Design and Analysis of Benchmarking Experiments for Distributed Internet Services

Eytan Bakshy
Facebook
Menlo Park, CA
eytan@fb.com

Eitan Frachtenberg
Facebook
Menlo Park, CA
eitan@frachtenberg.org

ABSTRACT

The successful development and deployment of large-scale Internet services depends critically on performance. Even small regressions in processing time can translate directly into significant energy and user experience costs. Despite the widespread use of distributed server infrastructure (e.g., in cloud computing and Web services), there is little research on how to benchmark such systems to obtain valid and precise inferences with minimal data collection costs. Correctly A/B testing distributed Internet services can be surprisingly difficult because interdependencies between user requests (e.g., for search results, social media streams, photos) and host servers violate assumptions required by standard statistical tests.

We develop statistical models of distributed Internet service performance based on data from Perflab, a production system used at Facebook which vets thousands of changes to the company's codebase each day. We show how these models can be used to understand the tradeoffs between different benchmarking routines, and what factors must be taken into account when performing statistical tests. Using simulations and empirical data from Perflab, we validate our theoretical results, and provide easy-to-implement guidelines for designing and analyzing such benchmarks.

Keywords

A/B testing; benchmarking; cloud computing; crossed random effects; bootstrapping

Categories and Subject Descriptors

G.3 [**Probability and Statistics**]: Experimental Design

1. INTRODUCTION

Benchmarking experiments are used extensively at Facebook and other Internet companies to detect performance regressions and bugs, as well as to optimize the performance of existing infrastructure and backend services. They often involve testing various code paths with one or more users on multiple hosts. This introduces multiple sources of variability: each user request may involve different amounts of computation because results are dynamically customized to their own data. The amount of data varies from user to user and often follows heavy-tailed distributions for quantities such as friend count, feed stories, and search matches [2].

Furthermore, different users and requests engage different code paths based on their data, and can be affected differently by just-in-time (JIT) compilation and caching. User-based benchmarks therefore must take into account a good mix of both service endpoints and users making the requests, in order to capture the broad range of performance issues that may arise in production settings.

An additional source of variability comes from benchmarking on multiple hosts, as is common for cloud and Web-based services. Testing in a distributed environment is often necessary due to engineering or time constraints, and can even improve the representativeness of the overall performance data, because the production environment is also distributed. But each host has its own performance characteristics, which can also vary over time. The performance of computer architectures has grown increasingly nondeterministic over the years, and performance innovations often come at a cost of lower predictability. Examples include: dynamic frequency scaling and sleep states; adaptive caching, branch prediction, and memory prefetching; and modular, individual hardware components comprising the system with their own variance [5, 16].

Our research addresses challenges inherent to any benchmarking system which tests user traffic on a distributed set of hosts. Motivated by problems encountered in the development of Perflab [14], an automated system responsible for A/B testing hundreds of code changes every day at Facebook, we develop statistical models of user-based, distributed benchmarking experiments. These models help us address real-life challenges that arise in a production benchmarking system, including high variability in performance tests, and high rates of false positives in detecting performance regressions.

Our paper is organized as follows. Using data from our production benchmarking system, we motivate the use of statistical concepts, including two-stage sampling, dependence, and uncertainty (Section 3). With this framework, we develop a statistical model that allows us to formally evaluate the consequences of how known sources of variation — requests and hosts — on the precision of different experimental designs of distributed benchmarks (Section 4). These variance expressions for each design also informs how statistical tests should be computed. Finally, we propose and evaluate a bootstrapping procedure in Section 5 which shows good performance both in our simulations and production tests.

2. PROBLEM DESCRIPTION

We often want to make changes to software—a new user interface, a feature, a ranking model, a retrieval system, or virtual host—and need to know how each change affects performance metrics

WWW 2015, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3469-3/15/05
http://dx.doi.org/10.1145/2736277.2741082.

such as computation time or network load. This goal is typically accomplished by directing a subset of user traffic to different machines (hosts) running different versions of software and comparing their performance. This scheme poses a number of challenges that affect our ability to detect statistically significant differences, both with respect to Type I errors—a "false positive" where two versions are deemed to have different performance when in fact they don't, and Type II errors—an inability to detect significant differences where they do exist. These problems can result in financial loss, wasted engineering time, and misallocation of resources.

