PROACTIVE EXPERIMENT-DRIVEN LEARNING FOR SYSTEM MANAGEMENT

by

Piyush Shivam

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Jeffrey S. Chase, Co-supervisor

_____
Shivnath Babu, Co-supervisor

_____
Kamesh Munagala

_____
John A. Board

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2007

ABSTRACT

PROACTIVE EXPERIMENT-DRIVEN LEARNING FOR SYSTEM MANAGEMENT

by

Piyush Shivam

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Jeffrey S. Chase, Co-supervisor

_____
Shivnath Babu, Co-supervisor

_____
Kamesh Munagala

_____
John A. Board

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2007

Abstract

The overall behavior of a system depends on a large number of factors related to the underlying hardware, system software, and running applications. In addition, system behavior may be influenced by interactions among these factors, where the impact of an individual factor on a system depends on the settings of other factors. A 'system knowledge base' that captures how different factors and multi-factor interactions affect the end-to-end behavior of a system is a prerequisite for managing systems effectively. This dissertation addresses the hypothesis that we can learn such a knowledge base in an automatic, proactive, and timely manner by planning and conducting experiments.

An experiment is a run of the system for a specific setting of the system's workload, resource allocation, and configuration. In this dissertation, we develop a general experiment-driven framework that incorporates: (a) policies for automatic planning of experiments to explore a large space of factors and interactions efficiently; and (b) mechanisms to conduct experiments for three important system domains: Web services, batch computing, and storage servers. The policies and mechanisms leverage techniques from design of experiments, active machine learning, and system virtualization to build a sufficiently accurate system knowledge base quickly. The dissertation makes the following contributions:

- Quantifies the linear and non-linear impact of a factor or an interaction on system behavior, and develops experiment-planning algorithms to estimate the impact of important factors and interactions in a system. We use this work to rank the factors and interactions that can affect the performance (e.g., throughput) of multitier Web services.

- Develops experiment-planning algorithms to build models that predict the sys-

tem behavior as a function of factors and interactions that affect this behavior. We explore a continuum of modeling alternatives ranging from a priori models to black-box models. We learn models to enable task and data placement of batch computing applications, and to predict performance measures of Web services like response time and throughput.

- Develops policies to determine how long to run an experiment and how many times to repeat an experiment to attain target levels of confidence and accuracy in experimental results at low cost. We use the policies to benchmark storage servers by systematically mapping a storage server's saturation throughput across a range of server workloads and configurations.

Our empirical evaluation with real and synthetic applications on physical as well as virtual hardware resources shows that our experiment-driven framework can learn an effective knowledge base by conducting only 1-5% of the total number of possible experiments.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank Jesus Christ, my friend, savior, and God whose assurance that He will never leave me nor forsake me kept me going through the many lows and highs of the graduate school life.

Without the love and support of my wife, Chris, this journey would not have been possible. Her unwavering belief in my potential as a researcher, and her patience, strength, and encouragement made the difficult days bearable and the good days much more meaningful. I am also thankful to our recently born daughter, Annika, who taught me invaluable time management skills by her determination to keep us awake at nights.

I am indebted to my advisors Jeff Chase and Shivnath Babu for believing in me and giving me the opportunity to work with them. I am thankful to Jeff for teaching me patiently to how to do good research, standing with me and encouraging me through the many disappointments of the graduate school life, and giving me the freedom to explore the research ideas. I am grateful to Shivnath for coming alongside at a critical time and helping me with the research by working long and hard to provide regular feedback and direction. Special thanks to my committee members, Kamesh Munagala and John Board, for their insightful feedback and comments.

My colleagues at Duke were an important part of this journey. Aydan Yumerefendi, Laura Grit, Varun Marupadi, Matthew Sayler, Rajiv Wickremesinghe, Jaidev Patwardhan, Adriana Iamnitchi, and Sudheer Sahu were always available to bounce ideas on topics ranging from research and life to how to keep awake three nights in a row. David Becker and David Irwin provided the much needed support for the experimental testbed that made this research possible.

Finally, I would like to express gratitude to family and friends who made life meaningful and fun outside the graduate school. I am thankful to my parents Pushpendra and Sarita, and my brother and sister-in-law, Puneet and Madhu, for their constant encouragement. I am thankful to International Bible Study for providing food, fun, and fellowship on many Friday nights over the last several years. I would especially like to thank Jonathan Pillai, Katie Rawson, Scott and Jenny Hawkins, Jennifer West, Curt Blazier, Matthews Abraham, Michael Rizk, and friends at ICF and Blacknall Church for supporting me with their love and prayers throughout this journey.

Chapter 1

Introduction

Over the last few decades the computing infrastructure has transformed from a collection of a few machines in a single room to planetary-scale systems such as grids and networked data centers that allow a user to access computing resources anytime and anywhere. However, the complexity of these systems raises serious management challenges [40, 67]. Failure to manage such systems in an efficient and effective manner continues to affect the bottom line of businesses [115], scientific research at educational institutions [41], and even national security [51].

The challenges in managing systems arise not just from their scale, but also from the interactions between system workload, system software, and hardware components [12, 24]. An adequate knowledge of how these interactions impact *end-to-end* system behavior is a fundamental prerequisite for addressing system management challenges. This dissertation presents an *experiment-driven framework* for building such knowledge proactively and automatically. It develops policies and mechanisms that leverage virtualization, design of experiments, and active machine learning to expose the interactions among the parameters in an efficient, accurate, and principled manner.

1.1 System Complexity

The behavior of a system is a complex function of the system workload, the hardware resources allocated to the system, and the system configuration. Each of these can be characterized by a vector of *factors* or *parameters*; if we represent the system behavior by a vector $\vec{B}$, system workload by a vector $\vec{W}$, hardware resources allocated to the

Table 1.1: Examples of parameters in $\langle \vec{B}, \vec{W}, \vec{R}, \vec{C} \rangle$ vectors.

System domain	**System behavior vector $\vec{B}$**	**Workload vector $\vec{W}$**	**Resource vector $\vec{R}$**	**Configuration vector $\vec{C}$**
Multitier Web Services	Response time and throughput, availability	Arrival rate, mix of requests, number of clients	CPU, memory	Buffer pool size
Batch Applications	Execution time	CPU-intensity, degree of parallelism	CPU, memory, and network bandwidth	File system
Storage Systems	Response time and throughput	Arrival rate, r/w ratio, locality	Number of disks, type of disks	Block size, file system

system by a vector $\vec{R}$, and system configuration by a vector $\vec{C}$ then:

$$\vec{B} = F(\vec{W}, \vec{R}, \vec{C}). \tag{1.1}$$

Table 1.1 illustrates a subset of parameters in $\langle \vec{B}, \vec{W}, \vec{R}, \vec{C} \rangle$ vectors for three important system domains. In this dissertation, the system behavior vector $\vec{B}$ consists of metrics that characterize the performance of the system. A discussion of these domains follows.

- **Multitier Web Services.** The behavior $\vec{B}$ of a multitier Web service is usually specified by measures such as the service's response time and throughput, its utilization of system resources, and its availability. The system workload $\vec{W}$ in this case represents parameters such as the arrival rate and mix of requests, and the number of concurrent clients. The system hardware $\vec{R}$ captures parameters such as the amount of CPU and memory assigned to each tier of the service, and the system configuration $\vec{C}$ captures the configuration parameters in each tier of the service.

- **Batch Applications.** The execution time $\vec{B}$ or makespan of a computational

workload depends on the provisioning (how much) and placement (where) of hardware resources $\vec{R}$ allocated to it, such as the CPU, memory, and network resources, and the characteristics $\vec{W}$ of the workload (CPU-intensity, memory reference locality, degree of parallelism, etc.), which in turn may depend on application inputs such as the data processed by the application.

- **Storage Systems.** The behavior $\vec{B}$ of a storage service, e.g., its response time and throughput, is a complex function of the characteristics of server's I/O workload $\vec{W}$ such as its locality and r/w ratio, the hardware resources $\vec{R}$ allocated to the storage server such as the number and type of disks (SCSI or IDE) and its network bandwidth, and configuration $\vec{C}$ comprising parameters such as the underlying file system type, block size, and RAID configuration.

The goal of system administrators is to keep the system behavior in its "sweet spot" range, as shown in Figure 1.1. The sweet spot is attained by making informed design, engineering, and management choices for the system hardware and system configuration, i.e., controlling the *settings* or values of parameters in the resource vector $\vec{R}$ and the configuration vector $\vec{C}$. There is also considerable interest in making systems self-managing, or *autonomic* [67]. A self-managing system would keep its behavior in the desired range by adjusting the settings of parameters in $\vec{R}$ and $\vec{C}$ automatically. Consider the following examples:

- *Capacity planning.* A common scenario is the provisioning and placement of resources for each tier of a multitier Web service in a data center. The goal is to control the settings of parameters in resource vector $\vec{R}$, e.g., the amount of CPU and memory resources of each tier, to obtain a desired $\vec{B}$ for a given $\vec{W}$. For example, the goal may be to keep the average response time below 50 ms for a given mix and arrival rate of requests [105, 112].

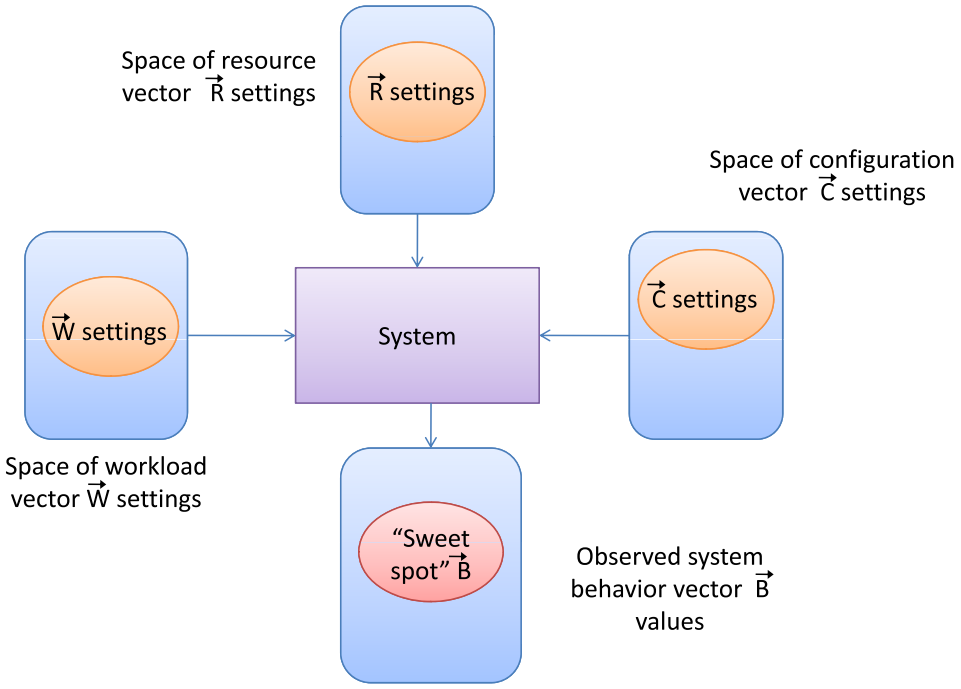

Figure 1.1: System behavior $\vec{B}$ is a function of the system workload $\vec{W}$, system resources $\vec{R}$, and system configuration $\vec{C}$. The space of settings of parameters in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$ can be quite large. The goal of system management is to control the settings of resource vector $\vec{R}$ and configuration vector $\vec{C}$ to keep the system in its "sweet spot".

- *System software configuration.* Tuning the parameters in the database tier of a multitier Web service is a common software configuration task. The goal is to choose a setting for the parameters in $\vec{C}$ for the system software to obtain a desired $\vec{B}$ for a given $\vec{W}$ [86].

- *Admission control.* Request throttling in Web services, as well as intelligent routing of requests among different servers, are common admission control scenarios. The goal is to configure the system, i.e., find a setting for the parameters in $\vec{C}$, for controlling the workload $\vec{W}$ that is allowed in the system so as to maintain a desired behavior $\vec{B}$ [6].

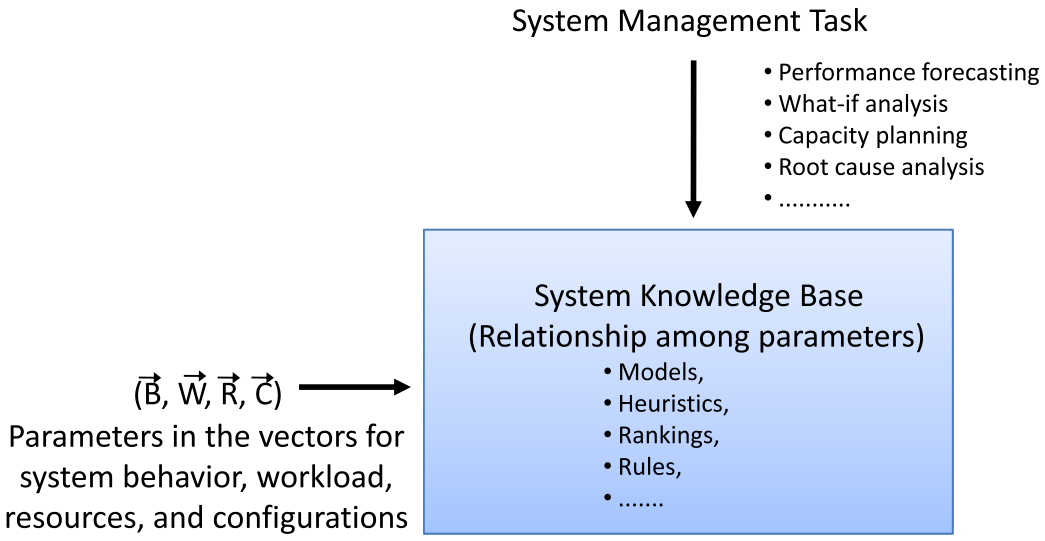

Figure 1.2: A system knowledge base encapsulates the relationship among system behavior, workload, resources, and configuration parameters. The relationship is useful for addressing system management tasks.

1.2 Building a System Knowledge Base

Controlling the settings of system parameters requires a knowledge of how the settings of parameters in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$ impact the system behavior $\vec{B}$. For example, the knowledge might consist of answers to questions such as:

- Which parameters have a significant impact on system behavior?
- Which parameters *interact* significantly?
- What is the relationship among the parameters comprising the system behavior $\vec{B}$ and system workload $\vec{W}$, resources $\vec{R}$, and configuration $\vec{C}$?

A *system knowledge base* encapsulates answers to such questions. It consists of any relationship among the system parameters that is useful for addressing system management tasks; see Figure 1.2. Such a knowledge base is a prerequisite for a *management controller* that must make informed management decisions for tasks like

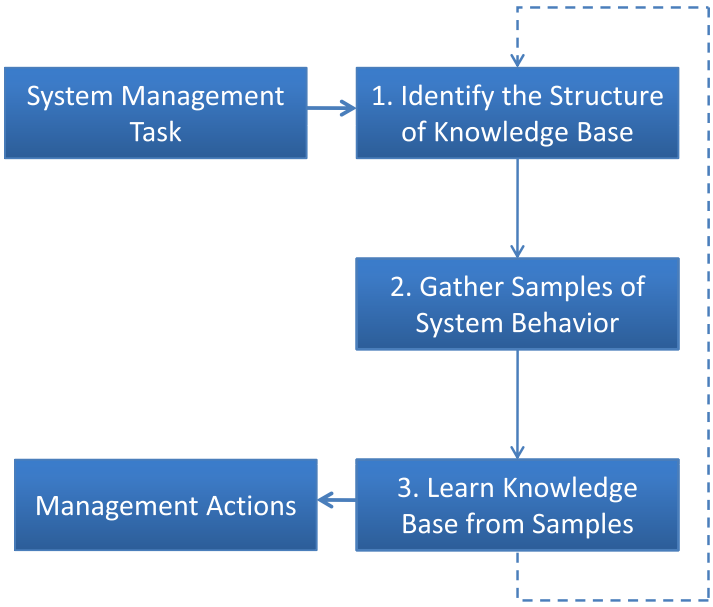

Figure 1.3: High-level steps involved in building the system knowledge base for system management.

resource provisioning, resource allocation, and application and system design [45, 60, 63]. The effectiveness of the management decisions by the controller depends on the accuracy of the knowledge base.

The management controller that uses the knowledge base may be different from the controller that builds it. This dissertation focuses on the mechanisms and the policies for the controller that builds the knowledge base. Figure 1.3 illustrates the high-level steps that are involved in building the knowledge base. The steps are described as follows:

1. **Identify the Structure of the Knowledge Base.** The first step consists of identifying the *structure* of the knowledge base that is required to address the management task. The structure determines the parameters that go into the knowledge base and the relationship among the parameters. The specific system domain and the management tasks determine the adequate structure.

   For a large variety of system management tasks, the knowledge base consists of

models that approximate the system behavior as a function of the parameters that affect the system. For example, the goal of resource provisioning in network computing systems is to assign compute, network, and I/O resources to applications in an efficient manner [81]. A model that can predict the impact of alternate resource assignment choices is useful for making efficient assignment choices [21]. Here, the structure of the knowledge base consists of the parameters that serve as input to the model, and one or more functions that capture the relationship among the parameters to predict the impact of any given resource assignment.

Sections 1.5 and 1.6 present concrete examples of the system knowledge base from three system domains: Web services, batch computing, and storage servers.

2. **Gather Samples of System Behavior.** The controller needs samples of system behavior to build the knowledge base that is identified in Step 1. A *sample* consists of an observation of system behavior for a given setting of system parameters, i.e., a $\langle \vec{W}_i, \vec{R}_i, \vec{C}_i, \vec{B}_i \rangle$ tuple, where i represents the ith sample, and $\vec{B}_i$ the behavior observed on a fixed setting of parameters that determine the workload $\vec{W}_i$, resources $\vec{R}_i$, and configuration $\vec{C}_i$.

3. **Learn the Knowledge Base from Samples.** The controller applies some induction or learning techniques on a table of samples to learn the knowledge base. The choice of techniques depends on the structure of the knowledge base and the desired accuracy of the knowledge base. For example, if the knowledge base consists of a system model then the techniques may range from simple linear regression to more sophisticated regression trees [24, 92]. In closed-loop systems, the knowledge base thus attained may further provide feedback to Step 1 in order to refine the structure of the knowledge base [67].

1.3 A Fundamental Challenge: *Sampling*

Each of the steps in Figure 1.3 is a research challenge in itself [45]. The research community has given significant attention to Steps 1 and 3, e.g., [24, 112, 66]. However, fundamental to the overall process is the set of samples of system behavior upon which the knowledge base is built. The samples must be representative of the system behavior that is or could be encountered in practice. Without such a set of samples, the controller may not be able to learn an accurate knowledge base, irrespective of the techniques used in Steps 1 and 3.

Moreover, the decisions made in Steps 1 and 3 may be incorrectly influenced by a set of samples that do not represent the complete system behavior. To illustrate, suppose the knowledge base consists of a simple linear regression model that predicts system performance as a function of some parameters as shown in Equation 1.2. In this model, the X_is are the input parameters to the model, and a_is are the model coefficients associated with the parameters.

$$P = a_1X_1 + a_1X_2 + \ldots + a_nX_n + c \tag{1.2}$$

In Step 1, the controller must decide which parameters or X_is to include in Equation 1.2. Based on the analysis of available samples the controller may incorrectly conclude that certain parameters have little or no influence on the system behavior, and hence exclude those parameters from the model.

Some of the challenges involved in building a representative set of samples are as follows:

- *Cost of Acquiring a Sample.* Acquiring a sample may have a significant cost; a sample may "cost" time or use of extra system resources. For example, a sample may represent a complete run of a batch application on a set of resources

assigned to run the application [102]. Higher costs—e.g., for long-running batch tasks—limit the rate at which samples can be acquired.

- *Number of Parameters.* The number of workload, resource, and system configuration parameters can be quite large (space of $\vec{W}, \vec{R}, \vec{C}$ in Figure 1.1). The samples must expose the impact of important parameters. As the *dimensionality* of the set of parameters increases, the number of samples needed to learn an accurate knowledge base can increase exponentially. Acquiring samples corresponding to a mere 1% of a 10-dimensional space with 10 distinct values per dimension and average sample-acquisition time of 5 minutes, takes around 951 years!

  For example, in RUBiS—a popular three-tier open source ecommerce benchmark [22]—the workload $\vec{W}$ involves a mix of 26 transaction types. The database tier of RUBiS also has hundreds of database configuration parameters. These combined with the hardware resource parameters, e.g., CPU and memory resources allocated to each tier, contribute to a large space of parameters that affect RUBiS' performance.

- *Parameter Interactions.* A subset of the parameters in $\vec{W}$, $\vec{R}$, and $\vec{C}$ vectors may also interact with each other. For example, in a multitier Web service, the memory size at the application tier can control the effectiveness of the buffer pool size in the database tier; or a typically CPU-bound workload might become I/O-bound for some value of system configuration parameters, and vice versa. The samples must expose the impact of all the relevant parameter interactions.

- *Operating Range of Parameters.* The samples must take into account the settings of each parameter in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$ such that the settings capture the operating range for each parameter.

1.4 Active Sampling

This dissertation presents an active sampling approach to acquire samples of system behavior proactively, automatically, and in a principled manner. Active sampling acquires samples of system behavior by planning *experiments*. An *experiment* consists of conducting a *run* of an application workload for a fixed setting of parameters in $\vec{W}$, i.e., $\vec{W}_i$, on a fixed setting of parameters in $\vec{R}$ and $\vec{C}$, i.e., $\vec{R}_i, \vec{C}_i$, to collect one sample of system behavior, i.e., a $\langle \vec{W}_i, \vec{R}_i, \vec{C}_i, \vec{B}_i \rangle$ tuple. Table 1.2 presents an example. The table shows a list of three parameters, *CPU*, *Memory*, and *Arrival Rate*, and the response time of a Web service on eight specific settings of these parameters. In this example, $\langle \vec{W}, \vec{R}, \vec{C} \rangle = \langle Arrival\ Rate, CPU, Memory \rangle$.

The state of the art often consists of observing the system in its normal operation passively to gather the samples. While recent research has shown promise with such an approach [45], it suffers from several drawbacks. Samples collected only by passive observations of system behavior may not be representative of the full workload and system operating range—e.g., system behavior on a flash crowd may never be observed—limiting the accuracy of the knowledge base learned from such samples. Moreover, passive sampling cannot establish cause and effect relationship between the change in settings of system parameters and system behavior since the effect of different parameters is not explicitly controlled [53]. Even when the samples are generated offline, either the techniques used to generate them are unscalable beyond a few parameters [121], or the samples do not adequately cover the entire system operating range, leading to point studies and brittle claims [110, 75].

Principled active sampling seeks to reduce the time before a reasonably accurate knowledge base is available while addressing the sampling challenges outlined above. Figure 1.4 illustrates the difference between the alternative sampling approaches. The x-axis shows the progress of time for collecting samples and learning the knowledge

Table 1.2: Samples of system behavior for different settings of three parameters.

CPU	Memory	Arrival Rate	Response Time
1 GHz	1 GB	5 req/s	4 ms
2 GHz	1 GB	5 req/s	1 ms
1 GHz	2 GB	5 req/s	2 ms
2 GHz	2 GB	5 req/s	1 ms
1 GHz	1 GB	10 req/s	10 ms
2 GHz	1 GB	10 req/s	5 ms
1 GHz	2 GB	10 req/s	9 ms
2 GHz	2 GB	10 req/s	3 ms

base (e.g., a model), and the y-axis shows the accuracy of the knowledge base learned from the samples available so far. While passive sampling may never collect a fully representative set of samples, ad hoc active sampling can miss the relevant samples, or take a long time to converge to an accurate knowledge base.

For example, *one-parameter-at-a-time* active sampling conducts experiments that change the setting of parameters one parameter at a time while keeping the value of other parameters constant. Such an active sampling approach will expose the impact of each individual parameter on system behavior, but it will not expose the interactions among the parameters. On the other hand, *brute-force* active sampling conducts experiments that consider all possible combinations of the settings for each parameter. This approach exposes the impact of all the parameters and parameter interactions, but can take a long time to collect all the relevant samples (Section 1.3).

In addition to providing the relevant samples for learning an accurate knowledge base, active sampling also enables stronger claims about cause and effect relationship between parameters; system behavior observed from experiments that control the settings of parameters explicitly enable such relationships to be studied more effectively [85].

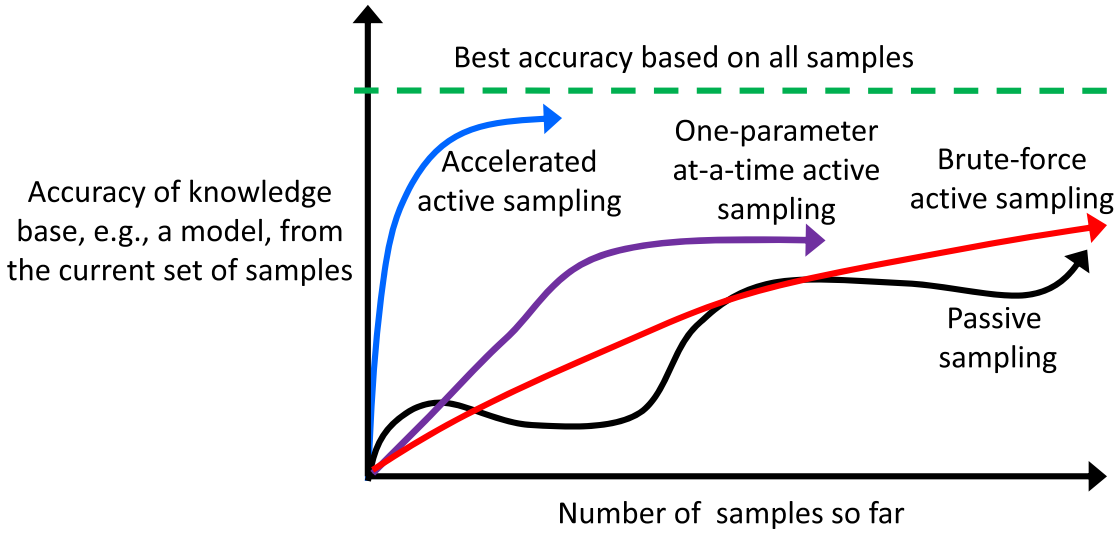

Figure 1.4: Quick learning of an accurate knowledge base from accelerated active sampling. While passive sampling may never gather enough samples to learn an accurate knowledge base, active sampling without proper planning can take a long time to collect the samples (brute-force active sampling), or even miss important samples (one-parameter-at-a-time active sampling). For all the sampling approaches, as the accuracy of the knowledge base converges to its best possible value, it may have periods where it drops depending on the samples that are used to learn the knowledge base and evaluate its accuracy.

1.5 Research Questions

The goal of this dissertation is to design, develop, evaluate, and apply policies and mechanisms to build an accurate knowledge base in an automated, efficient, and feedback-driven manner. It considers the research challenges involved in each step of Figure 1.3, while focusing on Step 2 (the sampling step) and its interactions with other steps. This dissertation addresses the following research questions:

1. **How to represent the knowledge base?** What is a practical way to represent the system knowledge base that captures: (a) the characteristics of the application workload, e.g., its CPU- or I/O-intensity; (b) the impact of resources assigned to the system, e.g., impact of varying CPU, memory, and storage hardware characteristics; and (c) the system configuration choices, e.g.,

the settings of the parameters in the system configuration files?

The goal is to formulate the knowledge base such that it is easy to understand, build, and use. This dissertation considers the following structures for the knowledge base which Section 2.1 further discusses.

- **Rankings.** A ranking of parameters and parameter interactions in order of their impact on system behavior.
- **Models.** A model that predicts system behavior as a function of parameters. This work explores a spectrum of approaches ranging from a priori application models to black-box models that require little or no prior information of applications and systems.
- **Response Surfaces.** A response surface maps the system behavior over a particular region of interest, i.e., a region defined by settings of parameters in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$.

2. **What are the mechanisms for conducting experiments?** The controller needs mechanisms to conduct experiments for gathering the samples. The goal is to generate the required samples in an on-demand fashion while making efficient use of the resources available to conduct the experiments. The following issues arise.

   - **Knobs.** Experiments require "knobs" to set the values of parameters in $\vec{W}$, $\vec{R}$, and $\vec{C}$. What are these knobs and what mechanisms are needed to expose these knobs to the controller?
   - **Infrastructure.** Once the controller has the knobs to set the values of parameters in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$, it needs an infrastructure to conduct the experiments. Experiments done on a production system may cause interference

with the production application runs, especially if the system has little or no virtualization support. What kind of 'workbench' does the controller need such that the 'workbench' enables automatic and feedback-driven experiments, and provides sensors to monitor the instrumentation data for each experiment?

Section 2.3 discusses the mechanisms for gathering samples that this dissertation considers.

3. **What are the policies for conducting the experiments?** The space of $\vec{W}$, $\vec{R}$, and $\vec{C}$ parameters and the corresponding parameter settings can be quite large (Section 1.1). Without a systematic exploration of these parameters, the controller may take a long time to collect the relevant samples for learning an accurate knowledge base. Hence, the controller requires policies for conducting the experiments that consider the following.

   - **Choice of Samples**. Which samples expose the impact of all the relevant parameters, interactions between them, and their operating range? How can the controller identify and collect such samples quickly?
   - **Sampling Order**. What is the order in which the controller should conduct the experiments such that it can learn a reasonably accurate knowledge base quickly?
   - **Feedback-driven Sampling**. How can the controller use the feedback from the samples collected so far to further guide the choice of new experiments?
   - **Cost-Aware Sampling**. How can the controller adapt the time and resources that are used to gather the relevant samples to the *desired accuracy* of the knowledge base?

Sections 2.2 and 2.5 provide an overview of the policies that this dissertation considers. Chapters 3, 4, and 5 present the policies in the context of different system domains.

1.6 Contributions

This dissertation makes the following contributions:

1. **A general experiment-driven framework for building the system knowledge base.** This dissertation presents a general experiment-driven framework for building the system knowledge base in an automatic, proactive, and timely manner by planning and conducting experiments. The framework incorporates: (a) mechanisms to conduct experiments for three important system domains: Web services, batch computing, and storage servers; and (b) policies for automatic planning of experiments to explore a large space of parameters and interactions efficiently.

   The framework combines techniques from two fields—*Design of Experiments* and *Active Machine Learning*—to develop experiment-planning policies. The dissertation demonstrates that the policies enable accurate learning of the knowledge base in an efficient, automated, and feedback-driven manner. Chapter 2 presents the overview of the overall framework.

2. **Experiment-planning policies to identify important parameters and interactions.** This dissertation develops experiment-planning policies to identify the important parameters and interactions among parameters that affect system performance by addressing the following queries.

   - Quantify the linear and non-linear *impact* or *effect* of each parameter in

workload $\vec{W}$, resource $\vec{R}$, and configuration $\vec{C}$ parameters on system performance P, and rank the parameters in order of their effect.

- Quantify the *interactions* among the parameters in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$ that affect P, and rank these interactions in order of their effect.

Chapter 3 presents and evaluates the experiment-planning policies that address the queries in the context of Web service management. The goal of a Web service provider is to meet the service level objectives of a client and make efficient use of resources to maximize its revenue. Hence, the service provider must not only understand the impact of many (often hundreds) service configuration parameters on the performance of a service for each hosted service, but also understand the interaction of the parameter settings with each other and with the allocated resources. This dissertation uses the experiment-driven framework to quantify the impact of these parameters and their interactions.

3. **Experiment-planning policies to learn models that predict system performance.** This dissertation presents experiment-planning policies that collect samples to parameterize a predetermined model accurately and quickly. The policies are model-agnostic and applicable for collecting samples to learn any given model. This work applies the policies to learn a number of models such as models that predict Web service response time and throughput (Chapter 3), models to predict the execution time of batch applications (Chapter 4), and models that guide the search for the saturation throughput of a storage server (Chapter 5).

4. **Experiment-planning policies for automated storage server benchmarking.**

   Obtaining the *saturation throughput* or the *peak rate* of storage servers for a

given workload and server configuration is a key building block for systematic mapping of server peak rates across a larger space of server workloads and configurations. This dissertation uses the experiment-driven framework to do systematic *response surface mapping* that plots the peak rate of a storage server over a space of workloads and server configurations. The benchmarking policies obtain accurate peak rate measures with a target confidence level and accuracy at low cost. Chapter 5 presents the policies in the context of NFS server benchmarking.

5. **Application performance models for resource planning of batch applications in a networked utility setting.**

   To schedule a batch application on the available resources automatically and efficiently, a system requires accurate estimates of the application's runtime on a candidate resource assignment. The dissertation presents an *end-to-end* application performance model that predicts the execution time of a batch application as a function of compute, memory, and network resources assigned to it and the properties of the data that the application processes. The work shows that the model can be learned from commonly available *noninvasive* instrumentation data that does not require any changes to the application or the underlying system. Chapter 4 presents the model, its validation, and its usage for resource planning of batch applications.

Chapter 2

Experiment-Driven Framework

Figure 2.1 presents the overall experiment-driven framework. The figure is similar to Figure 1.3 with an expanded sampling step (Step 2 in Figure 1.3). The sampling step consists of a *workbench* of hardware resources that is used for conducting the experiments. The experiments are issued by a *workbench controller* in order to collect the samples for learning the knowledge base. The controller incorporates: (a) mechanisms to configure and execute the experiment; and (b) policies that plan the experiments.

The experiment-planning policies determine the choice and sequence of experiments, the experiment runlength, and the number of times to repeat the experiment for statistically accurate results. The policies take into account the knowledge base being built, constraints such as the available resources in the workbench and the target accuracy of the knowledge base, and feedback from the analysis of samples from previous experiments.

Figure 2.2 shows the overall sequence of operations in the experiment-driven framework. The first step is to map the system management task to the knowledge base that is needed to address the task (Step 1 in Figure 1.3). Next, the workbench controller plans and conducts experiments according to the experiment-planning policies. The instrumentation data from the experiments is processed to build a table of samples (see Table 1.2 for an example). The feedback from the analysis of the sample table goes as an input to the experiment-planning policies to determine the next experiment(s). At any point during the execution of the operations, the available table of samples may be analyzed to learn the knowledge base (Step 3 in Figure 1.3).

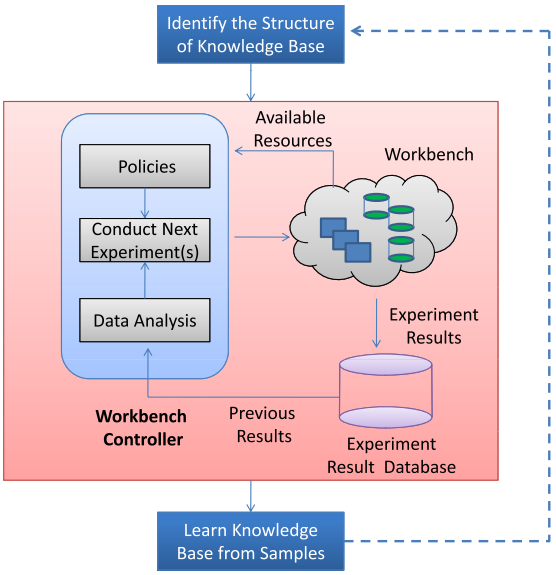

Figure 2.1: Experiment-driven framework for proactive learning of the system knowledge base.

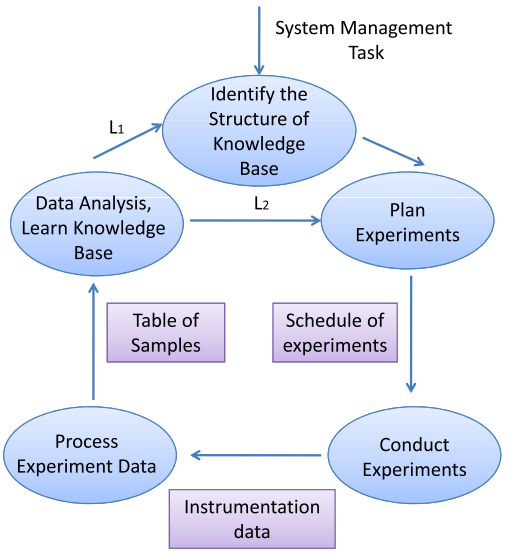

Figure 2.2: Sequence of operations in the experiment-driven framework.

Table 2.1: Mapping of common management tasks to the knowledge base that is useful for accomplishing the tasks.

Management Task	**Knowledge Base**
Identify important parameters that affect system behavior, e.g., [120, 57]	Numerical scores that quantify the impact of parameters
Online provisioning, e.g., [113, 31]	Predictive models
System benchmarking, e.g., [104, 36]	Response surfaces of system behavior

Sections 2.1-2.5 discuss each operation in detail.

2.1 Identify the Structure of the Knowledge Base

The first step is to identify the structure of the knowledge base that is needed to address a system management task. The appropriate choice and representation of the knowledge base depends on the management task, the overall management goals, and system parameters as shown in Figure 1.2. Section 1.2 characterizes the system knowledge base.

Table 2.1 presents a mapping of some common management tasks to the knowl-

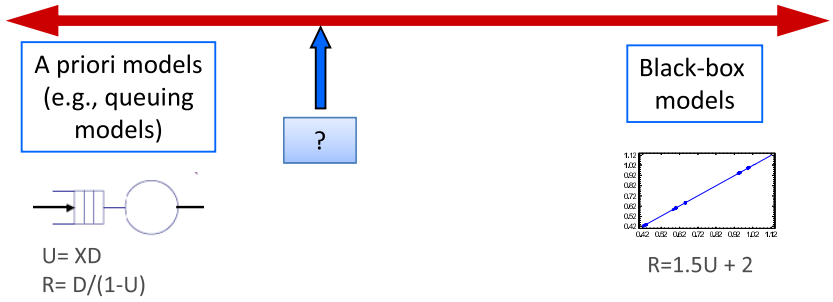

Figure 2.3: A spectrum of modeling alternatives ranging from a priori models to black-box models.

edge base that is useful to accomplish the tasks. A discussion of the mappings follows.

