Cutting Corners: Workbench Automation for Server Benchmarking

Piyush Shivam[†] Varun Marupadi[+] Jeff Chase[+] Thileepan Subramaniam[+] Shivnath Babu[+]

[†]*Sun Microsystems*
piyush.shivam@sun.com

[+]*Duke University*
{*varun,chase,thilee,shivnath*}*@cs.duke.edu*

Abstract

A common approach to benchmarking a server is to measure its behavior under load from a workload generator. Often a set of such experiments is required—perhaps with different server configurations or workload parameters—to obtain a statistically sound result for a given benchmarking objective.

This paper explores a framework and policies to conduct such benchmarking activities automatically and efficiently. The workbench automation framework is designed to be independent of the underlying benchmark harness, including the server implementation, configuration tools, and workload generator. Rather, we take those mechanisms as given and focus on automation policies within the framework.

As a motivating example we focus on rating the peak load of an NFS file server for a given set of workload parameters, a common and costly activity in the storage server industry. Experimental results show how an automated workbench controller can plan and coordinate the benchmark runs to obtain a result with a target threshold of confidence and accuracy at lower cost than scripted approaches that are commonly practiced. In more complex benchmarking scenarios, the controller can consider various factors including accuracy vs. cost tradeoffs, availability of hardware resources, deadlines, and the results of previous experiments.

1 Introduction

David Patterson famously said:

> For better or worse, benchmarks shape a field.

Systems researchers and developers devote a lot of time and resources to running benchmarks. In the lab, they give insight into the performance impacts and interactions of system design choices and workload characteristics. In the marketplace, benchmarks are used to evaluate competing products and candidate configurations for a target workload.

The accepted approach to benchmarking network server software and hardware is to configure a system and subject it to a stream of request messages under controlled conditions. The workload generator for the server benchmark offers a selected mix of requests over a test interval to obtain an aggregate measure of the server's response time for the selected workload. Server benchmarks can drive the server at varying load levels, e.g., characterized by request arrival rate for open-loop benchmarks [21]. Many load generators exist for various server protocols and applications.

Server benchmarking is a foundational tool for progress in systems research and development. However, server benchmarking can be costly: a large number of runs may be needed, perhaps with different server configurations or workload parameters. Care must be taken to ensure that the final result is statistically sound.

This paper investigates *workbench automation* techniques for server benchmarking. The objective is to devise a framework for an automated *workbench controller* that can implement various policies to coordinate experiments on a shared hardware pool or "workbench", e.g., a virtualized server cluster with programmatic interfaces to allocate and configure server resources [12, 27]. The controller plans a set of experiments according to some policy, obtains suitable resources at a suitable time for each experiment, configures the test harness (system under test and workload generators) on those resources, launches the experiment, and uses the results and workbench status as input to plan or adjust the next experiments, as depicted in Figure 1. Our goal is to choreograph a set of experiments to obtain a statistically sound result for a high-level objective at low cost, which may involve using different statistical thresholds to balance

This research was conducted while Shivam was a PhD student at Duke University. Subramaniam is currently employed at Riverbed Technologies. This research was funded by grants from IBM and the National Science Foundation through CNS-0720829, 0644106, and 0720829.

cost and accuracy for different runs in the set.

As a motivating example, this paper focuses on the problem of measuring the peak throughput attainable by a given server configuration under a given workload (the *saturation throughput* or *peak rate*). Even this relatively simple objective requires a costly set of experiments that have not been studied in a systematic way. This task is common in industry, e.g., to obtain a qualifying rating for a server product configuration using a standard server benchmark from SPEC, TPC, or some other body as a basis for competitive comparisons of peak throughput ratings in the marketplace. One example of a standard server benchmark is the SPEC SFS benchmark and its predecessors [15], which have been used for many years to establish NFSOPS ratings for network file servers and filer appliances using the NFS protocol.

Systems research often involves more comprehensive benchmarking activities. For example, *response surface mapping* plots system performance over a large space of workloads and/or system configurations. Response surface methodology is a powerful tool to evaluate design and cost tradeoffs, explore the interactions of workloads and system choices, and identify interesting points such as optima, crossover points, break-even points, or the bounds of the effective operating range for particular design choices or configurations [17]. Figure 2 gives an example of response surface mapping using the peak rate. The example is discussed in Section 2. Measuring a peak rate is the "inner loop" for this response surface mapping task and others like it.

This paper illustrates the power of a workbench automation framework by exploring simple policies to optimize the "inner loop" to obtain peak rates in an efficient way. We use benchmarking of Linux-based NFS servers with a configurable workload generator as a running example. The policies balance cost, accuracy, and confidence for the result of each test load, while meeting target levels of confidence and accuracy to ensure statistically rigorous final results. We also show how advanced controllers can implement heuristics for efficient response surface mapping in a multi-dimensional space of workloads and configuration settings.

2 Overview

Figure 1 depicts a framework for automated server benchmarking. An automated *workbench controller* directs benchmarking experiments on a common hardware pool (workbench). The controller incorporates policies that decide which experiments to conduct and in what order, based on the following considerations:

- **Objective**. The controller pursues benchmarking objectives specified by a user. A simple goal might be to obtain a standard NFSOPS rating for a given NFS filer configuration. More complex goals might involve varying the workload or mapping a response surface for different workloads or server configurations. The goals may also specify the response time metric used to obtain the peak rate, and/or thresholds for confidence and accuracy. An objective that we consider is to obtain peak rates with 90% accuracy. An alternative might be to obtain the most complete and/or accurate results achievable within some deadline.
- **Resources**. The controller runs experiments as resources become available. It may tailor the runs to the available resources or schedule multiple runs concurrently.
- **Previous results**. The controller is feedback-driven in that it may consider results of previous runs in designing new experiments. For example, policies in this paper consider the variance of response times at a given test load to determine how many trials are needed to obtain a sound result. The controller can also use results of previous runs to prune the sample space in mapping a response surface.

