Cutting Corners: Workbench Automation for Server Benchmarking

Piyush Shivam, Varun Marupadi, Jeff Chase,Thileepan Subramaniam, Shivnath Babu
{shivam,varun,chase,thilee,shivnath}@cs.duke.edu
Duke University
Durham NC

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 to gain 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 at some arrival rate (or test load) over a test interval to obtain an aggregate measure of the server's response time for the selected workload at that load level. A wide range of load generators exist for various server protocols and applications: when used correctly they are a foundational tool for progress in systems research and development [20].

Users of these benchmarks typically conduct a set of experiments to answer their specific questions. 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.

For example, one common goal is to find the peak throughput attainable by a given server configuration under a given set of workload conditions. That typically involves a set of runs with escalating load levels until the measured response time exceeds some defined threshold, indicating that the offered load has reached the *saturation throughput* or *peak rate* that the server can process. Organizations such as SPEC and TPC have defined standard *server benchmarks* and load generators 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 in use for decades to 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. For this purpose it is desirable to use a workload generator that is parameterized to emulate a wide range of request mixes and workload properties that might be encountered in practice, rather than a fixed workload standardized for commercial comparisons. Response surface methodology is a powerful tool to evaluate design and cost tradeoffs, explore the interactions of workloads and system choices, and to lo-

cate 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].

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 [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. We take the workbench test harness itself as given. In principle, our approach is compatible with advanced test harnesses such as Auto-pilot [26], which 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 for a controller 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 cost and accuracy for different runs in the set.

As a motivating example, this paper focuses on the problem of obtaining the peak rate for a server with a given server configuration and workload parameters. Even this relatively simple objective requires a costly set of experiments that have not been studied in a systematic way. It is important to optimize this benchmarking objective because it 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. In addition, this objective constitutes the "inner loop" for more complex response surface mapping tasks that are important in systems research. Figure 2 gives an example of response surface mapping using the peak rate. The example is discussed in Section 2.

This paper illustrates the power of a workbench automation framework by exploring simple policies to optimize the "inner loop" to obtain the peak rate in an efficient way that balances 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. Although the concepts and approach generalize to other examples, the experiments in this paper use standard Linux-based NFS servers and a parameterizable NFS workload generator called *fstress* [2], which was developed in previous research and is used by various industry partners. Fstress can emulate standard NFS file server workloads (SPECsfs97), as well as many other workloads

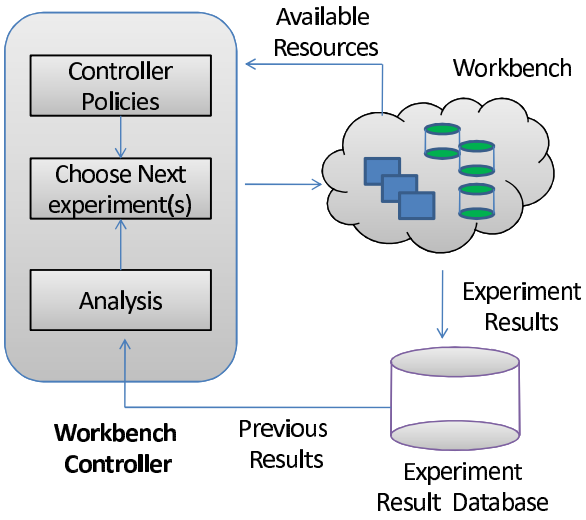

Figure 1: Automated Workbench and Controller.

that might be encountered in practice, according to parameters set by the workbench controller. 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 filer with a given 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 conduct 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

$\vec{W}$	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

Table 1: Some workload and configuration parameters that affect storage server performance.

a given test load to determine how many trials are needed to obtain a suitably accurate result. The controller can also use results of previous runs to prune the sample space in mapping a response surface.

Performance $\vec{P}$. 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 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 paper, saturation occurs when either of two conditions holds: (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.

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 $\vec{W}$. In the experiments in this paper, the controller uses a configurable synthetic workload generator called *Fstress* [2] to explore a space 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 that represent workloads encountered in practice (see Table 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. 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 NFS server parameters on a fixed set of Linux server configurations in the workbench.