Numerical scores of parameters. A large space of parameters and interactions among them can affect the system behavior. A common system management task is to identify the parameters and interactions that have a significant impact on system behavior [120, 57]. For example, tuning the settings of hundreds of configuration parameters in a database system [116] requires the knowledge of parameters that have the most impact on database performance [82].

By quantifying the impact of each parameter and interaction on system performance in terms of numerical scores, we can use the scores to identify the parameters and interactions that are important for system behavior. Hence, the knowledge base consists of the numerical scores for each parameter and interaction. Chapter 3 presents the use of the experiment-driven framework to rank the parameters and interactions in order of their impact on Web service response time and throughput. The numerical scores of parameters and interactions are the basis for such rankings.

Predictive models. Models are essential for tasks such as forecasting, diagnosis,

and repair of failure conditions [24], capacity planning [113], online provisioning [31] and *what-if* analysis [79]. This dissertation explores a spectrum of modeling alternatives ranging from a priori models to black-box models that require little or no prior knowledge of applications and systems. Figure 2.3 illustrates the spectrum.

A priori models are attractive in part because of their ability to capture complexity in the parameters and the structure of an analytical model. For example, queuing models can easily represent the impact of factors such as latency hiding, arrival rate, concurrency, and queuing on application behavior explicitly in the model parameters. As a result, the system can use such models to reason about application and system behavior beyond the samples used to learn the model parameters—increasing their explanatory power. Several researchers have explored such models for automated system management, e.g., [105, 112, 66, 113, 31]. However, a priori models are less general because they often require a prior understanding of application, e.g., the internal application structure and its prefetching and queuing behavior. In addition, the system may require the use of sophisticated instrumentation to gather the samples for parameterizing the model.

On the other hand, black-box models are general since they require little or no prior understanding of application and system behavior. As a result, black-box models are becoming popular for managing the increasing complexity in computer systems, e.g., [24, 45, 60, 123]. However, the accuracy of such models depends heavily on the range of system behavior seen in the samples used to parameterize them, and they might be inaccurate for extrapolation beyond that range [59]. Section 1.3 presents the challenges involved in gathering a representative set of samples.

Both a priori and black-box modeling must address a common set of questions: (a) what are the parameters that serve as input to the model; (b) what is the structure that relates the different parameters; and (c) how to identify and collect the samples

that are required to parameterize the model accurately. In this dissertation, we use the experiment-driven framework to address the three questions in the context of specific system domains, while focusing on developing policies for selecting the samples automatically and quickly to parameterize any given model.

Chapter 3 presents the use of the experiment-driven framework to identify the set of important parameters that affect system behavior in the context of modeling the performance of Web services. Chapter 4 presents the use of framework to iteratively refine the structure of the model that predicts the execution time of batch applications. Chapters 3, 4, and 5 present mechanisms and policies to collect the samples for learning accurate models quickly and automatically. Chapter 6 further discusses the role of the experiment-driven framework in addressing the questions raised above.

Response surfaces. Response surfaces map the performance of a system over some region of interest that is defined by the combinations of settings of parameters. Knowledge of such response surfaces is crucial for understanding the performance tradeoffs of adding resources and/or changing configurations for different workloads.

Chapter 5 presents the use of the experiment-driven framework to map such response surfaces accurately and efficiently in the context of benchmarking a storage server. It presents response surfaces that plot the performance of a storage server across different workloads as a function of two important parameters: the number of disks attached to the storage server and the number of I/O threads in the server. The surfaces show that adding more disks can improve the storage performance only if there is a sufficient number of I/O threads to issue requests to those disks, and that the appropriate number of I/O threads is workload-dependent.

Table 2.2: An example of an experiment design. 1 represents the choice of high value for a parameter, and -1 represents the choice of low value for a parameter. The high and low values for a parameter are chosen from its overall operating range.

CPU	Memory	Arrival rate
-1	-1	-1
1	-1	-1
-1	1	-1
1	1	-1
-1	-1	1
1	-1	1
-1	1	1
1	1	1

2.2 Plan Experiments

After identifying the structure of the knowledge base, the next step is to *plan* a set of experiments to collect the samples for building the knowledge base. The total space of possible experiments is exponential. For a total of n parameters in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$ where each parameter can be set to l possible values, the total number of experiments is l^n. The experiment-planning policies (Figure 2.1) must choose experiments that can generate the samples to build an accurate knowledge base quickly while making efficient use of resources for conducting the experiments. In this dissertation, the controller policies leverage the work done in two fields to explore the large experiment space: design of experiments [53] and active machine learning [96]. Both fields are based on rigorous theoretical foundations, and offer a range of techniques to guide the choice of experiments.

The controller policies produce an *experiment design* that determines the choice and sequence of experiments. An experiment design is a table of experiments, where each row of the table consists of one experiment. An experiment consists of setting each parameter in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$ vectors to a value from a legal range of values for the parameters, i.e., their operating range. Table 2.2 shows an example of an experiment

Table 2.3: Mapping of the knowledge base to the experiment design that generates samples to learn an accurate knowledge base quickly.

Knowledge Base	**Experiment Designs**	**Design Description**
Importance scores of parameters	Screening	Quantify the impact of parameters and interactions between the parameters
Model	Model Learning	Learn accurate models with a minimal number of samples
Response surfaces of system behavior	Response Surface Methodology (RSM)	Map system behavior over a particular region of interest, identify optimal operating range of parameters, optimize system behavior

design of 8 experiments with 3 parameters. The design considers only two values per parameter: the high value, represented by 1, and the low value, represented by -1. Note that the design considers all the combinations of high and low values, and hence the total number of experiments is: $2^3 = 8$. Such a design is called 2^n factorial design, where n is the number of parameters (Section 3.5).

The experiment design that the policies produce depends on the knowledge base that the controller is building, which in turn depends on the management task. Table 2.1 shows the mapping of some common system management tasks to the knowledge base that is useful for addressing the tasks. The controller policies must further map the knowledge base to the experiment design. Table 2.3 summarizes the broad category of designs that this dissertation considers, and shows the mapping for the knowledge base in Table 2.1. A discussion of these categories follows.

- **Screening [53].** As the name suggests, screening designs *screen* the large space of parameters to identify the parameters and interactions that have the most significant impact on system behavior. These designs use a small fraction of the total possible space of experiments. For example, Plackett-Burman (PB) [78] screening design uses $O(n)$ experiments to quantify the impact of n parameters on system behavior.

Chapters 3 and 4 present the screening designs in the context of ranking parameters and interactions in order of their impact on Web service performance and execution time of batch applications respectively. Chapter 4 also presents the use of such designs to rank a set of functions in order of their ability to capture the execution time of batch applications.

- **Model Learning [96].** The goal of these designs is to identify experiments to generate samples that can be used to learn a model with a specific structure using a minimal number of experiments. Such designs are useful for learning an accurate model under some time, resource, or other budget constraints. For example, if a user can afford to conduct only 10 experiments to learn a specific model, then these designs identify the 10 experiments that generate samples which will maximize the accuracy of the learned model.

  Chapters 3, 4, and 5 present the model-learning designs in the context of learning models that predict Web service performance, execution time of batch applications, and map a response surface for benchmarking storage servers respectively. While the designs in Chapter 3 focus on learning models with any given structure, the designs in Chapter 4 learn a model with a specific structure that incorporates prior knowledge about application behavior. The designs in Chapter 5 focus on mapping accurate response surfaces efficiently.

- **Response Surface Methodology (RSM) [78].** Typical applications of RSM designs include: (a) mapping the system behavior over a particular region of interest. The region of interest is defined by settings of parameters in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$; and (b) determining the values of parameters in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$ that result in optimal or target system behavior. Chapter 5 discusses the use of these designs in the context of storage server benchmarking to obtain benchmarking results with

Algorithm 1: Conduct an Experiment

Input: A specified value l_i for each parameter F_i in $\vec{F}$, where $\vec{F}$ is a subset of parameters in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$; system behavior metric of interest B.
Output: Measure B for $\langle F_1 = l_1, \ldots, F_n = l_n \rangle$ and return a complete sample.

1) Obtain hardware resources for the experiment;
2) Create a resource configuration, i.e., CPU, memory, network, and disk configuration, with the values set as per $\langle F_1 = l_1, \ldots, F_n = l_n \rangle$;
3) Instantiate the application on the resources;
4) Configure the system configuration and application workload according to the values in $\langle F_1 = l_1, \ldots, F_n = l_n \rangle$;
5) Start instrumentation;
6) Run the application workload for a predetermined period;
7) Stop instrumentation;
8) Collect instrumentation data during the experiment which when processed yields the $\langle F_1 = l_1, \ldots, F_n = l_n, B \rangle$ sample that goes into a database of samples;

target confidence and accuracy at low cost.

The experiment-planning policies determine how to use the designs in a unified manner to learn an accurate knowledge base quickly. Depending on the knowledge base that the controller is building, the policies may use the designs independently or in combination. Chapters 3 and 4 present experiment-planning algorithms that show how to use the screening and model-learning designs sequentially to guide the choice of experiments for: (a) ranking parameters and interactions in order of their impact on system performance; and (b) learning models that predict system performance.

2.3 Conduct Experiments

Algorithm 1 outlines the steps that controller takes for conducting an experiment. The controller obtains and prepares the resources according to the settings of parameters in $\vec{R}$, deploys the application on the resources, configures the system according to the settings of parameters in $\vec{C}$, and configures the application workload according to the settings of parameters in $\vec{W}$. It also starts, stops, and records instrumentation data which when processed yields a $\langle \vec{W}_i, \vec{R}_i, \vec{C}_i, \vec{B}_i \rangle$ sample for the *ith* experiment in the experiment design. The implementation of each step depends on the knowledge base that the controller is building and the specific system domains such as Web services and scientific applications. Chapters 3, 4, and 5 present concrete instances of each step.

The controller conducts the steps in Algorithm 1 programatically by leveraging existing mechanisms to generate a range of workloads, resources, and system configurations. The infrastructure for conducting the experiments consists of a testbed that also provides programmatic control to allocate resources for an experiment, deploy the experiment according to the settings of parameters in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$, collect and store streams of instrumentation data from the experiment, and repeat the experiment if necessary.

2.3.1 Mechanisms for Conducting Experiments

The controller requires mechanisms or "knobs" to set the values of parameters according to the settings of parameters in each experiment of the experiment design, i.e., knobs to set the value of parameters in the workload vector $\vec{W}$, the system resource vector $\vec{R}$, and the system configuration vector $\vec{C}$. We discuss each in turn.

Setting $\vec{W}$. Experiments require that the controller have the ability to generate a

range of settings in $\vec{W}$ to expose the wide variety of workloads that a system may encounter in practice, e.g., to expose the system to a flash crowd scenario. General knobs to set the parameters in $\vec{W}$ are elusive. One solution is to use application-specific workload generators to discover and vary the important parameters for a workload in a realistic fashion, in order to capture the system behavior for a given $\vec{W}$ [104].

A range of reconfigurable benchmarks already exist to emulate workloads for key application classes of interest. In Chapter 3 we use the synthetic workload generators that come prepackaged with Web services such as RUBiS [22] and TPC-W [109] to generate a wide variety of Web workloads. Chapter 4 uses GAMUT [77], an application emulator, that offers knobs to create batch workloads with varying CPU and I/O demands. In Chapter 5 we use *Fstress* [5], a synthetic storage workload generator that offers knobs to configure the properties of the workload's dataset and its request mix.

Setting $\vec{R}$. The heterogeneity in physical cluster resources [62] allows a range of settings for the resource vector $\vec{R}$. Alternatively, virtualization technologies such as Xen [33] and VMWare [106] offer knobs for controlling the settings of hardware resource vector $\vec{R}$ even on homogeneous resources. By using the controls offered by virtualization, it is easy to create virtual machines (VMs) with varying amounts of CPU, memory, and I/O resources. While the knobs offered by the virtualization technology are not exhaustive, e.g., ability to control CPU cache size does not exist, they make it possible to observe a workload's behavior on a range of system resources.

Setting $\vec{C}$. Most system and application software already provide knobs for setting the configuration parameters in the configuration vector $\vec{C}$. For example, a database

system provides means for tuning a large number of parameters in its $\vec{C}$ [116].

2.3.2 Experiment Workbench

In this dissertation, the controller uses an initial prototype of *Automat* [122] as a workbench for conducting experiments. Automat is a testbed targeted for research into mechanisms and policies for self-managing systems. Automat is layered upon Shirako, a toolkit for secure, on-demand leasing of shared networked resources [58]. Automat provides an infrastructure to configure a chosen setting of parameters in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$ automatically (Steps 1-4 in Algorithm 1). Automat can instantiate experiments in parallel; concurrent experiments cut down the elapsed time required to conduct a set of experiments. Furthermore, Automat provides support for pluggable monitoring to collect the instrumentation data from each experiment (Steps 5-8 in Algorithm 1).

2.3.3 Repeating Experiments

For each experiment, the controller must choose the *runlength*, which is the time interval over which to observe the system behavior, and the *number of independent trials* for that experiment. Given the inherent variability in the experimental process [72], one trial is usually insufficient to measure the system behavior accurately. The best that can be done in such a setting is to make a *probabilistic claim* about the *interval* in which the experiment result lies, based on the observations from multiple independent trials [59].

Chapter 5 presents experiment-planning policies to determine how long to run an experiment and how many times to repeat an experiment in the context of storage server benchmarking. The goal of the policies is to attain target confidence and accuracy in experimental results at low cost.

2.4 Collect and Process Instrumentation Data

The controller collects the instrumentation data from the experiments and processes it to yield a table of samples. Table 1.2 shows an example of a sample table. The overall framework is independent of the instrumentation methodology as long as the tools for collecting the data are available. This dissertation uses *noninvasive* instrumentation data—data that a system can easily collect using commonly available system tools, and without modifying the application or system. The premise is that for the knowledge base to be of practical use, the controller should be able to generate samples from instrumentation data that is widely available.

Sophisticated instrumentation, e.g., modifying application sources or binaries [19], may yield more accurate data. However, there are several barriers to using such instrumentation, e.g., slowdown of applications, sophisticated tools for instrumenting application source or binaries, and limited or no prior knowledge about the applications. At the same time, noninvasive instrumentation may be of little use if it yields insufficient data or inaccurate knowledge base.

2.5 Analyze Samples to Learn the Knowledge Base

Once the sample table is available, the controller can analyze it to build the knowledge base. The data analysis techniques depend on the knowledge base that the controller builds. For example, if the controller is building regression models, then the data analysis consists of learning the coefficients of a regression model, e.g., by using least squares estimation [59]. The data analysis has several other goals in addition to using standard techniques for learning the knowledge base.

- **Establish Accuracy**. The analysis must establish the accuracy of the knowledge base. To do so, the controller must identify appropriate accuracy metric(s).

Often one metric may be insufficient to evaluate the accuracy of the knowledge base, and hence it must consider several metrics. Chapters 3 and 4 present the multiple accuracy metrics that this dissertation considers.

- **Guide the Structure of the Knowledge Base**. Data analysis must guide the controller in refining the structure of the knowledge base in Step 1; see loop L_1 in Figure 2.2. The analysis of existing samples may give insights into a better choice and representation of the knowledge base. For example, suppose the goal is to build an accurate linear regression model, and experiments are conducted according an experiment design to collect samples for learning the model. The analysis of the samples might reveal the presence of significant non-linear effects. This allows the controller to refine the structure of the model to capture the non-linear system behavior as well. Chapter 4 uses the data analysis to iteratively refine the structure of the model that predicts the execution time of batch applications.

- **Guide the Choice of Experiments**. Data analysis must guide the controller in augmenting the experiment design based on the feedback from the previous experiments; see loop L_2 in Figure 2.2. For example, the analysis may reveal that a parameter that is excluded from a model has a significant impact on the system behavior. In such a case, the controller must generate more experiments to expose the impact of that parameter in the samples. Chapters 3, 4,and 5 show how the analysis of available samples guides further choice of experiments.

2.6 Summary

This chapter presents our experiment-driven framework that incorporates the mechanisms and policies to learn the system knowledge base proactively, efficiently, and

in a feedback-driven manner. The instances of the operations in the framework, as shown in Figure 2.2, depend on the management task, the knowledge base being built, system domain, and constraints such as available resources in the workbench and the target accuracy of the knowledge base. Chapters 3-5 present the use of the overall framework to learn the system knowledge base in the context of Web services, batch applications, and storage servers.

Chapter 3

Management Queries for Web Service Management

This chapter presents the use of experiment-driven framework for building the knowledge base required to process the management queries in the context of Web service management. The policies for building the knowledge base are general and applicable to other system domains as well.

3.1 Background

Databases with custom-written clients dotted the computing landscape for a long time. The advent of the Web brought about *Web services* (or Internet services) composed of "thin clients" that accessed databases through Web interfaces. As the scale of these services increased, stored procedures made their way out of the database into a new tier, the application server, running "business logic." This multitier, database-backed architecture supports many popular Web services such as Amazon, eBay, and Yahoo!. Many enterprises are moving towards a service-oriented architecture by putting their databases behind Web interfaces, thereby providing a well-defined, interoperable method for interacting with their data.

Web services usually have *service level objectives*, or *SLOs*, that specify the measures and properties for an acceptable level of service [54]. For example, an SLO for an online brokerage may stipulate that a certain percentile of all transactions complete within 1 second, regardless of the middleware, databases, or network components involved in overall transaction processing.

Many services have difficulty meeting their SLOs. A recent study [100] found

Table 3.1: Example of factors that affect Web service performance.

$\vec{W}$	Arrival rate of Web requests, Web request mix, number of clients
$\vec{R}$	CPU speed, memory size, number of disks
$\vec{C}$	Database configuration factors such as buffer pool size for index blocks, open table descriptors, and sort operations

that 72% of the top-40 Web sites suffer user-visible problems such as slowdowns and failures. Walmart.com experienced a 10-hour outage during the 2006 U.S. Thanksgiving holiday season, potentially losing customers and revenue [115]. Customers of the most popular online U.S. tax filing service, TurboTax, were unable to file their returns on the eve of the deadline [111]. The prospect of such problems places a huge burden on administrators who manage Web services. The increasing complexity and scale of Web services is making the administrator's task even more difficult. This chapter presents the use of the experiment-driven framework from Chapter 2 to build the knowledge base that will enable administrators to make informed system management decisions.

3.2 Problem Statement

The performance of a Web service is a function of its workload, the configuration of each Web service tier, and the hardware resources allocated to each tier. Each of these may be characterized by a vector of factors, as summarized in Table 3.1.

Performance $\vec{P}$. We characterize the performance of a Web service by metrics such as average service response time, average service throughput, and average number of errors returned by the service in a given time interval.

Workload $\vec{W}$. A Web service workload consists of: arrival rate of clients, the class

or type of clients, and the number of concurrent clients that access the Web service. The service priority for clients characterizes the client class, and the usage profile determines the client type. As an example, consider the eBay-like auction service *RUBiS* [22]. RUBiS's workload $\vec{W}$ is characterized by three types of user sessions $\vec{W} = \langle browse, bid, sell \rangle$ where each of *browse*, *bid*, and *sell* represent the number of concurrent clients doing a distinct activity. *browse* sessions are from unregistered users who browse the site. *buy* sessions are from registered users who, in addition to browsing, may bid on items and consult a summary of their current bids, their rating, and the comments left by other users. *sell* sessions are from paid users who put up items for sale.

Resources $\vec{R}$. This workbench controller (Chapter 2, Figure 2.1) uses virtualization technology to control the hardware resources allocated to each service. It allocates virtual machines (VMs) with varying amounts of CPU, memory, and I/O resources to each tier of the service.

Configuration $\vec{C}$. Setting the values of $\vec{C}$ involves modifying the factors in the service configuration or making choices with respect to the system software such as the communication protocol being used between the tiers, the operating system running on the hardware, and the file system used on the storage devices.

Supporting Management Queries on System Behavior. Let $\vec{P}$ be the performance metrics of interest for a Web service. Let $\vec{F} = \langle F_1, \ldots, F_n \rangle$ be the subset of factors in the larger factor space in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$. This work builds the knowledge base required to address the following example queries:

Q_1: Quantify the *impact* or *effect* of each factor in $\vec{F}$ on $\vec{P}$, and rank the factors in

order of effect.

Q_2: Are there strong *interactions* among factors in $\vec{F}$ that affect $\vec{P}$? Rank these interactions in order of their effect on $\vec{P}$.

Q_3: Approximate the function that determines $\vec{P}$ for any given setting of factors in $\vec{F}$.

Impact of Factors: Understanding the effect of various factors on system performance is a prerequisite for system administration tasks like tuning [116] and capacity planning [31]. For example, as part of system tuning, an administrator may need to find out which factors in the resource vector $\vec{R}$ or the configuration vector $\vec{C}$ to change to improve system performance metrics $\vec{P}$ for a given setting of factors in the workload vector $\vec{W}$. Or, in capacity planning, the task may be to find what resources to add to handle a projected increase in the workload $\vec{W}$ at minimum cost.

Impact of Interactions: A big hurdle in isolating the effect of individual factors is the presence of unknown interactions among factors. An interaction between two factors F_i and F_j means that the changes in performance across multiple settings of F_i are significantly different for different settings of F_j. Such interactions make typical "tune-one-factor-at-a-time" efforts ineffective.

Modeling: While the effects of factors and interactions give insight into system behavior, a complex administrative task like SLO maintenance under dynamic workloads may need a model of system behavior that can predict how varying settings of factors in $\vec{R}$ and $\vec{C}$ affects $\vec{P}$ for a range of settings of factors in $\vec{W}$.

	Levels				Contrast Coefficient Vectors										
	F_1	F_2	F_3	P	F_1		F_2	F_3	F_1F_2		F_1F_3		F_2F_3	$F_1F_2F_3$	
					$F_{1,Lin}$	$F_{1,Quad}$	$F_{2,Lin}$	$F_{3,Lin}$	$F_{1,Lin}$ $F_{2,Lin}$	$F_{1,Quad}$ $F_{2,Lin}$	$F_{1,Lin}$ $F_{3,Lin}$	$F_{1,Quad}$ $F_{3,Lin}$	$F_{2,Lin}$ $F_{3,Lin}$	$F_{1,Lin}$ $F_{2,Lin}$ $F_{3,Lin}$	$F_{1,Quad}$ $F_{2,Lin}$ $F_{3,Lin}$
1	L1	L1	L1	1	-1	+1	-1	-1	+1	-1	+1	-1	+1	-1	+1
2	L1	L1	L2	3	-1	+1	-1	+1	+1	-1	-1	+1	-1	+1	-1
3	L1	L2	L1	2	-1	+1	+1	-1	-1	+1	+1	-1	-1	+1	-1
4	L1	L2	L2	3	-1	+1	+1	+1	-1	+1	-1	+1	+1	-1	+1
5	L2	L1	L1	10	0	-2	-1	-1	0	+2	0	+2	+1	0	-2
6	L2	L1	L2	8	0	-2	-1	+1	0	+2	0	-2	-1	0	2
7	L2	L2	L1	1	0	-2	+1	-1	0	-2	0	+2	-1	0	2
8	L2	L2	L2	1	0	-2	+1	+1	0	-2	0	-2	+1	0	-2
9	L3	L1	L1	2.5	+1	+1	-1	-1	-1	-1	-1	-1	+1	+1	+1
10	L3	L1	L2	3.5	+1	+1	-1	+1	-1	-1	+1	+1	-1	-1	-1
11	L3	L2	L1	3	+1	+1	+1	-1	+1	+1	-1	-1	-1	-1	-1
12	L3	L2	L2	3	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
Contrasts					3	-19	-15	2	-1	33	-2	8	0	0	-6
Effects					.75	-4.75	-2.5	0.33	-0.25	8.25	-0.5	2.0	0	0	-1.5
Factorial Sum of Squares (SS)					16.167		18.75	0.333	45.50		3.167		0	1.5	
SS Decomposition					1.125	15.042	18.75	0.333	0.125	45.375	0.5	2.667	0	0	1.5

Figure 3.1: Running example with three factors F_1, F_2, and F_3 with 3, 2, and 2 levels respectively.

3.3 Impact of Factors and Interactions

This section defines more rigorously the notion of *interaction* between factors, and the *impact* or *effect* of factors and factor interactions on *system response*, e.g., Web service performance. A complete treatment of these concepts is available in standard texts on design and analysis of experiments, e.g., [72, 78, 85]. This section illustrates the application of these concepts for system management using a running example in Figure 3.1, which we created to explain the concepts.

Let $\vec{F} = \langle F_1, F_2, \ldots, F_n \rangle$ denote the n factors of interest for a service, and P denote the performance metric of interest. Each factor F_i has an *operating range* that determines the values F_i can take. Each value that F_i can take is called a *level* of F_i. For simplicity, we assume discrete numeric factors in this example, but the

techniques extend to continuous-valued and categorical factors as well. Factor F_i has l_i distinct levels, denoted $L_1, L_2, \ldots, L_{l_i}$. Consider three factors F_1, F_2, and F_3 with 3, 2, and 2 levels respectively. The columns labeled *Levels* in Figure 3.1 show the 12 possible combinations of these three factors, and the next column shows the corresponding value of the performance metric P. Each combination of levels and the system performance for that combination represents a sample. Assume that all these samples are already available. Recall from Section 2.2 that determining which samples to collect is a key challenge, which we address in Sections 3.5.1-3.5.3.

One way to characterize the effect of a factor F_i on P is to measure the mean change in P from the change in level(s) of F_i. For example, from the data in Figure 3.1, we can compute how much the mean value of P changes when F_1 is changed from it lowest level (L_1) to its highest level (L_3); the mean is taken over all samples with that specific level of F_1. F_1 is at level L_1 in the first four samples in Figure 3.1—with mean $P = \frac{1+3+2+3}{4} = 2.25$—and at level L_3 in the last four samples with mean $P = \frac{2.5+3.5+3+3}{4} = 3$. Thus, the change in the mean value of P when F_1 is changed from its lowest level to its highest level—called the *linear effect* of F_1—is 0.75. Figures 3.2(a)–(c) illustrate the linear effects of factors F_1–F_3.

Note that the absolute linear effect of factor F_1 is much lower than that of F_2. However, Figure 3.2(d), that shows the mean value of P for all three levels of F_1, illustrates that changing F_1 from L_1 to L_2 produces a significant change in P, and so does changing F_1 from L_2 to L_3. These changes cancel one another, making the linear effect small. Linear effects cannot capture such non-linear influence since the effects are estimated using only two levels. To estimate higher-order effects we need more than two levels. For example, the *quadratic effect* of F_1 is calculated as $(\bar{P}_3 - \bar{P}_2) - (\bar{P}_2 - \bar{P}_1)$, where $\bar{P}_i$ denotes the mean P at level L_i of F_1. As expected, F_1 has high magnitude of quadratic effect of -4.75.

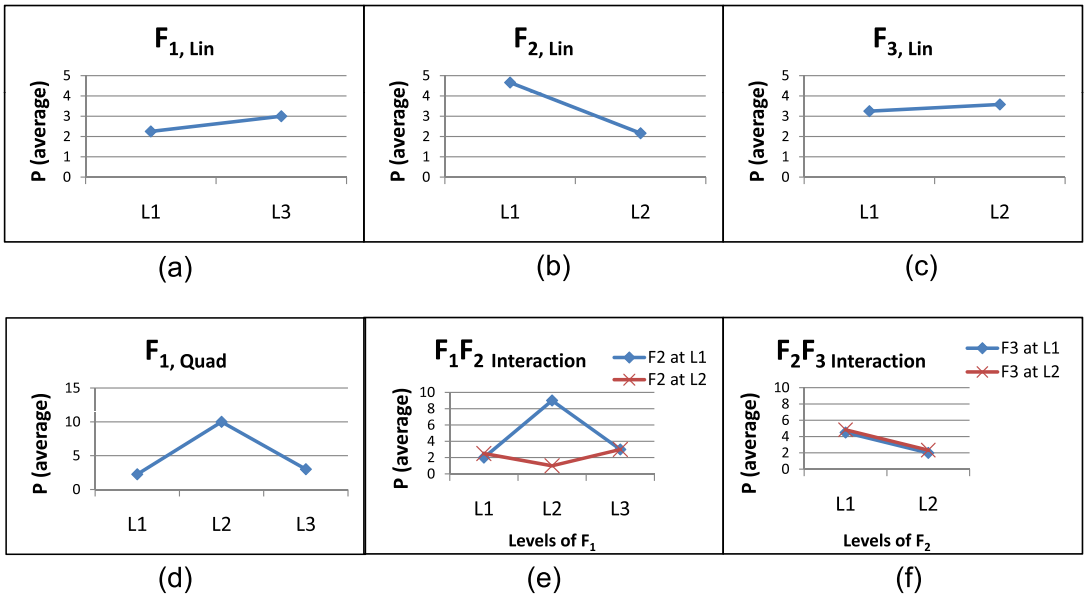

Figure 3.2: Impact of different factors and interactions in our running example. The performance is averaged across all the levels of other factors.

Figure 3.3 demonstrates the effect of three CPU levels on RUBiS' average response time. Here, the effect of changing CPU level from 20% to 60% is not the same as changing the CPU from 60% to 100%. If we consider only two CPU-levels, then we cannot capture the non-linear effect of CPU on the Web service performance. Hence, it may be important to consider more than two levels in the sample table for one or more factors to expose their non-linear effect.

There is an *interaction* between factors F_i and F_j when a change in F_i produces a different change in mean P at two different levels of F_j. Figure 3.2(e) visualizes such an interaction among factors F_1 and F_2 in the running example. The example shows the change in P as result of change in levels of F_1, for each level of F_2. Note the different shape of the two plots indicating interaction between the two factors. On the other hand, Figure 3.2(f) shows that there is almost no interaction between factors F_2 and F_3. Similarly, interactions can exist among three or more factors as well.

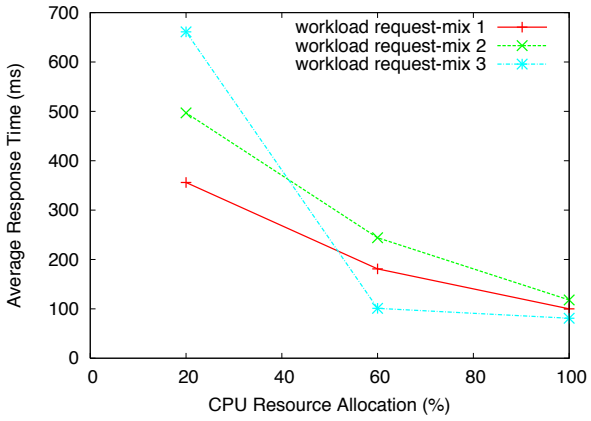

Figure 3.3: Non-linear effect of CPU on RUBiS's response time. More than two levels of factors are required to expose such non-linear effects.

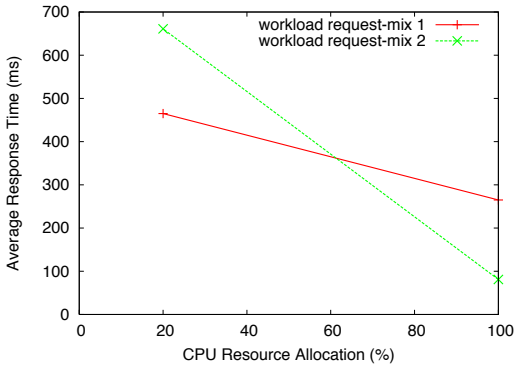

Figure 3.4: Interaction between the CPU and workload factors. The change in RUBiS' average response time for different CPU resource allocations is different at different settings of workload factors for the range of settings considered in this figure.

Figure 3.4 illustrates the effect of changing CPU-levels on RUBiS' average response time for two workloads. The effect of changing CPU-levels is different for the two workloads and depends on the type of workload. Hence the two lines intersect, and the Web service workload factors and the CPU factor interact with each other.

3.3.1 Computation of Impact

The running example illustrates the intuitive meaning of the effect of factors and interactions. This section presents a general method to measure the effect of factors and interactions from a given set of samples like those in Figure 3.1. We first describe

the method for two and three factors before generalizing it.

For the two-factor case, let P_{ij} denote the value of P in the sample where factor F_1 is at its ith level, and factor F_2 is at its jth level. Similarly, for the three-factor case, P_{ijk} denotes the value of P in the sample where factor F_1 is at its ith level, factor F_2 is at its jth level, and factor F_3 is at its kth level; and so on. For simplicity, we assume that there is only one sample per distinct combination of factor levels. The definition of effect is easy to extend to the case with multiple samples per distinct combination of factor levels [72].

We use a "star notation" to represent multiple samples, where a "$*$" instead of a level for a factor represents all the levels of that factor. For example, in the two-factor case, $\bar{P}_{i*}$ denotes the mean of P_{ij} across all samples where factor F_1 is at its ith level. Similarly, $\bar{P}_{**}$ represents the overall mean across all samples.

The general method to measure the effect of factors and interactions is based on *Factorial Analysis of Variance (Factorial ANOVA)* from Statistical Design of Experiments [53]. Here, the variation in performance that is present in the samples is partitioned into the variation that can be accounted for by selected factors and inter-factor interactions. The quantity used to measure variation is called *sum of squares (SS)*. The total variation in P is measured by the *total sum of squares* (SS_{tot}). For the two-factor case, SS_{tot} can be written as:

$$SS_{tot} = \sum_i \sum_j (P_{ij} - \bar{P}_{**})^2 \tag{3.1}$$

The $P_{ij} - \bar{P}_{**}$ term in Equation 3.1 can be further broken down as:

$$P_{ij} - \bar{P}_{**} = (\bar{P}_{i*} - \bar{P}_{**}) + (\bar{P}_{*j} - \bar{P}_{**}) + (P_{ij} - \bar{P}_{i*} - \bar{P}_{*j} + \bar{P}_{**}) \tag{3.2}$$

If the RHS in Equation 3.2 is squared and summed over i and j, then all the cross-product terms disappear, giving:

$$
\begin{aligned}
SS_{tot} &= \sum_i \sum_j (P_{ij} - \bar{P}_{**})^2 = \sum_i \sum_j (\bar{P}_{i*} - \bar{P}_{**})^2 + \\
&\sum_i \sum_j (\bar{P}_{*j} - \bar{P}_{**})^2 + \sum_i \sum_j (P_{ij} - \bar{P}_{i*} - \bar{P}_{*j} + \bar{P}_{**})^2 \\
&= SS_{F_1} + SS_{F_2} + SS_{F_1F_2} \qquad (3.3)
\end{aligned}
$$

$SS_{F_1} = \sum_i \sum_j (\bar{P}_{i*} - \bar{P}_{**})^2$ is the sum of squares of the deviation between the overall mean performance and the mean performance at each level of F_1. This quantity measures the total variation in P that can be accounted for by F_1 alone; so SS_{F_1} is an estimate of the effect of F_1 on P. Similarly, $SS_{F_2} = \sum_i \sum_j (\bar{P}_{*j} - \bar{P}_{**})^2$ is an estimate of the effect of F_2 on P.

$SS_{F_1F_2} = \sum_i \sum_j (P_{ij} - \bar{P}_{i*} - \bar{P}_{*j} + \bar{P}_{**})^2$ is an estimate of the effect of the interaction between F_1 and F_2 on P. Intuitively, the $(P_{ij} - \bar{P}_{i*} - \bar{P}_{*j} + \bar{P}_{**})$ term that calculates $SS_{F_1F_2}$ subtracts out the influence of F_1's level and F_2's level from each P_{ij}. Thus, $SS_{F_1F_2}$ captures the variation in P that cannot be accounted for by F_1 or F_2 individually.

In general, Factorial ANOVA partitions the total variation in P (SS_{tot}) into two components: (i) *explained* variation due to assignable causes such as certain factors and interactions, and (ii) *unexplained* variation that can arise due to measurement errors, or factors and interactions that are not considered. The difference between the total sum of squares SS_{tot} and the sum of all other SSs is represented by *error sum of squares* or SS_E. A small value of SS_E implies that most of the variability is explained by the factors and interactions considered in the analysis.

The process of decomposing the total variability in a data set, SS_{tot}, into different components is simply an arithmetic procedure. The only assumption is that the average or the mean system response across the levels of different factors is a

useful metric in calibrating the overall effect of a factor or an interaction. Further assumptions in ANOVA are needed only when the decomposed measures are used to construct statistical tests for *comparing the statistical significance* of a factor or an interaction. The decomposition of variability and the tests for statistical significance are often coupled tightly even though the tests are not really part of ANOVA [85].