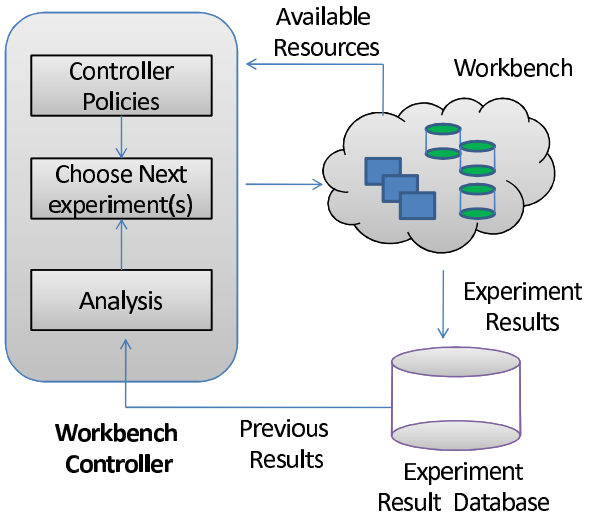

Figure 1: Automated Workbench and Controller.

$\vec{W}$	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 NFS server I/O daemons (nfsds), type of file system, block size

Table 1: Some workload and configuration factors that affect NFS file server performance.

We characterize the benchmark performance of a server by its *peak rate* or *saturation throughput*, denoted λ^*. λ^* is the highest request arrival rate λ that does not

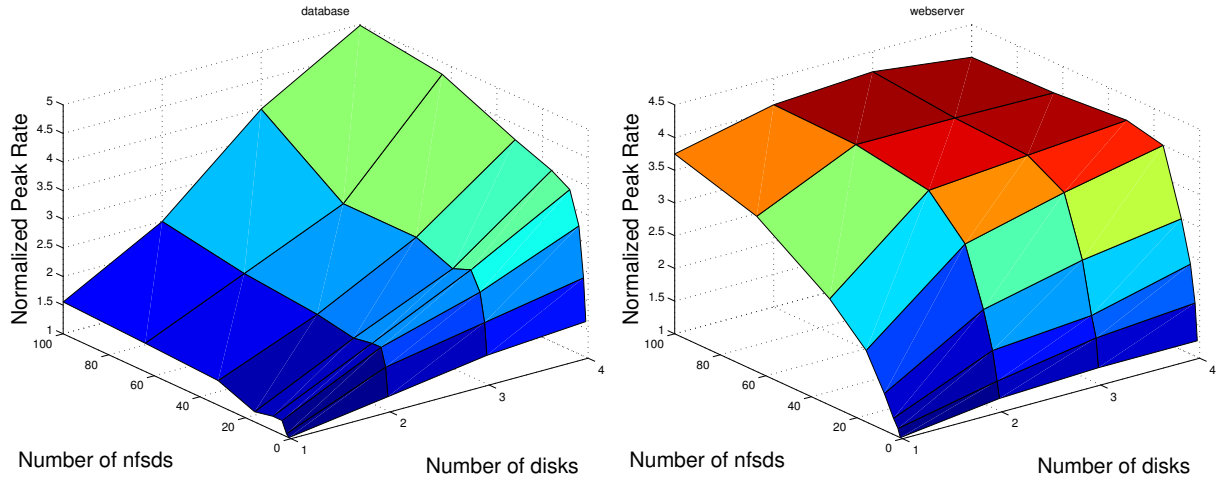

Figure 2: These surfaces illustrate how the peak rate, λ^*, changes with number of disks and number of NFS daemon (nfsd) threads for two canned *fstress* workloads (**DB_TP** and **Web server**) on Linux-based NFS servers. The workloads for this example are described in more detail later, in Table 3.

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, indicating that the offered load has reached the maximum that the server can process effectively.

The performance of a 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 metrics or *factors*, as summarized in Table 1.

Workload $\vec{W}$. Workload factors define the properties of the request mix and the data sets they operate on, and other workload characteristics.

Configurations ($\vec{C}$). The controller may vary server configuration parameters (e.g., buffer sizes, queue bounds, concurrency levels) before it instantiates the server for each run.

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. The prototype can instantiate Xen virtual machines sized along the memory, CPU, and I/O dimensions. The experiments in this paper vary the workload and configuration parameters on a fixed set of Linux server configurations in the workbench.

2.1 Example: NFS Server Benchmarking

This paper uses NFS server benchmarking as a running example. The controllers use a configurable synthetic NFS workload generator called *Fstress* [1], which was developed in previous research. Fstress offers knobs for various workload factors ($\vec{W}$), enabling the controller to configure the properties of the workload's dataset and its request mix to explore a space of NFS workloads. Fstress has preconfigured parameter sets that represent standard NFS file server workloads (e.g., SPECsfs97, Postmark), as well as many other workloads that might be encountered in practice (see Table 3).

Figure 2 shows an example of response surfaces produced by the automated workbench for two canned NFS server workloads representing typical request mixes for a file server that backs a database server (called **DB_TP**) and a static Web server (**Web server**). A response surface gives the response of a metric (peak rate) to changes in the operating range of combinations of factors in a system [17]. In this illustrative example the factors are the number of NFS server daemons (nfsds) and disk spindle counts.

Response surface mapping can yield insights into the performance effects of configuration choices in various settings. For example, Figure 2 confirms the intuition that adding more disks to an NFS server can improve the peak rate only if there is a sufficient number of nfsds to issue requests to those disks. More importantly, it also reveals that the ideal number of nfsds is workload-dependent: standard rules of thumb used in the field are not suitable for all workloads.

2.2 Problem Statement

The challenge for the automated feedback-driven workbench controller is to design a set of experiments to obtain accurate peak rates for a set of test points, and in particular for test points selected to approximate a response surface efficiently.

Response surface mapping is expensive. Algorithm 1 presents the overall benchmarking approach that is used by the workbench controller to map a response surface, and Table 2 summarizes some relevant notation. The overall approach consists of an outer loop that iterates over selected samples from $\langle F_1, \ldots, F_n \rangle$, 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. For each test load λ, the controller must choose the *runlength* r or observation interval, and the *number of independent trials* t to obtain a response time measure under load λ.