Configurations ($\vec{C}$). The controller may vary server configuration parameters before it instantiates the server for each run.

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, the figure 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 reveals that the ideal number of nfsds is workload-dependent: standard rules of thumb used in the field are not suitable for all workloads.

However, 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 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 latency of all the client requests while the workload is running against the server at load λ; and (c) the *number of independent trials* t with load λ.

The challenge for the controller is to design a set of experiments to obtain accurate peak rates for a set of test points selected to approximate the surface efficiently. Before refining the key problem for this paper (finding the peak rate), we first summarize what we mean by confidence and accuracy for each test point.

2.1 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 based on measurements from multiple independent trials [13]. 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 at that load level lies within the range $[25ms, 30ms]$. Such a 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. Basic statistics tells us how to compute confidence intervals and levels from a set of trials. For example, if the mean server response time from t trials is μ, and standard deviation is σ, then the confidence interval for μ at confidence level c

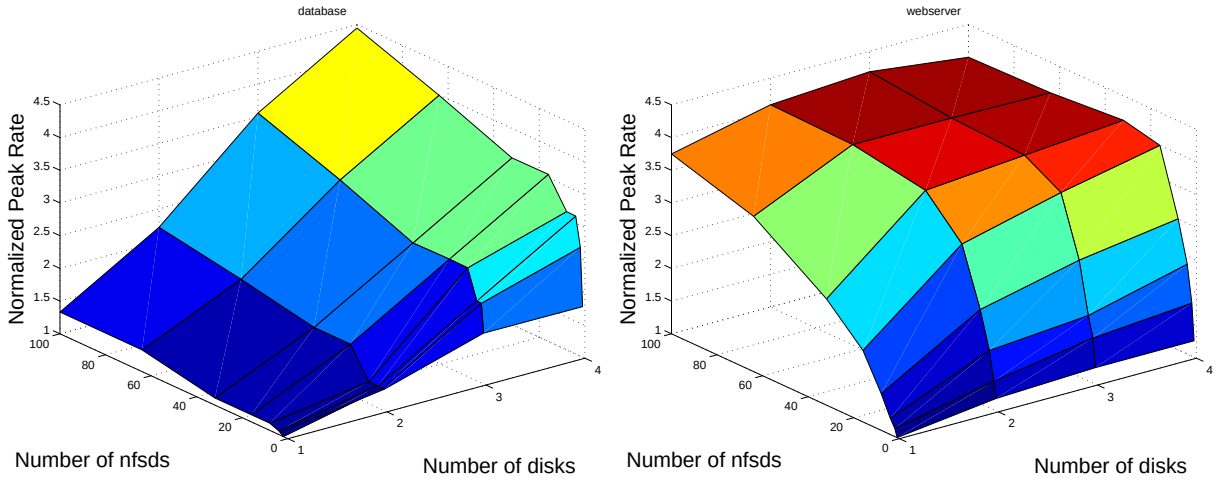

Figure 2: These surfaces illustrate how the peak rate, λ^*, changes with number of disks and number of NFS daemon (nfsd) threads for two canned *fstress* workloads (**DB_TP** and **Web server**). The workloads for this example are described in more detail later, in Table 3.

is given by:

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

z_p is a reading from a table of quantiles for the unit normal distribution, and p is a function of c, such that z_p increases with c. Appendix A explains how to compute confidence intervals at a desired confidence level.

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:

$$accuracy = 1 - error = (1 - \frac{high - low}{high + low}) \times 100\% \quad (2)$$

2.2 Problem Statement

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 1, 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 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 5 present algorithms that the controller uses to address these problems.

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 4).
P_{sat} L_{sat}	The threshold for 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.

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 a sample in $\langle F_1, \ldots, F_n \rangle$ by subjecting the server to a sequence of test loads. The inner loop must converge efficiently to 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. Common practice is to script a sequence of runs for a standard workload at escalating load levels. This *strawman* approach proceeds as follows (the notation is from Table 2):

- **Runlength** r. Use a *fixed* r for each test load.