Factorial ANOVA can be used to estimate the effect of any number of factors and multiway factor interactions. For example, for three factors F_1-F_3, the effect of each factor and interaction can be calculated as:

$$SS_{tot} = \sum_i \sum_j \sum_k (P_{ijk} - \bar{P}_{***})^2 = SS_{F_1} + SS_{F_2} + SS_{F_3} + SS_{F_1F_2} + SS_{F_1F_3} + SS_{F_2F_3} + SS_{F_1F_2F_3} \quad (3.4)$$

The above partitioning relies on the equality:

$$\begin{aligned} P_{ijk} - \bar{P}_{***} = {} & (\bar{P}_{i**} - \bar{P}_{***}) + (\bar{P}_{*j*} - \bar{P}_{***}) + \\ & (\bar{P}_{**k} - \bar{P}_{***}) + (\bar{P}_{ij*} - \bar{P}_{i**} - \bar{P}_{*j*} + \bar{P}_{***}) + \\ & (\bar{P}_{i*k} - \bar{P}_{i**} - \bar{P}_{**k} + \bar{P}_{***}) + (\bar{P}_{*jk} - \bar{P}_{*j*} - \bar{P}_{**k} + \bar{P}_{***}) + \\ & (P_{ijk} - \bar{P}_{ij*} - \bar{P}_{i*k} - \bar{P}_{*jk} + \bar{P}_{i**} + \bar{P}_{*j*} + \bar{P}_{**k} - \bar{P}_{***}) \end{aligned} \quad (3.5)$$

As in the two-factor case, when the RHS in Equation 3.5 is squared and summed over i, j, k, the cross-product terms disappear. For our running example, Figure 3.1 shows the SS using Factorial ANOVA for the three factors, three two-factor interactions, and single three-factor interaction. Note that the magnitude of these numbers is consistent with the intuitive definition of effect that Figure 3.2 illustrates.

The SS measure for each factor and interaction can be decomposed into linear,

quadratic, and other components. This decomposition is based on the method of *orthogonal contrast coefficients* [53] which we illustrate next for three factors, and then generalize. A column *vector of contrast coefficients* $\langle c_1, \ldots, c_m \rangle$ associates each sample in the table of m samples with an integer coefficient c_i. For example, the column for $F_{1,Lin}$ in Figure 3.1, shows one contrast coefficient vector $\langle c_1, \ldots, c_m \rangle$ for the linear effect of factor F_1 as follows:

$$c_i = \begin{cases} -1 & \text{if } F_1 = L_1 \ (F_1 \text{ is at level } L_1) \\ 1 & \text{if } F_1 = L_3 \ (F_1 \text{ is at level } L_3) \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

Consider a set of contrast coefficient vectors with the following properties:

1. The sum of all the contrast coefficients in the contrast coefficient vector is 0, i.e., for each $\langle c_1, \ldots, c_m \rangle$, $\sum_{i=1}^{i=m} c_i = 0$.

2. The sum of coefficients that result from the product of any two contrast coefficient vectors, $\langle c_1, \ldots, c_m \rangle$ and $\langle c'_1, \ldots, c'_m \rangle$, is 0, i.e., for each $\langle c_1, \ldots, c_m \rangle \neq \langle c'_1, \ldots, c'_m \rangle$, $\sum_{i=1}^{i=m} c_i c'_i = 0$.

The above properties makes the contrast coefficient vectors orthogonal. Each such vector $\langle c_1, \ldots, c_m \rangle$ enables us to compute one effect and one SS measure for a factor or interaction through a *contrast* defined as $C = \sum_i c_i P_i$. An effect can be calculated from a contrast C with suitable averaging, while an SS is calculated as $SS = \frac{C^2}{\sum_i \sum_j \sum_k c_i^2}$. We illustrate these properties using the running example.

Columns $F_{1,Lin}$, $F_{1,Quad}$, $F_{2,Lin}$, and $F_{3,Lin}$ in Figure 3.1 show four orthogonal contrast coefficient vectors. Their respective contrasts are also shown. The contrast from the coefficient vector in column $F_{1,Lin}$ can be used to compute the linear effect and linear sum of squares ($SS_{F_1,Lin}$) of factor F_1. Similarly, the contrast in column

$F_{1,Quad}$ can compute the quadratic effect and quadratic sum of squares ($SS_{F_1,Quad}$) of factor F_1. The contrasts in columns $F_{2,Lin}$ and $F_{3,Lin}$ can compute the linear effect and linear SS for factors F_2 and F_3 respectively.

For a factor with l levels, effects or SS of up to order $l-1$ can be calculated with appropriate orthogonal contrast coefficient vectors [53]. The SS for this factor that is calculated using Factorial ANOVA is the sum of the SS up to order $l-1$ that are calculated from the coefficient vectors. This property relates Factorial ANOVA to the method of orthogonal contrast coefficients. Factors F_2 and F_3 in our running example have two levels each, so we cannot compute their quadratic or higher order effect; thus $SS_{F_2} = SS_{F_2,Lin}$ and $SS_{F_3} = SS_{F_3,Lin}$. Similarly, since F_1 has three levels, we cannot compute its cubic or higher order effect; thus $SS_{F_1} = SS_{F_1,Lin} + SS_{F_1,Quad}$.

The interesting property of orthogonal contrast coefficient vectors is that the coefficient vector for any binary or multiway interaction among factors is easily derived by multiplying the individual coefficient vectors on an index-by-index basis. For example, the column $F_{2,Lin}F_{3,Lin}$ in Figure 3.1 shows the coefficient vector for the F_2F_3 interaction that is derived by multiplying the vectors in columns $F_{2,Lin}$ and $F_{3,Lin}$. The SS measure from this vector is equal to the SS derived for the F_2F_3 term using Factorial ANOVA. More specifically, since F_2 and F_3 have two levels each, we can write $SS_{F_2F_3} = SS_{(F_2,Lin)(F_3,Lin)}$.

Since F_1 has three levels, for the interaction between factors F_1 and F_2, we have $SS_{(F_1,Lin)(F_2,Lin)} + SS_{(F_1,Quad)(F_2,Lin)} = SS_{F_1F_2}$. The contrast coefficient vector for $SS_{(F_1,Lin)(F_2,Lin)}$ is the product of the coefficient vectors in columns $F_{1,Lin}$ and $F_{2,Lin}$, and that for $SS_{(F_1,Quad)(F_2,Lin)}$ is the product of the coefficient vectors in columns $F_{1,Quad}$ and $F_{2,Lin}$. Finally, we have $SS_{F_1F_2F_3} = SS_{(F_1,Lin)(F_2,Lin)(F_3,Lin)} + SS_{(F_1,Quad)(F_2,Lin)(F_3,Lin)}$.

3.3.2 Summary

In this section we use two methods—Factorial ANOVA and orthogonal contrast coefficients—to estimate the effect of each factor and inter-factor interaction on system performance P. The description revolves around two- and three-factor scenarios in our running example, but it is easy to generalize across any number of factors [53, 72]. For most of this chapter, we consider SS from Factorial ANOVA as the default measure of effect.

To estimate the effects, we need samples of system performance similar to samples shown in Figure 3.1. Section 3.4 presents the algorithm that collects the samples required to estimate the effects automatically and quickly.

3.4 Experiment-Driven Collection of Samples

Section 3.2 presents three example management queries on system behavior. A key challenge is to identify and collect samples that can be used to compute an accurate query result quickly. Algorithm 2 presents the experiment-planning algorithm for data collection and analysis to process a given management query. The steps are instances of the operations in the experiment-driven framework from Chapter 2; recall Figure 2.2. The algorithm leverages the experiment designs from Section 2.2 and techniques from Section 3.3 in a unified manner to generate the samples for computing the query result.

For the ranking-queries Q_1 and Q_2, the typical approach consists of conducting experiments from one of the screening designs alone, e.g., [120]. Algorithm 2 combines the screening designs with model-learning designs in a sequential manner such that the controller can conduct more experiments to improve the accuracy of the ranking queries even after exhausting the experiments from the screening designs. Moreover, the algorithm also aligns the choice of the screening design with the overall goals of

Algorithm 2: Experiment-Driven Sample Collection

Input: (i) Complete set of factors $\vec{F}$; (ii) Web-service management query, Q_i; (iii) Maximum allowed number of experimental runs, *MAX_RUNS*.

Output: A table of samples which is analyzed to compute the query result for query Q_i.

1) Pick a **screening experiment design** for bootstrapping (Section 3.5);

2) Conduct experiments according to the experiment design (Section 3.6);

3) Build an initial table of samples from the experiments in Step 2;

4) **while** ($num\_exps < MAX\_RUNS$)

a. Design and conduct the next experiment(s) using a **model-learning** design (Section 3.7);

b. Add the sample(s) from Step **a** to the table of samples;

c. num_exps++;

end

the ranking queries. As Section 3.5 explains, ranking the factors and interactions based on their linear effect requires less experiments than ranking based on their non-linear effect.

Similarly, for the model-learning query Q_3, the algorithm uses the screening designs to generate a set of initial samples that can determine the important factors and interactions that the model must consider. Hence, the screening step can ensure that the model not only includes all the important factors and interactions, but also eliminates unimportant factors and interactions to reduce the complexity of the model [59]. Sections 3.5-3.8 present the algorithm in detail. Section 3.9 evaluates the algorithm by evaluating the accuracy of the query results that are attained from the samples output by the algorithm.

3.5 Bootstrapping Using Screening Designs

Step 1 in Algorithm 2 consists of choosing an experiment design (Section 2.2) for bootstrapping the query processing. The bootstrapping involves choosing an initial set of experiments to generate samples which can be analyzed to guide the choice of future experiments (Section 2.5). The algorithm uses *screening* designs for this purpose (Section 2.2). In this work, we investigate the use of the following screening designs since these are some of the most popular screening designs used for product design, process control, and process trouble shooting in engineering disciplines [83].

3.5.1 Screening Designs for Linear Impact

Section 3.3.1 shows how to quantify the effect of factors and interactions based on their linear impact using two levels per factors. While the two levels can be any pair from the different levels per factor, they usually are the lowest and highest levels that are assigned contrast coefficients -1 and 1 respectively.

2^n Full Factorial Designs: The straightforward scheme is to conduct all 2^n possible experiments given two levels each for the n factors $F_1, \ldots, F_n$; called the 2^n Full Factorial Design [53]. Figure 3.5 shows such a design for three factors F_1-F_3. This table is the subset of Figure 3.1 from which the four samples where factor F_1 is at level L_3 are dropped; and the L_1 and L_2 levels for each factor are mapped to the contrast coefficients -1 and 1 respectively.

Section 3.3 shows how to compute the SS measures for all factors and interactions—which includes three single factor SS, three two-way interaction SS, and one three-way interaction SS—for 2^3 samples in Figure 3.5.

2^{n-p} Fractional Factorial Designs: The experimental effort in a 2^n design quickly

becomes exorbitant as n grows. For example, if $n = 10$ then 1024 experiments are required, which at 10 minutes per experiment, requires around seven days to complete. Note that 2^n design captures the effects of all the factors and interactions. However, the behavior of most systems is primarily a function of the effects of single factors and effects of some low-order (e.g., pairwise) interactions [78]. A design that captures the effects of single factors and low-order interactions requires a small fraction of the 2^k factorial design. Such designs are called 2^{k-p} *Fractional Factorial Designs*.

For $p \geq 1$, such designs can estimate important factor and interaction SS measures at only a $\frac{1}{2^p}$ fraction of the 2^n full factorial design [53]. However, this reduction in the number of experiments exposes a tradeoff: a 2^{n-p} design cannot isolate the SS measure for every factor and multi-factor interaction.

Consider our running example. Suppose we use a 2^{3-1} design where we collect only the samples in rows 2, 3, 5, and 8 in Figure 3.5, i.e., we conduct the experiments $\langle -1, -1, 1\rangle$, $\langle -1, 1, -1\rangle$, $\langle 1, -1, -1\rangle$, $\langle 1, 1, 1\rangle$ to collect the respective samples. The other four experiments are not conducted.

The row labeled *Aliased SS* in Figure 3.5 gives the factor and interaction SS for the four samples. Notice that $SS_{F_1} = SS_{F_2F_3}$, $SS_{F_2} = SS_{F_1F_3}$, and $SS_{F_3} = SS_{F_1F_2}$. The reason for this pattern is clear if we examine the contrast coefficient vectors restricted to the four samples. The contrast coefficient vector for F_1 is the same as the coefficient vector for F_2F_3, namely, $\langle -1, -1, 1, 1\rangle$. Identical coefficient vectors will give rise to identical SS measures (Section 3.3), so we cannot separate out the individual effects of the respective factors and interactions. This phenomenon is called *aliasing*. In the 2^{3-1} design discussed above, the effect of the factors F_1, F_2, and F_3 are aliased respectively with that of the interactions F_2F_3, F_1F_3, and F_1F_2.

	Levels				Contrast Coefficient Vectors						
	F_1	F_2	F_3	P	F_1	F_2	F_3	F_1F_2	F_1F_3	F_2F_3	$F_1F_2F_3$
					$F_{1,Lin}$	$F_{2,Lin}$	$F_{3,Lin}$	$F_{1,Lin}$ $F_{2,Lin}$	$F_{1,Lin}$ $F_{3,Lin}$	$F_{2,Lin}$ $F_{3,Lin}$	$F_{1,Lin}$ $F_{2,Lin}$ $F_{3,Lin}$
1	L1	L1	L1	1	-1	-1	-1	+1	+1	+1	-1
2	L1	L1	L2	3	-1	-1	+1	+1	-1	-1	+1
3	L1	L2	L1	2	-1	+1	-1	-1	+1	-1	+1
4	L1	L2	L2	3	-1	+1	+1	-1	-1	+1	-1
5	L2	L1	L1	10	+1	-1	-1	-1	-1	+1	+1
6	L2	L1	L2	8	+1	-1	+1	-1	+1	-1	-1
7	L2	L2	L1	1	+1	+1	-1	+1	-1	-1	-1
8	L2	L2	L2	1	+1	+1	+1	+1	+1	+1	+1
Contrasts					11	-15	1	-17	-5	1	3
Effects					2.75	-3.75	0.25	-4.25	-1.25	0.25	0.75
Factorial Sum of Squares (SS)					15.125	28.125	0.125	36.125	3.125	0.125	1.125
Aliased SS					9	25	16	16	25	9	

Figure 3.5: Running example with two levels per factor.

3.5.2 Controlling Aliasing Through Resolutions

We now present the concept of *design resolution* [78] which is a way to characterize 2^{n-p} designs such that the aliases in the design are known a priori. Resolutions are indicated using roman numerals. Commonly used resolutions are III, IV, and V.

Resolution III A 2^{n-p} design is of resolution III when no single-factor effect is aliased with another single-factor effect, but each single-factor effect is aliased with one or more two-factor interactions. Note that the example 2^{3-1} design has resolution III. For n factors, we can generate resolution III designs that have $O(n)$ experiments only.

PBDF (Plackett-Burman Design with Foldover) is an example of a popular resolution III screening design [59, 120] that ranks N independent factors $X_1, \ldots, X_N$ based on their effect on a dependent parameter Y. PBDF requires that $N + 1$ be a multiple of 4, with dummy parameters added as needed. PBDF identifies a set of

Table 3.2: Experiment design for PBDF

X_1	X_2	X_3	X_4	X_5	X_6	X_7	Y
+1	+1	+1	-1	+1	-1	-1	y_1
-1	+1	+1	+1	-1	+1	-1	y_2
-1	-1	+1	+1	+1	-1	+1	y_3
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	
-1	-1	+1	-1	+1	+1	-1	y_{15}
+1	+1	+1	+1	+1	+1	+1	y_{16}

experiments where the configuration of $X_1, \ldots, X_N$ for each experiment is given by a specific experiment design. The general design is described in [120]. This chapter presents an example.

Table 3.2 shows a part of the design when $N = 7$. This design has $2 * 8 = 16$ rows because 8 is the nearest higher multiple of 4 for $N = 7$, and PBDF requires twice that many rows. Each row specifies the configuration of $X_1, \ldots, X_N$ for a run. The value of Y given by the run is also shown. A "+" ("−") value for a parameter represents a value that is higher (lower) that the normal range of values for that parameter. PB designs can have a complicated aliasing structure and hence must be used carefully [78].

After performing all 16 runs, the *effect* of each parameter on Y is computed by multiplying the parameter's value in each row with the value of Y for the row, taking the sum across all rows, and taking the absolute value of the sum. For example, the effect of parameter X_1 is $|y_1 - y_2 - y_3 \cdots - y_{15} + y_{16}|$. Finally, the parameters are ranked in decreasing order of their effect on Y.

Resolution IV A 2^{n-p} design is of resolution IV when no single-factor effect is aliased with that of another single-factor or two-factor interaction, but some two-factor interactions are aliased with one or more two-factor interactions. Single-factor effects may be aliased with three-factor or higher-order interactions. While resolution

IV designs need more experiments than resolution III designs for the same number of factors, they still have $O(n)$ experiments.

Resolution V A 2^{n-p} design is of resolution V when no single-factor effect is aliased with that of another single-factor or two-factor interaction, and no two-factor interaction is aliased with another two-factor interaction. Two-factor interactions may be aliased with three-factor or higher-order interactions. Resolution V designs are good for studying all the single-factor and two-factor interaction effects independent of each other. If all interactions of three factors and higher are insignificant compared to single-factor and two-factor effects—which is often true in practice—these designs give accurate estimates of all single-factor and two-factor effects. Resolution V designs require $O(n^2)$ experiments.

In practice, we observe that the effects of higher-order interactions are usually much less than that of single-factor and two-factor effects. Figure 3.5 illustrates this phenomenon. Note that $\frac{SS_{F_1F_2F_3}}{SS_{tot}} = \frac{1.125}{83.875}$ is 1.3%. This phenomenon motivates an iterative technique to process queries Q_1 and Q_2 (Section 3.2) for linear effect measures: start with a low resolution 2^{n-p} design (say, resolution V) and keep increasing the resolution until we capture all dominant effects. The challenge is to determine if we have missed any dominant effects.

Recall the method of Factorial ANOVA from Section 3.3. Factorial ANOVA estimates the component SS_{expl} of the total variation SS_{tot} that is explained by a selected set of factors and interactions. The magnitude of the unexplained variation $SS_{tot} - SS_{expl}$ is a conservative estimate of the total effect of factors and interactions that are ignored. If the fraction of unexplained variation $\frac{SS_{tot}-SS_{expl}}{SS_{tot}}$ is small, then we can stop; otherwise we can increase the resolution and do experiments to collect more samples.

Despite their limitations in estimating the effect of all factors and interactions accurately, the two-level fractional designs are most widely used in engineering [83]. This is because in most engineering disciplines the knowledge gleaned from two-level fractional designs is sufficient for most management tasks. The effect of high-order interactions is often negligible.

3.5.3 Screening Designs for Linear and Quadratic Impact

A factor or interaction could have a non-linear effect on performance, like the overall effect of factor F_1 that the running example illustrates in Figure 3.1 and Figure 3.2(d). We cannot estimate such an effect without conducting experiments with at least three levels of the factors. In this section, we describe experiment designs to estimate quadratic effect in addition to linear effect; and in Section 3.7, we present model-learning designs that can capture effect arising from the full operating range of input factors.

3^n **Full Factorial Designs:** A straightforward design to capture quadratic effect is to conduct the 3^n experiments with each of the n factors set to three levels each: low, high, and a level in between. Because of its exponential nature, this design becomes impractical much faster than 2^n designs.

Central Composite Designs (CCD): A CCD is a design that augments a 2^{n-p} design with a selected set of new experiments that capture quadratic effect in addition to linear effect. A CCD for factors $F_1, \ldots, F_n$ consists of the three distinct sets of experimental runs. We use the contrast coefficients $-1, 0, 1$ for the low, median, and high levels respectively for each factor.

- A set of *corner runs* from a 2^{n-p} fractional factorial design, typically of resolu-

tion IV or V. Recall from Section 3.5.1 that these experiments have the form $\langle c_1, c_2, \ldots \rangle$ where $c_i \in \{-1, 1\}$.

- A *center run*, which is an experiment with each factor F_i set to its median level. (This experiment may be repeated a few times for precision.) This experiment has the form $\langle 0, 0, \ldots, 0 \rangle$.

- A set of $2n$ *axial runs*, which are experiments identical to the center point except for one factor, which will take on the low and high levels. All factors are set in this way to give the $2n$ experiments of the form: $\langle -1, 0, \ldots, 0 \rangle$, $\langle 1, 0, \ldots, 0 \rangle$, $\langle 0, -1, \ldots, 0 \rangle$, $\langle 0, 1, \ldots, 0 \rangle$, $\cdots$, $\langle 0, 0, \ldots, -1 \rangle$, $\langle 0, 0, \ldots, 1 \rangle$.

These three sets of runs play important and slightly different roles in a CCD. The corner runs help estimate factors and interactions with high linear effect. The center and corner runs together help test whether there is significant non-linear behavior; and if such behavior exists, the center and axial runs help estimate factors and interactions with high quadratic effect.

3.6 Conducting an Experiment

Algorithm 3 outlines the steps involved in conducting a single experiment. Broadly, it consists of obtaining resources for the experiment, instantiating the Web service and the clients, configuring the service and the client(s) according to the settings of each factor in $\vec{F}$, and running the workload for a predetermined period. The instrumentation on the client and server consists of measures that are useful for computing the response time of each Web transaction. Section 2.3 describes how the workbench controller configures resources and configuration. To configure the workload, we use synthetic workload generators that can generate a wide variety of Web service workloads (Section 3.9.1).

Algorithm 3: Conduct a Web service experiment

Input: A specified level l_i for each factor in $\vec{F}$, where $\vec{F}$ is a subset of factors in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$; performance metric of interest P. **Output:** Measure P for $\langle F_1 = l_1, \ldots, F_n = l_n \rangle$ and return a complete sample.

1) Obtain CPU, and Memory resources for the Web service and workload client(s);
2) Create a resource configuration for the Web service such that the CPU, memory, network, and disk levels are set as per $\langle F_1 = l_1, \ldots, F_n = l_n \rangle$;
3) Instantiate the Web service on the resources;
4) Configure the Web service client(s) according to the levels in $\langle F_1 = l_1, \ldots, F_n = l_n \rangle$;
5) Coordinate between the Web service and its clients such that the clients are launched only after the service is ready to receive the requests;
6) Start instrumentation on the client and server;
7) Run the client workload for a predetermined period;
8) Stop instrumentation;
9) Collect instrumentation data during the experiment which is processed to yield the $\langle F_1 = l_1, \ldots, F_n = l_n, P \rangle$ sample which goes into a table of samples;

3.7 Model-Learning Designs

Section 3.5 presents screening experiment designs that are useful for bootstrapping the experiment-driven collection of samples (Step 1 in Algorithm 2). The screening experiment designs, which consider only 2 or 3 levels per factor, can generate samples for computing the linear and quadratic effect of each factor. The linear and quadratic effect measures often suffice to give a good understanding of how various factors and interactions affect system behavior. Hence, we can obtain the query result for the ranking-queries Q_1 and Q_2 (Section 3.2) by using the screening designs alone.

However, system administrators and self-managing system components may also

be interested in ranking factors and interactions based on their *full effect* which accounts for all input factor levels. In this section, we propose model-learning experiment designs that generate samples which can be analyzed to compute the result for such queries. Section 2.2 introduces the model-learning experiment designs. The same designs can also generate samples to compute the results for query Q_3 which asks to approximate a function that determines Web service performance for any given combination of input factor levels.

To use such designs, we first choose the structure of the model that we want to learn, e.g., a linear regression model. Then we learn the model using a set of bootstrap or initial samples. The set of initial samples are generated from Steps $1-3$ of Algorithm 2. The next step is to use the learned model to output the next set of experiment(s) to conduct. The next set of experiments are chosen from the pool of total possible experiments T. For n factors, each with l levels, the total pool of possible experiments consists of n^l, excluding the experiments done so far. Since the next experiment is chosen from all the possible samples, model-learning designs consider all levels per factor for each factor automatically. Section 3.7.1 presents the algorithm for choosing the next experiment.

In this work we consider first-order, first-order with interaction, and second-order regression models as described below. We consider these models for generating the samples in Step 4 of Algorithm 2 to compute the result of the ranking-queries Q_1 and Q_2 as well as query Q_3 where the goal is to learn a model with a given structure. However, the model-learning designs are agnostic of the model structure, and can support any structure such as the more sophisticated regression trees [71].

Regression models capture the dependence of P on the input factors $F_1, \ldots, F_n$ and inter-factor interactions as follows:

- *First-order model*: $P = \beta_0 + \sum_i \beta_i F_i + \epsilon$

Algorithm 4: Active Machine Learning Algorithm.

Inputs: Current sample table, Model M, pool of total possible experiments, T. **Output:** An experiment.

1) Use 10-fold cross-validation to learn 10 models $\langle M_1, \ldots, M_{10} \rangle$ on the current set of data samples that were obtained in the previous steps of Algorithm 2;

2) Use $\langle M_1, \ldots, M_{10} \rangle$ to obtain 10 predictions $\langle p_1, \ldots, p_{10} \rangle$ for each experiment i in T;

3) Compute the variance in the predictions, v_i for each experiment i in T to obtain $v_1, \ldots, v_n$ for n experiments in T.

4) Choose the experiment i, such that $v_i = max(v_1, \ldots, v_n)$.

- *First-order model with interaction terms*: $P = \beta_0 + \sum_i \beta_i F_i + \sum_{i \neq j} \beta_{ij} F_i F_j + \epsilon$
- *Second-order model*: $P = \beta_0 + \sum_i \beta_i F_i + \sum_{i \neq j} \beta_{ij} F_i F_j + \sum_i \beta_{ii} F_i^2 + \epsilon$

Here, ϵ is an error term that is assumed to be an IID (independent and identically distributed) Gaussian random variable [59]. The β parameters are regression coefficients that Algorithm 2 estimates during *model-fitting* on samples collected through experiments. The method of *least squares* is a common technique for model-fitting [53]. Let p_i, $1 \leq i \leq m$, denote the observed value of P in each of m samples collected so far. Also, let q_i denote the model-predicted value of P in each case. Least squares chooses the values of β parameters that minimizes the model's squared error: $\sum_{i=1}^{m} \epsilon^2 = \sum_{i=1}^{m} (p_i - q_i)^2$.

3.7.1 Active Machine Learning

Active machine learning algorithms, e.g., [96], iteratively identify the next experiment from the total space of experiments T, that will maximize the accuracy of the model being learned. These algorithms assume that the structure of the model being learned

is known a priori. We use a *committee-based* approach [30, 57] for selecting the next experiment from T. Algorithm 4 presents the overall approach. The idea behind the *committee-based* approach is to select the experiment that causes maximum disagreement among a *committee* of models that are learned from the current sample table. The premise is that adding that point to the sample table will result in capturing a previously unseen behavior of the Web service, and hence enable learning a model that is accurate across a wide range of Web service behavior. The selected experiment is conducted, its results added to the sample table, and the process repeats.

For learning models that predict Web service performance, we find the committee-based approach to be sufficient for learning an accurate model quickly. However, there are several alternatives to the committee-based approach that are available in the literature for active machine learning, e.g., [87] as well as design of experiments, e.g., [9]. An exhaustive comparison of different approaches remains an interesting area for future work.

3.8 Query Processing

At any point during the execution of Algorithm 2, we can compute the query result for Q_1, Q_2, or Q_3 based on the current set of samples from the experiments thus far. For example, for the ranking queries Q_1 and Q_2, we can compute the SS effect measure to quantify the impact of factors and interactions; and for the modeling query Q_3, we can fit a model with a given structure. First-order model with interaction terms is our default model for all the queries.

The algorithm can run as long as there are resources to conduct more experiments. Algorithm 2 represents the experimental budget as an upper bound MAX_RUNS on the number of runs that can be conducted. Other conditions may include the desired accuracy of the query results. Section 4.6.6 presents techniques to evaluate the current

accuracy of a model that is being learned using the experiment-driven framework.

3.9 Evaluation

3.9.1 Experimental Setup

We use two multitier Web services, TPC-W and RUBiS, to evaluate the experiment-driven approach (Algorithm 2). RUBiS [22] is an open source multitier Web service that implements the core functionality of an auction site like eBay: selling, browsing, bidding. We identify three types of clients in RUBiS: browse-only client, bid-only client, and sell-only client. We add an additional type of client, called AboutMe in RUBiS, because users of auctions sites tend to spend a lot of time on their personalized home pages on the site (e.g., My eBay). These personalized home pages tend to have many customization options, and can generate significant load on the service.

TPC-W is a transactional Web ecommerce benchmark [109]. It specifies an ecommerce workload that simulates the activities of a retail website, placing heavy load on the backend database. It implements the core functionality of an online bookstore such as Amazon. Clients in TPC-W show two distinct types of behavior, browsing and ordering. Table 3.3 summarizes the $\vec{P}, \vec{W}, \vec{R}, \vec{C}, \vec{F}$ vectors for the Web services.

We consider 4 workload and 2 resource factors for RUBiS, and 2 workload, 2 resource, and 3 database configuration factors for TPC-W. We use three performance metrics: the average Web service response time, Web service throughput, and the *number of errors* seen by the clients. A number of client requests can receive an invalid response from the server. The invalid response can occur for several reasons, e.g., if the service is overloaded and cannot process the client request. We count the number of such responses during each experiment and represent it as the number-of-errors performance metric.

We use Automat [122] as the controller workbench to conduct the experiments

$\vec{P}_{RUBiS,TPC-W}$	Average service response time, service throughput, number of service errors
$\vec{W}_{RUBiS}$	Number of Sell [4], Browse [4], Bid [4], AboutMe [4] clients
$\vec{W}_{TPC-W}$	Number of Browse [3], Order [3] clients
$\vec{R}$	CPU [5], Memory [4]
$\vec{C}_{TPC-W}$	`mysqld_key_buffer_size` [3], `mysqld_table_cache` [3], `mysqld_sort_buffer_size` [3]
$\vec{F}_{RUBiS}$	$\langle \vec{W}_{RUBiS}, \vec{R} \rangle$, 6 factors
$\vec{F}_{TPC-W}$	$\langle \vec{W}_{TPC-W}, \vec{R}, \vec{C}_{TPC-W} \rangle$, 7 factors

Table 3.3: Factors, levels, and performance metrics for RUBiS and TPC-W. The number of levels for the factors [l] is given.

(Section 2.3, Figure 2.1). Section 5.4 presents techniques to identify an experiment's runlength and number of times to repeat the experiment automatically to ensure statistically accurate experiment result. For this study we ran each experiment once and for a fixed duration of 5 minutes.

3.9.2 Validation Methodology

For validating the query results computed by our approach, we compare the query result that we obtain by analyzing the samples from using Algorithm 2 to the query result that we obtain from the analysis of the complete dataset that contains all the sample from the full space of experiments. The result from the analysis of all the samples is the *correct* query result. For example, RUBiS has $\vec{W} = \langle bid, buy, sell, AboutMe \rangle$ where each factor has 4 levels in our experimental setup; and $\vec{R} = \langle CPU, Memory \rangle$ where CPU and *Memory* have 5 and 4 levels respectively. Thus, the total number of samples is: $4 \times 4 \times 4 \times 4 \times 5 \times 4 = 5120$. Similarly, for TPC-W the total space of samples is: $3 \times 3 \times 3 \times 3 \times 3 \times 4 \times 5 = 4860$.

The graphs that we report have x and y axes similar to Figure 1.4. The x-axis shows the number of samples generated from the experiments so far by Algorithm 2. The y-axis shows the accuracy of the query result computed from these samples by comparing this result to the correct result computed from all samples. We use

three accuracy metrics for the ranking-queries Q_1 and Q_2, and one metric for the modeling-query Q_3 as follows:

- **Effect Difference (ED).** Suppose we are ranking m factors or interactions. Let $est_1, \ldots, est_m$ be the respective effect measures estimated from the samples collected so far, and let $cor_1, \ldots, cor_m$ be corresponding correct measures computed from all samples. We normalize each set of measures by dividing by the maximum value, denoted est_{max} and cor_{max} respectively. We define the (normalized) *Effect Difference* as $ED = \frac{1}{m}\sum_{i=1}^{m} |\frac{est_i}{est_{max}} - \frac{cor_i}{cor_{max}}|$. $0 \leq ED \leq 1$, with lower values indicating more accurate ranking.

  Top-k Ranking Distance (RD(k)). Suppose we are ranking m factors (or interactions). Let r_i represent the rank estimated based on the current samples for the factor (interaction) that is at rank i in the correct ranking based on all samples. (Ranks start at 1 and go up to m.) The (normalized) top-k ranking distance *RD(k)* is defined as $\frac{\sum_{i=1}^{k} |r_i - i|}{\sum_{i=1}^{k} m - i}$. $0 \leq RD(k) \leq 1$, with lower values indicating more accurate ranking of the k most important factors.

  Order Preserving Degree (OPD) [123]. A ranking result *preserves* the relative order of factors or interactions i, j iff $r_{est_i}(<, =, >)r_{est_j}$ holds when the corresponding $r_{cor_i}(<, =, >)r_{cor_j}$ holds, where r_{est} and r_{cor} respectively represent ranks obtained using the current samples and all samples. Given m factors or interactions, $OPD = \frac{|OPP|}{m^2}$ where OPP is the set of order-preserving pairs, i.e., i, j that have the same relative order in both rankings. $0 \leq OPD \leq 1$, with higher values indicating more accurate ranking.

- **Mean Absolute Percentage Error (MAPE).** To compute the accuracy of a model generated for query Q_3, we choose $T = 1000$ test samples at random from the full space of samples, and compute the model-predicted performance

p_{est} for each test sample. MAPE is defined as $\frac{1}{T}\sum_{i=1}^{T}\frac{|p_{est}-p_{obs}|}{p_{obs}}$, where p_{obs} is the actual performance observed for the sample.

3.9.3 Computing and Validating Query Results

We use Algorithm 2 to generate the samples for computing the results for the three types of Web service management queries (Section 3.2). This section evaluates the algorithm to demonstrate that it can generate the samples for computing accurate query results by conducting less than 1-2% of the total possible space of experiments. We compute all the accuracy metrics after obtaining each new sample that an experiment generates.

Ranking Single Factors (Q_1). Figure 3.6 shows the accuracy metrics ED, $RD(k)$, and OPD for rankings factors in order of their effect on Web service throughput for both RUBiS and TPC-W. In Step 1 of Algorithm 2, the bootstrap experiment design is a resolution V 2^{n-p} design, which is our default for the all queries. For the accuracy metric $RD(k)$, $k = 3$ for RUBiS, and 4 for TPC-W, i.e., we investigate the accuracy of ranking top 3 out of 6 factors and top 4 out of 7 factors for RUBiS and TPC-W respectively.

The figure shows that the accuracy of rankings approaches that of rankings with the complete dataset with less than 1% of the total possible experiments. Data analysis techniques like Factorial ANOVA (Section 3.3) and model-fitting (Section 3.7) require a minimum number of samples to start generating results, causing the initial abrupt jumps in accuracy. Note that for RUBiS, $RD(k)$ and OPD metric show a slight deterioration towards the end. This happens because the effects of some of the factors are quite similar; for such factors, even a minor deviation from the complete dataset-based effect can result in a ranking that is different from the complete dataset-based ranking.

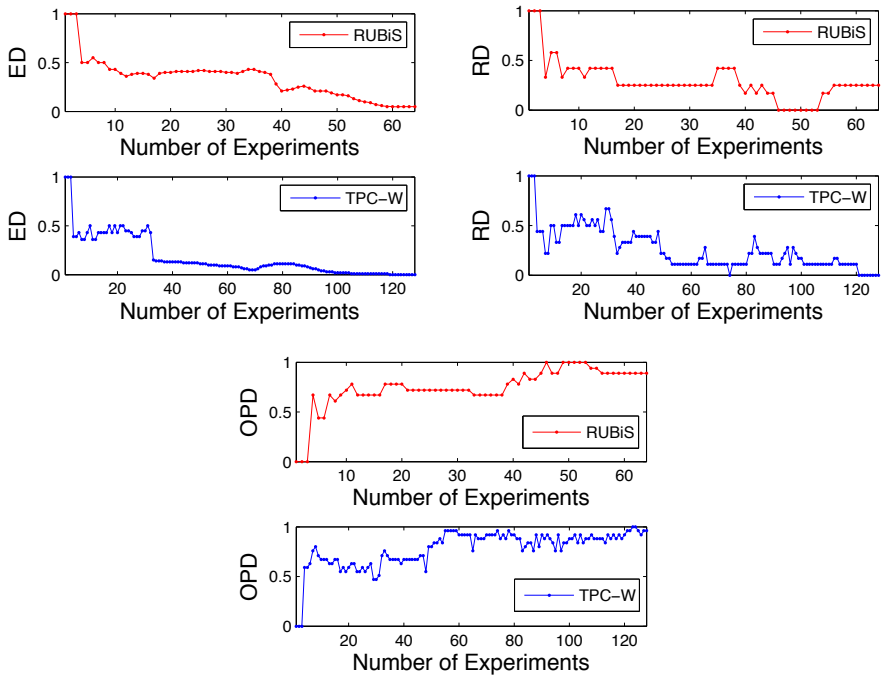

Figure 3.6: Quick convergence of the rankings of factors to their best possible value for the throughput performance metric.

The results for other performance metrics are also similar. Figures 3.7 shows the accuracy of rankings for the average response time metric, and Figure 3.8 shows the accuracy for the number-of-errors metric. In both cases, Algorithm 2 uses less than 1% of the total possible experiments to generate samples that result in a ranking accuracy that approaches the accuracy with the complete dataset.

Ranking Factor Interactions (Q_2). Figure 3.9 shows that the ranking accuracy metrics of two-factor interactions approaches that of rankings with the complete dataset with less than 1% of the experiments for both RUBiS and TPC-W for the throughput performance metric. RUBiS has a total of 15 pairwise interactions with 6 factors. TPC-W has a total of 21 pairwise interactions with 7 factors. For both Web services we compute $RD(k)$ metric with $k = 10$.