The goal of the automated feedback-driven controller is to address the following problems.

1. **Find Peak Rate** (§3). For a given sample from the outer loop of Algorithm 1, minimize the benchmarking cost for finding the peak rate λ^* subject to a target confidence level c and target accuracy a (defined below). Determining the NFSOPS rating of an NFS filer is one instance of this problem.

2. **Map Response Surface** (§4). Minimize the total benchmarking cost to map a response surface for all $\langle F_1, \ldots, F_n \rangle$ samples in the outer loop of Algorithm 1.

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 to the peak rate. Sections 3 and 4 present algorithms that the controller uses to address these problems.

2.3 Confidence and Accuracy

Benchmarking can never produce an exact result because complex systems exhibit inherent variability in their behavior. The best we can do is to make a *probabilistic claim* about the *interval* in which the "true" value for a metric lies based on measurements from multiple independent trials [13]. Such a claim can be characterized by a *confidence level* and the *confidence interval* at this confidence level. For example, by observing the mean response time $\bar{R}$ at a test load λ for 10 independent trials, we may be able to claim that we are 95% confident (the confidence level) that the correct value of $\bar{R}$ for that λ lies within the range $[25ms, 30ms]$ (the confidence interval).

Basic statistics tells us how to compute confidence intervals and levels from a set of trials. For example, if the mean server response time $\bar{R}$ from t trials is μ, and standard deviation is σ, then the confidence interval for μ at confidence level c is given by:

$$[\mu - \frac{z_c \sigma}{\sqrt{t}}, \ \ \mu + \frac{z_c \sigma}{\sqrt{t}}] \qquad (1)$$

z_c is a reading from the table of standard normal distribution for confidence level c. If $t <= 30$, then we use *Student's t* distribution instead after verifying that the t runs come from a normal distribution [13].

The tightness of the confidence interval captures the *accuracy* of the true value of the metric. A tighter bound implies that the mean response time from a set of trials is closer to its true value. For a confidence interval $[low, high]$, we compute the percentage accuracy as:

$$accuracy = 1 - error = (1 - \frac{high - low}{high + low}) \qquad (2)$$

λ^*	Peak rate for a given server configuration and workload.
λ	Offered load (arrival rate) for a given test load level.
ρ	Load factor $= \lambda/\lambda^*$ for a test load λ.
R	Mean server response time for a test load.
R_{sat}	Threshold for R at the peak rate: the server is saturated if $\bar{R} > R_{sat}$.
s	Factor that determines the width of the peak-rate region $[R_{sat} \pm sR_{sat}]$ (§3.3).
a	Target *accuracy* (based on confidence interval width) for the estimated value of λ^* (§2.3).
c	Target *confidence level* for the estimated λ^* (§2.3).
t	Number of independent trials at a test load.
r	Runlength: the test interval over which to observe the server latency for each trial.

Table 2: Benchmarking parameters used in this paper.

3 Finding the Peak Rate

In the inner loop of Algorithm 1, the automated controller searches for the peak rate λ^* for some workload and configuration given by a selected sample of factor values in $\langle F_1, \ldots, F_n \rangle$. To find the peak rate it subjects the server to a sequence of test loads $\lambda = [\lambda_1, \ldots, \lambda_l]$. The sequence of test loads should converge on an estimate of the peak rate λ^* that meets the target accuracy and confidence.

We emphasize that this step is itself a common benchmarking task to determine a standard rating for a server configuration in industry (e.g., SPECsfs [6]).

3.1 Strawman: Linear Search with Fixed r and t

Common practice for finding the peak rate is to script a sequence of runs for a standard workload at a fixed linear sequence of escalating load levels, with a preconfigured runlength r and number of trials t for each load level. The algorithm is in essence a linear search for the peak rate: it starts at a default load level and increments the load level (e.g., arrival rate) by some fixed increment until it drives the server into saturation. The last load level λ before saturation is taken as the peak rate λ^*. We refer to this algorithm as *strawman*.

Strawman is not efficient. If the increment is too small, then it requires many iterations to reach the peak rate. Its cost is also sensitive to the difference between the peak rate and the initial load level: more powerful server configurations take longer to benchmark. A larger increment

Algorithm 1: 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 trials t for load λ;
      Choose runlength r for each trial;
      Configure server and workload generator for the sample; Run t independent trials 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

can converge on the peak rate faster, but then the test may overshoot the peak rate and compromise accuracy. In addition, *strawman* misses opportunities to reduce cost by taking "rough" readings at low cost early in the search, and to incur only as much cost as necessary to obtain a statistically sound reading once the peak rate is found.

A simple workbench controller with feedback can improve significantly on the strawman approach to searching for the peak rate. To illustrate, Figure 3 depicts the search for λ^* for two policies conducting a sequence of experiments, with no concurrent testing. For *strawman* we use runlength r = 5 minutes, $t = 10$ trials, and a small increment to produce an accurate result. The figure compares *strawman* to an alternative that converges quickly on the peak rate using binary search, and that adapts r and t dynamically to balance accuracy, confidence, and cost during the search. The figure represents the sequence of actions taken by each policy with cumulative benchmarking time on the x-axis; the y-axis gives the load factor $\rho = \frac{\lambda}{\lambda^*}$ for each test load evaluated by the policies. The figure shows that *strawman* can incur a much higher benchmarking cost (time) to converge to the peak rate and complete the search with a final accurate reading at load factor $\rho = 1$. The strawman policy not only evaluates a large number of test loads with load factors that are not close to 1, but also incurs unnecessary

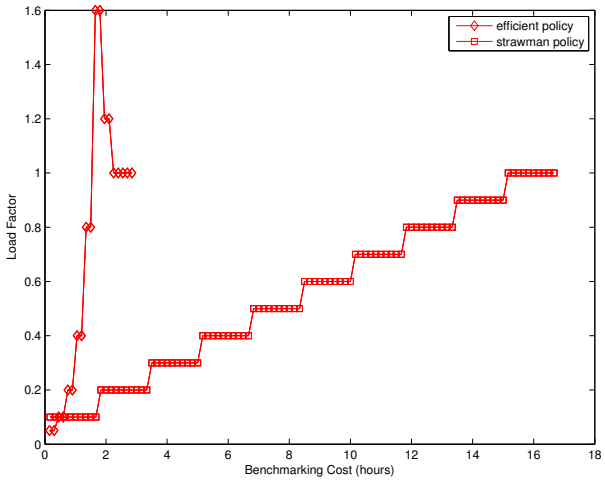

Figure 3: An efficient policy for finding peak rate converges quickly to a load factor near 1, and reduces benchmarking cost by obtaining a high-confidence result only for the load factor of 1. It is significantly less costly than the *strawman* policy: a linear search with a fixed runlength and fixed number of trials per test load.

cost at each load.