As mentioned earlier, A/B testing is challenging both from the perspective of reducing the variability in results, and from the perspective of statistical inference. Performance can vary significantly due to many factors, including characteristics of user requests, machines, and compilation. Experiments not designed to account for these sources of variation can obscure performance reversions. Furthermore, methods for computing confidence intervals for user-based benchmarking experiments which do not take into account these sources of variation can result in high false-positive rates.

2.1 Design goals

When designing an experiment to detect performance reversions, we strive to accomplish three main goals: (1) Representativeness: we wish to produce performance estimates that are valid (reflecting performance for the production use case), take samples from a representative subset of test cases, and generalize well to the test cases that weren't sampled; (2) Precision: performance estimates should be fine-grained enough to discern even small effects, per our tolerance limits; (3) Economy: estimates should be obtained with minimal cost (measured in wall time), subject to resource constraints.

In short, we wish to design a representative experiment that minimizes both the number of observations and the variance of the results. To simplify, we consider a single endpoint or service, and use the terms "request", "query", and "user" interchangeably.[1]

2.2 Testing environment

To motivate our model, as well as validate our results empirically, we discuss Facebook's in-house benchmarking system, Perflab [14]. Perflab can assess the performance impact of new code without actually installing it on the servers used by real users. This enables developers to use Perflab as part of their testing sequence even before they commit code. Moreover, Perflab is used to automatically check all code committed into the code repository, to uncover both logic and performance problems. Even small performance issues need to be monitored and corrected continuously, because if they are left to accumulate they can quickly lead to capacity problems and unnecessary expenditure on additional infrastructure. Problems uncovered by Perflab or other tests that cannot be resolved within a short time may cause the code revision to be removed from the push and delayed to a subsequent push, after the problems are resolved.

A Perflab experiment is started by defining two versions of the software/environment (which can be identical for an "A/A test"). Perflab reserves a set of hosts to run either version, and prepares them for a batch of requests by restarting them and flushing the caches of the backend services they rely on (which are copied and isolated from the production environment, to ensure Perflab has no side effects). The workload is selected as a representative subset of endpoints and users from a previous sample of live traffic. It is replayed repeatedly and concurrently at the systems under tests, which queue up the requests so that the system is under a relatively constant, near-saturation load, to represent a realistically high production load. This continues until we get the required number of observations, but we discard a number of initial observations ("warm-up" period) and final observations ("cool-down" period). This range was determined empirically by analyzing the autocorrelation function [13] for individual requests as a function of repetition number. Perflab then resets the hosts, swaps software versions so that hosts previously running version A now run version B and vice-versa, and reruns the experiment to collect more observations.

For each observation, we collect all the metrics of interest, which include: CPU time and instructions; database/key-value fetches and bandwidth; memory use; HTML size; and various others. These metrics detect not only performance changes, as in CPU time and instructions, but can also detect bugs. For example, a sharp drop in the number of database fetches (for a given user and endpoint) between two software versions is more often than not an undesired side effect and an indication of a faulty code path, rather than a miraculous performance improvement, since the underlying user data hasn't changed between versions.

[1]We discuss how our approach can be generalized to multiple endpoints in Section 7.

3. STATISTICS OF USER-BASED BENCHMARKS

In order to understand how best to design user-based benchmarks, we must first consider how to capture (sample) metrics over a population of requests. We begin by discussing the relationship between sampling theory and data collection for benchmarks. This leads us to confront sources of variability—first from requests, and then from hosts. We conclude this section with a very general model that brings these aspects together, and serves as the basis for the formal analysis of experimental designs for distributed benchmarks involving user traffic.

3.1 Sampling

3.1.1 Two-stage sampling of user requests

We wish to estimate the average performance of a given version of software, and do so by sending requests (e.g., loading a particular endpoint for various users) to a service, and measuring the sample average $\bar{y}$, for some outcome y_i for each observation. Our certainty about the true average value $\bar{Y}$ of the complete distribution Y increases with the number of observations. If each observations of Y is independent and identically distributed (iid), then the *standard error* of for our estimate of $\bar{Y}$ is $\text{SE}_{\text{iid}} = s/\sqrt{N}$, where N is the number of observations, and s is the sample standard deviation of our observed y_is. One can use the standard error to obtain the confidence interval for $\bar{Y}$ at significance level α by computing $\bar{y} \pm \Phi^{-1}(1 - \alpha/2)\text{SE}$, where Φ^{-1} is the inverse cumulative distribution of the standard normal. For example, the 95% confidence interval for $\bar{Y}$ can be computed as $\bar{y} \pm 1.96\text{SE}$.