Note that for TPC-W even though ED converges quickly an accurate ranking,

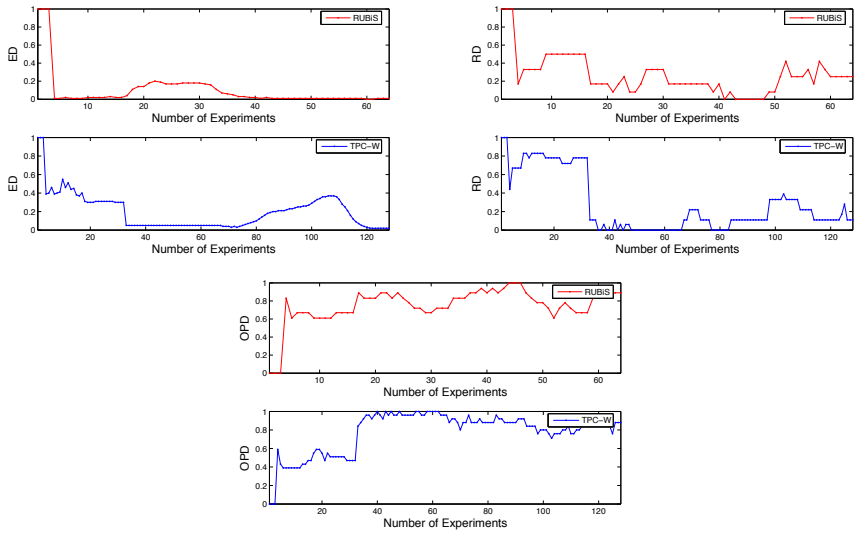

Figure 3.7: Quick convergence of the rankings of factors to their best possible value for the average response time performance metric.

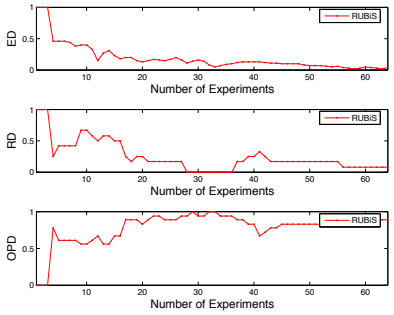

Figure 3.8: Quick convergence of the rankings of factors to their best possible value for the number-of-errors performance metric for RUBiS. The number of errors for TPC-W was always 0 for all the experiments with TPC-W.

the $RD(k)$ and OPD metrics converge slowly. The reason is that many of the lower-effect interactions for TPC-W have very similar values of effect, so even minor noise in their estimates can alter the ranking considerably. The results for other performance metrics are shown in Figures 3.10 and 3.11; with less than 1% of the total possible experiments, the ranking of interactions converges to the ranking with complete dataset.

Learning a Predictive Model (Q_3.) While validating query Q_3 results for Web

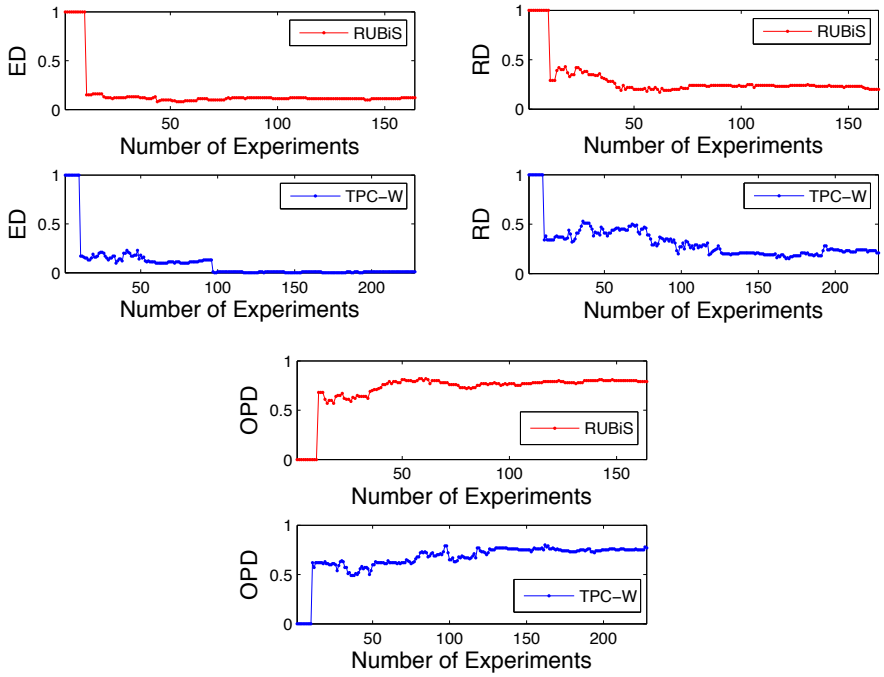

Figure 3.9: Quick convergence of the ranking of interactions to their best possible value for the throughput performance metric.

services, the goal is to demonstrate that Algorithm 2 can generate samples to quickly converge to an accurate model across models with different structures; the choice of model structure is not the focus here. Chapter 4 focuses on the appropriate model structure for predicting the execution time of batch applications. In chapter 4 we also investigate model accuracy by examining its mean absolute error, worst case error, and 80 and 90 percentile error in addition to MAPE (Section 4.8.1).

Figures 3.12 (a) and 3.12 (b) show that the accuracy of the model learned using samples generated by Algorithm 2 approaches that of the model learned with complete dataset with less than 1% of the experiments for TPC-W and RUBiS. Here, we use the first-order model with interaction terms (Section 3.7), and the model predicts the throughput metric. Figures 3.13 (a) and 3.13 (b) show the similar result for the average response time metric using the first-order model with interaction terms for RUBiS and TPC-W. Here, we observe that:

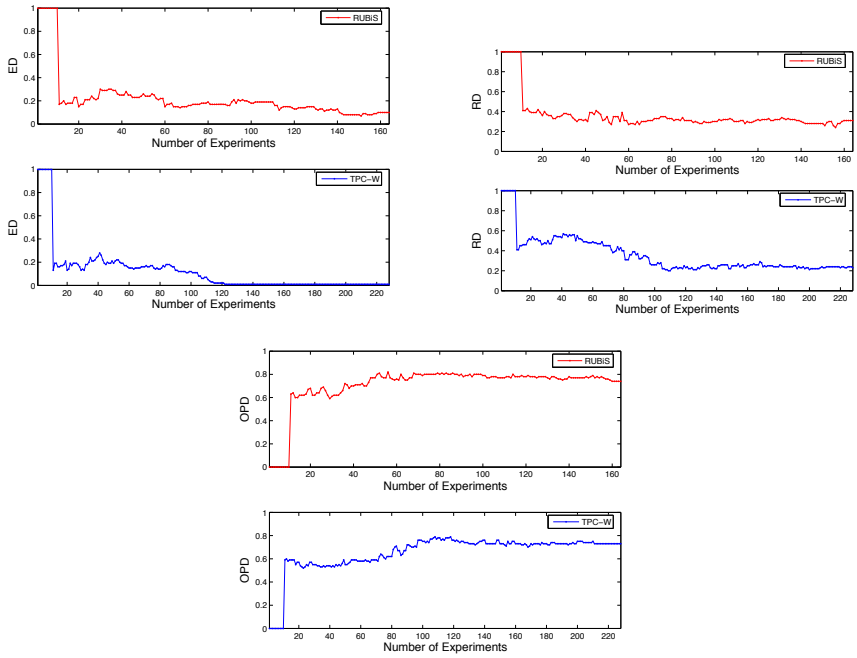

Figure 3.10: Quick convergence of the ranking of interactions to their best possible value for the average response time performance metric.

- The accuracy of model learned using samples that Algorithm 2 generates approaches that of the model learned with complete dataset with less than 1% of the total possible experiments for both TPC-W and RUBiS.

- The accuracy of the model that predicts RUBiS' average response time with complete dataset is 50%. The reason is that the first-order model with interactions is not the right model structure for modeling RUBiS' average response time metric. Models with more sophisticated structure are able to learn a more accurate model. Figure 3.14 shows the accuracy of the model that predicts RUBiS' average response time using a regression tree model [71], which results in better accuracy.

For the number-of-errors performance metric, we are unable to learn a reasonably accurate model with complete dataset either with the first-order model with interactions or regression trees. However, the accuracy of the model that is learned from

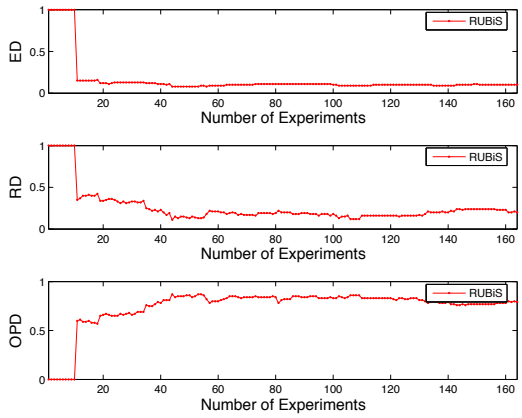

Figure 3.11: Quick convergence of the ranking of interactions to their best possible value for the number-of-errors performance metric for RUBiS. The number of errors is always 0 for all the experiments with TPC-W.

the samples that Algorithm 2 generates converges to the accuracy of the model from all the possible samples with less than 1% of the total possible experiments. Choice of an appropriate model structure for predicting the number-of-errors performance metric remains an interesting avenue for future work. Chapter 4 investigates the model structure for predicting the executing time of batch applications.

3.10 Related Work

To the best of our knowledge, this is the first work in systems management that explicitly quantifies the effect of interactions between the factors, and develops mechanisms and algorithms to capture the effect. Recent work in the computer architecture community has used design of experiments to explore the large space of factors that can affect the CPU performance. Joshua et al. [120] use design of experiments to rank the hardware factors in order of effect on CPU performance. However, the techniques used in [120] may not separate the effect of factors from that of interactions [76].

İpek et al. [57] use a predictive model to explore the CPU architecture design space; the samples for learning the model are collected using active machine learning

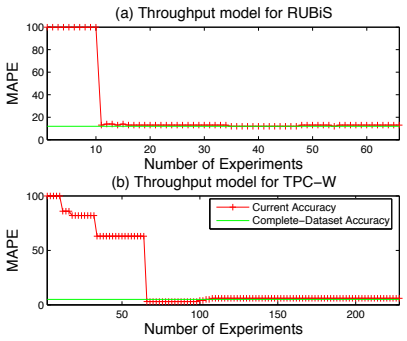

Figure 3.12: Accuracy of model that predicts RUBiS' and TPC-W's throughput. The model that is learned from samples that Algorithm 2 generates converges to the accuracy with the complete dataset with less than 1% of the total number of experiments.

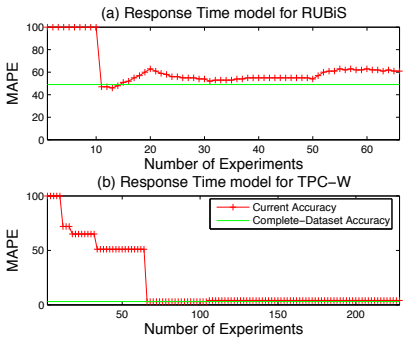

Figure 3.13: Accuracy of model that predicts RUBiS' and TPC-W's response time. The accuracy of the model that predicts RUBiS' average response time is only 50% even with the complete dataset. However, Algorithm 2 converges to this accuracy with less than 1% of the total number of experiments.

algorithm similar to the one that we use. However, the model does not consider interactions between the factors or the separation of the effect of factor from the effect of interactions between the factors. The techniques in our work can be used to identify the important factors and interactions, and hence complement the work in [57].

There is a large body of work on building performance models for Web service management, e.g., [113, 105, 66]. In most of the work, the samples for learning the models are collected passively from a service's normal operation. Section 1.3 presents the limitations of passive sampling for learning accurate models. Our work can be used to identify and collect the relevant samples for learning the models proposed in prior work.

3.11 Conclusions and Future Work

Managing system performance requires an accurate understanding of the factors and interactions that affect the system. This chapter presents mechanisms and

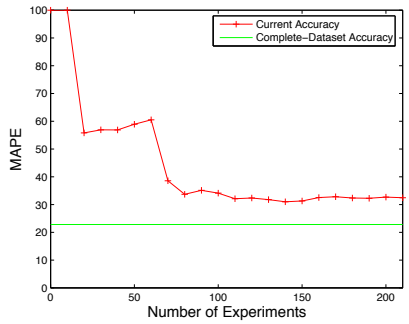

Figure 3.14: The more sophisiticated regression tree model has a better accuracy as compared to the first-order model with interactions (Figure 3.13) for predicting RUBiS' average response time.

experiment-planning algorithms to expose the important factors and interactions that affect the service performance, and learn models to predict system performance in the context of queries that arise in Web service management. This chapter does not explore the structure or use of models for Web service management, and focuses on learning the model with a given structure.

In general, identifying an appropriate model structure is an interesting and challenging problem. Chapter 6 further discusses the problem. The next chapter investigates the model structure for building models that can predict the execution time of batch applications as a function of the hardware resources assigned to the application and the properties of the data that the application process. The next chapter also presents the use of predictive models to do resource planning for batch applications in a utility computing setting.

Chapter 4

Resource Planning for Batch Applications

Previous chapter presents the use of experiment-driven framework to quantify the impact of important factors and interactions that affect system behavior, and learn models that can predict system behavior in the context of Web service management. This chapter focuses on identifying, building, and using predictive models to enable effective and efficient resource planning for batch applications in a distributed and shared collection of compute and storage resources—a networked computing utility. Shared computing utilities allocate compute, network, and storage resources to competing applications on demand. An awareness of the demands and behaviors of the hosted applications can enable the management controller to manage its resources more effectively.

This chapter presents an application performance model with a specific structure that captures the execution time of batch applications as a function of: (a) the compute, memory, and network resources assigned to the application; and (b) the properties of the data that the application process. It presents *NIMO*: a system that uses experiment-driven framework to learn such models proactively using only *noninvasive* instrumentation data that requires no changes to application or system. Finally, the chapter shows the use of the model to do resource planning for batch applications in a utility setting.

4.1 Background

High-performance computing has become a key driver for rapid advances in a range of sciences including astrophysics, bioinformatics, systems biology, and climate model-

ing [47, 101]. This new area of *computational science* has given rise to many resource-intensive scientific applications. For example, modern high-energy particle detectors generate up to 10^{15} bytes of data for analysis per year [10]. Other sources of important scientific applications include BIRN [16], GEON [43], and SDSS [99].

The typical scientific application can be represented as a *workflow* consisting of one or more *batch tasks* linked in a directed acyclic graph (DAG) representing task precedence and data flow (e.g., [15]). Complex scientific workflows are often run on networked computing utilities—systems that allocate compute, network, and storage resources on demand from a large heterogeneous resource pool. Examples of networked utilities include clusters of machines [25], computational grids [39], utility data centers, PlanetLab, and outsourced storage services [81].

A number of researchers have recently pointed out the critical need for automated systems to manage scientific workflows [47, 101]. One aspect of workflow management is *workflow planning* that involves finding an efficient *plan* for executing a workflow on a networked utility. Workflow planning is both important and challenging. Many scientific workflows perform complex computations, process very large amounts of data, or both. The difference in completion time can be on the order of days between a good execution plan for a workflow and a poor one [15]. These differences are magnified when workflows run on networked utilities composed of highly heterogeneous pools of geographically distributed resources.

A plan for a workflow G specifies a *resource assignment* for each batch task in the workflow. A task may be a batch application or a *data-staging task* interposed between a pair of application tasks. The resource assignment comprises the hardware resources—compute, network, and disk storage—that are assigned simultaneously to run G. G's performance can vary significantly across different resource assignments. To construct an effective assignment for G, the planning system must predict the in-

teraction of application characteristics (e.g., compute-to-communication ratio) with resource factors (e.g., CPU speed, cache, and I/O system behaviors). Specifically, the system needs a model that can predict G's total execution time on a candidate resource assignment. Execution time is a common performance metric for scientific application workflows. Accurate models are a prerequisite for selecting efficient resource assignments.

This work use the experiment-driven methodology (Chapter 2) to build simple and efficient predictive models with a specific structure based on limited prior application knowledge. It evaluates the effectiveness and accuracy of the models on a set of biomedical applications that run frequently on a shared production cluster at Duke, called the *DSCR*. It also illustrates the use of models to guide resource planning in the following scenarios:

- *Task placement.* Mapping individual tasks to candidate resources involves balancing multiple factors that affect performance on different CPU, host, network, and storage configurations. A system could use the models to evaluate alternatives and select the best candidates to maximize some objective.

- *On-time computing and SLOs.* A utility often must meet service-level objectives (SLOs) and performance targets for hosted applications. In a computational setting, specific deadlines may exist for tasks that deal with real-world events such as storm forecasting [93] and response. The system must find resource assignments that meet the performance constraints.

- *Storage outsourcing and data staging.* Storage access delays can be a key barrier to harnessing remote computing resources [46] or outsourced storage. This work shows how a system can use models to estimate the performance impact of remote I/O and the benefits of task migration or local staging.

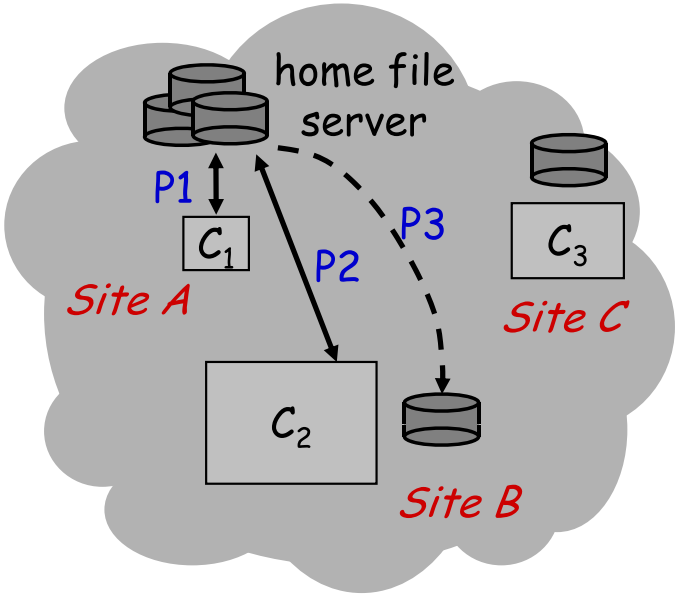

Figure 4.1: Three plans for executing a workflow G in a wide-area utility: P_1: run G locally at site A; P_2: run G at site B and access data remotely from A; P_3: stage the data at site C and run G locally at site C.

4.2 Motivating Example

Consider the scenario depicted in Figure 4.1, with three sites A, B, and C comprising a networked utility. Suppose a user at site A wants to run a workflow G on the utility. The input data for G is stored at A. Site B has the fastest compute resources, but insufficient storage to store G's input data locally. Site C has faster compute resources than A and sufficient local storage for G's data. The utility resource planner must choose a *plan* to execute G. Candidate plans include:

P_1: Run G locally at A.

P_2: Run G at B, so G gets the best compute resource available, but incurs remote I/O to A for data access.

P_3: Stage G's data to C from A, and run G locally at C.

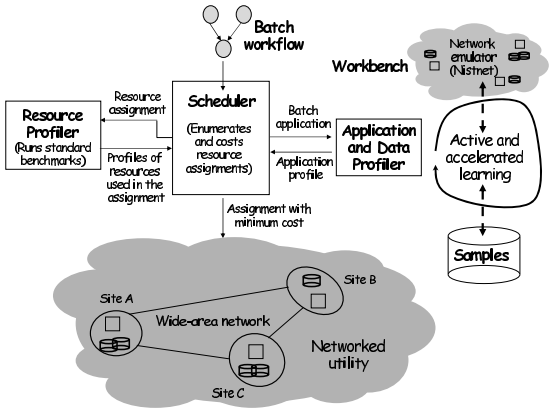

Figure 4.2: Architecture of NIMO.

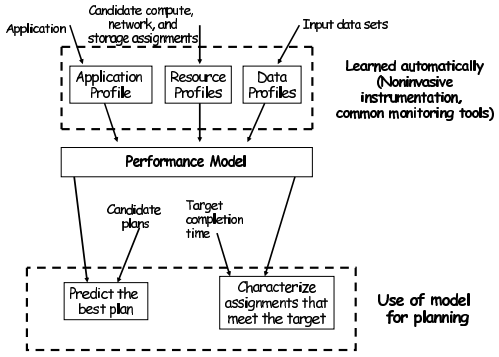

Figure 4.3: Overview of the model-guided planning approach.

Previous studies have shown that plan performance can vary significantly depending on application characteristics and underlying resource characteristics, e.g., [46, 21, 37, 89]. For example, plan P_2 may be more efficient than plans P_1 and P_3 if G does a lot of computation, but relatively little I/O.

4.3 Overview

This section presents the overview of the NIMO system. NIMO (NonInvasive Modeling for Optimization) is a workflow planning system that generates effective resource assignments or *plans* for workflows running on large-scale networked utilities. Figure 4.2 shows NIMO's overall architecture consisting of: (i) a *scheduler* that enumerates and selects plans for a workflow; (ii) a modeling engine, consisting of an *application profiler*, a *resource profiler*, and a *data profiler*, that learns the model to predict the execution time of a workflow for a plan. The modeling engine uses the workbench controller (Chapter 2, Figure 2.1) to collect training samples to learn the models proactively.

4.3.1 Scheduler

NIMO's scheduler is responsible for generating and executing a plan for a given workflow G. The scheduler enumerates candidate plans for G, estimates the execution time or *runtime* of each plan, and chooses the execution plan with the minimum total execution time. A *plan* P for workflow G is an execution strategy that specifies a resource assignment for each task in G. The scheduler uses a performance model $M(G, I, \vec{R})$ to estimate the execution time of G with input dataset I on a resource assignment $\vec{R}$. Thus $M(G, I, \vec{R})$ comprises the system knowledge base (Section 2.1) in this work.

The model is similar to the models that Chapter 3 presents, in that it predicts the performance of an application as a function of factors that affect application behavior. However, unlike Chapter 3 that focuses on the learning of models with any given structure, the model in this chapter has a specific structure based on limited prior application knowledge.

4.3.2 Modeling Engine

Figure 4.3 gives an overview of how NIMO estimates the performance of candidate plans. NIMO builds profiles of resources and frequently executed applications by analyzing instrumentation data gathered from previous runs. A performance model M for an application G predicts the performance of a plan for G given three inputs: (i) G's *application profile*, (ii) *resource profiles* of resources assigned to the plan, and (iii) *data profile* of the input data. Hence, in this work, the workload vector $\vec{W}$ consists of G's data profile, the resource vector $\vec{R}$ consists of resource profile of the resources assigned to the plan, and $\vec{C}$ is held constant (Equation 1.1).

Intuitively, the application profile captures how an application uses the input data and the resources assigned to it. Resource profiles specify parameters or *factors* that

characterize the function and power of those resources in an application-independent way. For example, a resource profile might represent a compute server with a fixed number of CPUs defined by factors such as clock rate and cache sizes, with an attached memory of a given size. Similarly, storage resources can be approximated by factors such as capacity, spindle count, seek time, and transfer speed. The data profile comprises the data characteristics of G's input dataset, e.g., the input data size. The profiles are described in Section 4.4.1.

4.3.3 Workbench

NIMO's modeling engine automatically learns the performance model for G from the samples attained by deploying G on selected resource assignments, either to serve a real request, or proactively to use idle or dedicated resources (a "workbench"; see Figure 2.1). NIMO's modeling engine actively initiates experiments—new runs of G on selected resource assignments—in the workbench guided by the workbench controller policies (Chapter 2). The goal is to obtain sufficient training samples for learning an accurate performance model for G quickly.

Instrumentation data is collected during an experiment, then aggregated to generate a sample as soon as the experiment completes. NIMO relies only on high-level metrics collected by commonly-available monitoring tools: (i) processor and disk usage data is collected using the popular `sar` utility [98]; and (ii) network I/O measures are derived from the `nfsdump/nfsscan` tools [35]. The goal is to use noninvasive instrumentation that requires no changes to operating system or application software.

4.4 Performance Model

A *workflow* G consists of one or more *tasks* linked in a directed acyclic graph (DAG) representing task precedence and data flow [15]. A *plan* makes a resource assignment

$\vec{R}$ to each task $G_i \in G$. Suppose that each task G_i accesses its inputs and outputs from the storage resources specified in $\vec{R}$, and that G_i *fires* only when its predecessors have produced its inputs, as is common [15].

In addition to the batch tasks in G, a plan may also interpose additional tasks for staging data between each pair of batch tasks in G. For example, a staging task G_{ij} between tasks G_i and G_j in the workflow DAG, copies the parts of G_j's input data produced by G_i from G_i's storage resource to that of G_j. Section 4.2 illustrates such a *staging task*.

Let G_i, $1 \leq i \leq l$ be the tasks—including both batch and staging tasks—in a plan. Let $\vec{R_i} = \langle C_i, N_i, S_i \rangle$ be the resource assignment made by P to task G_i. That is, when the scheduler schedules G_i on the networked utility, G_i executes on the compute resource C_i and accesses its input and output datasets from the storage resource S_i over the network resource N_i. (N_i will be null if S_i is local to C_i.)

The performance of the plan for executing G is determined by the performance of the tasks in its *critical path*, which can be found using existing techniques [38]. Given an estimate of the execution time of each task in the plan, the overall execution time of the plan is easy to estimate. Hence, this work focuses on modeling the performance of individual tasks G_i, or (equivalently) graphs G consisting of a single task.

4.4.1 Profiles

Consider a *batch* task G's execution as an interleaving of *compute phases*, in which the compute resource is doing useful work, and *stall phases*, in which the compute resource is stalled on I/O. For the execution of task G with input data I on the resource assignment $\vec{R}$, we define:

- The *compute occupancy* of G on $\vec{R}$, denoted o_a, is the average time spent computing per unit of data processed by G.

- The *stall occupancy* of G on $\vec{R}$, denoted o_s, is the average time for which the compute resource is idle per unit of data. Stall occupancy $o_s = o_n + o_d$, where o_n and o_d capture the portion of occupancy caused by delays in the network and storage (disk) resources respectively. Note that the "stall occupancy" components for network and storage do not represent the service demands or utilization at those resources; rather, they represent the compute stalls caused by delays at those stages, given the overall resource assignment.

The occupancies are determined by the interaction of the application and the resources, given the behavior of the application on the input data I. $\vec{R}$'s *resource profile* characterizes the resource properties, and the *data profile* represents the significant characteristics of I. G's *application profile* captures the interaction and the resulting application behavior in terms of the properties of the resource and data profiles. The profiles are defined as follows:

Resource Profile: A resource profile $\vec{\rho}$ is a vector $\langle \rho_1, \rho_2, \ldots, \rho_j \rangle$ where each ρ_i measures the value of some performance factor in $\vec{R}$. The performance factors are properties of the resources, and are quantifiable independently of any specific task. For example, for a compute resource, the factors may include processor speed, cache size, memory size, memory latency, and memory bandwidth.

Data Profile: The data profile $\vec{\lambda}$ of an input data I captures any characteristics of I that may correlate with resource demands, such as size, format type metadata, or histograms capturing data distribution.

Application Profile: G's application profile includes four *predictor functions* $f_a(\vec{\rho}, \vec{\lambda})$, $f_n(\vec{\rho}, \vec{\lambda})$, $f_d(\vec{\rho}, \vec{\lambda})$, and $f_D(\vec{\rho}, \vec{\lambda})$. f_a, f_n, and f_d are occupancy predictor functions that

predict G's occupancies o_a, o_n, and o_d respectively on a resource assignment $\vec{R}$ and input data I, as a function of $\vec{R}$'s resource profile $\vec{\rho}$ and I's data profile $\vec{\lambda}$. The data flow predictor function $f_D(\vec{\rho}, \vec{\lambda})$ estimates D, the total data flow processed by G (i.e., the total number of units of data read and written by G) for a given $\vec{R}$ and I.

Suppose the goal is to predict the completion time of G given G's application profile $\langle f_a, f_n, f_d, f_D \rangle$, the resource profile $\vec{\rho}$ of a candidate resource assignment $\vec{R}$, and input data I's data profile $\vec{\lambda}$. NIMO predicts G's completion time by the performance model $M(G, I, \vec{R})$:

$$Completion\ Time = f_D(\vec{\rho}, \vec{\lambda}) \times (f_a(\vec{\rho}, \vec{\lambda}) + f_n(\vec{\rho}, \vec{\lambda}) + f_d(\vec{\rho}, \vec{\lambda})) \tag{4.1}$$

4.4.2 Discussion of the Model Structure

In Figure 2.2 that shows the overall sequence of operations for building a system knowledge base, e.g., the model that this chapter proposes, the first step consists of identifying the structure of the knowledge base. For the model that this chapter proposes, we make a crucial choice to capture significant performance effects implicitly in the predictor functions, rather than explicitly in the structure of an a priori analytical model. The complexity arising from factors such as CPU caching, file caching, I/O patterns, latency hiding, concurrency, and queuing behavior at storage servers is captured *implicitly* in the training samples for learning the predictor functions and not in the model parameters. As long as the effects of these factors show in the training samples, the statistical learning techniques can capture them.

We considered the alternative of using a closed-loop queuing network model parameterized by analysis of the training samples. An explicit model would capture factors such as concurrency, latency hiding behavior, and queuing effects in the structure of the analytical model, so it can extrapolate beyond the samples used to learn

it. However, this approach is less general, and it may be difficult to obtain parameters such as service demands (utilization) from noninvasive instrumentation data, particularly for resources such as storage servers. In addition, this approach requires NIMO to infer the degree of concurrency in the closed-loop system, e.g., resulting from the depth of operating system prefetching, or the threading structure of the application.

By simplifying the structure of the model and relying on statistical learning, NIMO reduces the a priori knowledge required to model the application performance. This has several benefits. NIMO can build black-box models of applications, and hence apply to a large class of applications. Moreover, the model can be learned from high-level, commonly available, noninvasive instrumentation data. However, there is a cost for this generality. First, the training data and the statistical learning techniques required to learn an accurate model can be complex. Second, the model loses some of its ability to extrapolate beyond the behavior seen in the training data, and hence it is crucial that the training samples expose the application performance on the entire system operating range. Chapter 6 further discusses the spectrum of modeling alternatives.

4.5 Learning Data and Resource Profiles

In this work, the data profile for an input dataset I in NIMO is limited to I's total size in bytes. NIMO obtains resource profiles by running standard benchmark suites that are designed to expose the differences that are most significant for the performance of real applications. It uses *whetstone* [29] to calibrate processor speeds, *lmbench* [73] to calibrate memory latency and bandwidth, and *netperf* [80] to calibrate the network latency and bandwidth between compute and storage resources. The resource profiling approach is independent of the specific benchmarks as long

as they capture the underlying resource characteristics. Other researchers have: (1) confirmed that simple benchmarks can be used in profiling high-performance computing platforms [19]; (2) studied benchmark selection for comprehensive coverage [120]; and (3) devised strategies for robust resource profiling in the presence of competition for shared resources [114].

4.6 Experiment-Driven Learning of Models

NIMO's modeling engine uses the experiment-driven framework to learn the predictor functions comprising G's application profile. NIMO associates a specific dataset I along with a cost model for a task G. That is, a separate cost model is built for each task-dataset combination. The advantage is that the variable parameters in the predictor functions in G's application profile now consist of the resource-profile factors only, and not the data-profile factors. That is, the predictor functions have the simpler form $f(\vec{\rho})$ instead of $f(\vec{\rho}, \vec{\lambda})$. While the problem of automatically learning G's predictor functions simplifies, it largely remains the same and nontrivial. The disadvantage is that NIMO has to learn a new cost model for each new input dataset for G. However, many scientific applications are often run repeatedly on the same input dataset, and these runs tend to have similar resource-usage behavior [88].

If NIMO is given a reasonably large and representative set of samples of the form $\langle \rho_1, \rho_2, \ldots, \rho_k, o_a, o_n, o_d, D \rangle$, then the problem of learning accurate predictor functions $\langle f_a(\vec{\rho}), f_n(\vec{\rho}), f_d(\vec{\rho}), f_D(\vec{\rho}) \rangle$ reduces to a statistical learning problem of fitting accurate functions to predict each of o_a, o_n, o_d, and D using subsets of factors in $\rho_1, \rho_2, \ldots, \rho_k$. The challenge, however, is that NIMO does not initially have a representative sample set for training. Instead, NIMO collects each sample by conducting an experiment, i.e., proactively running G to completion on a selected resource assignment in the workbench. The total overhead of collecting training samples can be extremely high

because of the curse of dimensionality—resource profile $\vec{\rho} = \langle \rho_1, \ldots, \rho_k \rangle$ may contain many factors (i.e., k may be large)—and the high cost of data acquisition per sample—collecting a sample $\langle \rho_1, \ldots, \rho_k, o_a, o_n, o_d, D \rangle$ involves a complete run of G on $\vec{R}$, which could take up to hours or days for some scientific tasks.

Algorithm 5 illustrates the main steps that NIMO uses to learn G's application profile. (Details of these steps are discussed in Sections 4.6.1–4.6.6.) The algorithm is an instance of the sequence of operations in the experiment-driven framework; see Figure 2.2. The algorithm consists of an initialization step and a loop. The loop continuously refines the structure of the model (loop L_1 in Figure 2.2) and the accuracy of the predictor functions (loop L_2 in Figure 2.2) by learning from new samples acquired by running G on new resource assignments instantiated in the workbench.

4.6.1 Initialization

The initialization step of Algorithm 5 (Step 1) runs the task G on a designated *reference resource assignment* $\vec{R}_{ref} = \langle C_{ref}, N_{ref}, S_{ref} \rangle$ where C represents the compute resource assignment, N represents the network resource assignment, and S represents the storage resource assignment. Based on this run, NIMO measures the compute, network-stall, and disk-stall occupancies—called the *reference occupancies*—and the total data flow—called the *reference data flow*—of G on $\vec{R}_{ref}$. The details of this step follow from: (i) Algorithm 6, which shows how NIMO runs a task on a resource assignment instantiated in the workbench; and (ii) Algorithm 7, which shows how NIMO computes the occupancies and total data flow for a run. Specifically, the initialization step runs G on $\vec{R}_{ref}$ using Algorithm 6, then it measures G's occupancies and total data flow on $\vec{R}_{ref}$ using Algorithm 7.

Once NIMO computes the reference occupancies in Step 1 of Algorithm 5, it

Algorithm 5: Experiment-driven learning of predictor functions $f_a(\rho_1, \ldots, \rho_k)$, $f_n(\rho_1, \ldots, \rho_k)$, $f_d(\rho_1, \ldots, \rho_k)$, and $f_D(\rho_1, \ldots, \rho_k)$ for task G

1) **Initialize:** (Section 4.6.1) Obtain reference occupancies o_{aref}, o_{nref}, o_{dref} and reference data flow D_{ref} for G on a reference resource assignment $\vec{R}_{ref}$; Set $f_a(\vec{\rho}) = o_{aref}$, $f_n(\vec{\rho}) = o_{nref}$, $f_d(\vec{\rho}) = o_{dref}$, and $f_D(\vec{\rho}) = D_{ref}$ (constant functions);

2) **Design the next experiment:** (Sections 4.6.2–4.6.4)

   2.1 Select a predictor function for refinement, denoted f (Section 4.6.2);

   2.2 Should a factor from $\rho_1, \ldots, \rho_k$ be added to the set of factors already used in f? If yes, then pick the factor to be added (Section 4.6.3);

   2.3 Select new assignment(s) to refine f using the set of factors from Step 2.2 (Section 4.6.4);

3) **Conduct the chosen experiment:** (Section 4.6.5)

   3.1 Run G in the workbench using the assignment(s) picked in Step 2.3;

   3.2 After each run, generate the corresponding sample $\langle \rho_1, \ldots, \rho_k, o_a, o_n, o_d, D \rangle$, where o_a, o_n, o_d are the observed occupancies and D is the observed total data flow;

   3.3 Learn f (and other predictor functions) from the new sample set;

4) **Compute current prediction error:** (Section 4.6.6) Compute current prediction error of each predictor. If the overall error in predicting execution time is below a threshold, and a minimum number of samples have been collected, then stop, else go to Step 2.

initializes the predictor functions to constant functions that predict the compute, network-stall, and disk-stall occupancies and the total data flow of G on any resource assignment $\vec{R}$ as equal to the corresponding reference values; which is a reasonable thing to do based on the single run of G so far. NIMO refines the predictor functions in Algorithm 5 as it collects more samples.

There are many ways in which NIMO can choose the reference assignment $\vec{R}_{ref} = \langle C_{ref}, N_{ref}, S_{ref} \rangle$ from the different candidate assignments available in the workbench:

- *Random assignment (Rand)*: Pick each of C_{ref}, N_{ref}, and S_{ref} at random from

Algorithm 6: Running G on $\vec{R} = \langle C, N, S \rangle$

1) Instantiate a Network File System (NFS) server on the storage resource S in $\vec{R}$. Export a storage volume from S containing G's input dataset I, and mount this volume on C. Set G to access this volume;
2) Set routing tables in C and S so that all communication happens via a specific router r running NIST Net [102]. r is configured to emulate the network specifications (e.g., latency and bandwidth) of S;
3) Start monitoring tools (Section 4.3.3) to measure the execution time T and C's utilization U (required by Algorithm 7) for this run;
4) Start G on C. When the task finishes, stop the monitoring tools, and compute T and U.

among the corresponding resources in the workbench.

- *High-capacity assignment (Max)*: Pick the compute resource with the fastest processor speed, the network resource with minimum latency, and the storage resource with maximum transfer rate.
- *Low-capacity assignment (Min)*: Pick the compute resource with the slowest processor speed, the network resource with maximum latency, and the storage resource with minimum transfer rate.

4.6.2 Guiding the Sequence of Exploration for the Predictor Functions

In each iteration of Algorithm 5, Step 2.1 picks a specific predictor function to refine by collecting more samples for training. We consider both static and dynamic schemes to guide this sequence for exploring the predictor functions across iterations.