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 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

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

A simple workbench controller with feedback can improve significantly upon the strawman approach policies. We discuss the limitations of *strawman* as a prelude to the improved controller algorithm in Section 4.

Sequence of test loads. The number of loads, l, in the strawman depend on the increment size to generate successive test loads. If the increment is too low, then it will may take many iterations to reach the peak rate. If it is too high, the test may overshoot the peak rate and compromise accuracy.

Figure 3 illustrates the search for λ^* using the strawman approach for runlength r = 5 minutes, $t = 10$ trials, and a small increment to produce an accurate result. We represent 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 *strawman* to an efficient technique for finding the peak rate, using a straightforward search algorithm, as discussed later.

The figure shows that *strawman* can incur a much higher benchmarking cost to converge to the peak rate. The strawman considers a large number of load factors that are not close to 1, and unnecessarily incurs cost to reach the same accuracy at each load. Sections 4 and 5 present approaches that improve on these limitations.

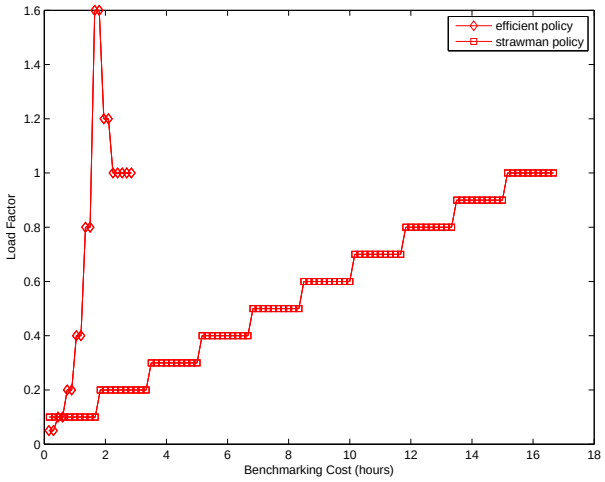

Figure 3: An efficient policy for finding peak rate converges quickly to a load factor near 1, and reduces benchmarking cost by obtaining a high-confidence result only for the load factor of 1. It is significantly less costly than a simple linear search with a fixed runlength, and fixed number of trials per test load (e.g., SPECsfs [7]).

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 4 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. The figure shows that t must adapt to the choice of the test load in the search process. Ideally, t is high near load factor 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 variability of response time.

Runlength. Figure 5 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 response times for different runlengths. It illustrates that the variability decreases with increased runlength. Thus, with short-duration runs more trials are needed to obtain an accurate measure of mean response time. We also observe that for a given 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.

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

An automated approach for finding the peak rate must select a suitable t and r for each test load λ, while adapting to: (a) the value of the load factor, and (b) the target