Often times data collected from benchmarks involving user traffic is not iid. For example, if we repeat requests for users multiple times, the observations may be clustered along certain values. Figure 1 shows how this kind of clustering occurs with respect to CPU time[2] for requests fulfilled to a heavily trafficked endpoint at Facebook. Here, we see 300 repetitions for 4, 16, and 256 different requests (recall that by different requests, we are referring to different users for a fixed endpoint). CPU time is distributed approximately

[2]Because relative performance is easier to reason about than absolute performance in this context, all outcomes in this paper are standardized, so that for any particular benchmark for an endpoint, the grand mean is subtracted from the outcome and then divided by the standard deviation.

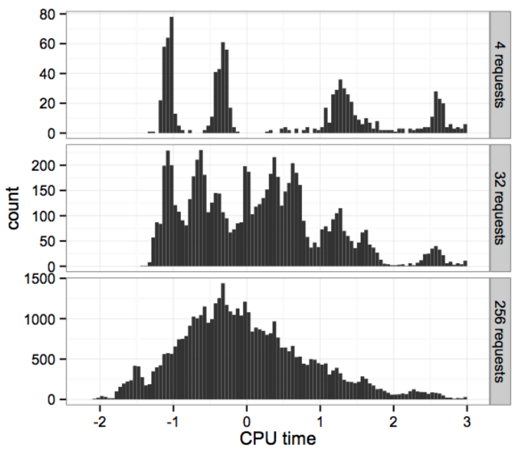

Figure 1: Clustering in the distribution of CPU times for requests. Panels correspond to different numbers of requests. Requests are repeated 300 times.

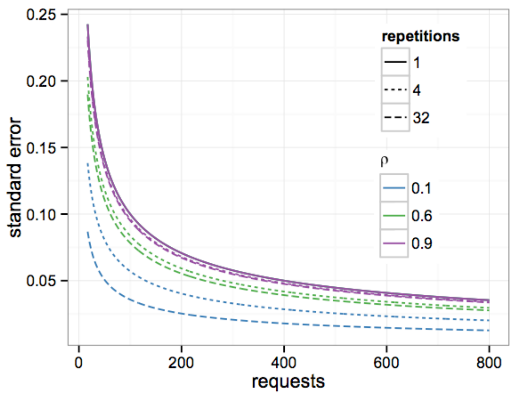

Figure 2: Standard error for clustered data (e.g., requests) as a function of the number of requests and repetitions for different intra-class correlation coefficients, ρ. Multiple repeated observations of the same request have little effect on the SE when ρ is large.

normally for any given request, but the amount of variability within a single request is much smaller than the overall variability between requests. When hundreds of different requests are mixed in, in fact, it's difficult to tell that this aggregate distribution is really the mixture of many, approximately normally distributed clusters.

If observations are clustered in some way (as in Figure 1), the effective sample size could be much smaller than the total number of observations. Measurements for the same request tend to be correlated, such that e.g., the execution time for requests for the same user are more similar to one another than requests related to different users. This is captured by the intra-class correlation coefficient (*ICC*) ρ, which is the ratio of between-cluster variation to total variation [28]:

$$\rho = \frac{\sigma_\alpha^2}{\sigma_\alpha^2 + \sigma_\varepsilon^2},$$

where σ_α is the standard deviation of the request effect and σ_ε is the standard deviation of the measurement noise. If ρ is close to 1, then repeated observations for the same request are nearly identical, and when it is close to 0, there is little relation between requests. It is easy to see that if most of the variability occurs between clusters, rather than within clusters (high ρ), additional repeated measurements for the same cluster do not help with obtaining a good estimate of the mean of Y, $\bar{Y}$. This idea is captured by the design effect [10], $d_{\text{eff}} = (1 + (T-1)\rho)$, which is the ratio of the variance of the clustered sample (e.g., multiple samples from the same request) to the simple random sample (e.g., random samples of multiple independent requests), where T is the number of times each request is repeated.

Under sampling designs where we may choose both the number of requests R, and repetitions T, the standard error is instead:

$$\text{SE}_{\text{clust}} = \sigma / \sqrt{\frac{1}{RT}(1 + (T-1)\rho)} \quad (1)$$

such that additional repetitions only reduce the standard error of our estimate if ρ is small. Figure 2 shows the relationship between the standard error and R for small and large vales of ρ. For most of the top endpoints benchmarked at Facebook—including news feed loads and search queries—nearly all endpoints have values of ρ between 0.8 and 0.97. In other words, from our experience with user traffic, *mere repetition of requests is not an effective means of increasing precision when the intraclass correlation, ρ, is large*.