Static Schemes: A static scheme first decides a *total ordering* of the predictor

Algorithm 7: Computing task G's occupancies on $\vec{R} = \langle C, N, S \rangle$

1) Using Algorithm 6, run G on $\vec{R}$, and measure C's average utilization U, G's execution time T, and the total data flow D (using network I/O traces);
2) Solve for o_a and o_s from $U = \frac{o_a}{o_a+o_s}$, $\frac{D}{T} = \frac{1}{o_a+o_s}$;
3) Use network I/O traces to derive the average time spent per I/O in the network resource N and in the storage resource S;
4) Split $o_s = o_n + o_d$ into o_n and o_d in proportion to the ratio of network and storage components of the average I/O time from Step 3, to obtain $\langle o_a, o_n, o_d, D \rangle$.

functions $f_a(\vec{\rho})$, $f_n(\vec{\rho})$, $f_d(\vec{\rho})$, and $f_D(\vec{\rho})$, then defines a fixed *traversal plan* for picking the predictor function to refine in each iteration. NIMO currently supports two techniques each for ordering and for traversal. The two ordering techniques are:

- *Domain-knowledge-based* where a domain expert specifies a total order of the predictor functions to NIMO. For example, the expert may know that the scientific task G is likely to be CPU-intensive for most resource assignments because G performs complex computations per unit of data in I, so $f_a(\vec{\rho})$ should come first in the total order and be refined first.

- *Relevance-based* where NIMO estimates the relevance of the predictor functions on G using the classic *Plackett-Burman design with foldover* (*PBDF*) experiment design [120] (Section 3.5.2). NIMO orders the predictor functions in decreasing order of effect. To order the four predictor functions using PBDF, NIMO performs eight runs of G on predefined resource assignments.

The two techniques to traverse a given total order are:

- *Round-robin* where NIMO chooses the predictor functions to refine across iter-

Algorithm 8: Dynamic scheme for picking the predictor function to refine in an iteration of Algorithm 5

1) Let $s_1, \ldots, s_m$ be the m training samples of the form $\langle \rho_1, \rho_2, \ldots, \rho_k, o_a, o_n, o_d, D \rangle$ collected so far;
2) Supplement each of the m samples with the predicted compute occupancy o_{ap} from the current $f_a(\rho_1, \ldots, \rho_k)$; similarly, the predicted network-stall occupancy o_{np} from the current $f_n(\rho_1, \ldots, \rho_k)$, the predicted disk-stall occupancy o_{dp} from the current $f_d(\rho_1, \ldots, \rho_k)$, and the predicted data flow D_p from the current $f_D(\rho_1, \ldots, \rho_k)$;
3) Use the m actual and predicted value-pairs $\langle o_a, o_{ap} \rangle$ to compute the current prediction error E_a of $f_a(\rho_1, \ldots, \rho_k)$ (see Section 4.6.6); similarly, compute E_n, E_d, and E_D from the respective $\langle o_n, o_{np} \rangle$, $\langle o_d, o_{dp} \rangle$, and $\langle D, D_p \rangle$ pairs;
4) Pick for refinement the predictor function with maximum current prediction error.

ations in a round-robin fashion from the given total order.

- *Improvement-based* where NIMO traverses the total order from beginning to end, and keeps refining the current predictor function until the reduction in the prediction error obtained in the last iteration drops below a predefined threshold. (Section 4.6.6 describes the computation of the current prediction error of a predictor function.) When the reduction in error drops below the threshold, NIMO moves on to the next predictor function in the total order. When it exhausts all predictor functions, it resumes at the beginning of the total order.

Dynamic Schemes: Dynamic schemes do not use a static ordering of the predictor functions. Instead, the function to refine in each iteration is based on the training samples collected so far. NIMO currently considers one dynamic scheme that, in each iteration, chooses to refine the predictor function with the maximum current prediction error, as illustrated in Algorithm 8.

4.6.3 Adding New Factors to Predictor Functions

Step 2.2 of Algorithm 5 decides when to add a new resource-profile factor to a predictor function $f(\vec{\rho})$, and if so, which of the k factors from $\rho_1, \ldots, \rho_k$ to add for maximum potential reduction in $f(\vec{\rho})$'s prediction error. (Recall from Step 1 of Algorithm 5 that $f(\vec{\rho})$ is initially set to a constant function having no variable parameters.) As in Section 4.6.2, NIMO's twofold strategy is to first define a total order over the $\rho_1, \ldots, \rho_k$ factors with respect to $f(\vec{\rho})$, and then to define a traversal plan based on this order to select factors for inclusion in $f(\vec{\rho})$.

Following a approach similar to the one in Section 4.6.2, a total ordering of the resource-profile factors $\rho_1, \ldots, \rho_k$ for predictor function $f(\vec{\rho})$ can be:

- *Domain-knowledge-based* where a domain expert specifies a total ordering of $\rho_1, \ldots, \rho_k$ for $f(\vec{\rho})$. For example, the expert may know that the task has a purely sequential I/O pattern. Thus, the memory-size factor may have minimal effect on the compute occupancy o_a, so this factor can be placed towards the end of the total order for $f_a(\vec{\rho})$.

- *Relevance-based* where NIMO first estimates the effect of each resource-profile factor on the occupancy predicted by $f(\vec{\rho})$ using PBDF. Then, it orders the resource-profile factors in decreasing order of effect.

Based on the total ordering of factors $\rho_1, \ldots, \rho_k$ for a predictor function $f(\vec{\rho})$, NIMO decides when to add the next factor in the total order to the current set of factors in $f(\vec{\rho})$. The *improvement-based* approach that NIMO uses here adds the next factor in the order when the reduction achieved in prediction error during an iteration with the current $f(\vec{\rho})$ (i.e., with the current set of factors) falls below a predefined threshold. When NIMO exhausts all factors, it resumes at the beginning of the total order.

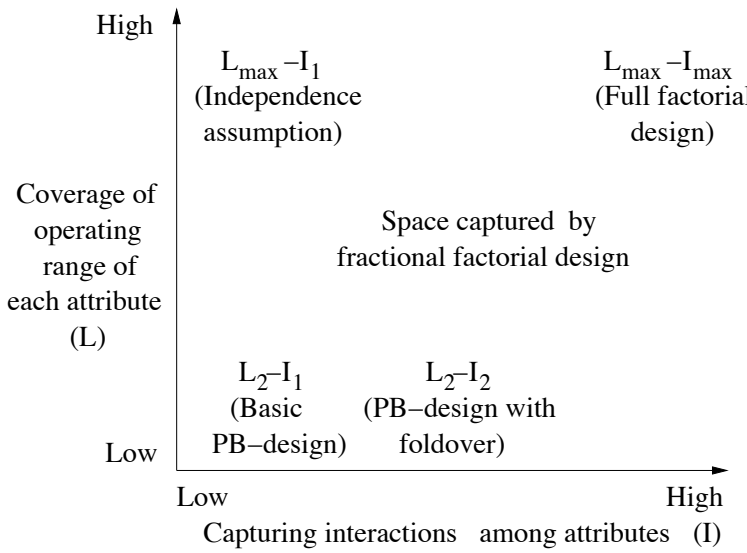

Figure 4.4: Techniques for selecting new sample assignments (PB = Plackett-Burman [120]).

4.6.4 Selecting New Sample Assignments

Step 2.3 of Algorithm 5 chooses new assignments to run task G and collect new samples for learning. To create a new assignment $\vec{R}$, NIMO needs to select a value of each factor ρ_i in $\vec{R}$'s resource profile, while accounting for the factors that Section 1.3 outlines.

Figure 4.4 shows the range of techniques for selecting new samples that this chapter considers. The techniques are shown in terms of their general performance and tradeoff on the two metrics above, namely, covering the operating range of factors, and capturing significant interactions among factors. The techniques are described by an L_α-I_β naming format, where (i) α represents the number of significant distinct values, or *levels*, in the factor's operating range covered by the technique, and (ii) β represents the largest degree of interactions among factors guaranteed to be captured by the technique. Among these techniques, the ones which we are currently working with are:

- L_{max}-I_1: This technique, described in Algorithm 9, systematically explores all

Algorithm 9: Selecting the next sample assignment using L_{max}-I_1

1) Let $f(\vec{\rho})$ be the current predictor function chosen for refinement in Step 2.1 of Algorithm 5. Let ρ_r be the factor most recently added to $f(\vec{\rho})$ in Step 2.2 of Algorithm 5. Let $IN \subset \vec{\rho}$ be the set of factors already considered for addition in $f(\vec{\rho})$, and $OUT \subset \vec{\rho}$ be the set of factors not considered for addition in $f(\vec{\rho})$. Let $\vec{R}_{ref}$ be the designated reference assignment chosen in the initialization step of Algorithm 5;

2) L_{max}-I_1 chooses the new assignment as follows:

    1. All factors in IN and OUT are set to the corresponding values in $\vec{R}_{ref}$;
    2. The value of ρ_r is set to the next unselected value from the binary-search sequence—lo, hi, $\frac{lo+hi}{2}$, $\frac{3lo+hi}{4}$, $\frac{lo+3hi}{4}$, $\frac{7lo+hi}{8}$, $\frac{5lo+3hi}{8}$, and so on—where lo and hi are the minimum and maximum values ρ_r can take. That is, lo is chosen for the first assignment, hi is chosen for the next assignment, and so on.

levels of a newly-added factor using a binary-search-like approach. However, it assumes that the effects of factors are independent of each other, i.e., there are no interactions among them. Hence it chooses values for factors one factor at a time.

- L_2-I_2: This technique is an adaptation of PBDF. Given the total number of factors, L_2-I_2 specifies the number of samples required and the values of factors in each sample. L_2-I_2 captures two levels (e.g., *low* and *high* [120]) per factor and up to pair-wise interactions among factors.

In our evaluation (Section 4.9), we find that these simple techniques are sufficient for selecting the experiments for learning a reasonably accurate execution time model quickly. Section 3.7 presents general and more sophisticated model-learning experiment designs for selecting the sample assignments.

Algorithm 10: Learning G's compute occupancy predictor function $f_a(\vec{\rho})$

1) Suppose m runs of G have been conducted, where run i is on $\vec{R}_i$. Let $\langle \rho_{1_i}, \ldots, \rho_{j_i} \rangle$ be the subset of $\vec{R}_i$'s resource-profile factors added to $f_a(\vec{\rho})$ so far;
2) Use Algorithm 7 to generate m training data points where the ith point is $\langle \rho_{1_i}, \ldots, \rho_{j_i}, o_{a_i} \rangle$;
3) Normalize the training points using a *baseline* assignment $\vec{R}_b$ with resource profile $\vec{\rho}_b$. (Currently, NIMO chooses $\vec{R}_b = \vec{R}_{ref}$.) Let G's compute occupancy on $\vec{R}_b$ be o_{a_b}, so the ith normalized training data point is $\langle \frac{\rho_{1_i}}{\rho_{1_b}}, \ldots, \frac{\rho_{j_i}}{\rho_{j_b}}, \frac{o_{a_i}}{o_{a_b}} \rangle$;
4) Use regression on the training data to learn a function $F(\vec{\rho})$ that predicts the value of $\frac{o_a}{o_{a_b}}$ from the normalized values of $\rho_1, \ldots, \rho_j$. Set $f_a(\vec{\rho}) = o_{a_b} \times F(\vec{\rho})$.

4.6.5 Conducting the Selected Experiment

In Step 3.1 of Algorithm 5, NIMO instantiates the assignment selected in Step 2.3 in the workbench and runs the task; details of running task G on a resource assignment $\vec{R}$ are given in Algorithm 6. The compute, network-stall, and disk-stall occupancies, and the total data flow, are collected from the run as described in Algorithm 7. These measures give a new sample of the form $\langle \rho_1, \ldots, \rho_k, o_a, o_n, o_d, D \rangle$. NIMO then analyzes all the samples collected so far, including the one collected most recently, to refine the predictor function chosen in Step 2.1 of Algorithm 5 with its current factor set as chosen in Step 2.2. If the latest run provides a new sample for another predictor f based on the current set of factors included in f, then NIMO refines f as well. The details of this step are given in Algorithm 10 for f_a; f_n, f_d, and f_D can be learned similarly.

4.6.6 Computing Current Prediction Error

NIMO considers two techniques for computing the current prediction error of a predictor function $f(\vec{\rho})$:

1. *Cross-validation*: In this technique, NIMO uses *leave-one-out* cross-validation to estimate the current prediction error of $f(\vec{\rho})$ [1]. For each sample s out of the m samples collected so far, NIMO learns $f(\vec{\rho})$ using all samples other than s (using Algorithm 10). NIMO then uses $f(\vec{\rho})$ to predict the corresponding occupancy for s, and computes the *absolute percentage error*. For example, if the predictor function is $f_a(\vec{\rho})$, and the actual and predicted occupancies for s are o_a and o_{ap} respectively, then the absolute percentage error is $\frac{|o_a - o_{ap}|}{o_a} \times 100\%$. The average of the m individual values of absolute percentage error, denoted *Mean Absolute Percentage Error (MAPE)*, is the current prediction error.

2. *Fixed test set*: In this technique, NIMO designates a small subset of resource assignments in the workbench as an *internal test set*. The test assignments may be a random subset of the possible assignments in the workbench, or chosen more robustly; When a fixed test set is used, the initialization step of Algorithm 5 begins by running the task on each assignment in the test set. NIMO computes the current prediction error of $f(\vec{\rho})$ as the MAPE in predicting occupancy on each assignment in the test set. Note that the samples collected for this test set are never used as training samples for any predictor function.

4.7 Experimental Evaluation

We validate the model, evaluate the techniques to learn them, and present the use of model for resource planning as follows:

- **Model Validation**. Section 4.8 presents the model validation and explores the sensitivity of the model accuracy to a number of factors. The validation consists of using a standard cross-validation methodology to evaluate the effectiveness of model induction on multiple metrics.

  The evaluation shows that the proposed models parameterized from noninvasive instrumentation data are sufficiently accurate to be of use in utility resource planning. The mean percentage error in application completion time predictions for a given assignment of compute, network, and storage resource is within 10%, and ranking accuracy of candidate assignments is close to 95-100% for all the real application tasks and for all but one synthetic application task. The worst reported mean percentage error in completion time prediction occurs for a synthetic application task doing sequential file reads—30%. Even in this case, the induced models are accurate to within 23% for 90% of the trials, and ranking accuracy of candidate assignments is close to 90% for all trials.

- **Experiment-Driven Model Learning**. Section 4.9 evaluates the algorithms for experiment-driven learning of models (Section 4.6). The evaluation shows that experiment-driven learning reduces the time to learn accurate models by an order of magnitude compared to approaches that sample a significant part of the entire sample space. The evaluation considers samples attained on physical resources as well as virtual machines.

- **Model-Guided Planning**. Section 4.10 illustrates the potential role of the induced performance models to guide resource planning in several scenarios for a network utility: task placement, on-time task completion, and storage outsourcing or data staging.

Table 4.1: Applications used in NIMO experiments.

Application	Description
fMRI [55]	Statistical parametric mapping to normalize functional MRI images of human brains
CardioWave [91]	Cardiac electrophysiology simulation (MPI)
BLAST [3]	Search genomic dataset for matches on proteins and nucleotides
NAMD [90]	Simulation of large biomolecular systems (MPI)
GAMUT [77]	An application emulation tool for generating work-mixes; generate 6 synthetic tasks with equivalent compute costs but different access patterns: rand./seq. read/write, and rand./seq. read and write in 1:1 ratio

4.8 Model Validation

This work considers a range of applications and resource assignments in the workbench (Figure 2.1). A recent survey of $280,000$ jobs submitted to the DSCR cluster from a diverse set of user groups (computational biology, physics, chemistry, biochemistry, biomedical, statistics, etc.) show that almost 90% of the jobs are sequential batch jobs. The techniques in this work are applicable to parallel applications, but we do not consider them in this work. All applications that we analyze in this work run on a single CPU. The parallel applications that we analyze in this work run on a single node.

We use standard cross-validation methodology to evaluate the effectiveness of model induction with various training sets, and the accuracy of the resulting performance predictions. Section 4.8.1 uses multiple accuracy metrics to profile the prediction accuracy, and Section 4.8.2 explores the sensitivity of the results to training set selection.

Workloads. Table 4.1 lists the applications used in the experiments. The *fMRI, CardioWave*, *BLAST*, and *NAMD* applications are used in biomedical research at

Duke and elsewhere. We consider one representative input dataset for each application. GAMUT is a tool for emulating cluster application workloads with controlled degrees of CPU, I/O (including I/O patterns), and memory usage. GAMUT is useful for generating synthetic workloads to explore the space of possible application behaviors comprehensively.

Candidate resources. NIMO's testbed contains five compute nodes—451 MHz, 797 MHz, 930 MHz, 996 MHz, 1396 MHz—with Intel PIII architecture, CPU cache size ranging from 256 KB to 512 KB, memory size ranging from 512 MB to 2 GB, and Linux 2.4.25 kernel. We used `NISTnet` [20] to impose varying network round-trip delays: $0, 2, \ldots, 16, 18$ ms. For the experiments reported in this work we held the storage system, memory size, CPU architecture (Pentium III), and network bandwidth constant. The five different CPU configurations and ten network latencies yield a total of 50 candidate resource assignments for a task in the testbed.

Model Construction. NIMO currently uses first-order multivariate linear regression [78] models for the structure of the predictor functions. (More sophisticated regression techniques, e.g., *transform regression* [123] or higher-order regression can be applied in NIMO without changing the overall approach.) A typical predictor function in our experiments has the form: $f(\vec{\rho}) = a_1 g_1(\rho_1) + a_2 g_2(\rho_2) + \cdots + a_k g_n(\rho_k) + c$, where each a_i is a regression coefficient, each ρ_i is a resource-profile factor, each g_i is a transformation function, and c is a constant. Apart from the default $g(\rho_i) = \rho_i$ transformation, this work also consider reciprocal transformations. For example, a reciprocal transformation is applied to the CPU speed factor because occupancy values are inversely proportional to CPU speed. The experiments reported in this section focus on learning the three occupancy predictor functions f_a, f_n, and f_d automatically, and assume that the data-flow predictor f_D is known.

The accuracy experiments use a 50-way cross-validation methodology as follows. For each application in Table 4.1 we use the following simple strategy for learning the model. Section 4.9 evaluates the experiment-driven approach for automated model-learning (Section 4.6).

1. We fix one of the 50 candidate assignments as the reference assignment $\vec{R}_{ref}$.

2. NIMO learns the predictor functions for the application using a *training set* of 14 (28%) of the 50 candidate assignments containing at most one non-reference resource.

3. The resource assignments not in the training set constitute the *test set* on which we compare model-predicted completion times with measured completion times to evaluate prediction accuracy.

4. For cross-validation and sensitivity analysis, we repeat Steps 1–2 50 times with different training and test sets. A different $\vec{R}_{ref}$ is selected for each trial.

As an illustration, Figure 4.5 shows two projections of a subset of training data collected for *fMRI*. These projections show that the dominant effect on compute occupancy in this case comes from the speed of the compute resource (as characterized by *whetstone* performance, Section 4.5). Similarly, in these assignments, changes in network latency affect only network occupancy, but have little effect on compute occupancy. Based on this training data, NIMO learns the following occupancy predictor functions for *fMRI*, with occupancy measured in microseconds per byte.

$$f_a = \frac{4.40}{\rho_{cpu\ speed}} - 0.2, f_n = 4.46\ \rho_{net\ lat} + 0.7, f_d = 0.32$$

Here, $f_D = 0.1064 \times 10^9$ bytes. $\rho_{cpu\ speed}$ and $\rho_{net\ lat}$ are factors of the resource profile of the candidate assignment normalized to the reference assignment. Depending on the

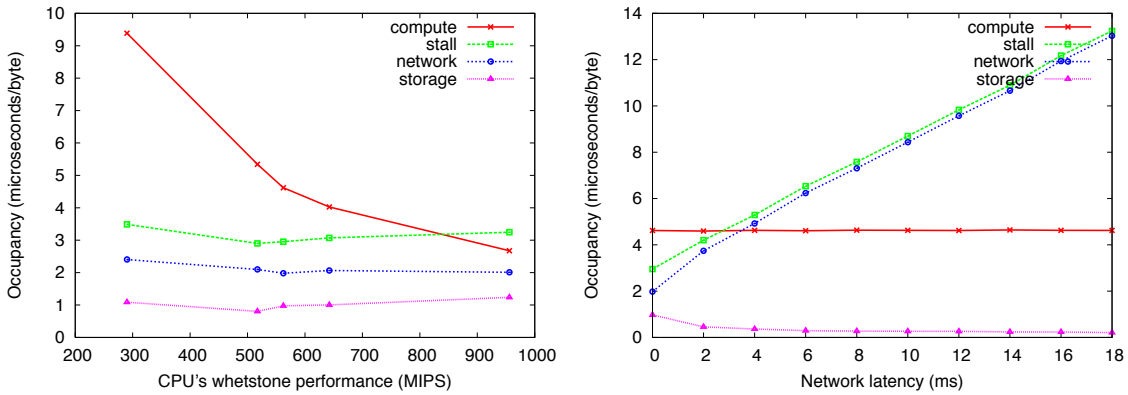

Figure 4.5: Impact of CPU speed and network latency on *fMRI*'s occupancies.

heterogeneity in the hardware resources, the predictor functions may depend on more factors, e.g., architecture type, memory size, number of disk spindles, and network bandwidth. Section 4.6.3 presents techniques that can automatically identify the most relevant factors. When we evaluate the above model on the test assignments, the predicted completion times are within 5% of the measured completion times.

4.8.1 Model Accuracy

Figure 4.6 summarizes the accuracy of the completion times predicted by the induced performance models. It reports the following metrics:

- **Absolute Error (AE) and Percentage Absolute Error (PE).** AE measures the absolute difference between predicted completion time and measured completion time. PE measures the corresponding percentage error. AE and PE are particularly relevant to the use of models to identify candidate resource assignments to meet a completion time target.

- **Top-k Ranking Distance (RD(k)).** Consider n resource assignments that are ranked in the order $\vec{R}_1, \ldots, \vec{R}_n$ based on increasing completion time for application G. Let p_i be the model-predicted rank of $\vec{R}_i$. The normalized

	AE (mins)				PE				OPD				RD(1)			
APPLICATION	**μ**	**σ**	**Wst**	**90 pc**	**μ**	**σ**	**Wst**	**90 pc**	**μ**	**σ**	**Wst**	**90 pc**	**μ**	**σ**	**Wst**	**90 pc**
fMRI	.9	.6	4.2	1.7	3	2	15	6	.98	0	.98	.98	.02	.02	.05	.05
CardioWave	.9	.7	2.9	1.2	11	7	27	22	.87	.02	.83	.85	.07	.03	.09	.09
BLAST	.1	.08	.36	1.1	1	.7	4	2	1	0	1	1	0	0	0	0
NAMD	.6	.4	1.5	1	8	6	19	13	.92	.02	.89	.90	.15	.09	.21	.21
GAMUT (seq. read)	.7	1.2	4.6	3.2	6	8	30	23	.92	.02	.89	.90	.15	.09	.21	.21
GAMUT (rand. read)	.1	.09	.4	.2	3	2	13	6	.98	.01	.97	.97	.02	.02	.04	.04
GAMUT (seq. write)	.1	.1	.5	.2	2	2	7	4	.98	.01	.97	.97	.03	.03	.09	.09
GAMUT (rand. write)	.4	.3	1.6	.9	5	3	15	10	.95	.02	.93	.93	.01	.03	.09	.04
GAMUT (rand. r/w)	.1	.07	.35	.19	1	1	6	2.3	.99	.01	.99	.99	.01	.02	.04	.04
GAMUT (seq. r/w)	.3	.3	1.6	.8	5	5	23	12	.90	.02	.88	.88	.09	.02	.16	.12

Figure 4.6: Summary of accuracy metrics for 4 real and 6 synthetic applications using 50-way cross validation. μ = Mean, σ = Standard Deviation, *Wst* = Worst Case Error, *90 pc* = 90th percentile.

top-k ranking distance *RD(k)*. for model predictions is $\frac{\sum_{i=1}^{i=k}|p_i - i|}{\sum_{i=1}^{i=k} n-i}$. When the model is used to select resource assignments in a utility setting, a low value of *RD(k)* indicates good performance. $0 \leq RD(k) \leq 1$, with $RD(k) = 0$ indicating accurate ranking of the k best assignments.

- **Order Preserving Degree (OPD) [123].** A model preserves the relative order of resource assignments i and j iff $t_{p_i}(<,=,>)t_{p_j}$ holds when $t_{o_i}(<,=,>)t_{o_j}$ holds, where t_p and t_o respectively represent model-predicted and measured completion times. Given n candidate assignments, $OPD = \frac{|OPP|}{n^2}$ where OPP is the set of order-preserving pairs. $0 \leq OPD \leq 1$, with $OPD = 1$ indicating accurate ranking.

For each metric, Figure 4.6 reports the mean, standard deviation, worst-case value (100th percentile), and 90th percentile value from a 50-way cross-validation. Note that mean percentage error from model predictions is under 11%, and ranking-related

errors are low (high OPD and low RD). The worst reported PE occurs for a synthetic application doing sequential file reads (GAMUT seq. read in the table). Even in this case, the induced models were accurate to within 23% for 90% of the trials. Section 4.8.2 explores this issue in more detail below.

4.8.2 Sensitivity Analysis

Choice of reference assignment

As noted, the induced predictor functions normalize resource occupancy as relative to occupancy observed on a designated reference assignment $\vec{R}_{ref}$ (see Algorithm 10). The 50-way cross-validation experiments in Figure 4.6 select a different reference assignment $\vec{R}_{ref}$ for each trial. The low variances in Figure 4.6 suggest that model induction is robust to the choice of reference assignment.

Queuing delays and concurrency

Although we do not investigate parallel applications in this work, our approach appears to be robust across degrees of concurrency in storage access. The key assumption is that the internal degree of concurrency is fixed in the application and the OS it runs on, i.e., the CPU initiates a bounded number of pending I/Os. Given this assumption, the parameterization captures the impact of queuing (e.g., in the storage system) implicitly.

We create a synthetic task (GAMUT rand. write in Figure 4.6) that saturates the storage resource to introduce a bottleneck. Figure 4.7 shows the impact of queuing delays on the occupancies. As seen in the figure, faster network or faster CPU increases the occupancy at the storage resource and vice versa. Such an effect is also created if concurrency at the compute resource introduces an I/O rate that cannot be handled by the storage resource, e.g., in the case of a parallel application with

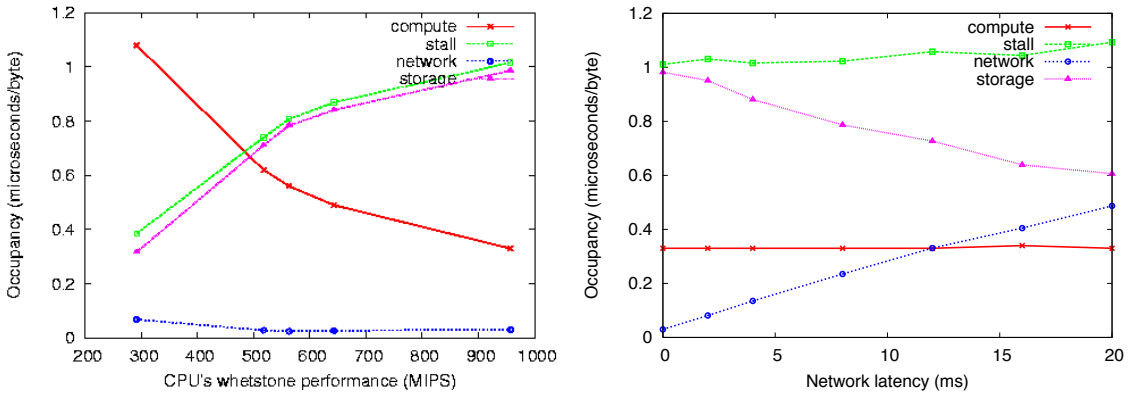

Figure 4.7: Impact of CPU speed and network latency on occupancies of GAMUT doing random file writes. Note that the storage resource is saturated and hence the storage occupancy changes with CPU speed as well as network latency.

high degree of parallelism.

NIMO learns the following predictor functions for this synthetic task. The mean percentage error is within 10% as shown in Figure 4.6. This suggests that our approach is reasonably robust with respect to queuing delays as well.

$$f_a = 0.019 + 0.59/\rho_{cpu\ speed} + 0.007 * \rho_{net\ lat}$$

$$f_n = -0.184 - 0.16/\rho_{cpu\ speed} + 0.42 * \rho_{net\ lat}$$

$$f_d = 1.04 + 0.34/\rho_{cpu\ speed} - 0.19 * \rho_{net\ lat}$$

Latency hiding and operating range

Figure 4.8 shows the impact of latency hiding in the case of a synthetic task doing pure sequential reads (GAMUT seq. read in Figure 4.6). The worst case percentage error for this task is 30%. We found that this error occurs because prefetching behavior of the file system hides the network latency up to a certain point, causing a non-linear impact on the network occupancy. This result highlights the importance of exposing the impact of entire operating range of any factor, e.g., network latency, on the application performance. Section 4.6 presents techniques that take into account

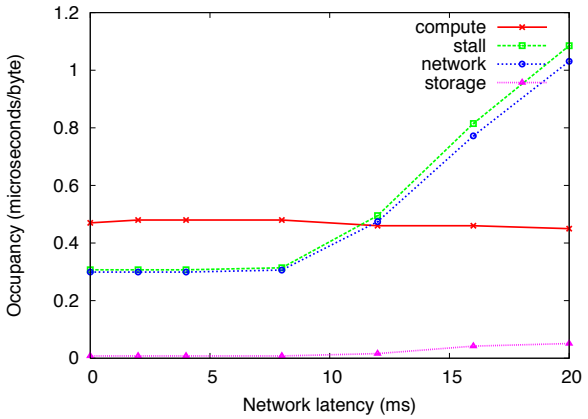

Figure 4.8: Impact of network latency on occupancies of GAMUT doing sequential file reads. Prefetching hides I/O latency up to a point, making a single linear function insufficient to predict network occupancy throughout the operating range.

the entire operating range of any given factor, and Section 4.9 evaluates them.

4.9 Experiment-Driven Learning of Models

The evaluation of experiment-driven learning has two goals: (i) to evaluate the different algorithmic choices introduced in Section 4.6; and (ii) to show that NIMO reduces the overall time to learn reasonably accurate models significantly . The experimental setup is similar to that of Section 4.8 with an addition of memory factor to the resource profile. Specifically, we use 5 CPU speeds, 5 memory sizes, and 6 network latencies, for a total of maximum of 150 candidate resource assignments for each batch task. For brevity, we use *BLAST* by for demonstrating the performance of our algorithms, and the results for other applications are similar.

The metric we use to report the current accuracy of a model M in our experiments is M's Mean Absolute Percentage Error (Section 4.6.6) in predicting total execution time on an *external test set* of 30 resource assignments chosen randomly from the workbench. Note that the external test set is different from the internal test set used by NIMO to compute the current prediction error (Section 4.6.6), and is never

Table 4.2: Choices for steps of Algorithm 5. * denotes the default in experiments unless otherwise noted

Step	Alternatives
Initialization	Min*, Rand, Max
Predictor refinement	Static + Round-Robin*, Static + Improvement-based, Dynamic
Factor addition	Relevance-based (PBDF)*, Static
Sample selection	L_{max}-I_1^*, L_2-I_2
Prediction error	Cross-Validation*, Fixed Test Set (Random), Fixed Test Set (PBDF)

exposed to NIMO for training or testing.

Experiment-driven learning of predictor functions depends on the choices made at the different steps of Algorithm 5 as explained in Section 4.6. This work evaluates various alternatives for each of the following five steps:

1. *Initialization*: The reference assignment that decides the starting point in the resource assignment search space (Section 4.6.1)

2. *Predictor refinement*: The order and traversal NIMO uses to refine the predictor functions (Section 4.6.2)

3. *Factor addition*: The order and traversal NIMO uses to add factors to each predictor function (Section 4.6.3)

4. *Sample selection*: The choice of value for each resource-profile factor to generate a new sample assignment for a run of the task (Sections 4.6.4 and 4.6.5)

5. *Prediction error*: The technique to compute the current prediction error at any point in time (Section 4.6.6)

While evaluating any of these 5 factors, we fix the choices for the other 4 factors to defaults as shown in Table 4.2.

4.9.1 Initialization

The reference assignment serves several purposes in NIMO: (i) starting sample assignment for the learning algorithm (Algorithm 5); (ii) baseline for normalizing training samples (Algorithm 10); and (iii) reference for setting factor values during sample selection (Algorithm 9). Different reference assignments may lead to completely different training samples, and hence, different MAPE statistics. We begin our experimental results with the evaluation of alternatives from Section 4.6.1 for choosing the reference assignment. Note that we fix the choice for each other step of Algorithm 5 to the default given in Table 4.2.

Figure 4.10 shows the impact of three alternatives for choosing the reference assignment on the overall accuracy and convergence time of the learned model for *BLAST*: (a) a randomly chosen assignment (*Rand*); (b) a high capacity assignment (*Max*); and (c) a low capacity assignment (*Min*). Each point in the figure corresponds to the MAPE when a new sample is added to the training data or a new factor is added to a predictor during learning.

We can make the following observations from Figure 4.10: (i) the plots start at different times; (ii) the MAPE values do not converge smoothly, e.g., there may be a sharp drop when a new training point is added; and (iii) while *Max* converges in the shortest time to a reasonably-accurate model, *Min* and *Rand* converge to models with lower errors. We explain these observations next.

Among the three alternatives, the reference assignment in *Max* has the maximum resource capacity, so it results in the shortest time to finish the first run and generate a training sample. Also, note that in the default L_{max}-I_1 strategy for sample selection, only one factor in any new sample assignment is set to a value different from the corresponding value in the reference assignment. Hence, *Max* will generate new training samples at a faster rate than *Min* or *Rand*.

The nonsmooth nature of the plots in Figure 4.10 is a consequence of NIMO's online exploration of the space of resource assignments to learn predictor functions with the right factors. The prediction errors may drop sharply, e.g., when a relevant factor is added to a predictor. Recall that the MAPE values in Figure 4.10 are based on an external test set that is never exposed to NIMO for training or testing.

Min and *Rand* converge to models with lower errors than *Max*. Our hypothesis is that the set of training samples produced when *Min* or *Rand* is used is more representative of the space of sample assignments than when *Max* is used. That is, *Min* and *Rand* may be leading to training sets that capture the operating range of relevant factors and the significant interactions among factors better.

4.9.2 Exploration Sequence for Predictors

The sequence in which NIMO explores predictor functions for refinement across iterations of Algorithm 5 determines the time to learn accurate models. In Figure 4.9 we evaluate the static and dynamic strategies from Section 4.6.2 for guiding predictor refinement through ordering and traversal. As usual, choices for the other steps are the defaults given in Table 4.2. The strategies we compare in Figure 4.9 are: (i) static order f_d, f_a, f_n + round-robin traversal; (ii) static order f_d, f_a, f_n + improvement-based traversal; and (iii) dynamic ordering and traversal.

The main observations from Figure 4.9 are: (i) round-robin traversal performs better than improvement-based traversal for static ordering; and (ii) the dynamic strategy takes the longest to converge and shows the most nonsmooth behavior.

Improvement-based traversal of predictors is sensitive to the order in which the predictors are refined as well as the improvement threshold used (Section 4.6.2). In Figure 4.9, the static order is the nonoptimal f_d, f_a, f_n order—the actual relevance order computed using PBDF is f_n, f_a, f_d—and the improvement threshold is 2%—i.e.,

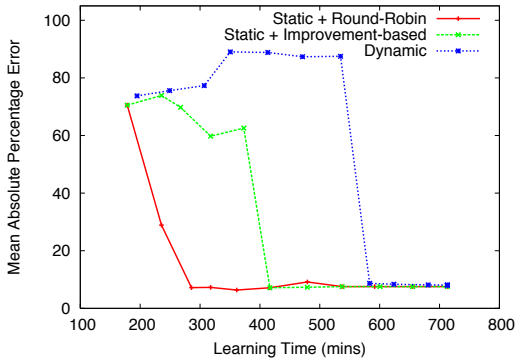

Figure 4.9: Impact of different alternatives for refining the predictor function (*BLAST* application).

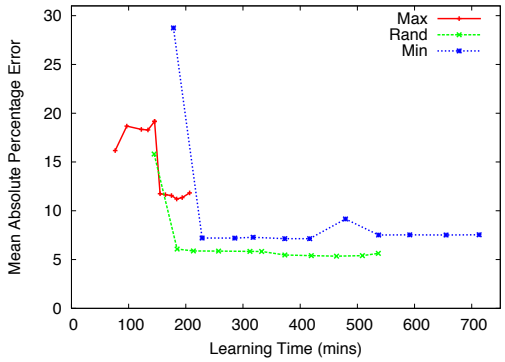

Figure 4.10: Impact of different alternatives for choosing the reference assignment (*BLAST* application).

we move to the next predictor in the order when the reduction in prediction error with the current predictor falls below 2%. The MAPE of the improvement-based strategy remains high until f_n starts being refined (around 400 minutes), when it drops sharply. On changing the static order to f_n, f_a, f_d, the improvement-based strategy learns an accurate model quickly (as shown by the *Min* plot in Figure 4.10). Round-robin traversal of the static order acquires samples for each predictor in turn, so it is less sensitive to the correctness of the order or the threshold.