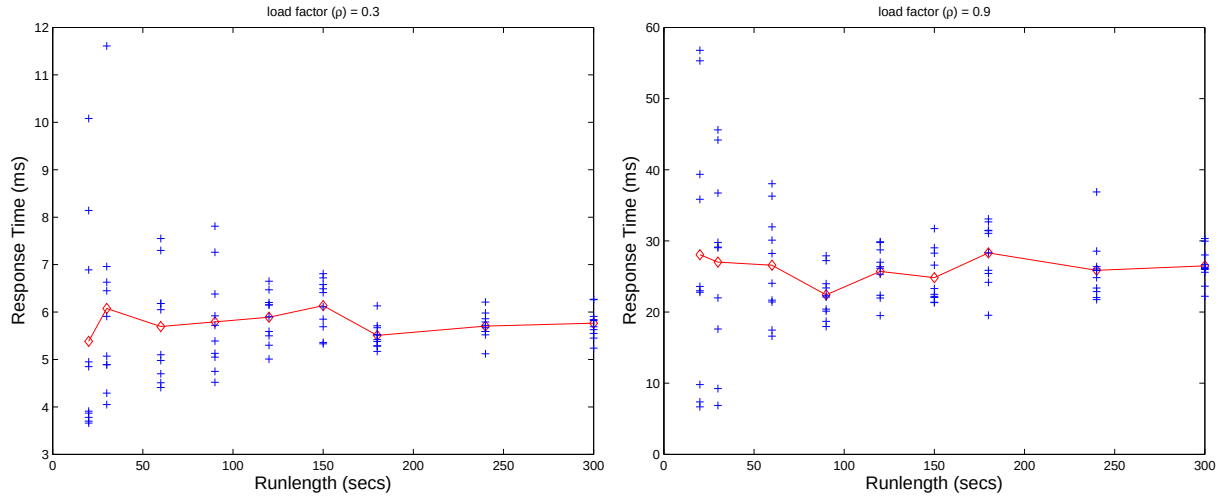

Figure 5: Mean server response time at different workload runlengths for the **DB_TP** *fstress* workload using 1 disk and 4 NFS daemon (nfsd) threads for the server. The variability in mean server response time for multiple trials decreases with increase in runlength. The results are representative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

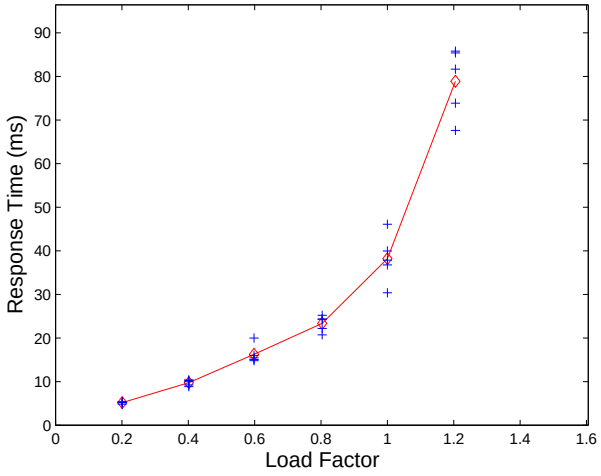

Figure 4: Mean server response time at different test loads for the **DB_TP** *fstress* workload using 1 disk and 4 NFS daemon (nfsd) threads for the server. The variability in mean server response time for multiple trials increases with load. The results are representative of other server configurations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

confidence and accuracy for λ^*. The goal is to converge quickly to an accurate reading at $\rho = 1$. More trials and longer runlengths are useful near a load factor of 1, but accuracy is less crucial during the search.

From Equations 1 and 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, 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.) From the data we can determine 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. 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).

We can longer runlengths to achieve a given target confidence level and/or accuracy. For example, in order to achieve accuracy $\geq$ 87% at 95% confidence, we need a runlength of 120 seconds or more. Another way to improve confidence and accuracy is to run more trials. Figure 6 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 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.

4 Search Algorithm for Peak Rate

Algorithm 2 illustrates our end-to-end search-based approach for estimating the peak rate for a given setting of