3.1.2 *Sampling on a budget*

Exploring the tradeoffs between collecting more requests vs. more repetitions needs to take into account not only the value of ρ but also the costs of switching requests. For example, it is often necessary to first "warm up" a virtual host or cache to reduce temporal effects (e.g., serial autocorrelation) [16] to get accurate estimates for an effect. That is, the fixed cost required to sample a request is often much greater than it is to gain additional repetitions. In situations in which ρ is small, it may be useful to consider larger numbers of repetitions per request. There is also an additional fixed cost S_f to set up an experiment (or a "batch"), which includes resetting the hardware and software as necessary and waiting for the runtime systems to warm up.

By considering the costs for each stage of a benchmarking routine, one can minimize the standard error. For example, if there is a fixed cost C_f for benchmarking a single request, a marginal cost for an additional repetition of a request, C_m, and an available budget, B, one could minimize Eq. 1 subject to the constraint that S_f + $R(C_f + TC_m) \leq B$.

3.2 Models of distributed benchmarks

While benchmarking on a single host is simple enough, we are often constrained to run benchmarks in a short period of time, which can be difficult to accomplish on only one host. Facebook, for example, runs automated performance tests on every single code commit to its main site, thousands of times a day [14]. It is therefore imperative to conduct benchmarks in parallel on multiple hosts. Using multiple hosts also has the benefit of surfacing performance issues that may affect one host and not another, for further investigation.[3] But multiple hosts also introduce new sources of variability, which we model statistically in the following sections.

To motivate and illustrate these models, let us look at an example of empirical data from a Perflab A/A benchmark (Figure 3). Two requests (corresponding to service queries for two distinct users under the same software version) are repeated across two different machines in two batches, running thirty times on each. For each batch, the software on the hosts was restarted, and caches and JIT environments were warmed up. The observations correspond to CPU time measured for an endpoint running under a PHP vir-

[3]Such investigation sometimes leads to identifying faulty systems in the benchmarking environment, but can also reveal unintended consequences of the software interacting with different subsystems.

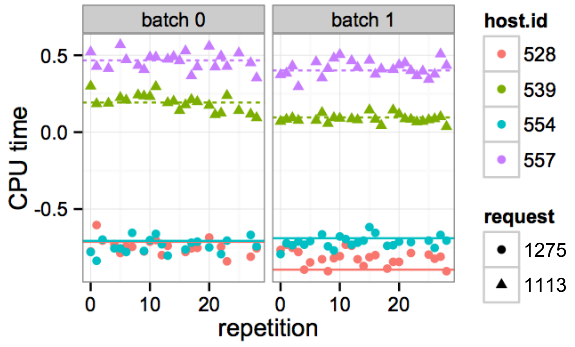

Figure 3: CPU time for two requests executed over 30 repetitions, executed on two hosts over two different batches. Each shape represents a different request, and each color represents a different host. Panels correspond to different batches. Lines represent a model fit based on Eq. 3.

tual machine. Note that each repetition of the same request on the same machine in the same batch is relatively similar and has few time trends. Furthermore, the between-request variability (circle vs. square shapes) tends to be greater than the per-machine variability (e.g., the green vs purple dots). Between-batch effects appear to be similar in size to the noise.

3.2.1 *Random effects formulation*

An observation from a benchmark can be thought of as being generated by several independent effects. In the simplest case, we could think of a single observation (e.g., CPU time) for some particular endpoint or service as having some average value (e.g., 500ms), plus some shift due to the user involved in the request (e.g., +500ms for a user with many friends), plus a shift due to the host executing the request (e.g., -200ms for a faster host), plus some random noise. This formulation is referred to as a crossed random effects model, and we refer to host and request-level shifts as a *random effects* [3, 26] or levels [28].