The remainder of the paper discusses the improved controller policies in more detail, and their interactions with the outer loop in mapping response surfaces.

3.2 Choosing r and t for Each Test Load

The runlength r and the number of trials t together determine the benchmarking cost incurred at a given test load λ. The controller should choose r and t to obtain the confidence and accuracy desired for each test load at least cost. The goal is to converge quickly to an accurate reading at the peak rate: $\lambda = \lambda^*$ and load factor $\rho = 1$. High confidence and accuracy are needed for the final test load at $\lambda = \lambda^*$, but accuracy is less crucial during the search for the peak rate. Thus the controller has an opportunity to reduce benchmarking cost by adapting the target confidence and accuracy for each test load λ as the search progresses, and choosing r and t for each λ appropriately.

At any given load level the controller can trade off confidence and accuracy for lower cost by decreasing either r or t or both. Also, at a given cost any given set of trials and runlengths can give a high-confidence result with wide confidence intervals (low accuracy), or a narrower confidence interval (higher accuracy) with lower confidence.

However, there is a complication: performance variability tends to increase as the load factor ρ approaches saturation. Figure 4 and Figure 5 illustrate this effect. Figure 4 is a scatter plot of mean server response time ($\bar{R}$) at different test loads λ for five trials at each load. Note that the variability across multiple trials increases

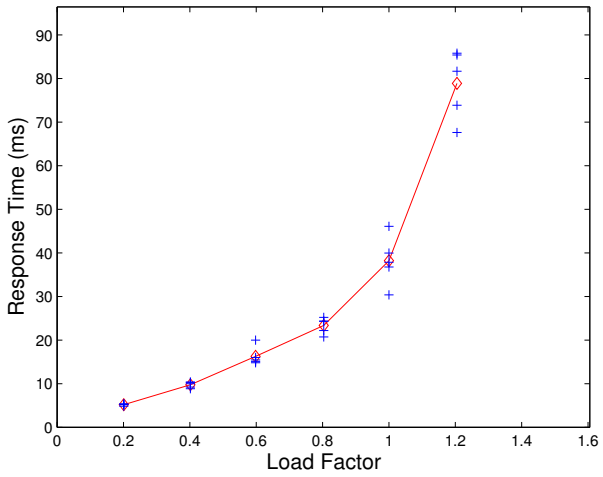

Figure 4: Mean server response time at different test loads for the **DB_TP** *fstress* workload using 1 disk and 4 NFS daemon (nfsd) threads for the server. The variability in mean server response time for multiple trials increases with load. The results are representative of other server configurations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

as $\lambda \rightarrow \lambda^*$ and $\rho \rightarrow 1$. Figure 5 shows a scatter plot of $\bar{R}$ measures for multiple runlengths at two load factors, $\rho = 0.3$ and $\rho = 0.9$. Longer runlengths show less variability at any load factor, but for a given runlength, the variability is higher at the higher load factor. Thus the cost for any level of confidence and/or accuracy also depends on load level: since variability increases at higher load factors, it requires longer runlengths r and/or a larger number of trials t to reach a target level of confidence and accuracy.

For example, consider the set of trials plotted in Figure 5. 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 $5 < \bar{R} < 6.5$. From the data we can determine the runlength needed to achieve target confidence and accuracy at this load level and number of trials t: 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. 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). As a result, we must increase the runlength and/or the number of trials to maintain target levels of confidence and accuracy as load factors increase. For example, we need a runlength of 120 seconds or more to achieve accuracy $\geq 87\%$ at 95% confidence for this number of trials at load factor 0.9.

Figure 6 quantifies the tradeoff between the runlength and the number of trials required to attain a target accuracy and confidence for different workloads and load factors. It shows the number of trials 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 shorter runlengths. It also shows a sweet spot for the runlengths that reduces the number of trials needed. A controller can use such curves as a guide to pick a suitable runlength r and number of trials t with low cost.

Algorithm 2: 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 3 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 $\lambda_{cur} = \lambda$;
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
$\lambda^* = \lambda_{cur}$;
Compute accuracy a_{λ^*} at confidence c;
end
end

3.3 Search Algorithm

Our approach uses Algorithm 2 to search for the peak rate for a given setting of factors.

Algorithm 2 takes various parameters to define the conditions for the reported peak rate:

- R_{sat}, a threshold on the mean server response time. The server is considered to be saturated if mean response time exceeds this threshold, i.e., $\bar{R} > R_{sat}$.
- P_{sat} and L_{sat} defining a threshold on percentile server response time. The server is considered to be saturated if the P_{sat} percentile response time exceeds L_{sat}. For example, if $P_{sat} = 0.95$ then the

