Cello96 workloads, whenever the synthetic workload gets into the on phase (t= 27 min to t=37 min, t=47 min to 50 min). Compared to the throttling-only scenario, the throughput and latency variations of the system are higher, taking a longer time to stabilize.

- Third, the constraint solver made a different throttling decision from Case 1: Cello96 was not throttled. This is because when the reasoning engine is triggered, the Cello96 was sending less than its SLA IOPS and was not meeting its SLA latency goals (t=9 min). As a result, both Cello96 and the synthetic workload were operating in the `failed` region, therefore, the reasoning engine will not throttle Cello96 as in Case 1 and only SPC was throttled.

- Fourth, between t=10 min to t=13 minutes, reasoning engine was triggered twice. This is because all workloads met their SLA goals after the first throttling (t=11 min) and the feedback stops. However, at t=12 min, SLAs for Cello96 and the synthetic workloads were violated again, a second call on reasoning engine was triggered and SPC was throttled again. After t=13 min, the system was stabilized.

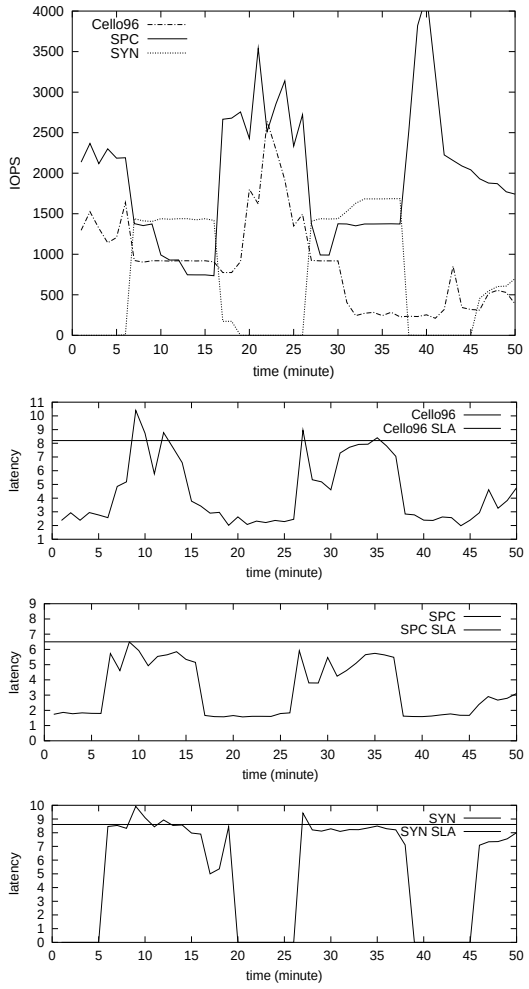


Figure 6.7: Throughput and latency values for real-world workload traces with throttling and periodic unthrottling

Case 3: Handling changes in the confidence value of the models at run-time

This test demonstrates how CHAMELEON caters to change in confidence value of the models at run-time; this change can be due to unpredictable system variations (hardware failures) or un-modeled properties of the system (such as changes in the workload access characteristics that change the workload models). It should be noted that refining the models to reflect the changes will not be instantaneous; in the meantime, CHAMELEON should have the ability to detect a fall in the confidence value and switch to a conservative management mode (using designer-defined policies or generate a log message for a human administrator).

Figure 6.8 show 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 (in reality, a similar scenario arise due

to changes in the cache allocation to individual workload streams sharing the controller) and the SPC is doing more random access (sequential random ratio increases from 0.11 to 0.5). In the future, we plan to run experiments with hardware failures induced on the RAID 1 logical volume.

The SLAs used for this test are: Cello96 has a SLA with 1000 IOPS with 7ms latency, SPC is 2000 IOPS with 8.8ms latency and the synthetic workloads has a SLA with 1500 IOPS and 9ms latency.

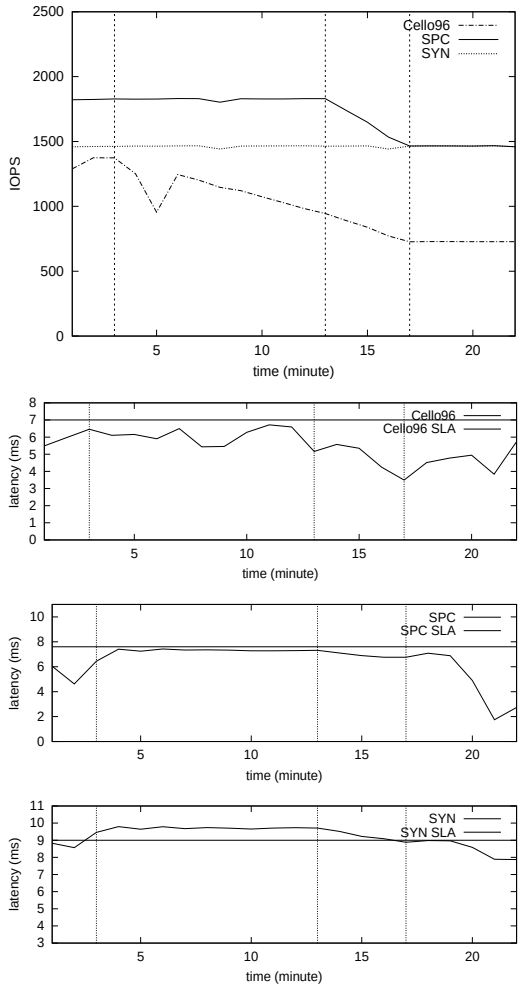


Figure 6.8: Handling a change in the confidence value of the models at run-time

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

Phase 1 (t= 3 min to t = 13 min): The feedback module continues to throttle for 3 consecutive increments; 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), but its confidence value is lower (because of the higher differences between predicted and observed model values). This repeats for consecutive invocations of the reasoning engine after which the feedback module switches to use the designer-defined policies.