The accuracy-driven dynamic strategy performs the worst in Figure 4.9. Recall from Section 4.6.2 that the dynamic strategy chooses to refine the predictor with the maximum current prediction error. In Figure 4.9, the dynamic strategy gets stuck initially in a local minima where it keeps refining f_a until all samples for the factors in f_a are exhausted, and f_n starts being refined (around 550 minutes). The problem with the dynamic strategy is that the current prediction error of a predictor f is not representative of f's relevance to the total task execution time.

4.9.3 Adding New Factors to Predictors

Accurate learning of predictor functions can happen only when relevant factors are added quickly to the functions. Recall from Section 4.6.4 that the factors in the

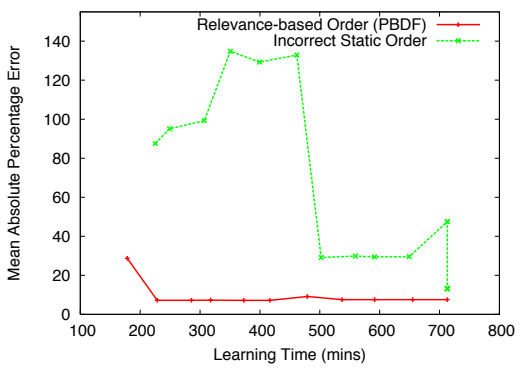

Figure 4.11: Impact of alternatives for adding new factors to a predictor function (*BLAST* application).

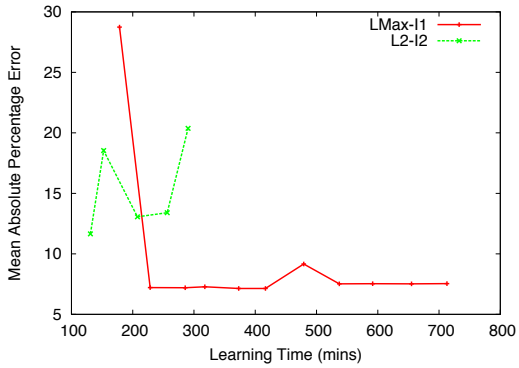

Figure 4.12: Impact of alternatives for selecting new sample assignments (*BLAST* application).

predictor function currently chosen for refinement dictate which sample assignments are selected for training. Our next experimental results show that adding factors to predictors in an incorrect order can delay convergence to accurate models. We consider the two alternatives from Section 4.6.3 for determining the order in which factors are added to predictors:

- Relevance-based ordering that determines relevant factors and their order using PBDF as: (i) f_a—*cpu speed*, *memory size*, (ii) f_n—*network latency*, *memory size*, and (iii) f_d—*network latency*.
- Static ordering set as: (i) f_a—*network latency*, *memory size*, *cpu speed*, (ii) f_n—*cpu speed*, *memory size*, *network latency*, and (iii) f_d—*cpu speed*, *memory size*, *network latency*. The static ordering is kept different from the relevance-based ordering to show the importance of adding factors in the right order.

Figure 4.11 compares the two alternatives. While the relevance-based order learns an accurate model quickly, the incorrect static order causes nonsmooth behavior and slow convergence.

4.9.4 Selecting New Sample Assignments

A good sample-selection strategy must cover the operating range of relevant factors and expose all significant interactions among factors while acquiring only a small number of samples. We evaluate two strategies from Section 4.6.4 for sampling new assignments: L_{max}-I_1 and L_2-I_2. Recall that the L_{max}-I_1 strategy covers the operating range of relevant factors, but it may fail to expose significant interactions among factors. The L_2-I_2 strategy adds training samples one at a time from the design matrix specified by PBDF. L_2-I_2 considers only two levels from the operating range of each factor, but it can expose significant two-way interactions among factors.

Figure 4.12 compares the two alternatives. Here we observe that L_{max}-I_1 converges quickly to an accurate model, while L_2-I_2 fails to converge. Our hypothesis is that the simple L_{max}-I_1 strategy is enough to expose any significant interactions among factors for *BLAST*. On the other hand, with only two levels considered for each factor, L_2-I_2 fails to obtain good regression functions for the predictors. We are now exploring sampling schemes that can guarantee the capture of significant interactions among factors and also provide good coverage of the operating range of relevant factors.

4.9.5 Computing Current Prediction Error

An important component of NIMO's accelerated learning algorithm is the computation of current prediction error for each predictor. This error is used by other steps of Algorithm 5, e.g., by the improvement-based traversal and the dynamic strategy for choosing the predictor function to refine. We consider the two strategies from Section 4.6.6 for computing the prediction error: (i) leave-one-out cross-validation using all samples collected so far; and (ii) using a fixed internal test set. The fixed test set is chosen in two ways: (a) a set of 10 assignments chosen randomly from

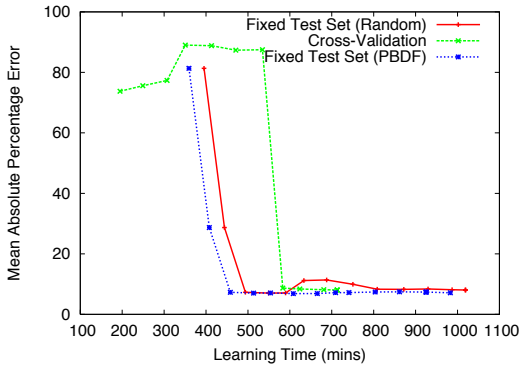

Figure 4.13: Impact of alternatives for computing the current prediction error (*BLAST* application).

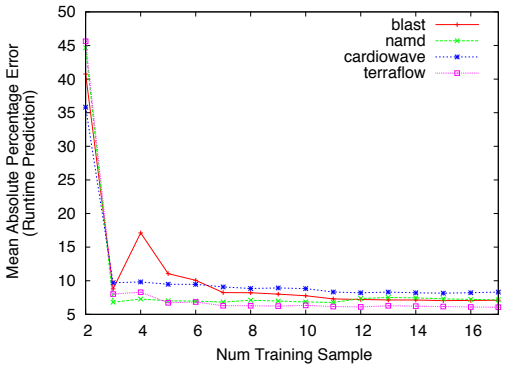

Figure 4.14: Accuracy of NIMO models based on samples attained on virtual machines.

the space of possible assignments; and (b) a set of 8 assignments chosen from the samples specified by PBDF. The results are shown in Figure 4.13. Here, we use the accuracy-driven dynamic strategy for refining the predictor functions to study the impact of internal test sets on MAPE. All other factors are set to their defaults as shown in Table 4.2.

Figure 4.13 shows the strengths and weaknesses of the approaches. Compared to fixed test sets, cross-validation starts producing results earlier, but it shows non-smooth behavior and slow convergence. Cross-validation produces its initial error estimates from the very few samples collected so far, causing the observed nonsmooth behavior. However, these estimates get more accurate over the course of experiment-driven learning as more samples are collected. The fixed test set approach requires an upfront investment of time to obtain the test samples, which delays the start of the learning process. However, fixed test sets give more robust estimates of prediction error because these sets are representative of the total sample space in terms of capturing operating ranges and factor interactions.

4.9.6 Experiment-Driven Learning on Virtual Machines

Virtual machine technology promises important benefits for grid computing and cluster batch job systems, including improved isolation, customizable workspaces, and support for checkpointing and migration [49]. Figure 4.14 shows the accuracy of the models that NIMO learns based on samples attained on Xen virtual machines [33]. The models predict the runtime of an application as a function of CPU, and memory shares assigned to it. 10 CPU $\times$ 9 memory configurations comprise the total number of virtual resource assignments. NIMO uses Algorithm 5 to collect the relevant training samples by executing the application on a few virtual resource assignments proactively. The accuracy of the model is evaluated on all virtual resource assignments that are not used for training. The figure shows that NIMO can learn reasonably accurate performance models with a few training sample runs on virtual machines as well.

4.9.7 Summary of Experimental Results

We apply the experiment-driven methodology for learning models for three real biomedical applications other than *BLAST*. Table 4.3 shows the time to learn an accurate model and the corresponding MAPE values for all four applications. The table shows that as the factor space gets larger, NIMO reduces the time to learn reasonably accurate models by an order of magnitude compared to approaches that first sample a significant part of the entire space and then build models all-at-once (the active sampling without acceleration strategy in Figure 1.4).

We summarize the results of our experiments evaluating the algorithmic choices presented in Section 4.6:

- The *Min* approach tends to select reference assignments that produce training sets that are representative of the total sample space.

Table 4.3: Gains from experiment-driven learning

Appl.	#Attrs	MAPE	NIMO's Learning Time (hrs)	Learning Time for All Samples (hrs)	Sample Space Used (%)
BLAST	3	10	12	130	3
fMRI	3	10	4	112	3
NAMD	2	4	2	16	5
C. Wave	2	10	2	16	5

- Unlike the improvement-based strategy, round-robin traversal is not sensitive to the (static) ordering of predictors, nor does it require a predefined threshold. Round-robin traversal also avoids the local-optima problem of dynamic approaches that are based on current prediction error.

- Adding factors in relevance order based on PBDF is a good approach for adding new factors to predictors. Other factor orders may significantly delay convergence to accurate models.

- In our experiments, the L_{max}-I_1 strategy for selecting new sample assignments performs better than L_2-I_2 mainly because of the limited factor operating range considered by L_2-I_2.

- A fixed internal test set, chosen randomly or using PBDF, is a reasonable choice for computing current prediction error. Cross-validation-based approaches show nonsmooth behavior and slow convergence.

4.10 Model-Guided Planning in Utilities

The primary goal of NIMO is to guide autonomic resource management in a networked utility setting. This section illustrates how an autonomic utility can use the models induced by the NIMO system for several planning scenarios.

Table 4.4: Candidate assignments for *fMRI*.

Candidate Assgs.	CPU speed	Network latency (ms)
A_1	996 MHz	4
A_2	797 MHz	4
A_3	1396 MHz	10
A_4	451 MHz	2

Table 4.5: Comparing assignment choices.

Choice of Assg.	Performance
Actual best = A_1	Completion time = 16.67 min
Model-predicted = A_1	Predicted time = 17.46 min
Fastest CPU = A_3	23% slower than A_1
Fastest network = A_4	50% slower than A_1

4.10.1 Selecting Task Placement

A utility resource manager can evaluate the models directly to compare the predicted performance of a set of competing candidate resource assignments. For example, the utility can predict which combination of CPU and I/O resources will yield the shortest completion time for a batch task.

Table 4.5 compares the measured performance of *fMRI* [55] runs on four candidate assignments A_1–A_4 shown in Table 4.4. It shows the predicted and measured completion times for the candidate (A_1) preferred by an induced model, and compares them to the candidates chosen by two simple alternative strategies: (i) select the assignment with the fastest CPU clock; and (ii) select the assignment with the lowest latency to network storage. Since the delivered performance depends on the combination of CPU and I/O resources, the naive approaches cannot identify the best candidate. The model captures the relative importance of the different factors alone and in combination for each application, so it can guide the choice of the best candidate.

4.10.2 On-Time Computing

A resource manager can also use the models to search for a candidate assignment that meets a specified completion time target t for a task or a sequence of tasks, such as the critical path of a complex workflow graph. For a chain of n tasks, eligible candidates must satisfy the inequality:

$$\sum_{i=1}^{n} D_i(o_{a_i} + o_{n_i} + o_{d_i}) \leq t$$

Identifying eligible candidates is a heuristic search problem in the general case, but it is often possible to solve directly for factor values that yield target occupancies for each resource, i.e., when the occupancy is driven by continuously valued factors such as CPU clock speed. Figure 4.15 plots occupancies for candidate assignments for an *fMRI* instance with a completion time target of $t = 20$ minutes. The figure shows the subset of candidates that meet the target completion time, the set that does not meet the target, and also the model-predicted "boundary" between these two. For each occupancy value of o_a and o_n, the resource manager can obtain the profile of the corresponding resources using the predictor functions f_a and f_n (Section 4.6).

4.10.3 Storage Outsourcing and Data Staging

A utility must estimate the performance impact of remote I/O and the benefits of application migration or local staging. To evaluate the potential of model-guided planning in this setting, we applied our approach to data published in a recent empirical study of storage outsourcing [81]. The study investigates the viability of storage outsourcing by measuring the impact of remote data placement, caching, and local data staging under various workloads with varying network latencies to the storage site. We induce performance models for two synthetic benchmarks—PostMark [61]

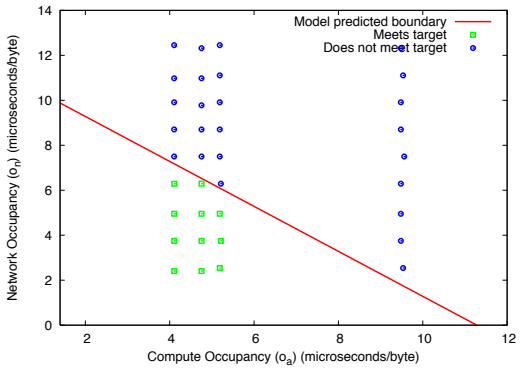

Figure 4.15: Model-predicted boundary separating assignments that meet and those that do not meet a target runtime for *fMRI*. Higher values of o_a and o_n indicate slower CPU and farther storage respectively.

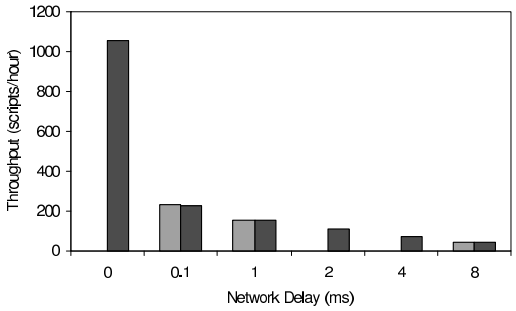

Figure 4.16: Accurate prediction of results from an empirical study of storage outsourcing. We parameterize the model using three configurations (single bars), and predict the throughput for the remaining ones (double bars). The maximum error in prediction is 10%.

and SPEC-SDET [42]—by taking data published for three remote storage configurations as training data, and then use the models to predict the results for the other configurations. The worst case percentage error (PE) was 6% and 10% for PostMark and SPEC-SDET respectively. Figure 4.16 shows the results for SPEC-SDET. We conclude that our approach has excellent potential to capture the phenomena explored in this comprehensive empirical study.

A utility may use the models to evaluate remote storage and data staging alternatives in conjunction with task placement. Data staging options—in which the candidate plan copies an input or output data set between sites before or after task execution—are modeled as one or more additional stages inserted into a task graph. The predicted completion time is the sum of the predicted times for the data staging steps and task execution.

Table 4.6 shows sample candidate assignments for *fMRI* involving both remote I/O and local I/O with data staging for the input dataset. The table shows that neither data staging nor remote I/O is always preferred. The models identify the best alternative and predict the overall completion time accurately even when data

staging is involved.

Table 4.6: Candidate assignments for *fMRI* with and without input data staging; the preferred choice of each pair is shown in bold.

Candidate Assg.	CPU speed (MHz)	Network latency (ms)	Data staging done?	Actual time (mins)	Model Predicted (mins)
1.1	451	14	Yes	54.47	53.48
1.2	451	14	**No**	35.16	37.65
2.1	996	14	**Yes**	23.43	22.78
2.2	996	14	No	28.25	26.71
3.1	996	2	Yes	24.63	24.44
3.2	996	2	**No**	14.58	15.40
4.1	451	16	**Yes**	28.21	25.51
4.2	451	16	No	37.43	39.81

4.11 Related Work

Much of the previous work on model-based provisioning and placement focuses on online Internet services, which are driven by the arrival patterns of requests and queuing behavior [31, 105, 112, 113]. In contrast, we focus on compute batch tasks that run to completion at machine speed, so we do not need to model request arrivals. Hippodrome [7] uses detailed performance models and an optimizing planner to assign storage resources in a shared utility. Hippodrome emphasizes storage performance and modeling of contention; our approach is complementary in that it addresses the interaction of computation and storage access and their impact on end-to-end application performance.

Several previous studies support the potential of statistical techniques to capture application performance behavior accurately. For example, two early surveys ([74, 88]) found that a large class of scientific applications exhibit marked regularity in CPU usage and I/O activity over the execution interval. Some applications show data-

dependent behavior—their resource usage depends on parameters or the contents of the input data—and several groups are exploring how to map such dependencies when they exist, e.g., [108, 118, 34]. Although this work does not consider data-dependent behavior, NIMO's data profiles can represent input data characteristics that affect application resource demands; these complementary projects may help to understand the most relevant characteristics and how to capture them.

Accurate prediction of completion time is a prerequisite for many provisioning and scheduling strategies, e.g., [21, 89]. AppLeS [21] is a grid scheduling framework that uses estimates of computation and file transfer times to decide task placement. Menasce et al. [14] use queuing analysis to predict the throughput of a stream of batch tasks competing for homogeneous shared resources, assuming the behavior of individual tasks is known. A number of systems approximate NP-hard optimal scheduling assignments given estimates of performance or utility, e.g., [65]. All these systems can benefit from our work in predicting the completion times of individual tasks on heterogeneous resources.

Some recent prediction work instruments the program source or its binary, e.g., [19], or assumes knowledge of the program internals. For example, Rosti et al. [95] instrument the source code of parallel applications to derive stochastic prediction models. In contrast, we rely only on noninvasive instrumentation, so our model predictions may be less accurate. [119] provides a black-box approach to predict completion time on heterogeneous platforms, but the approach requires a partial execution of each candidate application on each target system.

Remote storage access is a key barrier to harnessing remote computing resources effectively [81, 46]. Our approach models the impact of storage placement and access costs on end-to-end performance, as a basis for intelligent placement of tasks and data. Wolski et al. [37] evaluate the impact of various data-staging strategies on

wide-area task farming, and show that staging overhead can often be overlapped with computation. The models in this work are conservative with respect to this phenomenon, but could be extended to account for it.

Various mechanisms exist to realize a range of choices for data staging and task placement in networked systems. BAD-FS [15] assigns work to compute servers and storage servers to maximize the benefit of local caching and buffering. Logistical Networking [13] addresses the global scheduling of data movement and computation. Stork [68] is a batch scheduler that schedules data migration tasks as first-class citizens alongside computation. Abacus [4] uses analytical models to drive dynamic task placement for data-intensive cluster applications. Our work gives a model-guided approach to select among candidate task and data placements in such systems, and to induce the models automatically.

Database technology is well-suited to handle many aspects of workflow management as evident from the number of *WorkFlow Management Systems (WFMSs)* [101]—e.g., Griddb [70], GriPhyn [48], Kepler [2], and Zoo [56]—that exist to provide functionality such as modeling, execution, provenance, auditing, and visualization for workflows. Workflow planning is similar to query optimization in database systems, but it poses an entirely new set of challenges. A task in a workflow G is typically a script in a programming language like Perl or Matlab. Hence, a WFMS usually has no prior knowledge about G's resource usage characteristics, or its performance sensitivity to the diverse hardware platforms comprising the underlying networked utility. Consequently, G is a *black-box* to the WFMS, making it challenging to generate an accurate cost model for G.

Traditional work on cost modeling in relational databases, e.g., [50] assume that the execution plans are composed of operators belonging to a small well-defined family of operators. This assumption does not hold for scientific workflows. Statistical

learning methods have been used to develop cost models for: (i) complex user-defined functions, e.g., [52]; (ii) remote autonomous database systems in the multidatabase setting, e.g., [124]; and (iii) complex XML operators [123]. The general approach is to first identify a set of query and data features that potentially determine operator costs, and then to use given training data to learn the relationship among the values of the identified features and operator cost. NIMO differs from this category of work in two ways: (i) it addresses the problem of automatically acquiring the right training data to minimize the overall learning time; and (ii) it considers an application as a black-box and relies only on noninvasive measurement streams to make our work more widely applicable.

4.12 Conclusions and Future Work

This chapter presents the NIMO system that uses the experiment-driven framework to learn models for predicting the execution time of batch applications running on large-scale networked utilities. NIMO is noninvasive in that it uses training data from passive instrumentation streams collected using common profiling tools, requiring no changes to the operating system or applications. Our experimental results indicate that NIMO can learn fairly-accurate models quickly for real scientific applications.

There are many avenues for future work. To enable a fully automated model-learning, NIMO needs an algorithm that can automatically select the best combination of choices for each step of Algorithm 5 for any given application. Moreover, to handle an application whose resource usage is highly data dependent, NIMO needs to capture the data dependency using factors in the data profile of the application's input data. Identifying the right set of factors in the data profile for a black-box application remains an interesting open problem.

Chapter 5

Automated Server Benchmarking

"For better or worse, benchmarks shape a field."

-David Patterson

Systems researchers and developers devote a lot of time and resources to running benchmarks to gain insight into the performance impacts and interactions of system design and management choices, and workload characteristics. In the marketplace, benchmarks are used to evaluate competing products and candidate configurations for a target workload. This chapter presents the use of the experiment-driven framework from Chapter 2 for automated server benchmarking, in particular storage server benchmarking.

5.1 Background

Server benchmarking isolates the performance effects of choices in server design and configuration, since it subjects the server to a steady offered load independent of its response time. It reliably stresses the system under test to its saturation point where interesting performance behaviors may appear. Although application benchmarks are commonly used in systems research, they do not establish the bounds of the system's operating range or its behavior under stress. In the storage arena, NFS server benchmarking is a powerful tool for investigation of system behavior at all layers of the storage stack. A workload mix can be selected to stress any part of the system, e.g., the buffering/caching system, file system, or disk system. By varying the components alone or in combination, it is possible to focus on a particular component in the storage stack, or to explore the interaction of choices across the components.

For example, the SPEC SFS benchmark and its predecessors [64] have been in use for decades to establish NFSOPS ratings for network file servers and filer appliances using the NFS protocol. SPEC SFS is a *server benchmark*: client load generators subject the server under test to a request mix offered at some arrival rate (or test load) over a test interval to obtain an aggregate measure of the server's response time at that load level. To obtain an NFSOPS rating, the benchmark runs a sequence of trials at varying load levels to identify the point at which the response time measures begin to exceed specified maximum thresholds with a standard request mix. The NFSOPS rating is the rate of request completions at that load level, representing the *saturation throughput* or *peak rate* that the server can process. A similar methodology is used in other industry-standard server benchmarks, such as the TPC transaction processing benchmarks [27].

The work in this chapter presents the application of experiment-driven framework for storage server benchmarking. The objective is to devise policies for the automated *workbench controller* in Figure 2.1 that can obtain peak rate measures with a target confidence level and accuracy at low cost. In particular, the policies support systematic *response surface mapping* that plots the peak rate over a space of workloads and/or system management and design choices. Figure 5.1 gives an example of response surface mapping using the peak rate. The example is discussed in Section 5.2.

A large number of factors can affect storage server performance. Section 1.3 discusses the importance of sampling multi-dimensional space with care. In addition, it is crucial to optimize the "inner loop" of response surface mapping: the search for the peak rate for each sample point in the space. For example, suppose we are mapping the impact of five factors on a file server's peak rate, and that we sample five values for each factor. If the benchmarking process takes an hour to find the

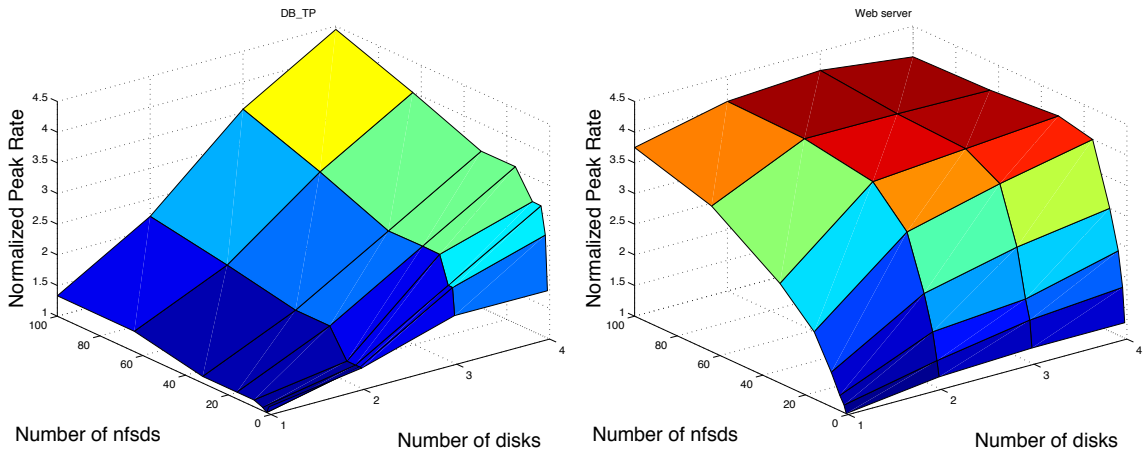

Figure 5.1: Surfaces that depict how the peak rate, λ^*, changes with number of disks and number of NFS daemon (nfsd) threads for two *Fstress* workloads (**DB_TP** and **Web server**).

peak rate for each factor combination, then the total time for benchmarking is 130 days. The benchmarking automation policies can shorten this time by pruning the sample space, planning experiments to run on multiple hardware setups in parallel, and optimizing the inner loop.

This chapter presents techniques that optimize the inner loop by searching for the peak rate in an efficient way that balances cost, accuracy, and confidence for the results of each test load, while meeting target levels of confidence and accuracy to ensure statistically rigorous final results. It also shows how the controller can use heuristics and established response surface methodology to prune the multi-dimensional sample space.

5.2 Overview

The performance of a storage server is a function of its workload, its configuration, and the hardware resources allocated to it. Each of these may be characterized by a vector of factors, as summarized in Table 5.1.

Table 5.1: Example of factors that affect storage server performance.

$\vec{W}$	read/write ratio, random/sequential ratio, metadata/data ratio, dataset size, file size distribution, directory structure, request mix
$\vec{R}$	CPU speed, memory size, number of disks
$\vec{C}$	Number of I/O daemons, type of file system, block size

Performance $\vec{P}$. This work characterizes the benchmark performance of a storage server by its *peak rate* or *saturation throughput*, denoted λ^*. λ^* is the highest request arrival rate λ that does not drive the server into a *saturation state*. The server is said to be in a saturation state if a response time metric exceeds a specified threshold. In this work, saturation occurs when either of two conditions hold: (i) the mean response time of the server, which is the aggregate server response time of client requests over some time interval, exceeds $> R_{sat} = 40$ ms, or (ii) the 95-percentile server response time exceeds a specified threshold latency $L_{sat} = 2000$ ms.

Workload $\vec{W}$. In this work, the workbench controller uses a configurable synthetic workload generator called *Fstress* [5] to explore a wide range of NFS workloads defined by various workload factors. Fstress offers knobs for the controller to configure the properties of the workload's dataset and its request mix, and preconfigured parameter sets intended to be representative of workloads encountered in practice; see Table 5.3.

Resources $\vec{R}$. The controller can vary the amount of hardware resources assigned to the system under test, depending on the capabilities of the workbench testbed. Our prototype can instantiate Xen virtual machines sized along the memory, CPU, and I/O dimensions. Experiments for this work use a small number of physical configurations in the workbench.

Configurations $\vec{C}$. To benchmark the storage server across a range of storage server configurations such as the file system type and block size, the controller modifies the relevant properties of the server configuration files before creating the server file system.

5.2.1 Storage Server Benchmarking

Figure 5.1 shows an example of a benchmarking result produced by the automated workbench. It shows the peak rate for different combination of NFS server daemons (nfsds) and disk spindles for two different Fstress workloads: a database workload (**DB_TP**) and a Web server (**Web server**) workload. Such a result is called a *response surface*: it gives the response of a metric (peak rate) to changes in the operating range of the factors in a system [78].

Knowledge of response surfaces is crucial for understanding the performance tradeoffs of adding resources and/or changing configurations for different workloads. For example, the figure shows that adding more disks can improve the peak rate only if there is a sufficient number of nfsds to issue requests to those disks, and that the appropriate number of nfsds is workload-dependent.

Algorithm 11 presents the overall benchmarking approach that is used by the workbench controller to map a response surface. Table 5.2 shows the measures that are relevant to server benchmarking. The overall approach consists of an outer loop that iterates over the samples in $\langle F_1, \ldots, F_n \rangle$, where $F_1, \ldots, F_n$ is a subset of factors in the larger $\langle \vec{W}, \vec{R}, \vec{C} \rangle$ space (Step 2). The inner loop (Step 3) finds the peak rate λ^* for each sample by generating a series of test loads for the sample. To find the peak rate, the controller must choose: (a) the test load λ at which to conduct the experiment; (b) the *runlength* r, which is the test interval over which to observe the server response time while the workload is running against the server at load λ; and

(c) the *number of independent trials* t with load λ.

Given the inherent variability in a storage system as well as in the experimental process, it is impossible to compute the true value of mean server response time for a given setting of the factors of interest. Figure 5.3 illustrates the variability in the mean server response times from multiple independent trials for a range of test loads. The best that can be done in this setting is to make a *probabilistic claim* about the *interval* in which the mean response time lies based on the mean response time measurements from multiple independent trials [59]. For example, by observing the mean response time at a test load λ for 10 independent trials, we may be able to claim that we are 95% confident that the mean server response time lies within the range $[25ms, 30ms]$.

Such a probabilistic claim can be characterized by a *confidence level*, and the *confidence interval* at this confidence level. In the example above, $[25, 30]$ represents the confidence interval at a 95% confidence level. For t trials of a workload at test load λ we compute the confidence interval as follows:

- Compute the mean server response time: $\mu = \sum_{i=1}^{t} R_i/t$, where R_i is the server response time for the i^{th} trial. Note that the mean server response time from t trials is a mean of the aggregate server response time at each trial.

- Compute the standard deviation for the server response time:

  $\sigma = \sqrt{\sum_{i=1}^{t}(R_i - \mu)^2/(t-1)}$.

- Confidence interval for the response time at confidence $100c\%$ is given as:

$$[\mu - z_p\sigma/\sqrt{t}, \quad \mu + z_p\sigma/\sqrt{t}] \tag{5.1}$$

  where $p = (1+c)/2$, and z_p is the quantile of the unit normal distribution at p. If the number of trials $t <= 30$, we replace z_p by $t_{p;t-1}$, which is the p-quantile

of a t-variate with $t-1$ degrees of freedom, assuming that the response time values from t trials come from a normal distribution. We verify that response times do come from a normal distribution using a normal probability plot.

Intuitively, a higher value of the confidence level implies that one is more certain of the interval in which the mean server response time lies. Similarly, a tighter bound on the confidence interval implies that the mean response time computed from the repeat observations is close to its true value. Hence, the tightness of the confidence interval captures the *accuracy* of the true value of mean response time. If the interval is narrow, then the accuracy is high, and if the interval is wide, the accuracy is low. For a confidence interval $[low, high]$, this work computes the percentage accuracy as:

$$accuracy = 1 - error = (1 - \frac{high - low}{high + low}) \tag{5.2}$$

5.2.2 Problem Statement

In this work, the goal of the automated feedback-driven controller is to address the following problems.

1. **Find Peak Rate**. For a given sample from the outer loop of Algorithm 11, minimize the benchmarking cost for finding the peak rate λ^* subject to a target confidence level c and target accuracy a. Determining the NFSOPS rating of an NFS filer is one instance of this problem.

2. **Map Response Surface**. Minimize the total benchmarking cost for mapping a response surface for all $\langle F_1, \ldots, F_n \rangle$ samples in the outer loop of Algorithm 11.

Minimizing benchmarking cost involves choosing values carefully for the runlength r, the number of trials t, and test loads λ so that the controller converges quickly

Table 5.2: Server benchmarking parameters that determine the benchmarking cost and accuracy.

λ^*	Peak rate of the server.
R_{sat}	Mean server response time threshold at peak rate.
s	Factor that determines the width of the peak-rate region $[R_{sat} \pm sR_{sat}]$ (see Section 5.4.1).
P_{sat} L_{sat}	The threshold on the percentage of requests that must complete under a specified threshold latency, L_{sat}.
a	Target *accuracy* (based on confidence interval width) for the estimated value of R_{sat}.
c	Target *confidence level* for the estimated R_{sat}.
r	Runlength of each trial of the workload at a test load.
t	Number of independent trials at a test load.
ρ	Load factor $= \lambda/\lambda^*$ where λ is a test load.
l	Number of test loads run before converging to λ^* with desired accuracy and confidence level.

to the peak rate. Sections 5.3 and 5.5 present algorithms that the controller uses to address these problems. Section 5.6 evaluates the algorithms to demonstrate that the controller maps the response surfaces efficiently, and the cost of finding the peak rate as well as mapping the surface adapts to the target confidence and accuracy specified by the storage server administrator.

5.3 Finding the Peak Rate

In the inner loop of Algorithm 11, the automated controller searches for the peak rate λ^* for a sample in $\langle F_1, \ldots, F_n \rangle$ by subjecting the server to a sequence of test loads. A principled approach for the inner loop must converge quickly and efficiently to an estimate of λ^* that meets the target accuracy and confidence. A *strawman* approach proceeds as follows (the notation is from Table 5.2):

- **Runlength** r. Use a *fixed* r for each test load.
- **Number of trials** t. Use a *fixed* t for each test load.

Algorithm 11: Mapping Response Surfaces

1) **Inputs**: (a) $\langle F_1, \ldots, F_n \rangle$, which is the subset of factors of interest from the full set of factors in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$; (b) Different possible settings of each factor;

2) *// Outer Loop: Map Response Surface.*
foreach *distinct sample* $\langle F_1 = f_1, \ldots, F_n = f_n \rangle$
do

3) *// Inner Loop: Find Peak Rate for the Sample.*
Design a sequence of test loads $[\lambda_1, \ldots, \lambda_l]$ to search for the peak rate λ^*;

foreach *test load* $\lambda \in [\lambda_1, \ldots, \lambda_l]$ **do**
Choose number of independent trials t for λ;
Choose runlength r for each trial;
Do t independent runs of length r each, with workload generated at load λ;

end
Set $\lambda^* = \lambda$, where $\lambda \in [\lambda_1, \ldots, \lambda_l]$ is the largest load that does not take the server to the saturation state;

end

- **Sequence of test loads** $[\lambda_1, \ldots, \lambda_l]$. Start at a default value, and use a *linear* increasing sequence of test loads λ where each load differs from the previous one by a small fixed increment. Stop when the current test load saturates the file server.

This section shows that the strawman can be highly inefficient in terms of the benchmarking cost and may lead to an inaccurate estimate of λ^*. However, the strawman serves as a good illustration of the insights behind the improved algorithm that Section 5.4 presents.

Sequence of test loads. The number of loads, l, in the strawman depend on the increment size used to get successive values of the test load. Since the peak rate can vary significantly not just across workloads, but even for the same workload as shown in Figure 5.1, choosing a good value of the increment becomes more of an art.

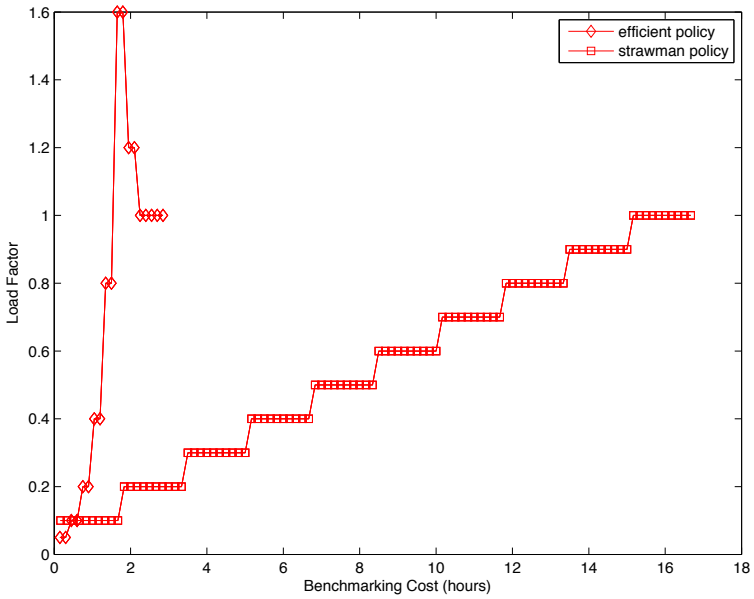

Figure 5.2: An efficient policy for finding peak rate converges quickly to a load factor near 1, and reduces benchmarking cost by obtaining a high-confidence result only for the load factor of 1. It is significantly less costly than a simple linear search with a fixed runlength, and fixed number of trials per test load (e.g., SPECsfs [26]).

Figure 5.2 illustrates the search for λ^* using the strawman approach for runlength $r = 5$ minutes, number of trials $t = 10$ trials, and a small increment in test load so that the search for λ^* yields an accurate result. This work represents the sequence of test loads by load factor, $\rho = \frac{\lambda}{\lambda^*}$. At load factor of 1, $\lambda = \lambda^*$, and the search for the peak rate terminates. The figure compares the strawman to an efficient search technique for finding the peak rate.

The figure shows that a linear increase in load factor can incur a much higher benchmarking cost to converge to the peak rate. The strawman not only considers a large number of load factors that are not close to 1, but also incurs the same benchmarking cost at all load factors. Since the goal is to find the peak rate, an efficient approach must quickly eliminate load factors that are not close to 1, and incur most of the benchmarking cost near $\rho = 1$; Sections 5.4 and 5.5 present such approaches.