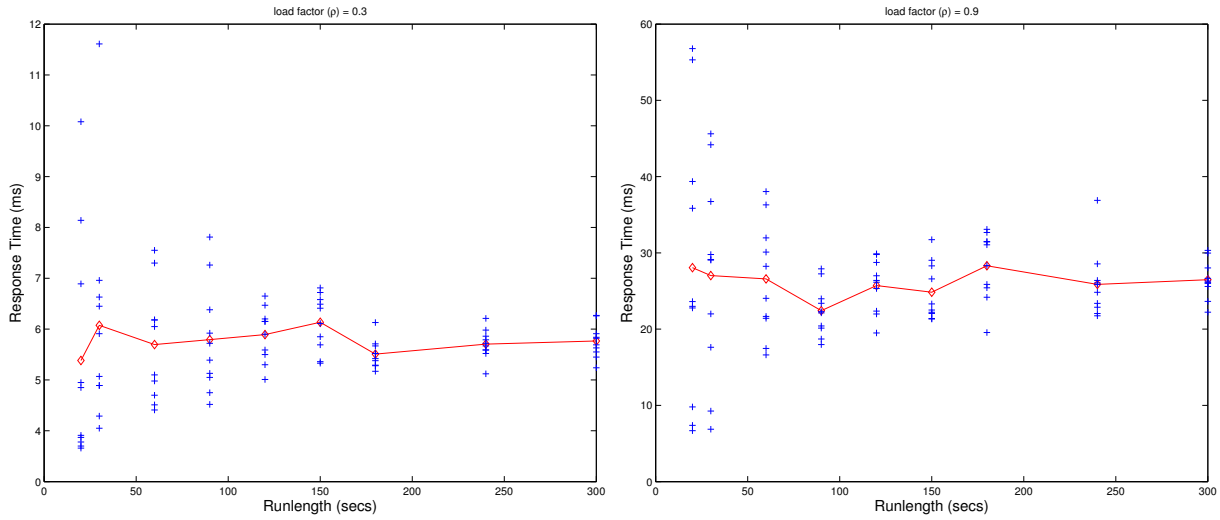

Figure 5: Mean server response time $\bar{R}$ at different workload runlengths for the **DB_TP** *fstress* workload using 1 disk and 4 NFS daemon (nfsd) threads for the server. The variability in mean server response time for multiple trials decreases with increase in runlength. The results are representative of other server configurations and workloads.

server is saturated if no more than 95% of responses show latency at or below L_{sat}. To simplify the presentation we use the R_{sat} threshold test on mean response time and do not discuss P_{sat} further.

- Width parameter s defining the *peak-rate region* $[R_{sat} \pm sR_{sat}]$. The reported peak rate λ^* can be any test load level that drives the mean server response time into 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 2 chooses (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. It automatically adapts the number of trials at any test load according to the load factor and the desired target confidence and accuracy. At each load level the algorithm conducts a small (often the minimum of two in our experiments) 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 just enough time to check whether it is in fact the peak rate.

More specifically, for each test load λ_{cur}, Algorithm 2 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 2.) Next, it checks if the confidence interval overlaps with the specified peak-rate region (Step 9).

If the regions overlap, then Algorithm 2 identifies the current test load λ_{cur} as an estimate of a potential peak rate with 95% confidence. 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 c (Section 2.1). If it reaches the target accuracy a, then the algorithm terminates (Step 4), otherwise it conducts more trials at the current test load (Step 6) to narrow the confidence interval, and repeats the threshold condition test. Thus the cost of the algorithm varies with the target confidence and accuracy.

If there is no overlap (Step 10), then Algorithm 2 moves on to the next test load. It uses any of several *load-picking* algorithms to generate the sequence of test loads, described in the rest of this section. All load-picking algorithms take as input the set of past test loads and their results. The output becomes the next test load in Algorithm 2. For example, Algorithm 3 gives a load-picking algorithm using a simple binary search.

To simplify the choice of runlength for each experiment at a test load (Step 5), Algorithm 2 uses the "sweet spot" derived from Figure 6 (Section 3.2). The figure shows that for all workloads that this paper considers, a runlength of 3 minutes is the sweet spot for the minimum number of trials.

3.4 The Binsearch Load-Picking Algorithm

Algorithm 3 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.

Binsearch allows the controller to find the lower and

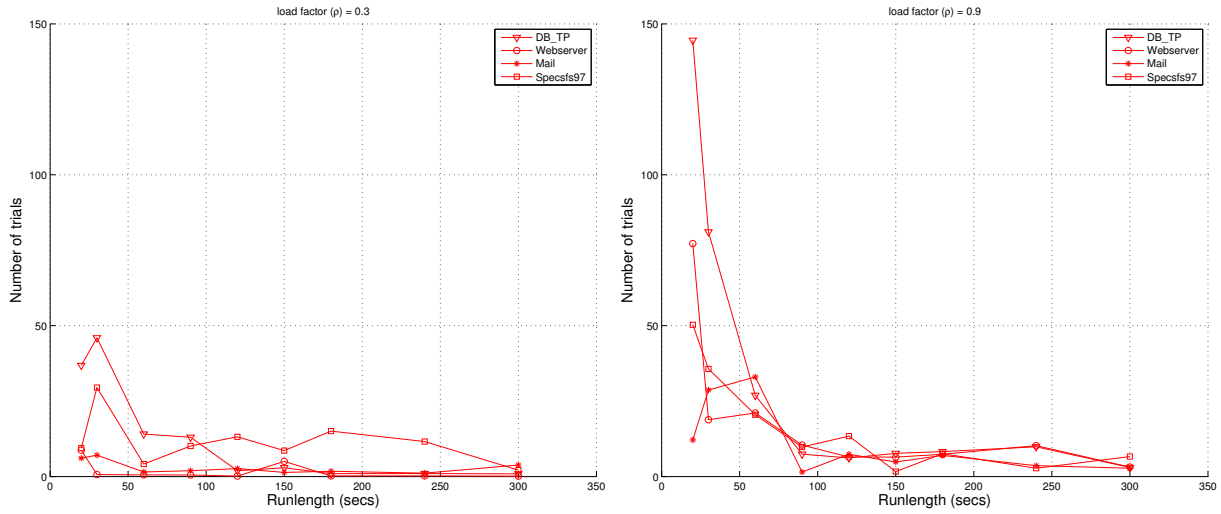

Figure 6: Number of trials to attain 90% accuracy for mean server response time at 95% confidence level at low and high load factors for different runlengths. The results are for server configuration with 1 disk and 4 nfsds, and representative of other server configurations.