Phase 2 (t= 13 min to t=17 min): A simple designer policy the CHAMELEON uses is to throttle all the non-violating workloads uniformly (*uniform pruning*). Both SPC and Cello96 are throttled in small steps (5% of their SLA IOPS) till the latency SLA of the synthetic workload is satisfied.

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

6.1.3 Computational complexity of the reasoning engine

The current implementation of CHAMELEON uses a piece-wise linear programming approach for constraint solving. The computational complexity of the constraint solver is a function of the number of variables involved. Figure 6.9 shows the amount of time CHAMELEON takes to generate the answer. This experiment was run on a P4 2.8 Ghz machine with 512MB memory.

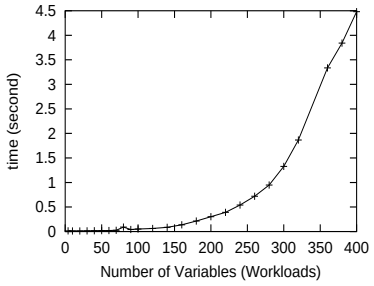


Figure 6.9: Computational complexity of the reasoning engine

6.1.4 Discussion of the experimental results

We were pleased that in our experiments, CHAMELEON was able to automatically execute throttling procedures that moved the system to its optimal state as defined by the value function. In all cases the right workloads were throttled and the amount of the throttling was the minimum needed to meet the targets. With our current system and for the workload perturbations we imposed, we saw reaction times between 3 and 12 minutes. While this is not instantaneous, it is almost certainly quicker than a human could react (notice the problem, decide what to do, execute) and also almost certainly more precise in execution (got to exactly the right place). Unthrottling was similarly successful, releasing

constraints when load on the high priority workloads decreased; it introduces an additional delta to the settling-time which is dependent on how proactively the token are issued and recovered. Again, we believe our reaction times were better than human-capable at 14 minutes. Thus, we are achieving the goals of allowing full hardware utilization when all workloads are meeting responsiveness requirements and we were able to appropriately restrain the lower priority or greedy ones when system limits were reached.

6.2 Experimental Evaluation using the POLUS simulator

The POLUS simulator is built using ABLE (Agent Building and Learning Environment) [15]. ABLE provides the basic building blocks for Polus namely learning algorithms such as neural networks, self-organizing map; JDBC connectivity for interfacing with the database, and data filters.

The Polus modules are implemented as Java-beans or agents. An agent can maintain state and has a thread of control. The architecture of the implementation is shown in Figure 6.10.

1. Polus specifications and goals are written in XML.
2. The Input agent parses the specifications and populates the database using JDBC.
3. The Monitor agent collects information from the managed system, uses data filters to eliminate noise and spikes in the data; generates a trigger for the decision-agent when the goals are not met.
4. The Decision-agent is responsible for searching the specification tables using forward chaining. It decides the action to be invoked and sends a message to the Actuator agent.
5. Learning agent: After the action is invoked, the Learning-agent gathers information from the monitor-agent. Employs various learning algorithms at different levels: CBR at the implications-level, re-enforcement learning at the preconditions level, neural nets at the base-level.
6. Actuator agent: Interfaces with the actions in the managed system and invokes them, depending on the message from the decision-agent.

6.2.1 Experimental Setup

The entities within the SAN file-system simulator are similar to those introduced in Chapter 5. The cost of atomic operations used in the simulator are shown in table 6.2. The simulator models the following actions that are invoked by

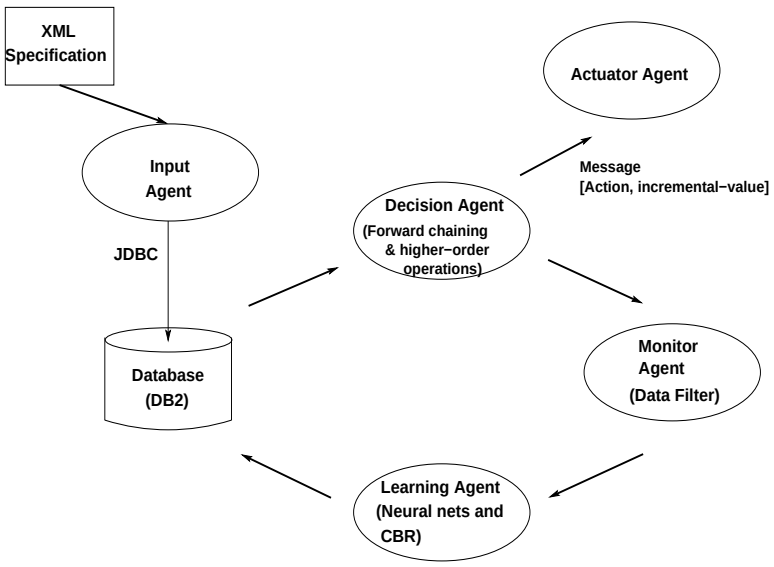


Figure 6.10: Polus simulator using ABLE

the management software via actuators: Pre-fetch size tuning, Data replication, Backup and Clean-delay interval. I/O operations within the SAN file system are invoked by the client and can have multiple possible paths, depending on whether the data is cache or not. The simulator considers the following paths:

MHDH Metadata and data hit in the client

MHDM Metadata hit and data miss in client

MMDM Metadata and data miss in the client

The summation of each of these probabilities of the invocation paths $P_{MHDH}+$ $P_{MHDM} + P_{MMDM} = 1$

The average I/O latency is given by:

$$\begin{aligned} L =& P_{MHDH} * L_{MHDH} + P_{MHDM} * L_{MHDM} + \\ & P_{MMDM} * L_{MMDM} \end{aligned}$$

where:

$$\begin{aligned} L_{MHDH} &= L_m + L_d \\ L_{MHDM} &= L_m + L_{DM} \\ L_{DM} =& P_{Controller-hit} * Q_C * S_{Controller} + \\ & (1 - P_{Controller-hit}) * Q_C * S_{Disk} \\ L_{MMDM} &= Queue\_depth_{Server} * S_{Server} + L_{DM} \end{aligned}$$