Formally, we can denote the request corresponding to an observation i as $r[i]$, and the host corresponding to i as $h[i]$. In the equation below, we denote the random effects (random variables) for requests, hosts, and noise for each observation i as $\alpha_{r[i]}$, $\beta_{h[i]}$, and ε_i, respectively. The mean value is given by μ. f denotes a transformation (a "link function"). For additive models, f is simply the identity function, but for multiplicative models (e.g., each effect induces a certain percent increase or decrease in performance), f may be the exponential function[4], $exp(\cdot)$:

$$Y_i = f(\mu + \alpha_{r[i]} + \beta_{h[i]} + \varepsilon_i)$$

Under the heterogeneous random effects model [26], each request and host can have its own variance. This model can be complex to manipulate analytically, and difficult to estimate in practice, so one common approach is to instead assume that random effects for requests or hosts are drawn from the same distribution.

Homogeneous random effects model for a single batch. Under this model, random effects for hosts and requests are respectively drawn from a common distribution.

[4]For the sake of simplicity we work with additive models, but it is often desirable to work with the multiplicative model, or equivalently $log(Y)$. The remaining results hold equally for the log-transformed outcomes.

$$Y_i = \mu + \alpha_{r[i]} + \beta_{h[i]} + \varepsilon_i$$
$$\alpha_r \sim \mathcal{N}(0, \sigma^2_\alpha), \quad \beta_h \sim \mathcal{N}(0, \sigma^2_\beta), \quad \varepsilon_i \sim \mathcal{N}(0, \sigma^2_\varepsilon). \quad (2)$$

In this homogenous random effects model each request has a constant effect, which is sampled from some common normal distribution, $\mathcal{N}(0, \sigma^2_\alpha)$. Any variability from the same request on the same host is independent of the request.

Homogeneous random effects model for multiple batches. Modern runtime environments are non-deterministic and for various reasons, restarting a service may cause the characteristic performance of hosts or requests to deviate slightly. For example, the dynamic request order and mix can vary and affect the code paths that the JIT compiles and keeps in cache. This can be specified by additional batch-level effects for hosts and requests. We model this behavior as follows: each time a service is initialized, we draw a batch effect for each request, $\gamma_{r[i],b[i]} \sim \mathcal{N}(0, \sigma_\gamma)$, and similarly for each host, $\eta_{h[i],b[i]} \sim \mathcal{N}(0, \sigma_\eta)$. That is, each time a host is restarted in some way (a new "batch"), there is some additional noise introduced that remains constant throughout the execution of the batch, either pertaining to the host or the request. Note that per-request and per-host batch effects are independent from batch to batch[5], similar to how ε is independent between observations:

$$Y_i = \mu + \alpha_{r[i]} + \beta_{h[i]} + \gamma_{r[i],b[i]} + \eta_{h[i],b[i]} + \varepsilon_i$$
$$\alpha_r \sim \mathcal{N}(0, \sigma^2_\alpha), \quad \beta_h \sim \mathcal{N}(0, \sigma^2_\beta),$$
$$\gamma_{r,b} \sim \mathcal{N}(0, \sigma^2_\gamma), \quad \eta_{h,b} \sim \mathcal{N}(0, \sigma^2_\eta), \quad \varepsilon_i \sim \mathcal{N}(0, \sigma^2_\varepsilon) \quad (3)$$

3.2.2 *Estimation*

How large are each of these effects in practice? We begin with some of observations from a real benchmark to illustrate the sources of variability, and then move on to estimating model parameters for several endpoints in production. Models such as Eq. 2 and Eq. 3 can be fit efficiently to benchmark data via restricted maximum likelihood estimation using off-the-shelf statistical packages, such as `lmer` in R. In Figure 3, the horizontal lines indicate the predicted values for each endpoint using the model from Eq. 3 fit to an A/A test (e.g., an experiment in which both versions have the same software) using `lmer`. In general, we find that the request-level random effects are much greater than the host level effects, which are similar in magnitude to the batch-level effects. We summarize estimates for top endpoints at Facebook in Table 1, and use them in subsequent sections to illustrate our analytical results.

endpoint	μ	σ_α	σ_β	σ_γ	σ_η	σ_ε
1	0.06	1.02	0.12	0.10	0.08	0.13
2	0.08	1.08	0.10	0.05	0.06	0.08
3	0.07	1.02	0.11	0.04	0.05	0.06
4	0.08	1.08	0.05	0.05	0.02	0.06
5	0.04	1.08	0.04	0.01	0.02	0.14

Table 1: Parameter estimates for the model in Eq. 3 for the five most trafficked endpoints benchmarked at Facebook.