Algorithm 3: 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 ($\lambda_{cur} == 0$);
   $\lambda_{next} = 50$ requests/sec;
   Phase = Geometric; Return λ_{next};
2) **Geometric Phase.**
   if (Phase == Geometric && $\bar{R}_{\lambda_{cur}} < R_{sat}$)
   Return $\lambda_{next} = \lambda_{cur} \times 2$;
   else
   binsearch$_{low} = \lambda_{prev}$, and Go to Step 3;
   end
3) **Binary Search Phase.**
   if ($\bar{R}_{\lambda_{cur}} < R_{sat}$) ;
   binsearch$_{low} = \lambda_{cur}$;
   else
   binsearch$_{high} = \lambda_{cur}$;
   end
   Return λ_{next} = (binsearch$_{high}$ + binsearch$_{low}$)/2;

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.

3.5 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.

3.6 Model-guided Load-Picking Algorithm

The general *shape* of the response-time vs. load curve is well known, and the form is similar for different workloads and server configurations. This suggests that a model-guided approach could fit the curve from a few test loads and converge more quickly to the peak rate. Using the insight offered by well-known open-loop queuing theory results [13], we experimented with a simple model to fit the curve: $R = 1/(a - b * \lambda)$, 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 $\langle \lambda, R_\lambda \rangle$.

Algorithm 4 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 should improve progressively, and the model should guide the search to an accurate peak rate. In many cases, this happens in a single iteration of model learning (Section 5).

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 gen-

Algorithm 4: 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 $\lambda_{next} = 50$ requests/sec;
   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 R_{sat}. Let the corresponding load be λ_i;
   Learn the model $R = 1/(a - b\lambda)$ with two tuples $\langle \lambda_{cur}, \bar{R}_{\lambda_{cur}} \rangle$ and $\langle \lambda_i, \bar{R}_i \rangle$;
   Return $\lambda_{next} = \frac{R_{sat}a-1}{R_{sat}b}$;

erated by previous model predictions. This creates the possibility of 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 $R = a + b\lambda + c\lambda^2$, which show similar limitations.

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 ongoing study. Section 5 reports our experience with the model-guided choice of test loads. Preliminary results suggest that the model-guided approaches are often superior but can be unstable depending on the initial samples used to learn the model.

3.7 Seeding Heuristics

The load-picking algorithms in Sections 3.5-3.6 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 1 are $\langle$ number of disks, number of nfsds $\rangle$ (as shown in Figure 2). 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, 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 load value close to $\lambda^*_{1,2}$.

In the common case, the peak rates for "nearby" samples will be close. If they are not, the load-picking algorithms 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 may use a default seed or an aggregate of peak rates from previous samples to start the search.

4 Mapping Response Surfaces

We now relate the peak rate algorithm that Section 3 describes to the larger challenge of mapping a peak rate response surface efficiently and effectively, based on Algorithm 1.

A large number of factors can affect performance, so it is important to sample the multi-dimensional space with care as well as to optimize the inner loop. 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 peak rate for each factor combination, then the total time for benchmarking is 130 days. An automated workbench controller can shorten this time by pruning the sample space, planning experiments to run on multiple hardware setups in parallel, and optimizing the inner loop.

We consider two specific challenges for mapping a response surface:

- Algorithm 2 from Section 3.3 is used for 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. Sections 3.4-3.7 describe the load-picking algorithms we consider.
- Algorithm 1 also depends on a policy to choose the samples in the outer loop. 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.

If the benchmarking objective is to understand the overall trend of how the peak rate is affected by 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 Algorithm 1 can leverage Response Surface Methodology (RSM) [17] to select the sample points efficiently (in Step 2). RSM is a branch of statistics that provides principled techniques to choose a set of samples to obtain good approximations of the overall response surface at low cost. For example, some RSM techniques assume 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. This approximation is a basis for selecting a minimal set of samples for the controller to obtain in order to learn a fairly accurate model (i.e., estimate values of the β parameters in the model). We evaluate one such RSM technique in Section 5.

It is important to note that these RSM techniques may reduce the effectiveness of the seeding heuristics described in Section 3.7. RSM techniques try to find sample points on the surface that will add the most information to the model. Intuitively, such samples are the ones that we have the least prior information about, and hence for which seeding from prior results would be least effective. We leave it to future work to explore the interactions of the heuristics for selecting samples efficiently and seeding the peak rate search for each sample.

5 Experimental Evaluation

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

Cost for Finding Peak Rate. Sections 3.3 and 4 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 search 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 5.1 presents the experiment setup. Section 5.2 presents the workloads that we use for evaluation. Section 5.3 evaluates our benchmarking methodology as described above.

5.1 Experimental Setup

Table 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 5.2). 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 in $\langle 1, 2, 3, 4 \rangle$, and a single factor in $\vec{C}$: the number of nfsd daemons for the NFS server chosen 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 2.6.18 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 3.4-4 to plot a response surface and/or search for the peak rate for a given sample with target confidence and accuracy.

5.2 Workloads

We use Fstress to generate $\vec{W}$ corresponding to four workloads as summarized in Table 3. A brief summary follows. Further details are in [1].

- **SPECsfs97**: The Standard Performance Evaluation Corporation introduced their System File Server benchmark (SPECsfs) [6] in 1992, derived from the earlier self-scaling LADDIS benchmark [15]. A recent (2001) revision corrected several defects identified in the earlier version [11].
- **Web server**: Several efforts (e.g., [2]) 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., [19, 8, 18, 9, 2]).
- **DB_TP**: We model our database workload after TPC-C [7], reading and writing within a few large files in a 2:1 ratio. I/O access patterns are random, with some short (256 KB) sequential asyn-

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

Table 3: Summary of *fstress* workloads used in the experiments.

chronous 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. [20].