The values for probabilities such as P_{MHDH}, and $P_{Controller-hit}$ are modeled by representing the caches as finite sized-arrays and keeping track of data blocks

Operation	Cost
Size of metadata object	1000 bytes
Latency to access from metadata cache (L_m)	20 μsec
Latency to read from datacache (L_d)	20 μsec
Service time of server (S_{server})	420 μsec
Service time from controller cache ($S_{controller}$)	0.6 $msec$
Service time of disk (S_{disk})	6 $msec$
Queue depth at controller (Q_C)	256 (max)
Size of metadata cache	128 MB
Size of data cache	1 GB
Size of controller cache	256 MB

Table 6.2: Cost of atomic operations in the file system simulator

Action	**Description**	**Invocation path parameters affected**	**Resources affected**
Prefetching	Readahead of data and metadata	P_{MHDH} and P_{MHDM}	Data cache, metadata cache and interconnect bandwidth
Replication	Creates replica of data on a different volume in the controller	$queuedepth_{controller}$	Storage space (ignoring transient effects)
Data backup	Consider only transient effects since we are not considering availability	P_{MHDH}, P_{MHDM}, P_{MMDM} $queuedepth_{controller}$ $queuedepth_{server}$	Memory, interconnect bandwidth and storage space
Clean delay	Frequency at which dirty buffers are flushed to disks	Only for writes – P_{MHDH} bursty traffic for storage controller: $queuedepth_{controller}$	Metadata cache and data cache (metadata cannot be evicted till data is written)

Table 6.3: Modeling actions within the file system simulator

in the elements. Similarly, the average queue depth such as Q_C are actually modeled by using service time to complete each request.

Each action is modeled to reflect its impact on the invocation path, system resources and changes in the workload characteristics (table 6.3). To activate the actions in the file system simulator, specifications are fed into Polus and rule-based management. The specifications for a rule-based system consist of ECAs that describe the system-behavior for different system-states. The ECA specifications in this example run into 7 pages (76 rules). The exact Polus specifications that are fed are given in figure 6.11.

Action	Implications	Preconditions	Base Invocation
Prefetch	<Throughput, impact = up>	<Workload = sequential/random, value = high> <precond dimension = memory, value >20%> <precond dimension = fc_bandwidth, value = *>	*Parameter:* prefetchSize *Function:* changePrefetchSize
Replication	<Availability, impact = up> <Latency, impact = up> <Throughput, impact = depends>	<Workload = read/write, value = high> <Workload = Queue-depth, value > 16> <Resource = storage-disks, value = *>	*Parameter:* numReplicas *Function:* invokeReplication
Clean delay	<Latency, impact = up> <Throughput, impact = depends> <Reliability, impact = down>	<Workload = read/write, value = low> <Workload = writes, value = async> <Resource = memory, value = *>	*Parameter:* cleanDelay interval *Function:* changeDelay
Data backup	<Latency, impact = down> <Throughput, impact = down> <Reliability, impact = up>	<Trigger = backup, value = *> <Resource = storage value > 35%>	*Parameter:* backupThroughput *Function:* invokeBackup

Figure 6.11: Specifications fed to the Polus framework

6.2.2 Experimental Results

The experimental analysis consists of a quantitative comparison of Polus and Rule-based systems for different system states. During evaluation, the values (thresholds and action invocation) for rule-based systems are assumed to be correct and empirically obtained from prior runs.

The file system is driven by a trace generator that imposes different states on the system. The generated state is a triplet of the form: <Workload characteristics, Available Resources, Goals >. The values for the goals are different than their current values. For each of these system states, we compare the response of both Polus and an ECA based rule system. Table 6.4 categorizes the possible system state.

The analysis of Polus and ECA for each of the categories is described as follows:

Category 1: Single action applicable

Analysis: The comparison is shown in figure 6.12. This is a simple category with a single candidate action. Polus generally selects the same action as an ECA-based system. ECA is assumed to have the right value for invocation while Polus uses the incremental approach for invocation. Learning improves the incremental approach by interpolating the starting point for incremental invocation.

Insights: The efficiency of the incremental algorithm is dependent on the impact function of the invoked action, which could be linear, quadratic, exponential and so on.

Categories	Description	Example (File system simulator)
Category1: Single action applicable	The system states in this category are such that only a single candidate action is applicable, i.e., searching the specifications leads to a single candidate action	Workload: Sequential, read dominated with read/write ratio of 0.9, avg. queue-depth = 6 Current Throughput = 80 MBps Goal = 100 MBps Polus specification search: The only action that becomes applicable is Prefetching.
Category 2: Multiple actions applicable (they appear to have similar preconditions and/or implications)	In this category more than one action have similar preconditions and/or implications and are indistinguishable. In reality, these actions are not similar This category becomes increasingly common during initial bootstrapping, i.e., the system hasn't learnt values for preconditions and implications	Workload: Read dominated, sequential/random ratio = 0.2, average queue-depth = 8. Current Throughput = 80 MBps Goal = 100 MBps Polus specification search: Prefetching and Replication are selected as candidate actions. In reality Prefetching is not applicable as the workload is not sequential, but Polus does not have the threshold value for the sequential/random ratio in prefetch specifications
Category 3: More than one goal not met	In this category, more than one action needs to be invoked as a single action cannot satisfy the goal requirements	Workload: Sequential, read/write= 0.3 Current Throughput = 80 MBps Goal = 100 MBps , Current Latency = 6msec Goal = 4.5 msec Polus specification search: Prefetching and Clean delay are both invoked as the former improves throughput while the later improves latency
Category 4: Recurrent action invocation (One action, leads to chaininvocation of actions)	In this category, invocation of an action leads to a chain-invocation of a series of actions. Ability to detect and prevent recurrent action invocation is a required property of the management software	Workload: Trigger for data backup with window = 4 hours Polus specification search: Theoretically, backup can be invoked since the goals are being met. But invoking backup at this time will cause latency goals to be overshot
Category 5: No action applicable (Negation of previous actions required)	Actions are negated under two scenarios: to make resources available for another actions, and the workload preconditions change	Workload: Changes from large block sequential to small block random Polus specification search: Prefetching hurts performance as memory and storage resources are used for acquiring data that is never used

Table 6.4: Categorizing the possible system states

Category 2: Multiple actions applicable

Analysis: This category (figure 6.13) exposes the "weak-spot" in Polus. When the candidate actions are indistinguishable, Polus tries them one-by-one till it either leads to a negative impact on the observable values or the goals are met. As shown in the graph, Polus initially selects the wrong action (i.e. prefetching). After the value dips further, Polus tries the next candidate action (i.e. replication). Learning adds the threshold values (in this example at the pre-conditions level) and enables distinguishing between the actions.