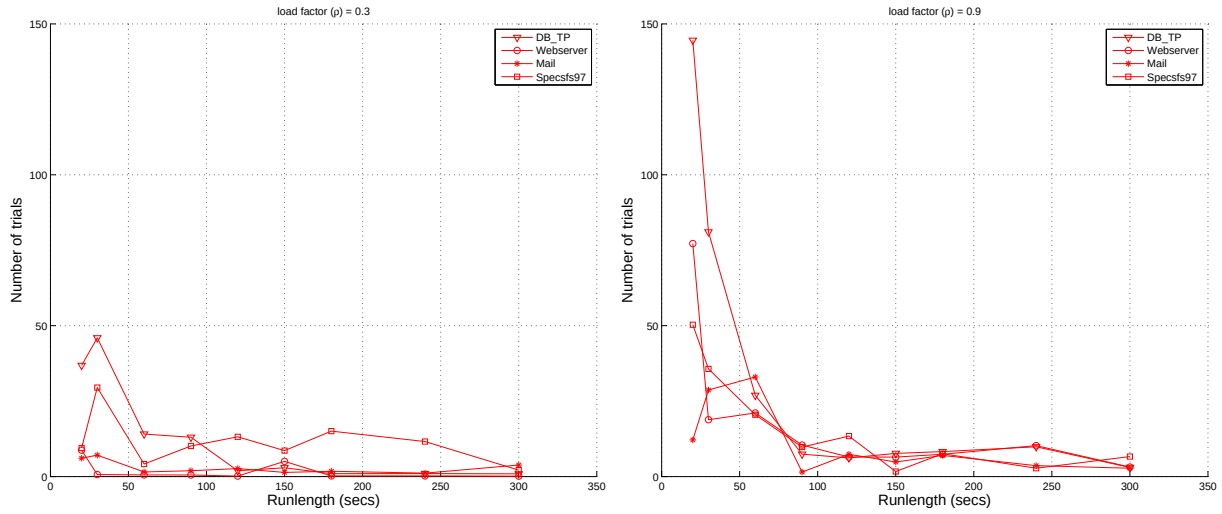

Figure 6: Number of trials to attain 90% accuracy for mean server response time at 95% confidence level at low and high load factors for different runlengths. The results are for server configuration with 1 disk and 4 nfsds, and representative of other server configurations.

factors. The measures used by this algorithm are summarized in Table 2.

The inputs to Algorithm 2 specify various constraints:

- R_{sat}, the threshold on the mean server response time. The server saturates when it 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 2 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.

4.1 Sequence of Test Loads

Algorithm 2 uses one of several *load-picking* algorithms. Section 5 describes the algorithms and their cost and accuracy tradeoffs. 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.

4.2 Number of Trials

For a 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 tests to see if the confidence interval overlaps with the peak-rate region input (Step 9). These steps establish with 95% confidence level whether the current test load is a potential peak rate. If there is no overlap, then Algorithm 2 moves on to the next test load as guided by Algorithm 3 (Step 2).

If the regions overlap, then Algorithm 2 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 2). 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 (Section 3.1). As a result, one of two things happens: (i) the overlap test of the confidence interval and the peak-rate region fails (Step 10), in which case the algorithm 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.

4.3 Runlength for Test Load

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 3.1). The figure shows that for all workloads that this paper considers, a runlength around 3 minutes attains the sweet spot for the number of trials.

4.4 Discussion

Algorithm 2 automatically adapts the number of trials at any test load according to the load factor and the desired confidence and accuracy. Section 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 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 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 2 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.

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 $\lambda_{cur} = \lambda$;

// 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

// A potential peak rate has been reached;

$\lambda^* = \lambda_{cur}$;

Compute accuracy a_{λ^*} at confidence c;

// Check if it meets target accuracy (Step 4);

end

end

5 Mapping Response Surfaces

We now relate the peak rate algorithm 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 4 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 5.1-5.4 describe the load-picking algorithms that this paper considers.
- Algorithm 1 also needs a policy for choosing the samples in the outer loop. Section 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 discusses some of these techniques.

5.1 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.

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

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;
   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;
   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;

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 $\bar{R}_{sat}$. 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 $\lambda_{next} = \frac{b}{\bar{R}_{sat} - a}$;

irrespective of the start test load or the peak rate.

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.

5.3 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 [13], 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 1.

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 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 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 ongoing study. Section 6 reports our experience with the model-guided choice of test loads.

5.4 Better Seeding

The load-picking algorithms in Sections 5.2-5.3 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 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. 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 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 much more efficient techniques exist than iterating over all samples as in Algorithm 1. We can leverage Response Surface Methodology (RSM) [17], a branch of statistics that gives 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 principled techniques 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). We evaluate one such RSM technique in Section 6.

6 Experimental Evaluation

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

Cost for Finding Peak Rate. Sections 4 and 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 6.1 presents the experiment setup. Section 6.2 presents the workloads that we use for evaluation. Section 6.3 evaluates our benchmarking methodology as described above.

6.1 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 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 5.1-5.5 to plot a response surface and/or search for the peak rate for a given sample with target confidence and accuracy.

6.2 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 [2].

- **SPECsfs97**: The Standard Performance Evaluation Corporation introduced their System File Server

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.

benchmark (SPECsfs) [7] in 1992, derived from the earlier self-scaling LADDIS benchmark [15]. A recent (2001) revision corrected several defects identified in the earlier version [12].

- **Web server**: Several efforts (e.g., [3]) 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., [18, 9, 1, 10, 3]).
- **DB_TP**: We model our database workload after TPCC [8], 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. [19].