5.3 Results

For evaluating the overall methodology and the policies outlined in Sections 3.3 and 4, we define the peak rate λ^* to be the test load that causes: (a) the mean server response time to be in the $[36, 44]$ ms region; or (b) the 95-percentile request response time to exceed 2000 ms to complete. We derive the $[36, 44]$ region by choosing mean server response time threshold at the peak rate R_{sat} to be 40 ms and the width factor $s = 10\%$ in Table 2. For all results except where we note explicitly, we aim for a λ^* to be accurate within 10% of its true value with 95% confidence.

5.3.1 Cost for Finding Peak Rate

Figure 7 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 4. 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 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 a load factor of 1 as compared to policies with seeding. All policies with seeding start at a 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 which to increase the test load. However, *Binsearch* jumps to the peak rate region in logarithmic number of steps. The *Model* policy is the quickest to jump near the load factor of 1, but incurs most of its cost there. This happens because the model learned is sufficiently accurate for guiding the search *near* the peak rate, but not accurate enough to search the peak rate quickly.

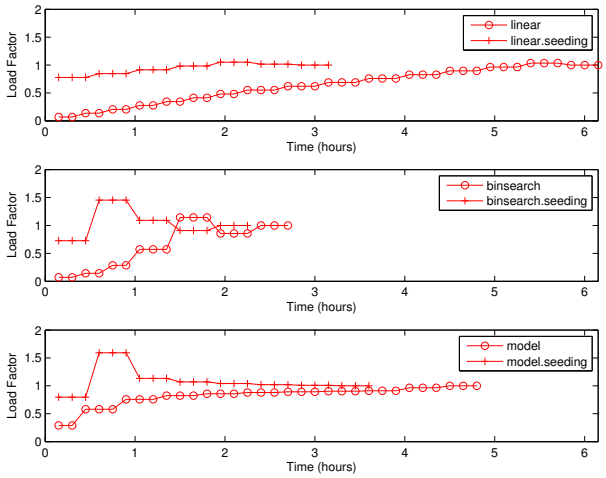

Figure 7: Time spent at each load factor for finding the peak rate for different policies for **DB_TP** with 4 disks and 32 nfsds. Seeded policies were seeded with the peak rate for 4 disks and 16 nfsds. The result is representative of other samples and workloads. All policies except linear quickly converge to the load factor of 1 and conduct more trials there to achieve the target accuracy and confidence.

5.3.2 Cost for Mapping Response Surfaces

Figure 8 compares the total normalized benchmarking cost for mapping the response surfaces for the three workloads using the policies outlined in Section 4. The costs are normalized with respect to the lowest total cost, which is 47 hours and 36 minutes taken by 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 unstable. This is not surprising given that the accuracy of the learned model guides the search. As Section 3.6 explains, if the model is inaccurate the search may converge slowly.

The linear policy is inefficient and 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. Note that for **Mail**, *Binsearch with Seeding* incurs a slightly higher cost than *Binsearch*. For some configurations, as Section 3.7 explains, seeding can incur additional cost to recover from a bad seed resulting in longer search times.

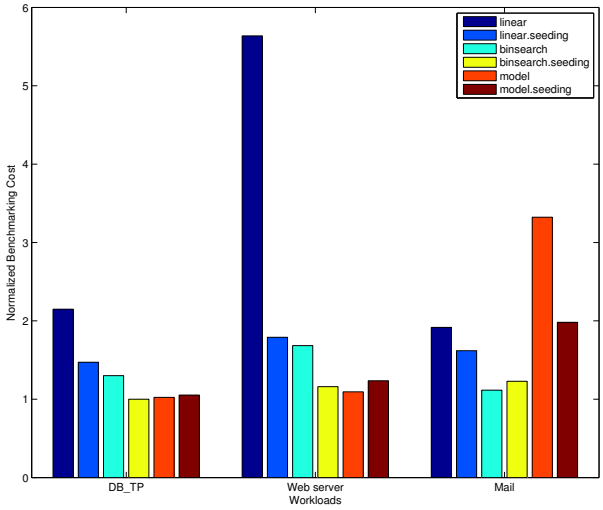

Figure 8: The total cost for mapping response surfaces for three workloads using different policies.

Reducing the Number of Samples. To evaluate the RSM approach presented in Section 4, we approximate the response surface by a quadratic curve in two dimensions: peak rate = func(number of disks, number of nfsds). We use a D-optimal design [17] 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 each.

After learning the equation, we use it to predict the peak rate at all the other samples in the surface. Table 4 presents the mean absolute percentage error in predicting the peak rate across all the samples. The results show that D-optimal designs do a very good job of picking appropriate samples, and that very little more can be learned by small increases in the number of points sampled. Improving the accuracy of the surface with limited numbers of sampled points is an area of ongoing research.

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

Table 4: Mean Absolute Prediction Error (MAPE) in Predicting the Peak Rate

5.3.3 Cost Versus Target Confidence and Accuracy

Figure 9 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%.

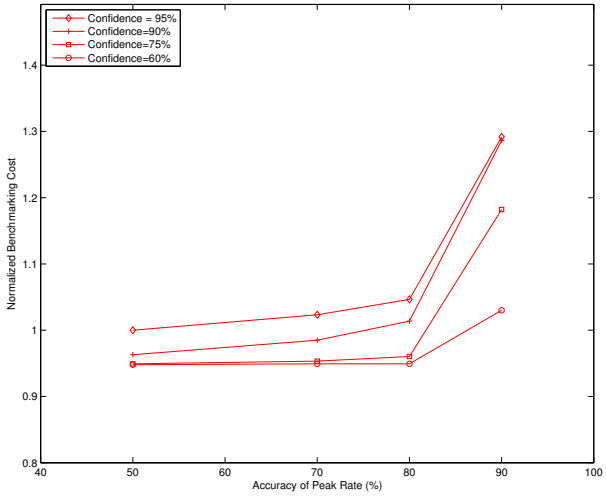

Figure 9: The total benchmarking cost adapts to the desired confidence and accuracy. The cost is shown for mapping the response surface for **DB_TP** using the *Binsearch* policy. Other workloads and policies show similar results.

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 2) by defining the peak rate region. For example, $s = 10\%$ 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 experiments so far, we fix $s = 10\%$.