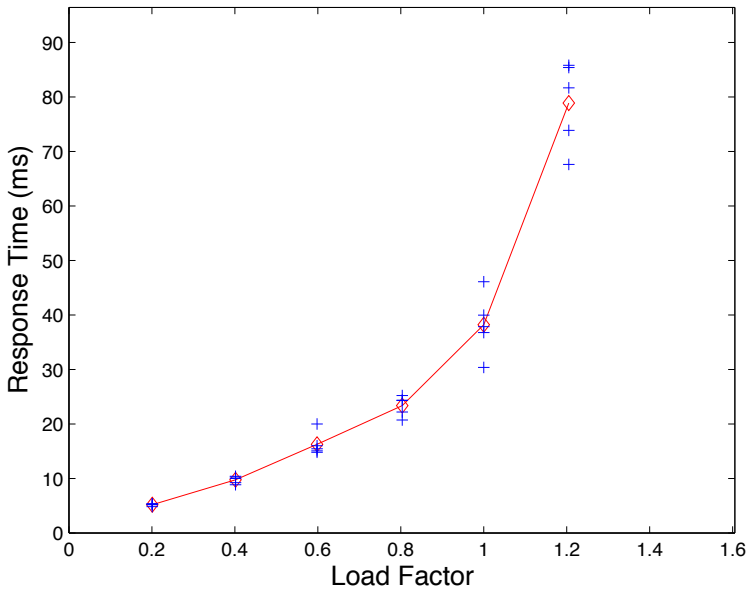

Figure 5.3: Mean server response time at different test loads for the **DB_TP** *Fstress* workload using 1 disk and 4 NFS daemon (nfsd) threads for the server. The variability in mean server response time for multiple trials increases with load. The results are representative of other server configurations and workloads.

Number of trials. The runlength r and the number of independent trials t for each test load determine the benchmarking cost incurred at that load. Figure 5.3 shows a scatter plot of mean server response time at different test loads for 5 trials at each load. Note that the variability across multiple trials increases with load, and that multiple trials are essential at high loads to get reasonable estimates. The figure shows that that the number of trials, t, must adapt to the choice of the test load in the search process. Ideally, t must be high near a load factor of 1, and low otherwise. For the strawman approach, 10 trials may be too many at low load factors and too little near $\rho = 1$, depending on the corresponding variability of response time.

Runlength. Figure 5.4 shows the scatter plot of mean server response times at two load factors, $\rho = 0.3$ and $\rho = 0.9$. The figure plots the mean server response times against different runlengths of the workload. It shows that the variability in mean server response time for multiple trials decreases with increase in an experiment's

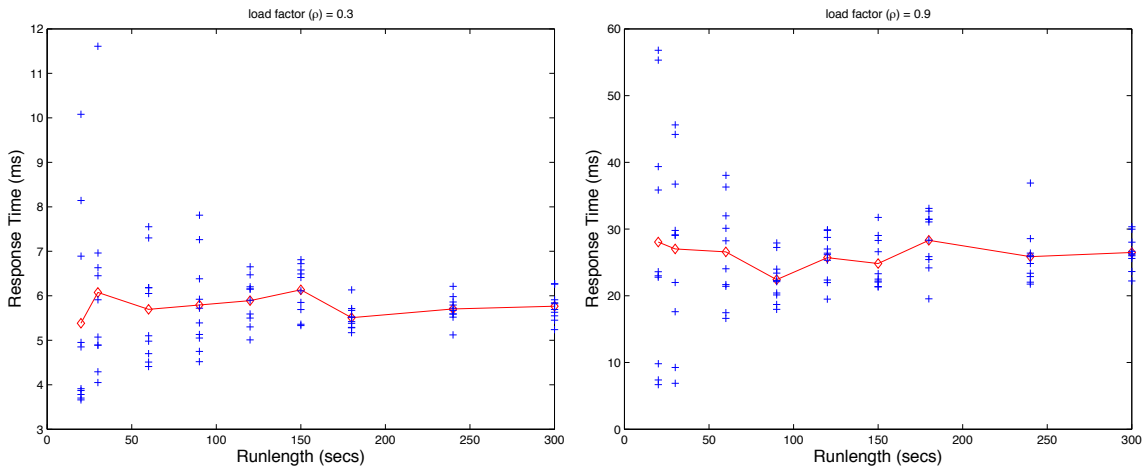

Figure 5.4: Mean server response time at different workload runlengths for the **DB_TP** *Fstress* workload using 1 disk and 4 NFS daemon (nfsd) threads for the server. The variability in mean server response time for multiple trials decreases with increase in runlength. The results are representative of other server configurations and workloads.

runlength. Hence, when the workload is run for a short duration, more trials are needed to obtain an accurate measure of mean server response time; and vice versa. Also, for the same runlength, the variability is higher at higher load factors, especially when the runlength is small. Thus, for the strawman approach, with 10 trials per test load, $r = 5$ minutes may be too high at low load factors.

5.3.1 Choosing the Runlengths and Number of Trials to Meet Target Confidence and Accuracy

An automated approach for finding the peak rate must automatically determine the choice of t and r for each test load λ, while adapting to: (a) the value of the load factor, and (b) the target confidence and accuracy of the estimated peak rate λ^*. The goal is to converge quickly to $\rho = 1$. Higher number of trials and longer runlengths are useful near a load factor of 1, but must be minimal at other load factors.

From Equations 5.1 and 5.2 for confidence intervals and accuracy, it follows that, for a given confidence level c, more trials tend to give tighter confidence intervals,

and hence higher accuracy. Similarly, as the confidence level increases, the width of the confidence interval also increases, requiring more trials to maintain a target accuracy. For the scatter plot in Figure 5.4, at load factor 0.3 and runlength of 90 seconds, the data gives us 70% confidence that $5.6 < \bar{R} < 6$, or 95% confidence that $5 < \bar{R} < 6.5$. ($\bar{R}$ is the mean server response time.) The data also determines the runlength needed to achieve target confidence and accuracy: a runlength of 90 seconds achieves an accuracy of 87% with 95% confidence, but it takes a runlength of 300 seconds to achieve 95% accuracy with 95% confidence.

Accuracy and confidence decrease with higher load factors unless the number of trials t and/or the runlength of each trial r is increased. For example, at load factor 0.9 and runlength 90, the data gives us 70% confidence that $21 < \bar{R} < 24$ (93.3% accuracy), or 95% confidence that $20 < \bar{R} < 27$ (85.1% accuracy). The controller can run the experiment with a longer runlength in order to achieve a given target confidence level and/or accuracy. For example, in order to achieve accuracy $\geq 87\%$ at 95% confidence, the controller need a runlength of 120 seconds or more.

In addition to increasing the runlength, increasing the number of trials is another way to improve the confidence level and accuracy. Figure 5.5 quantifies the tradeoff between the runlength and the number of trials required to attain a target accuracy and confidence for different workloads. It shows the number of trails required to meet an accuracy of 90% at 95% confidence level for different runlengths. The figure shows that to attain a target accuracy and confidence, one needs to conduct more independent trials at smaller runlengths, and vice versa. It also shows that there is a sweet spot for the runlength of an experiment that causes significant reduction in the number of trials. Figure 5.5 shows that runlength ≥ 3 minutes is sufficient to attain the sweet spot for the number of trials in our setting. In general, the controller can use such curves as a guide to pick a suitable runlength.

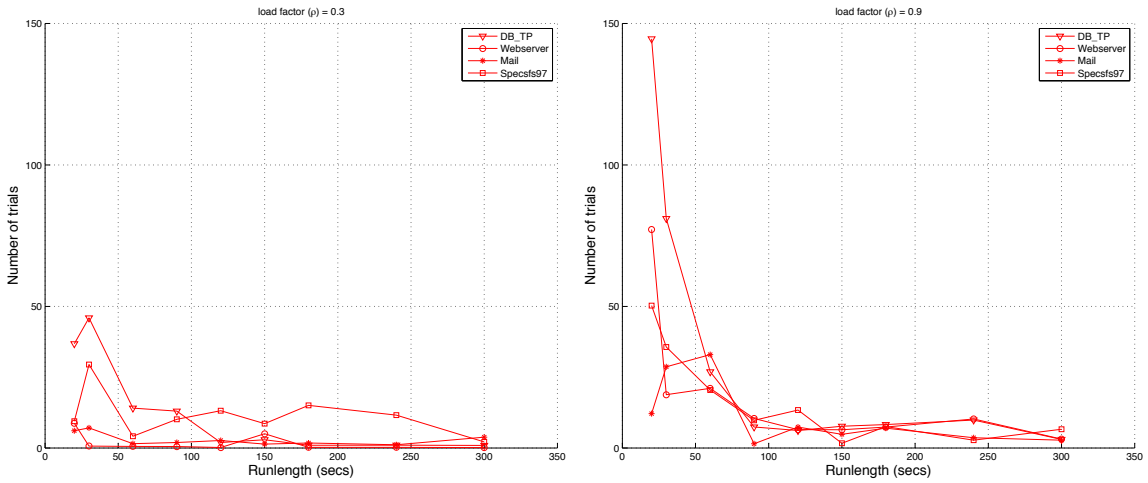

Figure 5.5: Number of trials required to get 90% accuracy for mean server response time at 95% confidence level at low and high load factors for different runlengths. The results are for server configuration with 1 disk and 4 nfsds, and representative of other server configurations.

5.4 Search Algorithm for Peak Rate

Algorithm 12 illustrates our end-to-end search-based approach for estimating the peak rate for a given setting of factors, while meeting target levels of confidence and accuracy. The measures used by this algorithm are summarized in Table 5.2.

5.4.1 Inputs

The inputs to Algorithm 12 specify the high level goals of storage server benchmarking such as the operating bounds of the server and the target confidence and accuracy of the estimated peak rate.

- R_{sat}, the threshold on the mean server response time. The server saturates when its mean response time exceeds this threshold, i.e., $\bar{R} > R_{sat}$.
- P_{sat}, the threshold on the percentage of requests that must complete under a threshold latency L_{sat}. The server saturates when this threshold is not met, i.e., $P < P_{sat}$.

- Width parameter s that defines the *peak-rate region* $[R_{sat} \pm sR_{sat}]$. The peak rate λ^* is any test load that causes the mean server response time to be in this region. (The region $[P_{sat} \pm sP_{sat}]$ is defined similarly.)

- Target confidence c in the peak rate that the algorithm estimates.

- Target accuracy a of the peak rate that the algorithm estimates.

Algorithm 12 consists of three key steps that involve choosing: (a) a sequence of test loads to try; (b) the number of independent trials at any test load; and (c) the runlength of the workload at that load.

5.4.2 Sequence of Test Loads

For choosing a test load to try, Algorithm 12 uses one of several *load-picking* algorithms; Algorithm 13, termed *Binsearch*, is one such example. Section 5.5 describes these algorithms in detail and their cost and accuracy tradeoffs. All load-picking algorithms take as input the value of past test loads that Algorithm 12 attempts, and the corresponding mean server response times. The output consists of a load that is a potential peak rate. This load then becomes the next test load in Algorithm 12.

5.4.3 Number of Trials

For the current test load λ_{cur}, Algorithm 12 first conducts two trials to generate an initial confidence interval for $\bar{R}_{\lambda_{cur}}$, the mean server response time at load λ_{cur}, at 95% confidence level. (Steps 6 and 7 in Algorithm 12). Next, a test is done to check whether the confidence interval overlaps with the peak-rate region input to the algorithm (Step 9). Overlap is tested by comparing the bounds of the peak-rate region and the confidence interval. These steps establish with 95% confidence level whether the current test load is a potential peak rate. If there is no overlap, then

Algorithm 12 moves on to the next test load as guided by a load-picking algorithm such as Algorithm 13 (Step 2).

If there is an overlap of the regions, then Algorithm 12 identifies the current test load λ_{cur} as an estimate of a potential peak rate. It then computes the accuracy of the mean server response time $\bar{R}_{\lambda_{cur}}$ at the current test load, at the target confidence level of $c\%$ (Section 5.2.1). If the accuracy at the target confidence matches the target accuracy a, then the algorithm terminates (Step 4), otherwise it conducts more trials at the current test load (Step 6).

As the algorithm conducts more trials at the current test load, the accuracy improves because the confidence interval gets narrower (Section 5.3.1). As a result, one of two things happen: (i) the overlap test of the confidence interval and the peak-rate region fails (Step 10), in which case the algorithm correctly moves on to the next test load; or (ii) the overlap test does not fail and after some number of trials, the algorithm attains the target accuracy.

5.4.4 Runlength for Test Load

To simplify the choice of runlength for each experiment at a test load (Step 5), Algorithm 12 uses the sweet spot that is derived from Figure 5.5 (Section 5.3.1). The figure shows that for all workloads that this work considers, a runlength around 3 minutes attains the sweet spot for the number of trials.

5.4.5 Discussion

Algorithm 12 automatically adapts the number of trials at any test load according to the load factor and the desired confidence and accuracy. Section 5.6 presents empirical results that demonstrate the same.

For very low or very high load factors, the algorithm conducts a small (often the

minimum of two in our evaluation) number of trials to establish with 95% confidence that the current test load is not the peak rate (Step 10). However, as soon as the algorithm identifies a test load λ to be a potential peak rate, which happens near a load factor of 1, it spends more time at λ to check whether it is in fact the peak rate. Since the algorithm computes the confidence interval after each trial, it conducts the minimum number of trials to establish whether λ is the peak rate.

The while condition in Step 4 of Algorithm 12 matches the current accuracy of the potential peak rate with the target accuracy at a target confidence. Since the accuracy and confidence improve with more trials, if the target confidence and accuracy are low, the algorithm will automatically conduct less trials before it terminates. Thus, the benchmarking cost will be low if the desired target confidence and accuracy are low, and vice versa.

5.5 Mapping Response Surfaces

Algorithm 11 shows how to map a response surface for a space of samples corresponding to a subset of interesting factors in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$: the outer loop iterates over the full space of samples, and the inner loop finds the peak rate for each sample. Mapping a response surface poses two challenges:

- Algorithm 12 from Section 5.4 is used to implement the inner loop. However, the algorithm needs a good load-picking policy to generate a sequence of test loads. An efficient controller policy will generate a new test load based on the feedback of the previous results, e.g., the server response time and throughput observed on the earlier test loads (see loop L_2 in Figure 2.2). Sections 5.5.1-5.5.4 describe the load-picking algorithms that this work considers.

- Algorithm 11 also needs a policy for choosing the samples in the outer loop.

Algorithm 12: Searching for the Peak Rate

1) **Initialization.** Peak Rate, $\lambda^* = 0$; Current accuracy of the peak rate, $a_{\lambda^*} = 0$; Current test load, $\lambda_{cur} = 0$; Previous test load, $\lambda_{prev} = 0$;
2) Use Algorithm 13 to choose a test load λ by giving current test load λ_{cur}, previous test load λ_{prev}, and mean server response time $\bar{R}_{\lambda_{cur}}$ at λ_{cur} as inputs;
3) Set $\lambda_{prev} = \lambda_{cur}$ and $\lambda_{cur} = \lambda$;

   // Conduct trials until the target accuracy for the peak rate is reached at the desired confidence.
4) **while** ($a_{\lambda^*} < a$ at confidence c)
5) Choose the runlength r for the trial;
6) Conduct the trial at λ_{cur}, and measure server response time from this trial, $R_{\lambda_{cur}}$;
7) Compute mean server response time at λ_{cur}, $\bar{R}_{\lambda_{cur}}$, from all trials at λ_{cur}. Repeat Step 6 if the number of trials, t, at λ_{cur} is 1;
8) Compute confidence interval for the mean server response $\bar{R}_{\lambda_{cur}}$ at target confidence level c.
9) Check for overlap between the confidence interval for $\bar{R}_{\lambda_{cur}}$ and the peak rate region.
10) **if** (no overlap with 95% confidence)
        Go to Step 2 to choose the next test load;
     else
        $\lambda^* = \lambda_{cur}$; *// A potential peak rate has been reached*;
        Compute accuracy a_{λ^*} at confidence c;
     end
end

Section 5.1 explains that exhaustive enumeration of the full factor space in the outer loop can incur an exorbitant benchmarking cost. Depending on the goal of the benchmarking exercise, the controller can choose more efficient techniques. Section 5.5.5 discusses some of these techniques.

Algorithm 13: Binsearch **Input**: Previous load λ_{prev}; Current load λ_{cur}; Mean response time $\bar{R}_{\lambda_{cur}}$ at λ_{cur}; **Output**: Next load λ_{next}

1) **Initialization.**
   if $(\lambda_{cur} == 0)$
      Set λ_{next} to a default value;
      Start Geometric Phase, and return λ_{next};
2) **Geometric Phase.**
   if $(\bar{R}_{\lambda_{cur}} < R_{sat})$
      $Return \lambda_{next} = \lambda_{cur} \times 2$;
   else
      // End Geometric Phase; Start Binary Search;
      $\text{binsearch}_{low} = \lambda_{prev}$, and go to Step 3;
   end
3) **Binary Search Phase.**
   if ($\bar{R}_{\lambda_{cur}} < R_{sat}$) $\text{binsearch}_{low} = \lambda_{cur}$;
   else
      $\text{binsearch}_{high} = \lambda_{cur}$;
   end
   Return $\lambda_{next} = (\text{binsearch}_{high} + \text{binsearch}_{low})/2$;

5.5.1 The Binsearch Load-Picking Algorithm

Algorithm 13 outlines the *Binsearch* algorithm. Intuitively, Binsearch keeps doubling the current test load until it finds a load that saturates the server. After that, Binsearch applies regular binary search, i.e., it recursively halves the most recent interval of test loads where the algorithm estimates the peak rate to lie.

This algorithm allows the controller to find the lower and upper bounds for the peak rate within a logarithmic number of test loads. The controller can then estimate the peak rate using another logarithmic number of test loads. Hence, the total number of test loads is always logarithmic irrespective of the start test load or the peak rate.

5.5.2 The Linear Load-Picking Algorithm

The *Linear* algorithm is similar to Binsearch except in the initial phase of finding the lower and upper bounds for the peak rate. In the initial phase it picks an increasing sequence of test loads such that each load differs from the previous one by a small fixed increment.

5.5.3 Model-guided Load-Picking Algorithm

The general *shape* of the response-time Vs. load curve is well known, and it does not change drastically for different workloads or server configurations. Using the insight offered by the open-loop queuing theory results [59], we capture the curve by a model: $R = a + \frac{b}{\lambda}$, where R is the response time, λ is the load, and a and b are constants that depend on the settings of factors in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$. To learn the model, the controller needs tuples of the form $\langle \lambda, R_\lambda \rangle$. Since the controller can record the server response times at different test loads, it can learn the model online as it collects $\langle \lambda, R_\lambda \rangle$ tuples for a given sample in the outer loop of Algorithm 11.

Algorithm 14 outlines the *model-guided* algorithm. If there are insufficient tuples for learning the model, it uses a simple heuristic to pick the test loads for generating the tuples. After that, the algorithm uses the model to predict the peak rate $\lambda = \lambda^*$ for $R = R_{sat}$, returns the prediction as the next test load, and relearns the model using the new $\langle \lambda, R_\lambda \rangle$ tuple at the prediction. The whole process repeats until the search converges to the peak rate. As the controller observes more $\langle \lambda, R_\lambda \rangle$ tuples, the model-fit will improve progressively, and hence the model will guide the search to an accurate peak rate. In many cases, this happens in a single iteration of model learning (Section 5.6).

However, unlike the previous approaches, a model-guided search is not guaranteed to converge. Model-guided search is dependent on the accuracy of the model, which

in turn depends on the choice of $\langle \lambda, R_\lambda \rangle$ tuples that are used for learning. The choice of tuples is generated by previous model predictions. This creates the possibility of the learning an *incorrect* model which in turn yields incorrect choices for test loads. For example, if most of the test loads chosen for learning the model happen to lie significantly outside the peak rate region, then the model-guided choice of test loads may be incorrect or inefficient. Hence, in the worst case, the search may never converge or converge slowly to the peak rate. We have experimented with other models including polynomial models of the form $R = a + b\lambda + c\lambda^2$; they are all prone to similar pitfalls.

To avoid the worst case, the algorithm uses a simple heuristic to choose the tuples from the list of available tuples. Each time the controller learns the model, it chooses two tuples such that one of them is the last prediction, and the other is the tuple that yields the response time closest to threshold mean server response time R_{sat}. More robust techniques for choosing the tuples is a topic of future work. Section 5.6 reports our experience with the model-guided choice of test loads.

5.5.4 Better Seeding

The load-picking algorithms in Sections 5.5.2-5.5.3 generate a new load given one or more previous test loads. How can the controller generate the first load, or *seed*, to try? One way is to use a conservative low load as the seed, but this approach increases the time spent ramping up to a high peak rate. When the benchmarking goal is to plot a response surface, the controller uses another approach that uses the peak rate of the "nearest" previous sample as the seed.

To illustrate, assume that the factors of interest, $\langle F_1, \ldots, F_n \rangle$, in Algorithm 11 are $\langle$ number of disks, number of nfsds $\rangle$ (as shown in Figure 5.1). Suppose the controller uses Binsearch with a low seed of 50 to find the peak rate $\lambda^*_{1,1}$ for sample $\langle 1, 1 \rangle$. Now,

Algorithm 14: Model-Guided **Input**: Previous loads $\lambda_1, \lambda_2, ..., \lambda_{cur-1}$; Current load λ_{cur}; Mean response times $\bar{R}_{\lambda_1}, \bar{R}_{\lambda_2}, ..., \bar{R}_{\lambda_{cur}}$ at $\lambda_1, \lambda_2, ..., \lambda_{cur}$; **Output**: Next load λ_{next}

1) **Initialization.**
   if ($\lambda_{cur} == 0$)
      Return a default λ_{next};
   end
   if (number of test loads == 1)
      if ($\bar{R}_{\lambda_{cur}} < R_{sat}$)
         Return $\lambda_{next} = \lambda_{cur} \times 2$;
      else
         Return $\lambda_{next} = \lambda_{cur}/2$;
      end
   end

2) **Model Learning and Prediction.**
   Choose a value of $\bar{R}_i$ from $\bar{R}_{\lambda_1}, ..., \bar{R}_{\lambda_{cur-1}}$ that is nearest to $\bar{R}_{sat}$. Let the corresponding load be λ_i;
   Learn the model $R = a + \frac{b}{\lambda}$ with two tuples $\langle \lambda_{cur}, \bar{R}_{\lambda_{cur}} \rangle$ and $\langle \lambda_i, \bar{R}_i \rangle$;
   Return $\lambda_{next} = \frac{b}{R_{sat} - a}$;

for finding the peak rate $\lambda^*_{1,2}$ for sample $\langle 1, 2 \rangle$, it can use the peak rate $\lambda^*_{1,1}$ as seed. Thus, the controller can jump quickly to a test load that is close to $\lambda^*_{1,2}$.

In the common case, the peak rates for "nearby" samples will be close. Even if they are not, the load-picking algorithms will still guide the search in the right direction. However, they may incur additional cost to recover from a bad seed. The notion of "nearness" is not always well defined. While the distance between samples can be measured if the factors are all quantitative, if there are categorical factors—e.g., file system type—the nearest sample may not be well defined. In such cases the controller uses a default seed to start the search.

5.5.5 Approximating the Response Surface

If the overall goal of server benchmarking is to understand the overall trend of how the peak rate is affected by settings of certain factors of interest $\langle F_1, \ldots, F_n \rangle$—rather than finding accurate peak rate values for each sample in $\langle F_1, \ldots, F_n \rangle$—then more efficient techniques exist than iterating over all samples as in Algorithm 11. We can leverage Response Surface Methodology (RSM) [78], a branch of statistics that provides techniques to choose a small set of samples carefully so that the controller can approximate the overall response surface efficiently.

By assuming that a low-degree multivariate polynomial model—e.g., a quadratic equation of the form $\lambda^* = \beta_0 + \sum_{i=1}^{n} \beta_i F_i + \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} \beta_{ij} F_i F_j + \sum_{i=1}^{n} \beta_{ii} {F_i}^2$—approximates the surface in the n-dimensional $\langle F_1, \ldots, F_n \rangle$ space, RSM provides experiment designs (Section 2.2) for selecting a minimal set of samples for which the controller must obtain the λ^* to learn a fairly-accurate model, i.e., estimate values of the β parameters in the model (see Section 3.7). We evaluate one such RSM design in Section 5.6.

5.6 Experimental Evaluation

We evaluate the benchmarking methodology and policies with multiple workloads on the following metrics.

Cost for Finding Peak Rate. Sections 5.4 and 5.5 present several policies for finding the peak rate. We evaluate those policies as follows:

- The sequence of load factors that the policies consider before converging to the peak rate for a sample. An efficient policy must quickly direct the benchmarking effort to load factors that are near or at 1.

- The number of independent trials for each load factor. The number of trials

should be less at low load factors and high around load factor of 1.

Cost for Mapping Response Surfaces. We compare the total benchmarking cost for mapping the response surface across all the samples.

Cost Versus Target Confidence and Accuracy. We demonstrate that the policies adapt the total benchmarking cost to target confidence and accuracy. Higher confidence and accuracy incurs higher benchmarking cost and vice-versa.

Section 5.6.1 presents the experiment setup. Section 5.6.2 presents the workloads that we use for evaluation. Section 5.6.3 evaluates our benchmarking methodology as described above.

5.6.1 Experimental Setup

Table 5.1 shows the factors in the $\langle \vec{W}, \vec{R}, \vec{C} \rangle$ vectors for a storage server. We benchmark an NFS server to evaluate our methodology. In our evaluation, the factors in $\vec{W}$ consist of samples that yield four types of workloads: SPECsfs97, Web server, Mail server, and DB_TP (Section 5.6.2). The controller uses Fstress to generate samples of $\vec{W}$ that correspond to these workloads. We report results for a single factor in $\vec{R}$: the number of disks attached to the NFS server ranging from $\langle 1, 2, 3, 4 \rangle$, and a single factor in $\vec{C}$: the number of nfsd daemons for the NFS server ranging from $\langle 1, 2, 4, 8, 16, 32, 64, 100 \rangle$ to give us a total of 32 samples.

The workbench tools can generate both virtual and physical machine configurations automatically. In our evaluation we use physical machines that have 800 MB memory, 2.4 GHz x86 CPU, and run the 2.4.18 Linux kernel. To conduct an experiment, the workbench controller first prepares an experiment by generating a sample in $\langle \vec{W}, \vec{R}, \vec{C} \rangle$. It then consults the benchmarking policy(ies) in Sections 5.5.1-5.5.5 to plot a response surface and/or search for the peak rate for a given sample with target confidence and accuracy.

Table 5.3: Summary of *Fstress* workloads used in the experiments.

workload	file popularities	file sizes	dir sizes	I/O accesses
SPECsfs97	random 10%	1 KB – 1 MB	large (thousands)	random r/w
Web server	Zipf ($0.6 < \alpha < 0.9$)	long-tail (avg 10.5 KB)	small (dozens)	sequential reads
DB_TP	few files	large (GB - TB)	small	random r/w
Mail	Zipf ($\alpha = 1.3$)	long-tail (avg 4.7 KB)	large (500+)	seq r, append w

5.6.2 Workloads

We use Fstress to generate $\vec{W}$ corresponding to four workloads as shown in Table 5.3. A brief summary follows. Further details are in [5].

- **SPECsfs97**: The Standard Performance Evaluation Corporation introduced their System File Server benchmark (SPECsfs) [26] in 1992, derived from the earlier self-scaling LADDIS benchmark [64]. A recent (2001) revision corrected several defects identified in the earlier version [44].

- **Web server**: Several efforts (e.g., [8]) attempt to identify durable characterizations of the Web. We derive the distributions for various parameters and the operation mix from the previous published studies (e.g., [94, 28, 84, 32, 8]).

- **DB_TP**: We model our database workload after TPCC [27], reading and writing within a handful of large files in a 2:1 ratio. I/O access patterns are random, with some short (256 KB) sequential asynchronous writes with *commit* (fsync) to mimic batch log writes.

- **Mail**: Electronic mail servers frequently handle many small files, one file per users' mailbox. Servers append incoming messages, and sequentially read the

mailbox file for retrieval. Some users or servers truncate mailboxes after reading. The workload model follows that proposed by Saito et al. [97].

5.6.3 Results

For evaluating the overall methodology and the policies outlined in Sections 5.4 and 5.5, we define the peak rate λ^* to be the test load that causes: (a) the mean server response time to be in $[36, 44]$ ms region; or (b) more than 10% of the requests to complete over 2000 ms. We derive the $[36, 44]$ region by choosing mean server response time threshold at the peak rate to be, $R_{sat} = 40$ and the width factor $s = 10\%$ in Table 5.2. For all results except where we note explicitly, we aim for a λ^* to be accurate within 90% of its true value with 95% confidence. The default value of initial load in Algorithms 13 and 14 is 50 requests/sec.

Cost for Finding Peak Rate

Figure 5.6 shows the choice of load factors for finding the peak rate for a sample with 4 disks and 32 nfsds using the policies outlined in Section 5.5. Each point on the curve represents a single trial for some load factor. More points indicate higher number of trials at that load factor. For brevity, we show the results only for **DB_TP**. Other workloads show similar behavior.

For all policies, the controller conducts more trials at load factors at or near 1 than at other load factors to find the peak rate with the target accuracy and confidence. All policies without seeding start at a low load factor and take longer to reach close to load factor of 1 as compared to policies with seeding. All policies with seeding start at load factor close to 1, since they use the peak rate of a previous sample with 4 disks and 16 nfsds as the seed load.

Linear takes a significantly longer time because it uses a fixed increment by

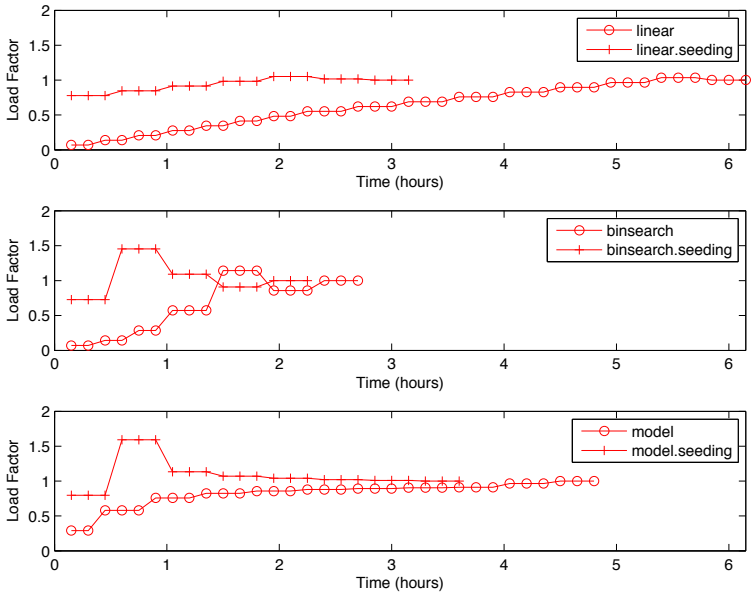

Figure 5.6: Time spent at each load factor for searching the peak rate for different policies for **DB_TP** with 4 disks, and 32 nfsds. The result is representative of other samples and workloads. All policies except linear quickly converge to the load factor of 1 and conduct more trials there to achieve the target accuracy and confidence.

which to increase the test load. However, *Binsearch* jumps to the peak rate region in logarithmic number of load factors. The *Model* policy is the quickest to jump near the load factor of 1 because the controller learns an accurate model quickly.

Cost for Mapping Response Surfaces

Figure 5.7 compares the total normalized benchmarking cost for mapping the response surfaces for the three workloads using the policies outlined in Section 5.5. The costs are normalized with respect to the lowest total cost, which is the *Binsearch with Seeding* policy to find the peak rate for **DB_TP**. *Binsearch*, *Binsearch With Seeding*, and *Linear with Seeding* cut the total cost drastically as compared to the linear policy.

We also observe that *Binsearch*, *Binsearch with Seeding*, and *Linear With Seeding* are robust across the workloads, but the model-guided policy is sensitive to some workloads. This is not surprising given that the accuracy of the model guides the

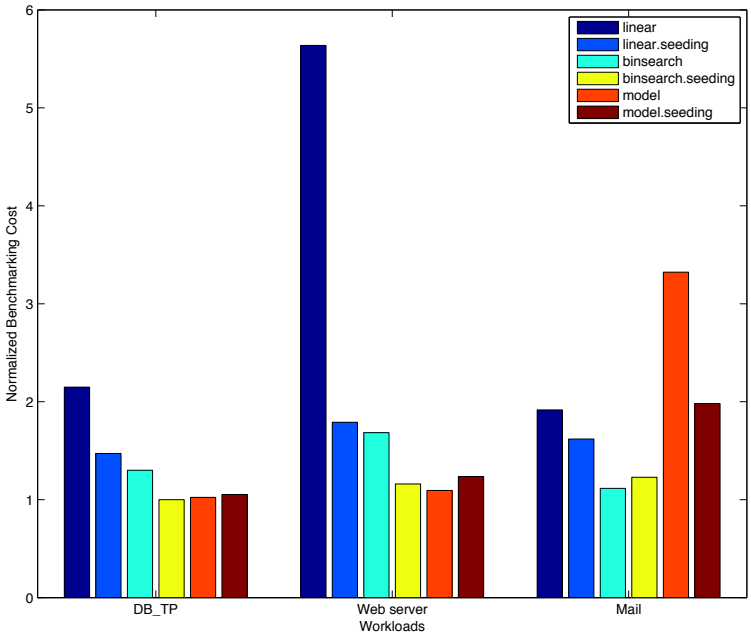

Figure 5.7: The total cost for mapping response surfaces for three workloads using different policies.

search. While an accurate model can guide the search quickly to the peak rate, an inaccurate model can direct the search in the wrong direction. Thus the model-guided policy may take longer to find the peak rate.

The linear policy is not only inefficient, but also highly sensitive to the magnitude of peak rate. The benchmarking cost of *Linear* for **Web server** peaks at a higher absolute value for all samples than for **DB_TP** and **Mail**, causing more than a factor of 5 increase in the total cost for mapping the surface.

Reducing the Number of Samples. To evaluate the RSM approach that Section 5.5.5 presents, we approximate the response surface by a quadratic curve in two dimensions: peak rate = func(number of disks, number of nfsds). We use D-optimal experiment design [78] from RSM to obtain the best of 6, 8, and 10 samples out of a total of 32 samples for learning the response surface equation. We use *Binsearch* to obtain the peak rate for the chosen samples.

Table 5.4: Mean Absolute Prediction Error (MAPE) in Predicting the Peak Rate.

Workload	Num. of Samples	MAPE
DB_TP	6, 8, 10	14, 14, 15
Web server	6, 8, 10	9, 9, 9
Mail	6, 8, 10	3.3, 2.8, 2.7

D-optimal design is an example of a model-learning design (Section 3.7). The experiments in such designs are chosen from the total space of experiments such that the resulting samples minimize the generalized variance of the parameter estimates for the model that is being learned.

After learning the model, we use it to predict the peak rate at all the other samples in the surface. Table 5.4 presents the mean absolute percentage error in predicting the peak rate across all the samples. The results show that if the goal is simply to approximate the surface, we can reduce the size of the sample space significantly.

Cost Versus Target Confidence and Accuracy

Figure 5.8 shows how the benchmarking methodology adapts the total benchmarking cost to the target confidence and accuracy of the peak rate. The figure shows the total benchmarking cost for mapping the response surface for the **DB_TP** using the *Binsearch* policy for different target confidence and accuracy values.

Higher target confidence and accuracy incurs higher benchmarking cost. At 90% accuracy, note the cost difference between the different confidence levels. Other workloads and policies exhibit similar behavior, with **Mail** incurring a normalized benchmarking cost of 2 at target accuracy of 90% and target confidence of 95%.

So far, we configure the target accuracy of the peak rate by configuring the accuracy, a, of the response time at the peak rate. The width parameter s also controls the accuracy of the peak rate (Table 5.2) by defining the peak rate region. For ex-

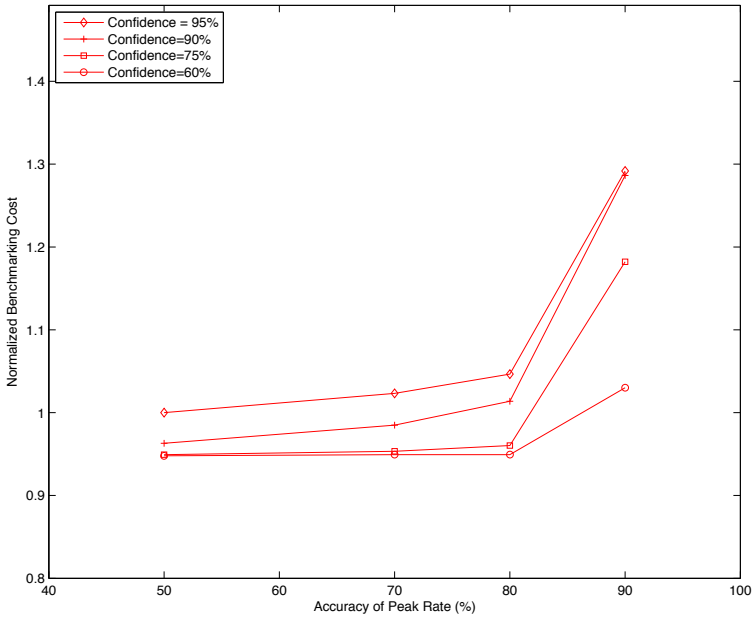

Figure 5.8: The total benchmarking cost adapts to the desired confidence and accuracy. The cost is shown for mapping the response surface for **DB_TP** using the *Binsearch* policy. Other workloads and policies show similar results.

ample, $s = 10\%$ implies that if the mean server response time at a test load is within 10% of the threshold mean server response time, R_{sat}, then the controller has found the peak rate. As the region narrows, the target accuracy of the peak rate region increases. In our evaluation so far, we fix $s = 10\%$.