6.3 Results

For evaluating the overall methodology and the policies outlined in Sections 4 and 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 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.

6.3.1 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 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 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.

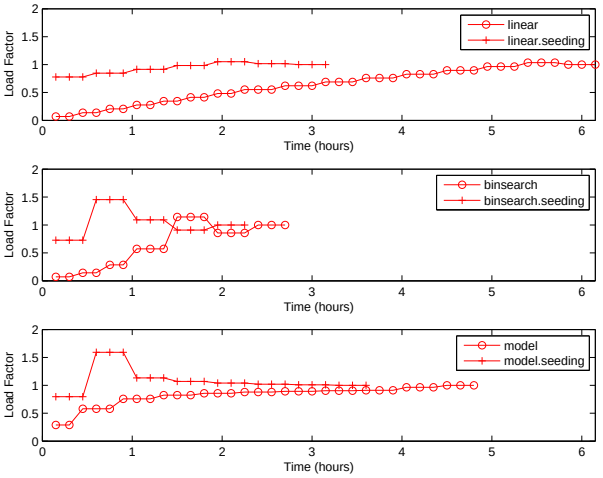

Figure 7: Time spent at each load factor for searching the peak rate for different policies for **DB_TP** with 4 disks, and 32 nfsds. The result is representative of other samples and workloads. All policies except linear quickly converge to the load factor of 1 and conduct more trials there to achieve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

6.3.2 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 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** . The benchmarking results for **SPECsfs97** are still in progress. *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 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.

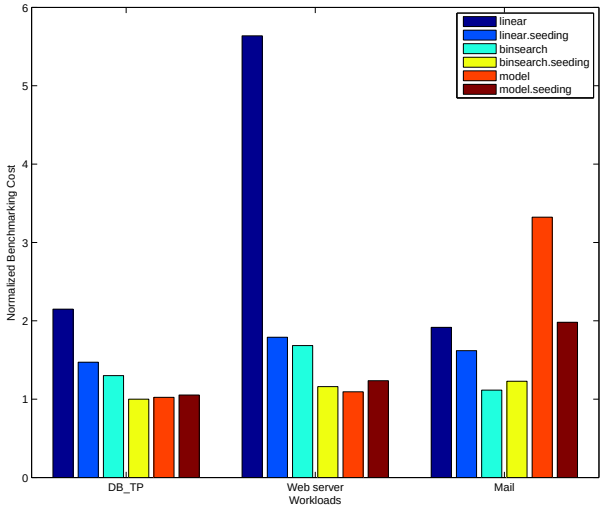

Figure 8: The total cost for mapping response surfaces for three workloads using different policies.

Reducing the Number of Samples. To evaluate the RSM approach presented in Section 5.5, 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 the chosen samples.

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 if the goal is simply to approximate the surface, we can drastically reduce the size of the sample space.

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

6.3.3 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%.

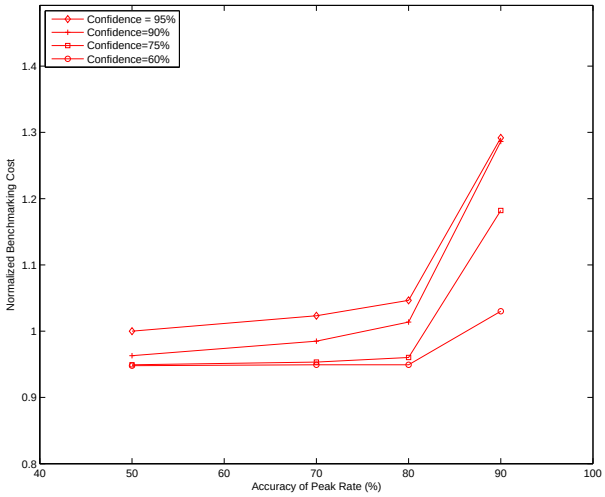

Figure 9: The total benchmarking cost adapts to the desired confidence and accuracy. The cost is shown for mapping the response surface for **DB_TP** using the *Binsearch* policy. Other workloads and policies show similar results.