Insights: In systems with larger action-sets, it is quite possible that Polus never converges due to side-effects of trying wrong actions.

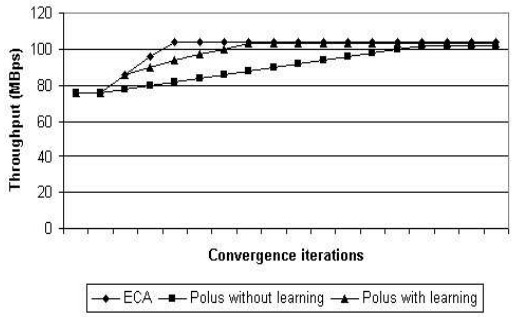


Figure 6.12: Comparing Polus and ECA for category 1 (single candidate action). In the graph, the throughput goal = 100 MBps.

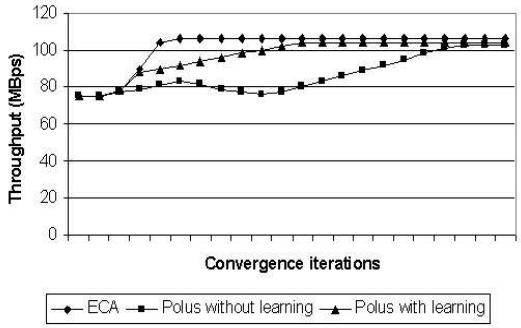


Figure 6.13: Comparing Polus and ECA for category 2 (multiple candidate actions). In the graph, the throughput goal = 100 MBps

Category 3: More than one goal not met

Analysis: Rule-based systems will invoke a single action in each iteration without analyzing the combined impact of the actions. On the other hand, Polus considers different permutations to combine the actions (figure 6.14). This is beneficial when the two actions act on the same resources, such that the invocation of one action beyond a threshold can violate the pre-conditions of other actions. As shown in the graph, ECA does not meet the latency goal due to lack of memory resources. In its previous iteration Prefetching was invoked for throughput goals and the rules did not consider the combined state while deciding the value for prefetching. Learning refines the attributes of actions allowing better combination strategies.

Insights: Higher-order operation are powerful in deriving permutations that cannot be possibly defined statically.

Category 4: Recurrent action invocation

Analysis: The comparison in shown in figure 6.15. Polus uses look ahead while invoking actions to estimate the impact of the action on the goals. Rule-based

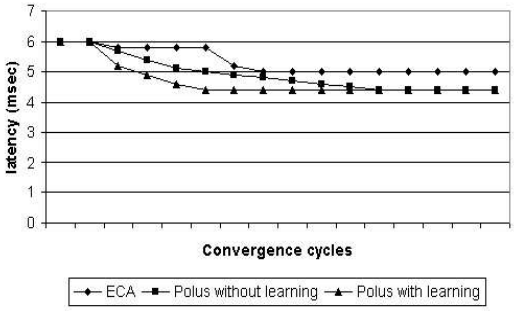


Figure 6.14: Comparing Polus and ECA for category 3 (more than one goal not met). In the graph, the latency goal = 4.5 msec

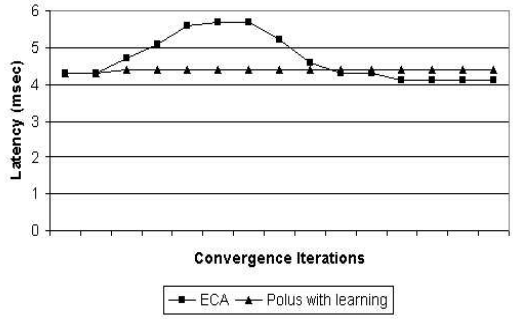


Figure 6.15: Comparing Polus and ECA for category 4 (recurrent action invocation). In the graph, the latency goal = 4.5 msec

systems don't have an equivalent of this (though it is possible to write separate rules to cater for this). As shown in the graph, ECA invokes the Backup action that leads to invocation of a series of actions (Replication in this example). Polus does a look-ahead and does not invoke Back-up during the current system-state. (For Back-up we are assuming a time-window)

Insights: Look-ahead is a required operation and effective only with some learning of the action model. Hence there is only a single curve in the graph.

Category 5: Negation of previous actions

Analysis: Both Polus and Rule-based systems can cater to negation of actions (figure 6.16). ECA can accommodate by writing separate rules. Learning does not play a significant role in this category

Insights: Explicit need to write separate rules in rule-based systems for negation whereas the Polus reasoning engine can account for this without any additional specifications.

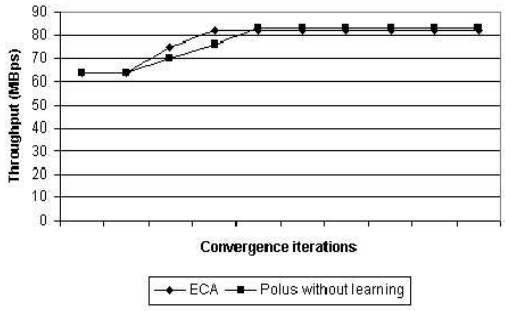


Figure 6.16: Comparing Polus and ECA for category 5 (negation of previous actions required). In the graph, the throughput goal = 80 MBps

7 Conclusion

OAA automation is undoubtedly a complex but important problem. Existing commercial management products are largely policy-based and have not got much traction – the OAA loop in production systems is largely driven by human administrators, with little or no automation. This thesis is the stepping stone in using model-based techniques for OAA automation. Polus addresses several real-world challenges related with the model-based automation: representation of domain-details as models, automated creation and maintence of models, fomalizing decision-making as constrainted optimization, and handling inaccuracies of real-world models. The prototype implementation of Polus in a production storage system [99] serves as a proof-of-concept – it identified, analyzed, and corrected performance violations within a real storage system in 3-14 minutes, which compares very favorably with the time a human administrator would have needed.