Figure 5.9 shows the benchmarking cost adapting to the target accuracy of the peak rate region for different policies at a fixed target confidence interval for **DB_TP** ($c = 95$) and fixed target accuracy of the mean server response time at the peak rate ($a = 90\%$). The results for other workloads are similar. All policies except the model-guided policy incur the same benchmarking cost near or at the peak rate since all of them do binary search around that region. Since a narrower peak rate region causes more trials at or near load factor of 1, the cost for these policies converge.

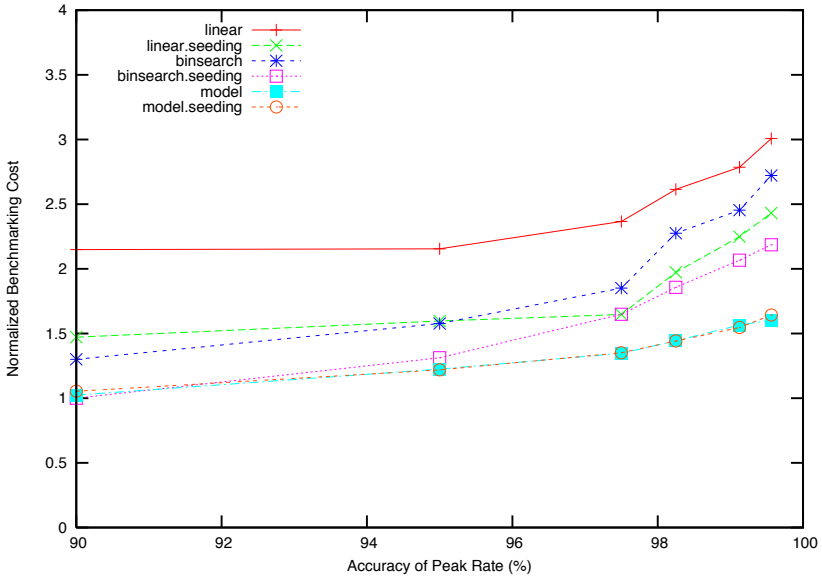

Figure 5.9: Benchmarking cost adapts to the target accuracy of the peak rate region for all policies. As the region narrows, the majority of the cost is incurred at or near the peak rate. Linear and Binsearch incur the same cost close to the peak rate, and hence their cost converges as they conduct more trials near the peak rate. The cost is shown for **DB_TP**. Other workloads show similar results.

5.7 Related Work

Several researchers have made a case for statistically significant results from system benchmarking, e.g., [18]. Auto-pilot [117] is a system for automating the benchmarking process such that a benchmarking experiment can obtain results with the target confidence and accuracy for a single test load on the system. We use this idea as a basis for an efficient and accurate search for the peak rate through a larger space of a test loads, e.g., to obtain the saturation throughput for a server under a given workload, resource allocation, and configuration.

While there are large numbers and types of benchmarks, (e.g., [23, 61, 17, 64]) that test the performance limits of a system in a variety of ways, there is a lack of a general benchmarking methodology that provides benchmarking results from these benchmarks efficiently with confidence and accuracy. Our methodology and techniques for balancing the benchmarking cost and accuracy are applicable to all

these benchmarks.

Zadok et al. [110] present an exhaustive nine year study of file system and storage benchmarking that includes benchmark comparisons, their pros and cons [103], and makes recommendations for systematic benchmarking methodology that considers a range of workloads for benchmarking the server. Smith et al. [104] make a case for benchmarks the capture realistic application behavior. Ellard et al. [36] show that benchmarking an NFS server is challenging because of the interactions between the server software configurations, workloads, and the resources allocated to the server. One of the challenges in understanding the interactions is the large space of factors that govern such interactions. Our benchmarking methodology benchmarks a server across the multi-dimensional space of workload, resource, and configuration factors efficiently and accurately, and avoids brittle claims [75] and lies [107] about a server performance.

Synthetic workloads emulate characteristics observed in real environments. They are often self-scaling [23], augmenting their capacity requirements with increasing load levels. The synthetic nature of these workloads enables them to preserve workload features as the file set size grows. In particular, the SPECsfs97 benchmark [26] (and its predecessor LADDIS [64]) creates a set of files and applies a pre-defined mix of NFS operations. Fstress is a synthetic, flexible, self-scaling file service benchmark similar to SPECsfs97. Like SPECsfs97, Fstress uses probabilistic distributions to govern workload mix and access characteristics. Fstress adds file popularities, directory tree size and shape, and other controls. Fstress includes several important workload configurations, such as Web server file accesses, to simplify file system performance evaluation under different workloads [104] while at the same time allowing standardized comparisons across studies.

5.8 Conclusions and Future Work

This work presents efficient and effective controller policies for benchmarking servers, in particular storage servers. The policies plot the saturation throughput or peak rate over a space of workloads and system configurations. The overall approach consists of iterating over the space of workloads and configurations to find the peak rate for samples in the space. The policies find the peak rate efficiently while meeting target levels of confidence and accuracy to ensure statistically rigorous benchmarking results.

The controller may use a variety of heuristics and methodologies to prune the sample space to map a complete response service. This work illustrates one such approach, and an exhaustive comparison of different policies to prune the sample space remains an interesting future work.

Chapter 6

Spectrum of Models: A Discussion

Predictive system models are a prerequisite for a large variety of system management tasks [45]. Section 2.1 introduces the spectrum of models that this dissertation considers. This chapter further explores the spectrum in the context of the experiment-driven framework.

6.1 Models

A model consists of a *structure* and *parameters*. The structure of a model defines the relationship among the model parameters. To use a model that predicts the system behavior as a function of parameters that affect the system, we must have sufficiently accurate values for the model parameters as well as the correct model structure. Section 2.1 introduces the spectrum of models ranging from a priori to black-box models. In an a priori model both the model structure and model parameters are known based on the knowledge of the internal details of the system. On the other hand, in a black-box model, nothing is assumed about the internal working of the system apart from what can be observed from *outside* the system. Hence, in such models, neither the model structure nor the value of model parameters is known.

In practice, models lie somewhere in between the two extremes depending on how much prior information is available about the model structure and parameters. In a priori models the model structure is known, but accurate values of model parameters are unknown. Similarly in black-box models there might be some prior knowledge of system internals that can guide the *guess* for the appropriate model structure. Sections 6.1.1 and 6.1.2 present a brief description of the two extremes and discuss

the role of the experiment-driven framework from Chapter 2 in the overall spectrum.

6.1.1 A Priori Models

Consider a simple system with a single hardware device, where a stream of requests access the device for some service. Figure 6.1 illustrates the response time of such a system as a function of the system's utilization. Equation 6.1 presents an example of a simple a priori model that captures the response time behavior that the figure illustrates. The equation represents a queuing theory model of a single service center [69].

The model outputs the response time R of a system as a function of: (a) service demand of the system, D; and (b) utilization of the system, U. The service demand D captures the service requirement for each unit of work that the system does. The system utilization captures the *busyness* of the system for a given interval of time; $U = 1$ implies that the system is fully utilized or busy for the entire duration of the interval. The structure of the model shows that system response time R is directly proportional to the service demand D, and inversely related to the available capacity of the system $1 - U$.

$$R = \frac{D}{1 - U} \tag{6.1}$$

In the model, the parameter for system utilization U serves as an input to the model, and the parameter for service demand D is the unknown parameter. The model can easily capture the system behavior in Figure 6.1 assuming that accurate value of the system service demand D is known. In practice, it is rarely the case that the value of model parameters is known a priori even if the structure is well known. Hence, the model parameters must be estimated from the observations of samples of system behavior. For the model in Equation 6.1, we need to estimate the value of

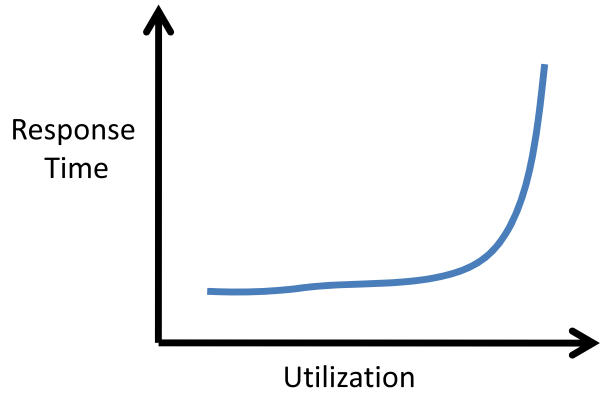

Figure 6.1: System response time as a function of the system utilization.

the system service demand D by observing the system utilization U and the system response time R (or the system throughput [69]) when the system is in operation. Once an accurate estimate of the system service demand is available, the model can be used to predict the system response time R as a function of the system utilization U.

The simple model in Equation 6.1 illustrates that in order to estimate the values of unknown model parameters for a priori models we need samples of system behavior. As Section 1.3 discusses, gathering relevant samples of system behavior is a challenging problem. For example, the service demand of a disk depends on factors such as the layout of the data on the disk, the type of workload, the disk scheduling algorithm, the operating system prefetching policies, and the caching hierarchy. The samples must expose the impact of the important factors and interactions to obtain a sufficiently accurate estimate of the disk service demand.

We can use the experiment-driven framework from Chapter 2 to obtain the relevant samples quickly and efficiently to learn accurate estimates for the parameters. In this work, we show that the experiment-driven framework is applicable for learning the parameters for such models in two system domains. Chapter 4 presents an appli-

cation performance model for batch applications, where the model structure is based on prior knowledge of the behavior of batch applications (Section 4.4.1). Chapter 5 presents a server response time model, where the model structure is based on prior knowledge of the behavior of response time curve as a function of load (Section 5.5.3).

A priori models are widely used for modeling computer systems. They offer valuable insight into the system behavior independent of their use to predict the performance effects of management choices. In our work leading up to this dissertation we developed the *LAWS* model that bounds the maximum or *peak* performance benefit of low-overhead I/O as a result of innovations in networking hardware (e.g., faster networks), or software (e.g., by protocol offload) for the entire space of applications. The key parameters of LAWS consist of four ratios—Lag, Application, Wire, and Structural or LAWS—that capture speed differences between the application host and the network, the CPU-intensity of the application, and structural factors that may eliminate a portion of network I/O overhead. We found LAWS to be useful for understanding trends, gaining insights into system behavior as a result of different design choices, and for interpreting previous and current empirical studies.

Moreover, if the samples to learn the model parameters are available, then such models can be used to extrapolate system behavior beyond the behavior seen in the samples. For example, Equation 6.1 captures the non-linear behavior of system response time, and it can predict the response time correctly even if the samples used to learn the service demand do not exhibit significant non-linear response time behavior.

However, since systems consist of several interacting components, it is often difficult to understand the internal details of the system in order to create the correct model structure. Moreover, systems change over time as system administrators respond to varying workload demands. In such situations, existing a priori models may

become brittle unless they adapt to the changes quickly [24].

6.1.2 Black-Box Models

Due to the challenges involved in formulating the correct structure for a priori models, there is a considerable interest in black-box models that assume little or no prior knowledge about the internals of a system. In such models, a model structure is *guessed* and the model parameters are learned from the samples of system behavior gathered during the system's operation. Black-box models are general and applicable to a wide variety of systems. They easily adapt to changing system environments and can be induced automatically. Recent work in black-box modeling has shown great promise for the use of such models for automated system management (e.g., [24, 121]).

Equation 6.2 illustrates a simple black-box model that predicts the system response time as a function of the system utilization to capture the behavior that Figure 6.1 illustrates. The model consists of two unknown parameters: a which is the coefficient for the system utilization U; U goes as an input to the model, and c which is a constant. Note that unlike the model in Section 6.1.1 that requires the knowledge of the system service demand, the model in Equation 6.2 does not assume any prior information about the system's behavior. It *guesses* that the response time is a linear function of the system utilization, and hence the model structure represents a *linear additive model* [72].

$$R = aU + c \tag{6.2}$$

To learn the parameters of a black-box model, similar to that of a priori models, we need samples of system behavior. For the model in Equation 6.2, the samples consist of $\langle R, U \rangle$ tuples. Once such samples are available, it is easy to learn a and c by applying statistical learning techniques such as least squares regression [53]. Note

however that unlike a priori models, such models are good mainly for interpolation, and their accuracy diminishes outside the operating range that is observed in the samples. For example, if the model in Equation 6.2 is learned using samples that only reflect the portion of Figure 6.1 that can fit a line, then the model will predict inaccurate response time for system utilization outside the range observed in the samples. Section 4.8.2 presents an empirical result where inadequate samples can lead to inaccurate black-box models, and hence wrong inferences about the system behavior.

The experiment-driven framework collects samples of system behavior proactively to ensure that the samples expose the operating range of the relevant parameters and interactions. Chapter 3 presents examples of black-box models that capture the response time and throughput of Web services. We show that the experiment-planning policies not only gather the relevant samples that are required to learn accurate models with a given structure, but can also aid to simplify and verify the model structure for black-box models.

For example, Sections 3.5.1, 3.5.3, and 4.6.2 present experiment-planning policies to prune the large space of factors that affect system performance. Knowledge of important factors can reduce the space of total factors that the model must consider. Section 4.9 illustrates that reducing the factor space enables quick learning of sufficiently accurate models.

Similarly, Sections 3.5.2 and 3.5.3 present experiment-planning policies to identify if the factors interact with each other and if the factors have a non-linear impact on system behavior. Considering the important interactions and non-linear effects ensures that the model can capture a wide range of system behavior that may or may not have appeared in a system's normal operation.

6.2 Summary

This chapter discusses the spectrum of models that Section 2.1 introduces. The discussion shows that the challenge of gathering representative samples of system behavior in order to estimate model parameters is common to both a priori and black-box models. Hence, the experiment-driven framework is applicable for learning models across the modeling spectrum.

Chapter 7

Conclusions and Future Work

Computer systems are becoming increasingly complex to manage due to their scale, the large number of factors that affect their behavior, and unknown interactions among such factors. A 'system knowledge base' that captures how different factors and multi-factor interactions affect the end-to-end behavior of a system is a prerequisite for managing systems effectively. This dissertation addresses the challenge of developing policies and mechanisms to enable building a sufficiently accurate knowledge base automatically, efficiently, and proactively.

We argue that an accurate system knowledge base is difficult to learn by passively observing the system in its normal operation, as is typical today. First, passive observations may not be available to start with in order to learn the knowledge base. Even if such observations are available, they may not expose the impact of all the relevant factors and often unknown interactions between the factors in a scalable manner. This dissertation makes a case for planning and conducting experiments to build the knowledge base proactively.

We develop an experiment-driven framework that incorporates: (a) policies for planning the experiments; and (b) mechanisms for conducting the experiments. The policies that plan the experiments to explore a large range of factors and interactions leverage existing work in two fields: design of experiments and active machine learning. Both these fields are built on rigorous theoretical foundations, and offer efficient techniques to explore the multi-dimensional space of factors and interactions. The mechanisms for conducting the experiments leverage system virtualization technologies to make efficient use of resources that are used for conducting the experiments.

We demonstrate that the experiment-driven framework can learn the knowledge base in an automated, timely, and proactive manner in three important system domains. We apply the experiment-driven framework to: (a) quantify the impact of important factors and interactions in a system; (b) build models that predict system behavior as a function of factors and interactions that affect this behavior; and (c) balance the effort spent in building the knowledge base with the desired confidence and accuracy in the knowledge base.

Empirical results for three system domains—Web services, batch computing, and storage servers—demonstrate that our approach is practical for building the knowledge base across a range of systems. Figure 7.1 summarizes the key results across the domains. One limitation of our approach is that it requires prior knowledge of factors that may impact the system behavior, and their potential operating range. While such knowledge is available for a wide range of systems, there may be settings where it is difficult to glean. Our approach also depends on the availability of the mechanisms and resources to conduct the experiments. In this work we show that such mechanisms already exist for three important system domains.

7.1 Future Work

Based on the work in this dissertation, we identify the following interesting avenues for future research.

7.1.1 Choice of Models

Models of system behavior are a prerequisite for automated system management. Chapter 6 discusses the spectrum of modeling alternatives ranging from a priori models to black-box models in the context of the experiment-driven framework. We observe that without the sufficient and necessary samples to learn the model pa-

System Domains	Number of Factors (workload, resource, configuration)	Knowledge Base	Total Number of Experiments	Experiment-Driven Framework
Web Service TPC-W (Amazon.com)	2 workload (Browse, Order) 2 resource (CPU, Memory) 3 configuration	Rank of Factors and Interactions, Model for throughput, response Time	4860 (~34 days) 10 min/exp	**>= 90% accuracy with < 1-2% experiments**
Web Service RUBiS (eBay.com)	4 workload 2 resource (CPU, Memory)	Rank of Factors and Interactions, Model for throughput, response time, number of errors	5120 (~34 days) 10 min/exp	**>=90% accuracy with <1-2% experiments**
Batch Applications (5 Biomedical, 6 Synthetic)	3 resource (CPU, Memory, Network)	Models to predict execution time	1650 (~30 days)	**>= 90% accurate model with < 3-5% experiments**
Storage Server	Workload factors to generate 3 workloads (Mail, Web server, Database) 2 configuration (Disks, nfsds)	Response surface with 90% accuracy and confidence	7000 (~30 days)	**2000 ~5 days**

Figure 7.1: Summary of results from the application of experiment-driven framework to learn the system knowledge base across different system domains.

rameters, the models will be inaccurate irrespective of where they are in the overall spectrum. Hence, the experiment-driven approach is applicable to the entire modeling spectrum. The experiment-planning policies in this dissertation are model-agnostic and can be used to learn any kind of model.

However, the appropriate choice of models in the overall spectrum remains an interesting question to explore. The appropriate choice depends on the system domain, the management tasks, the availability of prior information about the system, and the techniques used to gather the samples for learning the model. The discussion in Sections 4.6.2 and 6.1.2 touches upon the use of the experiment-driven framework to guide the choice of models, but an automatic selection of models remains an interesting avenue for future work.

7.1.2 Choice of Experiment Designs

A key challenge while planning experiments is to choose experiments that can generate samples to learn a sufficiently accurate knowledge base quickly while making efficient use of resources that are available for conducting the experiments. The experiment-planning policies in this dissertation leverage standard techniques from two fields—design of experiments and active machine learning—to guide the choice of experiments. In our evaluation, we find that the standard techniques are sufficient for generating samples to learn the knowledge base.

However, both design of experiments and active machine learning are themselves active research areas. Hence, the state of the art in these fields may offer techniques that are more efficient and scalable than the standard techniques. Exploring the state of the art in these fields is an interesting area for future work.

7.1.3 Online versus Offline

The experiment-driven framework is applicable to online as well as offline scenarios, i.e., it can be used online on production systems as well as offline on a separate workbench. The experiment-planning algorithms in this dissertation focus on the latter. In online systems, the framework must consider an additional challenge of not causing drastic changes in the behavior of production systems. The literature in design of experiments offers a separate method called evolutionary operation (EVOP) to deal with this scenario [78]. EVOP does not require sudden changes to the system that can disrupt its normal operation. It will be interesting to explore the practicality of such methods for computer systems.

7.1.4 Exploitation versus Exploration

Exploration versus exploitation is a classic dilemma in active machine learning [11]. Exploration improves the accuracy of the knowledge base by conducting more experiments. However, excessive exploration provides diminishing returns at a cost: resources may be wasted to improve the accuracy marginally.

We are exploring this dilemma in the context of batch computing systems [49]. Chapter 4 presents experiment-driven learning of models that predict the execution time of a batch application when it is submitted to a batch scheduling system. On each submission, the choice is between: (a) exploiting the model, i.e., using the currently available model to schedule the application on available resources; and (b) exploring the model, i.e., conducting an experiment to collect one more training sample for improving the accuracy of the model. Exploration may result in longer application execution time, but improve the accuracy of the model for subsequent submissions. Similarly, exploitation with a sufficiently accurate model will result in better scheduling, but an inaccurate model might cause longer execution and inefficient use of resources.

Bibliography

[1] A. Allen. *Probability, Statistics, and Queuing Theory with Computer Science Applications*. Academic Press, 1990.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *Proceedings of the International Conference on Statistical and Scientific Database Management*, June 2004.

[3] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25(17):3389–3402, September 1997.

[4] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *Proceedings of the USENIX Annual Technical Conference*, Jun 2000.

[5] D. C. Anderson and J. S. Chase. Fstress: A Flexible Network File Service Benchmark. Technical Report CS-2002-01, Duke University, Department of Computer Science, Jan 2002.

[6] D. C. Anderson, J. S. Chase, and A. M. Vahdat. Interposed Request Routing for Scalable Network Storage. *ACM Transactions on Computer Systems*, 20(1):25–48, 2002.

[7] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the Conference on File and Storage Technologies*, January 2002.

[8] M. Arlitt and C. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, April 1996.

[9] A. C. Atkinson and A. N. Donev. *Optimum Experimental Designs*. Oxford University Press, USA, 1992.

[10] Grid Physics Network in Atlas. `www.usatlas.bnl.gov/computing/grid/griphyn`.

[11] Y. Baram, R. El-Yaniv, and K. Luz. Online Choice of Active Learning Algorithms. *The Journal of Machine Learning Research*, 5:255–291, 2004.

[12] R. Barrett, E. Kandogan, P. Maglio, E. Haber, L. Takayama, and M. Prabaker. Field studies of computer system administrators: Analysis of system management tools and practices. In *Proceedings of Conference on Computer Supported Cooperative Work*, November 2004.

[13] M. Beck, T. Moore, J. S. Plank, and M. Swany. Logistical Networking: Sharing More Than the Wires. In S. Hariri, C. A. Lee, and C. S. Raghavendra, editors, *Active Middleware Services*. Kluwer Academic, Norwell, MA, 2000.

[14] M. Bennani and D. A. Menascé. Resource Allocation for Autonomic Data Centers Using Analytic Performance Models. In *Proceedings of the International Conference on Autonomic Computing*, May 2005.

[15] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit Control in a Batch-Aware Distributed File System. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, March 2004.

[16] Biomedical Informatics Research Network. `www.nbirn.net`.

[17] T. Bray. Bonnie file system benchmark, 1996. `http://www.textuality.com/bonnie`.

[18] A. B. Brown, A. Chanda, R. Farrow, A. Fedorova, P. Maniatis, and M. L. Scott. The Many Faces of Systems Research: And How to Evaluate Them. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, September 2005.

[19] L. Carrington, M. Laurenzano, A. Snavely, R. Campbell, and L. Davis. How Well Can Simple Metrics Represent the Performance of HPC Applications? In *Proceedings of the International Conference on Supercomputing*, November 2005.

[20] M. Carson and D. Santay. NIST Net: A Linux-Based Network Emulation Tool. *SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[21] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of the International Conference on Supercomputing*, November 2000.

[22] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. In *Proceedings of the SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications*, November 2002.

[23] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation: Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1993.

[24] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, December 2004.

[25] Condor High Throughput Computing. `www.cs.wisc.edu/condor`.

[26] S. P. E. Corporation. SPEC SFS Release 3.0 Run and Report Rules, 2001.

[27] T. P. P. Council. TPC Benchmark C Standard Specification, August 1992. Edited by François Raab.

[28] M. Crovella, M. Taqqu, and A. Bestavros. *In A Practical Guide To Heavy Tails*, chapter 1 (Heavy-Tailed Probability Distributions in the World Wide Web). Chapman & Hall, 1998.

[29] H. J. Curnow and B. A. Wichmann. A Synthetic Benchmark. *The Computer Journal*, 19(1):43–49, February 1976.

[30] I. Dagan and S. Engelson. Committee-based Sampling for Training Probabilistic Classifiers. In *Proceedings of the International Conference on Machine Learning*, July 1995.

[31] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, March 2003.

[32] R. Doyle, J. Chase, S. Gadde, and A. Vahdat. The Trickle-Down Effect: Web Caching and Server Request Distribution. In *Proceedings of the International Workshop on Web Caching and Content Delivery*, June 2001.

[33] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the Symposium on Operating Systems Principles*, October 2003.

[34] L. Eeckhout, H. Vandierendonck, and K. Bosschere. Quantifying the Impact of Input Data Sets on Program Behavior and its Applications. *Journal of Instruction-Level Parallelism*, 5:1–33, February 2003.

[35] D. Ellard and M. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the Annual Large Installation System Administration Conference*, October 2003.

[36] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps. In *Proceedings of the FREENIX Technical Conference*, June 2003.

[37] W. Elwasif, J. Plank, and R. Wolski. Data Staging Effects in Wide Area Task Farming Applications. In *Proceedings of the International Symposium on Cluster Computing and the Grid*, May 2001.

[38] B. A. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the International Symposium on Computer Architecture*, June 2001.

[39] I. Foster and C. Kesselman. The Grid2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 2004.

[40] A. Fox and D. Patterson. Self-Repairing Computers. *Scientific American*, June 2003.

[41] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of the Symposium on Operating Systems Principles*, October 2003.

[42] S. Gaede. Perspectives on the SPEC SDET Benchmarks. `http://www.spec.org/osg/sdm91/sdet/index.html`.

[43] Cyberstructure for the Geosciences. `www.geongrid.org`.

[44] S. Gold. Defects in SFS 2.0 Which Affect the Working-Set, July 2001. `http://www.spec.org/osg/sfs97/sfs97_defects.html`.

[45] M. Goldszmidt, I. Cohen, A. Fox, and S. Zhang. Three research challenges at the intersection of machine learning, statistical induction, and systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, June 2005.

[46] J. Gray. Distributed Computing Economics. IEEE TFCC Newsletter http://www.clustercomputing.org/content/tfcc-5-1-gray.html, March 2003.

[47] J. Gray, D. Liu, M. Nieto-Santisteban, A. Szalay, G. Heber, and D. DeWitt. Scientific Data Management in the Coming Decade. Technical Report MSR-TR-2005-10, Microsoft Research, January 2005.

[48] Grid Physics Network. `www.griphyn.org`.

[49] L. Grit, D. Irwin, V. Marupadi, P. Shivam, A. Yumerefendi, J. Chase, and J. Albrecht. Harnessing Virtual Machine Resource Control for Job Management. In *Workshop on System-level Virtualization for High Performance Computing, held in conjunction with EuroSys 2007*, March 2007.

[50] L. Haas, M. Carey, M. Livny, and A. Shukla. Seeking the Truth About Ad Hoc Join Costs. *International Journal on Very Large Data Bases*, 3(6):241–256, 1997.

[51] Homeland Security Computers Hacked, September 2007. http://www.nationalterroralert.com/updates/2007/09/25/homeland-securit%y-computers-hacked/.

[52] Z. He, B. Lee, and R. Snapp. Self-Tuning UDF Cost Modeling Using the Memory-Limited Quadtree. In *Proceedings of the International Conference on Extending DataBase Technology*, March 2004.

[53] C. R. Hicks and J. Kenneth V. Turner. *Fundamental Concepts in the Design of Experiments*. Oxford University Press, 1999.

[54] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical report, IBM Corp., 2001. http://www.research.ibm.com/autonomic.

[55] S. A. Huettel, A. W. Song, and G. McCarthy. *Functional Magnetic Resonance Imaging*. Sinauer Associates, Inc., 2004.

[56] Y. Ioannidis, M. Livny, A. Ailamaki, A. Narayanan, and A. Therber. Zoo: A Desktop Experiment Management Environment. In *Proceedings of the SIGMOD International Conference on Management of Data*, June 1997.

[57] E. Ïpek, S. McKee, et al. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, October 2006.

[58] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing Networked Resources with Brokered Leases. In *Proceedings of the USENIX Annual Technical Conference*, June 2006.

[59] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, May 1991.

[60] C. Karamanolis, M. Karlsson, and X. Zhu. Designing Controllable Computer Systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, June 2005.

[61] J. Katcher. Postmark: A New File System Benchmark. Technical Report 3022, Network Appliance, Oct 1997.

[62] Y.-S. Kee, H. Casanova, and A. Chien. Realistic Modeling and Synthesis of Resources for Computational Grids. In *Proceedings of the International Conference on Supercomputing*, November 2004.

[63] K. Keeton, T. Kelly, A. Merchant, C. Santos, J. Wiener, X. Zhu, and D. Beyer. Don't settle for less than the best: use optimization to make decisions. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[64] B. Keith and M. Wittle. LADDIS: The Next Generation in NFS File Server Benchmarking. In *Proceedings of the USENIX Annual Technical Conference*, June 1993.

[65] T. Kelly. Utility-Directed Allocation. In *Proceedings of the Workshop on Algorithms and Architectures for Self-Managing Systems*, July 2003.

[66] T. Kelly. Detecting Performance Anomalies in Global Applications. In *Proceedings of the USENIX Workshop on Real, Large Distributed Systems*, December 2005.

[67] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003.

[68] T. Kosar and M. Livny. A Framework for Reliable and Efficient Data Placement in Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 65(10):1146–1157, 2005.

[69] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[70] D. Liu and M. Franklin. The Design of Griddb: A Data-Centric Overlay for the Scientific Grid. In *Proceedings of the International Conference on Very Large Data Bases*, September 2004.

[71] W.-Y. Loh. Regression Tree Models for Designed Experiments. *IMS Lecture Notes*, 49:210–228, 2006.

[72] R. L. Mason, R. F. Gunst, and J. L. Hess. *Statistical Design and Analysis of Experiments, with Applications to Engineering and Science*. Wiley-Interscience, 2003.

[73] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference*, January 1996.

[74] E. L. Miller and R. H. Katz. Input/Output Behavior of Supercomputing Applications. In *Proceedings of the International Conference on Supercomputing*, November 1991.

[75] J. C. Mogul. Brittle Metrics in Operating Systems Research. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, March 1999.

[76] D. C. Montgomery. *Design and Analysis of Experiments*. Wiley, 2000.

[77] J. Moore, J. Chase, K. Farkas, and P. Ranganathan. Data Center Workload Monitoring, Analysis, and Emulation. In *Proceedings of Computer Architecture Evaluation using Commercial Workloads*, February 2005.

[78] R. H. Myers and D. C. Montgomery. *Response Surface Methodology: Process and Product in Optimization Using Designed Experiments*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[79] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous Resource Monitoring for Self-predicting DBMS. In *Proceedings of the International Symposium on Modeling, Analysis, and Simulation*, September 2005.

[80] Netperf: A Network Performance Benchmark. `http://www.cup.hp.com/netperf/NetperfPage.html`.

[81] W. T. Ng, B. Hillyer, E. Shriver, E. Gabber, and B. Ozden. Obtaining High Performance for Storage Outsourcing. In *Proceedings of the Conference on File and Storage Technologies*, January 2002.

[82] R. J. Niemiec. *Oracle9i Performance Tuning Tips & Techniques*. McGraw-Hill Osborne Media, 2003.

[83] Engineering Statistics Handbook. `http://www.itl.nist.gov/div898/handbook/index.htm`.

[84] National Laboratory for Applied Network Research (NLANR). `http://moat.nlanr.net`.

[85] G. W. Oehlert. *A First Course in Design and Analysis of Experiments*. W. H. Freeman, 2000.

[86] T. Osogami and S. Kato. Optimizing System Configurations Quickly by Guessing at the Performance. In *Proceedings of the SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2007.

[87] T. T. Osugi. Exploration-Based Active Machine Learning. Master's thesis, University of Nebraska, August 2005.

[88] B. K. Pasquale and G. C. Polyzos. A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload. In *Proceedings of the International Conference on Supercomputing*, November 1993.

[89] T. Phan, K. Ranganathan, and R. Sion. Evolving Toward the Perfect Schedule: Co-scheduling Job Assignments and Data Replication in Wide-Area Systems Using a Genetic Algorithm. In *Proceedings of the International Workshop on*

Job Scheduling Strategies for Parallel Processing, volume 3834, pages 173–193, June 2005.

[90] J. C. Phillips, R. Braun, et al. Scalable Molecular Dynamics with NAMD. *Journal of Computational Chemistry*, 26:1781–1802, 2005.

[91] J. Pormann, J. Board, D. Rose, and C. Henriquez. Large-Scale Modeling of Cardiac Electrophysiology. In *Proceedings of Computers in Cardiology*, September 2002.

[92] R. Powers, M. Goldszmidt, and I. Cohen. Short term performance forecasting in enterprise systems. In *Proceedings of the SIGKDD International Conference on Knowledge Discovery in Data Mining*, August 2005.

[93] D. A. Reed, L. Ramakrishnan, et al. Service-Oriented Environments in Research and Education for Dynamically Interacting with Mesoscale Weather. In *Proceedings of Computing in Science and Engineering*, Nov-Dec 2005.

[94] C. Roadknight, I. Marshall, and D. Vearer. File Popularity Characterization. In *Proceedings of the Workshop on Internet Server Performance*, May 1999.

[95] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante. Models of Parallel Applications with Large Computation and I/O Requirements. *IEEE Transactions on Software Engineering*, 28(3):286–307, 2002.

[96] N. Roy and A. McCallum. Toward Optimal Active Learning Through Sampling Estimation of Error Reduction. In *Proceedings of the International Conference on Machine Learning*, June-July 2001.

[97] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-based Mail Service. In *Proceedings of the Symposium on Operating Systems Principles*, pages 1–15, December 1999.

[98] Performance Monitoring Tools for Linux. `http://perso.wanadoo.fr/sebastien.godard`.

[99] Sloan Digital Sky Survey. `www.sdss.org`.

[100] Business Internet Group of San Francisco, The Black Friday Report on Web Application Integrity, 2004. `http://www.tealeaf.com/downloads/news/analyst_report/BIG-SF_BlackFrida%yReport.pdf`.

[101] S. Shankar, A. Kini, D. DeWitt, and J. Naughton. Integrating Databases and Workflow Systems. *SIGMOD Record*, 3(34):5–11, 2005.

[102] P. Shivam, S. Babu, and J. Chase. Learning Application Models for Utility Resource Planning. In *Proceedings of the International Conference on Autonomic Computing*, June 2006.

[103] C. Small, N. Ghosh, H. Saleed, M. Seltzer, and K. Smith. Does Systems Research Measure Up, November 1997.

[104] K. A. Smith. *Workload-Specific File System Benchmarks*. PhD thesis, Harvard University, Cambridge, MA, January 2001.

[105] C. Stewart and K. Shen. Performance Modeling and System Management for Multi-component Online Services. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, May 2005.

[106] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the USENIX Annual Technical Conference*, June 2001.

[107] D. Tang and M. Seltzer. Lies, Damned Lies, and File System Benchmarks. In *VINO: The 1994 Fall Harvest*. Harvard Division of Applied Sciences Technical Report TR-34-94, December 1994.

[108] V. Taylor, X. Wu, and R. Stevens. Prophesy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications. In *Proceedings of ACM Sigmetrics Performance Evaluation Review*, March 2003.

[109] TPC-W: e-Commerce Benchmark. `http://www.tpc.org/tpcw/`.

[110] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A Nine Year Study of File System and Storage Benchmarking. Technical Report FSL-07-01, Computer Science Department, Stony Brook University, May 2007.

[111] TurboTax Outage, April 2007. `http://news.com.com/TurboTax+e-filing+woes+draw+customer+ire/2100-1038%\_3-6177341.html`.

[112] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An Analytical Model for Multi-tier Internet Services and its Applications. In *Proceedings of the SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2005.

[113] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, December 2002.

[114] S. Vazhkudai and J. M. Schopf. Using Regression Techniques to Predict Large Data Transfers. *International Journal of High Performance Computing Applications*, cs.DC/0304037, 2003.

[115] Christmas shopping crush stalls Walmart.com, November 2006. http://news.zdnet.co.uk/internet/0,1000000097,39284866,00.htm.

[116] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. *Proceedings of the International Conference on Very Large Data Bases*, August 2002.

[117] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A Platform for System Software Benchmarking. In *Proceedings of the USENIX Annual Technical Conference*, April 2005.

[118] J. Xu, S. Adabala, and J. A. B. Fortes. Towards Autonomic Virtual Applications in the In-VIGO System. In *Proceedings of the International Conference on Autonomic Computing*, May 2004.

[119] L. T. Yang, X. Ma, and F. Mueller. Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution. In *Proceedings of the International Conference on Supercomputing*, November 2005.

[120] J. J. Yi, D. J. Lilja, and D. M. Hawkins. A Statistically Rigorous Approach for Improving Simulation Methodology. In *Proceedings of International Symposium on High-Performance Computer Architecture*, February 2003.

[121] L. Yin, S. Uttamchandani, and R. Katz. An Empirical Exploration of Black-Box Performance Models for Storage Systems. In *Proceedings of the International Symposium on Modeling, Analysis, and Simulation*, September 2006.

[122] A. Yumerefendi, P. Shivam, D. Irwin, P. Gunda, L. Grit, A. Demberel, J. Chase, and S. Babu. Towards an Autonomic Computing Testbed. In *Proceedings of the Workshop on Hot Topics in Autonomic Computing*, June 2007.

[123] N. Zhang, P. Hass, V. Josifovski, G. Lohman, and C. Zhang. Statistical Learning Techniques for Costing XML Queries. In *Proceedings of the International Conference on Very Large Data Bases*, Aug-Sep 2005.

[124] Q. Zhu and P. Larson. Building Regression Cost Models for Multidatabase Systems. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, December 1996.

Biography

Piyush Shivam was born in Moradabad, India in 1977. From 1995 to 1999 he attended the Birla Institute of Technology and Science (BITS), Pilani, India and received his undergraduate degree in Information Systems. He attended The Ohio State University from 2000 to 2002 for his MS in Computer and Information Science. From 2002 to 2007 he attended graduate school at Duke University, receiving his Ph.D. in Computer Science.

His research addresses the challenges in managing computer systems across a range of system domains. His work includes developing network protocols for high performance computing, end-to-end performance modeling of computer applications and systems, and exploring the role of statistical learning techniques for automated system management. He is the author of several refereed conference and workshop papers.