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 that the benchmarking cost adapting to 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.

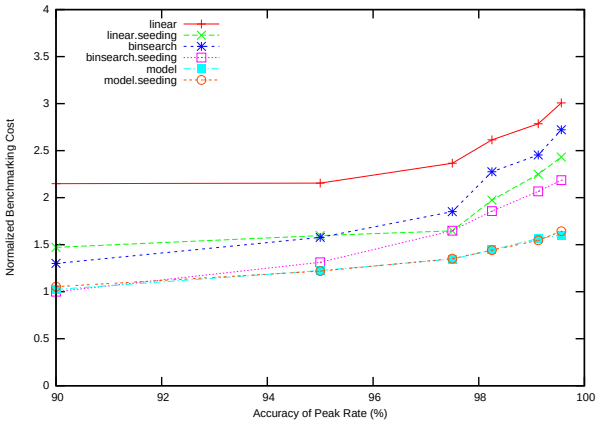

Figure 10: Benchmarking cost adapts to the target accuracy of the peak rate region for all policies. As the region narrows, the majority of the cost is incurred at or near the peak rate. Linear and Binsearch incur the same cost close to the peak rate, and hence their cost converges as they conduct more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

7 Related Work

Several researchers have made a case for statistically significant results from system benchmarking, e.g., [5]. Auto-pilot [26] 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., [6, 14, 4, 15]) 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. [25] present an exhaustive nine year study of file system and storage benchmarking that includes benchmark comparisons, their pros and cons [21], 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 realistic application behavior. Ellard et al. [11] 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 [6], 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 [7] (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 [2], 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 [22] 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.

8 Conclusion

This paper focuses on the problem of workbench automation for storage 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 ap-

proach 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.

APPENDIX: Confidence Intervals

Given N observations of response time from N runs at given arrival rate λ, the confidence interval for the response time at that λ with a desired confidence level, $c\%$, is computed as follows:

- Compute the mean server response time: $\mu = \sum_{i=1}^{N} R_i/N$, where R_i is the server response time for the i^{th} run.
- Compute the standard deviation for the server response time: $\sigma = \sqrt{\sum_{i=1}^{N}(R_i - \mu)^2/(N-1)}$.
- Confidence interval for the response time at confidence $100c\%$ is given as: $[\mu - z_p\sigma/\sqrt{N}, \mu + z_p\sigma/\sqrt{N}]$, where $p = (1+c)/2$, and z_p is the quantile of the unit normal distribution at p.

  If $N <= 30$, we replace z_p by $t_{p;n-1}$, which is the p-quantile of a t-variate with $n-1$ degrees of freedom, assuming that the response time values from N runs come from a normal distribution. We verified that response times do come from a normal distribution using a normal proability plot.

References

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

[2] 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.

[3] 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*, pages 126–137, April 1996.

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

[5] 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 *HOTOS '05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 26–26. "USENIX Association", 2005.

[6] 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*, pages 1–12, May 1993.

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

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

[9] 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.

[10] 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.

[11] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps. In *Proceedings of the FREENIX 2003 Technical Conference*, pages 101–114, June 2003.

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

[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*, pages 111–128, June 1993.

[16] J. C. Mogul. Brittle Metrics in Operating Systems Research. In *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, page 90, Washington, DC, USA, 1999. IEEE Computer Society.

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

[19] 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*, pages 1–15, December 1999.

[20] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *Networked Systems Design and Implementation (NSDI)*, Apr. 2006.

[21] C. Small, N. Ghosh, H. Saleed, M. Seltzer, and K. Smith. Does systems research measure up, November 1997.

[22] K. Smith. *Workload-Specific File System Benchmarks*. PhD thesis, Harvard University, June 2001.

[23] K. A. Smith. *Workload-Specific File System Benchmarks*. PhD thesis, Harvard University, Cambridge, MA, Jan 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. `www.fsl.cs.sunysb.edu/docs/fsbench/fsbench.pdf`.

[26] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: a platform for system software benchmarking. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, pages 53–53. USENIX Association, 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 *Proc. of Work. on Hot Topics in Autonomic Computing*, Jun 2007.