This chapter summarizes the key concepts of Polus and enumerates the future work to manage increasingly complex systems. Given the deep rooted skeptism in model-based automation or simply system automation in general, the details are summarized in the format of detractor's viewpoint for the OAA automation challenges, and the response of the advocate using the Polus approach.In looking ahead at the next set of challenges, there are many unsolved issues, but none we think, for which we don't have ideas for solutions.

7.1 Assesment of the Polus Methodology and Future Research Challenges

Detractor's viewpoint*: Real-world systems are too unpredictable for the models to be accurate; hence model-based OAA automation is not possible.*

Advocate's viewpoint: Yes, models will be inaccurate; in fact in a production system, it will never be possible to capture all the system and environment parameters that affect the output of the model. For example, the effect of data migration is primaryily dependent on the current target load; but second-order parameters such as the ON-OFF phase of the migrated workload and its overlap with the current load pattern on the target component plays an important role in deciding the effect of data migration.

We believe that the models don't have to be perfectly accurate for the OAA automation to be operational – they only have to be accurate enough so that the relative ordering of action selection is maintained. Also, the execution of the action is variably aggressive based on the accuracy of the models – if the confidence value of the model is low, and if the cost to invoke the selected action is high, the control is passed to the human administrator. Further, for actions that can be executed incrementally (e.g. throttling), the feedback loop checks for divergent system behavior resulting from model inaccuracies.

It should be noted that the confidence-value of the model is not constant within the state-space – it is dependent on whether a similar state has been seen in the past, and hence varies from decision to decision.

Research Directions:

Making models accurate is clearly an ongoing area of research [], and will help in making OAA automation more effective.

1. *Formalisms for capturing complex dependencies*: Techniques to capture complex dependencies as a part of the model: effect of invoking an action on the cost-benefit of other actions [94]; effect of workload interleaving on the component response time (queueing delays, cache sharing, scheduling policies, ON-OFF workload phases).

2. *Time-series analysis of production workloads*: Evaluating the effectiveness, error range, and limitations of popular time-series techniques such as ARIMA [96] when applied for production workloads.

3. *Heuristics to identify divergent system behavior*

Detractor's viewpoint*: Production systems have a large number of candidate corrective options – it is computationally infeasible to find optimal solutions*

Advocate's viewpoint: The OAA loop today is managed by human administrators; they generally select actions by using best-practices, or coarse rules-of-thumb derived from years of experience. The process of action selection is coarse-grained and does not necessarily aim to find an optimal action, but rather find a feasible action "that works." Similarly, the objective of OAA automation isn't necessarily be to find the optimal solution, but rather to avoid the worst ones. As such, the complexity of action selection can be reduced by pruning of corrective options based on attributes such as cost, benefit, transient time, confidence-value.

Research Directions :

4. *Empirical heuristics to prune the solution-space*: Techniques to prune groups of corrective options early in the decision-making (analogous to pruning within the database optimizer [88]).

5. *Decentralized decision-making*: Decentralized approach for analysis, decision-making, and coordination (generally with only a partial information about the global system state).

Detractor's Viewpoint*: Creating and refining models requires extensive monitoring data that may be difficult to collect at run-time and is not currently available*

Advocate's Viewpoint: Existing management frameworks [17, 58] collect extensive monitoring data today; this data serves as the "magnifying glass" for human administrators, and is presented within a monitoring dashboard as thousands of gauges.

Research Directions:

6. *Standard for exporting monitoring data across heterogeneous hardware and software products*: For example, SMI-S [92] is such an ongoing standard within storage systems.

7. *Integrating legacy knowledge-base for system management*: Common model-based schema to represent hard-wired policies, best-practice templates, documented rules-of-thumb; also the ability to switch between different information sources at run-time.

Detractor's Viewpoint*: Updating models within the system is a human-intensive task, and generally makes it infeasible to maintain the management framework e.g. The XCON Expert System [18] was difficult to maintain and was eventually abandoned*

Advocate's viewpoint: Automated model updates is critical for the success of OAA automation. In our approach, models are represented as mathematical functions that can be constantly updated by applying SLT to the new monitor data. Armando Fox's group at Stanford have applied SLT in different domains, and have demonstrated promising results [42].

Research Directions:

8. *Feature-set selection (i.e the input parameters) for Models*: Enumerating a minimal set of parameters that can produce models with a reasonable accuracy (i.e. the balancing point between accuracy and simplicity of models).

9. *Techniques for bootstrapping models, and experimental evaluation of model confidence-values within production systems.*

Detractor's Viewpoint*: Enterprise customers won't be comfortable in the OAA automation taking over complete control, and hiding the details of decision-making; also a management software cannot really outperform the human administrator.*

Advocate's Viewpoint: Taking over control from the administrator is a gradual process, that might take months and even years. At the time of initial deployment, the models are bootstrapped by running calibration tests – these tests explore different quadrants of the state-space and serve as baseline values. As similar system-states are visited repeatedly, the confidence-value of the models improves, and allows automation of the common-tasks. Whenever the system operates in a relatively new quadrant of the state-space, the management software should flag for administrator input (unless the cost of invoking as well as reversing the action effects is small). Thus, the longer the system runs, the more useful the management software becomes; also the initial versions of these frameworks will have an extensive *explain* facility, to describe the reasons for selecting a particular action.