4. EXPERIMENTAL DESIGN

Benchmarking experiments for Internet services are most commonly run to compare two different versions of software using a

[5]This formal assumption corresponds to no carryover effects from one batch to another [7]. As we describe in Section 2.2, the system we use in production is designed to eliminate carryover effects.

mix of requests and hosts (servers). How precisely we are able to measure differences can depend greatly on which requests are delivered to what machines, and what versions of the software those machines are running. In this section, we generalize the random effects model for multiple batches to include experimental comparisons, and derive expressions for how the standard error of experimental comparisons (i.e., the difference in means) depends on aspects of the experimental design. We then present four simple experimental designs that cover a range of benchmarking setups, including "live" benchmarks and carefully controlled experiments, and derive their standard errors. Finally, we will show how basic parameters, including the number of requests, hosts, and repetitions, affect the standard error of different designs.

4.1 Formulation

We treat the problem formally using the potential outcomes framework [24], in which we consider outcomes (e.g., CPU time) for an observation i (a request-host pair running within a particular batch), running under either version (the experimental condition), whose assignment is denoted by $D_i = 0$ or $D_i = 1$. We use $Y_i^{(1)}$ to denote the potential outcome of i under the treatment, and $Y_i^{(0)}$ for the control. Although we cannot simultaneously observe both potential outcomes for any particular i, we can compute the average treatment effect, $\delta = \mathbb{E}[Y_i^{(1)} - Y_i^{(0)}]$ because by linearity of expectation, it is equal to the difference in means $\mathbb{E}[Y_i^{(1)}] - \mathbb{E}[Y_i^{(0)}]$ across different populations when D_i is randomly assigned. In a benchmarking experiment, we identify δ by specifying a schedule of delivery of requests to hosts, along with hosts' assignments to conditions. The particulars of how this assignment procedure works is the experimental design, and it can substantially affect the precision with which we can estimate δ. We generalize the random effects model in Eq. 2 to include average treatment effects and treatment interactions:

$$\begin{gathered}Y_i^{(d)} = \mu^{(d)} + \alpha_{r[i]}^{(d)} + \beta_{h[i]}^{(d)} + \gamma_{r[i],b[i]} + \eta_{h[i],b[i]} + \varepsilon_i \\ \vec{\alpha}_r \sim \mathcal{N}(0, \Sigma_\alpha), \quad \vec{\beta}_h \sim \mathcal{N}(0, \Sigma_\beta), \\ \gamma_{r,b} \sim \mathcal{N}(0, \sigma_\gamma^2), \quad \eta_{h,b} \sim \mathcal{N}(0, \sigma_\eta^2), \quad \varepsilon_i \sim \mathcal{N}(0, \sigma_\varepsilon^2) \qquad (4)\end{gathered}$$

Our goal therefore is to identify the true difference in means, $\delta = \mu^{(1)} - \mu^{(0)}$. Unfortunately, we can never observe δ directly, and instead must estimate it from data, with noise. Exactly how much noise there is depends on which hosts and requests are involved, the software versions, and how many batches are needed to run the experiment. More formally, we denote the number of observations for a particular request–host–batch tuple $\langle r, h, b \rangle$ running under the treatment condition d, by $n_{rhb}^{(d)}$. We express the total noise from requests, hosts, and residual error under each condition as:

$$\begin{aligned}\phi_R^{(d)} &\equiv \sum_r \left[n_{r\bullet\bullet}^{(d)} \alpha_r^{(d)} + \sum_b n_{r\bullet b}^{(d)} \gamma_{r,b} \right] \\ \phi_H^{(d)} &\equiv \sum_h \left[n_{\bullet h\bullet}^{(d)} \beta_h^{(d)} + \sum_b n_{\bullet hb}^{(d)} \eta_{h,b} \right] \\ \phi_E^{(d)} &\equiv \sum_{i=1}^{2N} \varepsilon_i^{(d)} \mathbb{1}[D_i = d],\end{aligned}$$

where, e.g., $n_{r\bullet b}^{(d)}$ represents the total number of observations involving a request r executed in batch b under condition d. We take the total number of observations per condition to be equal so that $n_{\bullet\bullet\bullet}^{(0)} = n_{\bullet\bullet\bullet}^{(1)} = N$.