Figure 10 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.

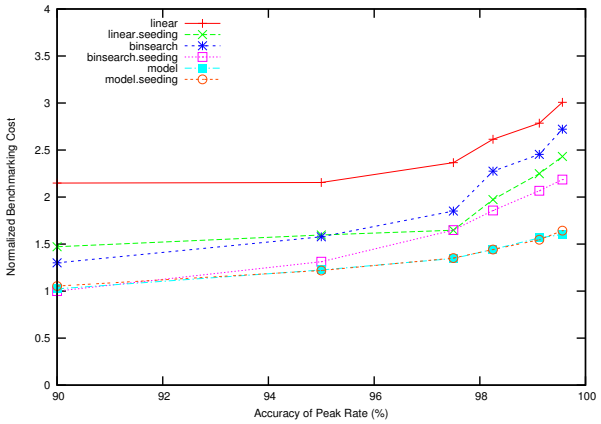

Figure 10: Benchmarking cost adapts to the target accuracy of the peak rate region for all policies. As the region narrows, the majority of the cost is incurred at or near the peak rate. Linear and Binsearch incur the same cost close to the peak rate, and hence their cost converges as they conduct more trials near the peak rate. The cost is shown for **DB_TP**. Other workloads show similar results.

6 Related Work

Several researchers have made a case for statistically significant results from system benchmarking, e.g., [4]. Auto-pilot [26] is a system for automating the benchmarking process: it supports various benchmark-related tasks and can modulate individual experiments to obtain a target confidence and accuracy. Our goal is to take the next step and focus on an automation framework and policies to orchestrate sets of experiments for a higher level benchmarking objective, such as evaluating a response surface or obtaining saturation throughputs under various conditions. We take the workbench test harness itself as given, and our approach is compatible with advanced test harnesses such as Auto-pilot.

While there are large numbers and types of benchmarks, (e.g., [5, 14, 3, 15]) that test the performance of servers 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. [25] present an exhaustive nine-year study of file system and storage benchmarking that includes benchmark comparisons, their pros and cons [22], and makes recommendations for systematic benchmarking methodology that considers a range of workloads for benchmarking the server. Smith et al. [23] make a case for benchmarks the capture composable elements of realistic application behavior. Ellard et al. [10] 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" [16] and "lies" [24] about a server performance.

Synthetic workloads emulate characteristics observed in real environments. They are often self-scaling [5], 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 [6] (and its predecessor LADDIS [15]) creates a set of files and applies a pre-defined mix of NFS operations. The experiments in this paper use Fstress [1], a synthetic, flexible, self-scaling NFS workload generator that can emulate a range of NFS workloads, including 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 [23] while at the same time allowing standardized comparisons across studies.

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. Relative to other methodologies such as application benchmarking, it reliably stresses the system under test to its saturation point where interesting performance behaviors may appear. In the storage arena, NFS server benchmarking is a powerful tool for investigation 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.

7 Conclusion

This paper focuses on the problem of workbench automation for server benchmarking. We propose an automated benchmarking system that plans, configures, and executes benchmarking experiments on a common hardware pool. The activity is coordinated by an automated controller that can consider various factors in planning, sequencing, and conducting experiments. These factors include accuracy vs. cost tradeoffs, availability of hardware resources, deadlines, and the results reaped from previous experiments.

We present efficient and effective controller policies

that 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, and this is a topic of ongoing study.

References

[1] 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, January 2002.

[2] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, April 1996.

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

[4] 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 10th conference on Hot Topics in Operating Systems*, June 2005.

[5] P. Chen and D. Patterson. A new approach to I/O performance evaluation—self-scaling I/O benchmarks, predicted I/O performence. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993.

[6] S. P. E. Corporation. SPEC SFS release 3.0 run and report rules, 2001.

[7] T. P. P. Council. TPC benchmark C standard specification, August 1992. Edited by François Raab.

[8] 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.

[9] R. Doyle, J. Chase, S. Gadde, and A. Vahdat. The trickle-down effect: Web caching and server request distribution. In *Proceedings of the Sixth International Workshop on Web Caching and Content Delivery*, June 2001.

[10] D. Ellard and M. Seltzer. NFS tricks and benchmarking traps. In *Proceedings of the FREENIX 2003 Technical Conference*, June 2003.

[11] S. Gold. Defects in SFS 2.0 which affect the working-set, July 2001. `http://www.spec.org/osg/sfs97/sfs97_defects.html`.

[12] D. Irwin, J. S. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing Networked Resources with Brokered Leases. In *Proc. of the USENIX Annual Technical Conf.*, Jun 2006.

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

[14] J. Katcher. Postmark: A new file system benchmark. Technical Report 3022, Network Appliance, October 1997.

[15] B. Keith and M. Wittle. LADDIS: The next generation in NFS file server benchmarking. In *Proceedings of the USENIX Annual Technical Conference*, June 1993.

[16] J. C. Mogul. Brittle metrics in operating systems research. In *Proceedings of the the 7th Workshop on Hot Topics in Operating Systems*, March 1999.

[17] 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.

[18] National laboratory for applied network research (NLANR). `http://moat.nlanr.net`.

[19] C. Roadknight, I. Marshall, and D. Vearer. File popularity characterisation. In *Proceedings of the 2nd Workshop on Internet Server Performance*, May 1999.

[20] Y. Saito, B. Bershad, and H. Levy. Manageability, availability and performance in Porcupine: A highly scalable, cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, December 1999.

[21] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *Proeceedings of the 3rd Symposium on Networked Systems Design and Implementation*, April 2006.

[22] C. Small, N. Ghosh, H. Saleed, M. Seltzer, and K. Smith. Does systems research measure up. Technical Report TR-16-97, Harvard University, Department of Computer Science, November 1997.

[23] K. A. Smith. *Workload-Specific File System Benchmarks*. PhD thesis, Harvard University, Cambridge, MA, January 2001.

[24] 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.

[25] 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.

[26] 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, FREENIX Track*, April 2005.

[27] 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.