The objective of management software is not to outperform the administrator. This is analogous to Deep Blue computer defeating the chess grand master [2]. The strength of management software is the exhaustive exploration of the corrective options; while the strength of humans is a better perception of domain specific details (which in the case of automation is more critical). Thus, the so called *reduction in TCO argument* [53] for management software will not be because of better decision-making by automated frameworks, but because of: reduced need on **skilled** administrators, increase in the number of resources handled by an existing admin, and faster response to exception events such as load surges, component failures.

Research Directions:

10. *Learning from the human administrator*: Each time the control is passed to the administrator, the management software should record the actions invoked, and possibly generalize rules-of-thumb to be used in conjunction with model-based decision-making.

11. *Explain facility for management frameworks*: Book-keeping of the decision-making process, in order to generate an explanation for the sequence of intermediate options that led to the final action selection (analogous to *db2expln* in databases such as DB2).

7.2 Summary

In moving forward with production system OAA automation, we envision two roadmaps: The first roadmap (referred to as *information route)* is for the specification, creation, and evolution of the domain-specific models – the key questions

to be addressed are *how accurate the models can be* within a production system, and *how easy it would be to maintain* these models with the continious system evolution. The second roadmap (referred to as *execution route)* deals with algorithms for action selection, pruning the solution-space, and handling uncertainty in the domain-specific information – the key questions to be addressed is *how accurate the models need to be* in order to preserve the relative order of action selection, and *how well can the action execution handle errors* in the model predictions. This chapter enumerates several interesting questions along each of these routes. Based on our initial exploration in the domain of storage management, we believe that OAA automation is an *undiscovered reality* – something that can be realized at the crossroads of these two roadmaps.

From the real-world deployment perspective, the acceptance of OAA automation frameworks is going to be a gradual process. The deep-rooted skepticism towards automation will initially limit these frameworks as tools to aid the administrator, rather than remove the later from the loop – tools that "Give information rather than data."

References

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

[2] *Deep Blue Versus Kasparov: The Significance for Artificial Intelligence, Collected Papers from the 1997 AAAI Workshop*. AAAI Press, 1997.

[3] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[4] Eugene Agichtein and Luis Gravano. Snowball: Extracting relations from large plain-text collections. In *Proceedings of the Fifth ACM International Conference on Digital Libraries*, 2000.

[5] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *The VLDB Journal*, pages 496–505, 2000.

[6] G. Alvarez, E. Borowsky, S. Go, T. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, November 2001.

[7] G. Alvarez, K. Keeton, E. Riedel, and M. Uysal. Characterizing data-intensive workloads on modern disk arrays. In *4th. Workshop on Computer Architecture Evaluation using Commercial Workloads*, January 2001.

[8] G.A. Alvarez, E. Borowsky, S. Go, T.H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An Automated Resource Provisioning Tool for Large-scale Storage Systems. *ACM Transactions on Computer Systems*, 19(4):483–518, November 2001.

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

[10] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proc. 1st Conf. on File and Storage Tech.*, pages 175–188, June 2002.

[11] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 175–188, January 2002.

[12] Marc Badel, Erol Gelenbe, Jacques Leroudier, and Dominique Potier. Adaptive optimization of a time-sharing system's performance. *Proceedings of the IEEE*, 63(6):958–956, 1975.

[13] M. Bearden, S. Garg, W.J. Lee, and A. van Moorsel. User-centric QoS policies, or saying what and how. Work-in-progress report, Proc. 11th Workshop on Distributed Systems: Operations and Management (DSOM), December 2000.

[14] Kevin Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cube. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM Press, 1999.

[15] J.P. Bigus, D.A. Schlosnagle, J.R. Pilgrim, W.N. Mills III, and Y. Diao. ABLE: A Toolkit for Building Multiagent Autonomic Systems. *IBM Sys. J.*, 41(3), September 2002.

[16] Parker R. Blevins and C. V. Ramamoorthy. Aspects of a dynamically adaptive operating system. *IEEE Trans. Computers*, 25(7):713–725, 1976.

[17] BMC Software. *Patrol for Storage Networking*, 2004.

[18] D. G. Bobrow, S. Mittal, and M. J. Stefik. Expert systems: perils and promise. *Commun. ACM*, 29(9):880–894, 1986.

[19] E. Borowsky, R. Golding, P. Jacobson, A. Merchant, L. Schreier, M. Spasojevic, and J. Wilkes. Capacity planning with phased workloads. In *Proceedings of the first international workshop on Software and performance*, pages 199–207. ACM Press, 1998.

[20] Eric Allen Brewer. *Portable high-performance supercomputing: high-level platform-dependent optimization*. PhD thesis, 1995.

[21] J. S. Brown. The low road the middle road and the high road. In P. H. Winston and K. H. Prendergast, editors, *The AI Business*, pages 81–90. MIT Press, Cambridge, MA, 1984.

[22] Kurt P. Brown. Goal-oriented memory allocation in database management systems. Technical Report CS-TR-1995-1288, 1995.

[23] Kurt P. Brown, Michael J. Carey, and Miron Livny. Goal-oriented buffer management revisited. *SIGMOD Rec.*, 25(2):353–364, 1996.

[24] Kurt P. Brown, Manish Mehta, Michael J. Carey, and Miron Livny. Towards automated performance tuning for complex workloads. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 72–84. Morgan Kaufmann Publishers Inc., 1994.

[25] B.G. Buchanan and E.H. Shortliffe, editors. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, Reading, Massachusetts, 1984.

[26] Chris Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[27] Maria Calzarossa and Giuseppe Serazzi. Workload characterization: A survey. *Proc. IEEE*, 81(8):1136–1150, 1993.

[28] D. Chambliss, G. Alvarez, P. Pandey, R. Menon D. Jadav, J. Xu, and T.P. Lee. Performance virtualization for large-scale storage systems. In *Proc. Symp. on Reliable Distributed Sys. (SRDS)*, October 2003.

[29] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centres. In *Symposium on Operating Systems Principles*, pages 103–116, 2001.

[30] Surajit Chaudhuri, Eric Christensen, Goetz Graefe, Vivek R. Narasayya, and Michael J. Zwilling. Self-tuning technology in microsoft sql server. *IEEE Data Eng. Bull.*, 22(2):20–26, 1999.

[31] Surajit Chaudhuri and Vivek R. Narasayya. Autoadmin 'what-if' index analysis utility. In *SIGMOD Conference*, pages 367–378, 1998.

[32] Guillermo A. Alvarez Chenyang Lu and John Wilkes. Aqueduct: online data migration with performance guarantees. pages 175–188, January 2002.

[33] Fernando J. Corbato, Marjorie Merwin-Daggett, and Robert C. Daley. An experimental time-sharing system. In *Proceedings of the AFIPS Fall Joint Computer Conference*, 1962.

[34] International Business Machines Corp. Autonomic Computing:IBMs Perspective on the State of Information Technology. http://www.research.ibm.com/autonomic/manifesto/, 2001.

[35] Storage Performance Council. Spc i/o traces. http://www.storageperformance.org/.

[36] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–32, 2001.

[37] Murthy V. Devarakonda, Jack P. Gelb, Avi Saha, and Jimmy P. Strickland. A policy-based storage management framework. In *POLICY*, pages 232–235, 2002.

[38] FalconStor Software. Ipstor: build an end-to-end IP-based network storage infrastructure. White paper. http://www.falconstor.com, 2001.

[39] E. A. Feigenbaum. The art of artificial intelligence: Themes and case studies of knowledge engineering. In *Proc. of the 5th IJCAI*, pages 1014–1029, Cambridge, MA, 1977.

[40] E. A. Feigenbaum. The art of artificial intelligence: Themes and case studies of knowledge engineering. In *Proc. of the 5th IJCAI*, pages 1014–1029, Cambridge, MA, 1977.

[41] Dror G. Feitelson and Michael Naaman. Self-tuning systems. *IEEE Softw.*, 16(2):52–60, 1999.

[42] A. Fox, E. Kiciman, D. A. Patterson, R. H. Katz, and M. I. Jordan. Combining statistical monitoring and predictable recovery for self-management. ¡i¿ACM SIGSOFT Proceedings of the Workshop on Self-Managed Systems (WOSS)¡/i¿, 2004., 2004.

[43] Gene F. Franklin, Abbas Emami-Naeini, and J. David Powell. *Feedback Control of Dynamic Systems*. Addison-Wesley Longman Publishing Co., Inc., 1993.

[44] G. Ganger, J. Strunk, and A. Klosterman. Self-* Storage: Brick-Based Storage with Automated Administration. Technical Report CMU-CS-03-178, Carnegie-Mellon University, August 2003.

[45] M.R. Genesereth and M.L. Ginsberg. Logic Programming. *Comm. ACM*, 28(9), September 1985.

[46] J. Ghosh and A. Nag. *An Overview of Radial Basis Function Networks.* Radial Basis Function Neural Network Theory and Applications, Physica-Verlag, 2000.

[47] J.S. Glider, F. Fuente, and W.J. Scales. The software architecture of a san storage control system. *IBM System Journal*, 42(2):232, 2003.

[48] Ashvin Goel, David Steere, Calton Pu, and Jonathan Walpole. Adaptive resource management via modular resource management. Technical report, 1998.

[49] R. Golding, E. Shriver, T. Sullivan, and J. Wilkes. Attribute-managed Storage. In *Proc. Workshop on Modeling and Specification of I/O*, October 1995.

[50] Jim Gray. What next?: A dozen information-technology research goals. *J. ACM*, 50(1):41–57, 2003.

[51] C. Green. Application of theorem proving to problem solving. In B. L. Webber and N. J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 202–222. Kaufmann, Los Altos, CA, 1981.

[52] Irene Greif and Carl Hewitt. Actor semantics of planner-73. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 67–77. ACM Press, 1975.

[53] Gartner Group. Total Cost of Storage Ownership A User oriented Approach. Research note, Gartner Group, 2000.

[54] F. Hayes-Roth. Rule-based Systems. *Comm. ACM*, 28(9), September 1985.

[55] C. Hewitt. Planner: A language for proving theorems in robots. In *Proc. of the 1st IJCAI*, pages 295–301, Washington, DC, 1969.

[56] C. Hewitt. Procedural embedding of knowledge in planner. In *Proc. of the 2nd IJCAI*, pages 167–182, London, UK, 1971.

[57] Lan Huang, Gang Peng, and Tzi cker Chiueh. Multi-dimensional storage virtualization. *SIGMETRICS Perform. Eval. Rev.*, 32(1):14–24, 2004.

[58] IBM Redbook. *IBM Tivoli Storage Manager*, 2004.

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

[60] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, aug 1988.

[61] Raj Jain. *The Art of Computer System Performance Analysis.* Wiley, 1991.

[62] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. *SIGMETRICS Perform. Eval. Rev.*, 32(1):37–48, 2004.

[63] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. In *Proc. of the 12th. Int'l Workshop on Quality of Service*, June 2004.

[64] S. Keshav. A control-theoretic approach to flow control. *Proceedings of the conference on Communications architecture and protocols*, pages 3–15, 1993.

[65] T. Kohonen. *Self-Organizing and Associative Memory 3rd ed.* Springer-Verlag, 1988.

[66] R. Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Proceedings of the Sixth IFIP Congress (Information Processing 74)*, pages 569–574, Stockholm, Sweden, August 1974.

[67] Robert Kowalski. The limitation of logic. In *Proceedings of the 1986 ACM fourteenth annual conference on Computer science*, pages 7–13. ACM Press, 1986.

[68] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models.* Prentice-Hall, Inc., 1984.

[69] D.B. Leake. *Case-Based Reasoning: Experiences, Lessons and Future Directions.* AAAI Press, 1996.

[70] C. Lumb, A. Merchant, and G. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 131–144, April 2003.

[71] E. Lupu M. Sloman. Security and management policy specification. *IEEE Network*, March 2002.

[72] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 238–251. ACM Press, 1997.

[73] John McCarthy. Phenomenal data mining: from data to phenomena. *SIGKDD Explor. Newsl.*, 1(2):24–29, 2000.

[74] Daniel A. Menasc, Daniel Barbar, and Ronald Dodge. Preserving qos of e-commerce sites through self-tuning: a performance model approach. In *EC '01: Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 224–234. ACM Press, 2001.

[75] J.C. Meza. Opt++:an object-oriented class library for nonlinear optimization. Technical report.

[76] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, New York, 1997.

[77] B. Moore. Network Working Group – RFC3060. Policy Core Information Model – Version 1 Specification. http://www.ietf.org/rfc/rfc3060.txt, 2001.

[78] D. Narayanan, J. Flinn, and M. Satyanarayanan. Using history to improve mobile application adaptation. In *WMCSA '00: Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'00)*, page 31. IEEE Computer Society, 2000.

[79] Allen Newell, J. C. Shaw, and H. A. Simon. Chess-playing programs and the problem of complexity. pages 39–70, 1995.

[80] H. Newman, M. Ellisman, and J. Orcutt. Data-intensive e-science frontier research. *CACM*, 46(11):68–77, 2003.

[81] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Sixteen ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, 1997.

[82] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, Armando, E. Kcman, M. Merzbacher, D. Oppenheimer, and N. Sastry. Recovery oriented computing (roc): Motivation, definition, techniques, and case studies.

[83] D.A. Pease, J.Menon, B. Rees, L.M. Duyanovich, and B.L. Hillsber. IBM Storage Tank-A heterogeneous scalable SAN file system. *IBM Sys. J.*, 42(2):250–267, 2003.

[84] David Reiner and Tad Pinkerton. A method for adaptive performance improvement of operating systems. *SIGMETRICS Perform. Eval. Rev.*, 10(3):2–10, 1981.

[85] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning Internal Representations Through Error Propagation. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing: Experiments in the Microstructure of Cognition, Vol. 1.* MIT Press, 1986.

[86] L. Russell, S. Morgan, and E. Chron. Clockwork: A new movement in autonomic systems. *IBM Systems Journal*, 42(1):250–267, 2003.

[87] S.J. Russell and P. Norvig. *Artificial Intelligence: a modern approach.* Prentice-Hall, 1995.

[88] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD Conference*, pages 23–34, 1979.

[89] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.

[90] M. I. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. *Proceedings of the Sixth workshop on Hot Topics in Operating Systems*, 1997.

[91] E. Shriver, A. Merchant, and J. Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. In *Proc. of SIGMETRICS*, pages 182–191, 1998.

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

[93] David G. Sullivan, Margo I. Seltzer, and Avi Pfeffer. Using probabilistic reasoning to automate software tuning. *SIGMETRICS Perform. Eval. Rev.*, 32(1):404–405, 2004.

[94] David Gerand Sullivan. *Using probabilistic reasoning to automate software tuning*. PhD thesis, September 2003.

[95] S. Thrun. Lifelong learning: A case study. Thrun, S., Lifelong Learning: A Case Study, Carnegie Mellon University: CS-95-208, 1995.

[96] 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. ACM Press, 2001.

[97] J. Turner. New directions in communications. *IEEE Communications*, 24(10):8–15, October 1986.

[98] S. Uttamchandani, K. Voruganti, S. Srinivasan, J. Palmer, and D. Pease. Polus: Growing storage QoS management beyond a 4-year old kid. In *FAST04*, March 2004.

[99] S. Uttamchandani, L. Yin, G. Alvarez, J. Palmer, and G. Agha. Chameleon: a self-evolving, fully-adaptive resource arbitrator for storage systems. In *To Appear in Proc. of Usenix Annual Technical Conference (Usenix)*, April 2005.

[100] M. Uysal, G. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proc. of the 9th Intl. Symp. on Modeling, Analysis and Simulation on Computer and Telecommunications Systems*, pages 183–192, August 2001.

[101] Vladimir N. Vapnik. *The nature of statistical learning theory*. Springer-Verlag New York, Inc., 1995.

[102] D. Verma. Simplifying network administration using policy based management. (2), March 2002.

[103] D. Verma and S. Calo. Goal Oriented Policy Determination. In *Proc. 1st Workshop on Algorithms and Architectures for Self-Managing Sys.*, pages 1–6. ACM, June 2003.

[104] Rich Vuduc, James Demmel, and Jeff Bilmes. Statistical models for automatic performance tuning. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*, pages 117–126. Springer-Verlag, 2001.

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

[106] D. A. Waterman and Frederick Hayes-Roth. *Pattern-Directed Inference Systems*. Academic Press, Inc., 1978.

[107] Gerhard Weikum, Christof Hasse, Axel Mönkeberg, and Peter Zabback. The comfort automatic tuning project. *Inf. Syst.*, 19(5):381–432, 1994.

[108] Matt Welsh, David E. Culler, and Eric A. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *SOSP*, pages 230–243, 2001.

[109] R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation*, pages 309–328. Kaufmann, Los Altos, CA, 1985.

[110] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. *Lecture Notes in Computer Science 2092*, pages 75–91, 2001.

[111] T. Winograd. Frame representations and the declarative/procedural controversy. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation*, pages 357–370. Kaufmann, Los Altos, CA, 1985.

[112] J. A. Zinky, D. E. Bakken, and R. D. Schantz. Architectural support for Quality-of-Service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.

Vita

Sandeep Uttamchandani was born in Pune, India, on 13 January, 1979. He graduated from University of Pune in 2000 with a B.E. in Computer Science and Engineering. He then joined the University of Illinois at Urbana Champaign for graduate studies in Computer Science under the supervision of Prof. Gul Agha. He obtained a MS in Computer Science in August 2001, and a PhD in Computer Science in October 2005. After completing his MS, he joined IBM Research Center in San Jose, California, and continued making progress on his PhD off-campus, by frequently interacting with Prof. Gul Agha, Dr. Carolyn Talcott, and Dr. Jai Menon. Sandeep Uttamchandani's primary research interest is in specifications and reasoning techniques for automated system management.