We can then write down our estimate, $\hat{\delta}$, as

$$\hat{\delta} = \delta + \frac{1}{N} \left[(\phi_R^{(1)} - \phi_R^{(0)}) + (\phi_H^{(1)} - \phi_H^{(0)}) + (\phi_E^{(1)} - \phi_E^{(0)}) \right].$$

To obtain confidence intervals for $\hat{\delta}$, we also need to know its variance, $\mathrm{V}[\hat{\delta}]$. Following Bakshy & Eckles [4], we approach the problem by first describing how observations from each error component are repeated within each condition. We define the duplication coefficients [22, 23]:

$$\nu_R^{(d)} \equiv \frac{1}{N} \sum_r \left(n_{r\bullet\bullet}^{(d)}\right)^2 \qquad \nu_H^{(d)} \equiv \frac{1}{N} \sum_h \left(n_{\bullet h\bullet}^{(d)}\right)^2,$$

which are the average number of observations sharing the same request (ν_R) or host (ν_H). We then define the between-condition duplication coefficient [4], which gives a measure of how balanced hosts or requests are across conditions:

$$\omega_R \equiv \frac{1}{N} \sum_r n_{r\bullet\bullet}^{(0)} n_{r\bullet\bullet}^{(1)} \qquad \omega_H \equiv \frac{1}{N} \sum_h n_{\bullet h\bullet}^{(0)} n_{\bullet h\bullet}^{(1)}.$$

Furthermore, we only consider experimental designs in which the request-level and host-level duplication are the same in both conditions, so that $\nu_R^{(0)} = \nu_R^{(1)}$ and $\nu_H^{(0)} = \nu_H^{(1)}$, and omit the superscripts in subsequent expressions.[6]

Noting that because batch-level random effects are independent, the variance of their sums over batches can be expressed in terms of duplication coefficients, so that e.g.,

$$\mathrm{V}[\frac{1}{N} \sum_b n_{r\bullet b}^{(d)} \gamma_{r,b}] = \frac{1}{N^2} \left(n_{r\bullet\bullet}^{(d)}\right)^2 \sigma_\gamma^2 = \frac{1}{N} \nu_R \sigma_\gamma^2,$$

and because all random effects are independent, it is straightforward to show that the variance of $\hat{\delta}$ can be written in terms of these duplication factors:

$$\begin{aligned}\mathrm{V}[\hat{\delta}] = \frac{1}{N} \Bigg[&\left(\nu_R (\sigma_{\alpha^{(1)}}^2 + \sigma_{\alpha^{(0)}}^2 + 2\sigma_\gamma^2) - 2\omega_R \sigma_{\alpha^{(0)},\alpha^{(1)}} \right) \\ &+ \left(\nu_H (\sigma_{\beta^{(1)}}^2 + \sigma_{\beta^{(0)}}^2 + 2\sigma_\eta^2) - 2\omega_H \sigma_{\beta^{(0)},\beta^{(1)}} \right) \qquad (5) \\ &+ \left(\sigma_{\varepsilon^{(0)}}^2 + \sigma_{\varepsilon^{(1)}}^2 \right) \Bigg].\end{aligned}$$

This expression illustrates how repeated observations of the same request or host affect the variance of our estimator for $\hat{\delta}$. Host-level variation is multiplied by how often hosts are repeated in the data, and similarly for requests. Variance is reduced when requests or hosts appear equally in both conditions, since, for example, ω_R is greatest when $n_{r\bullet\bullet}^{(0)} = n_{r\bullet\bullet}^{(1)}$ for all r. We use these facts to explore how different experimental designs—ways of delivering requests to hosts under different conditions—affect the precision with which we can estimate δ.

4.2 Four designs for distributed benchmarks

In this section, we describe four simple experimental designs that reflect basic engineering tradeoffs, and analyze their standard errors under a common set of conditions. In particular, we consider our certainty about experimental effects when:

[6]Note that the individual components, e.g., which requests appear in the treatment and control need not be the same for the duplication coefficients to be equal.

i	Host	Req.	Ver.	Batch
1	h_0	r_0	0	1
2	h_0	r_1	0	1
3	h_1	r_2	1	1
4	h_1	r_3	1	1

(a) Unbalanced

i	Host	Req.	Ver.	Batch
1	h_0	r_0	0	1
2	h_0	r_1	0	1
3	h_1	r_0	1	1
4	h_1	r_1	1	1

(b) Request balanced

i	Host	Req.	Ver.	Batch
1	h_0	r_0	0	1
2	h_1	r_1	0	1
3	h_0	r_2	1	2
4	h_1	r_3	1	2

(c) Host balanced

i	Host	Req.	Ver.	Batch
1	h_0	r_0	0	1
2	h_1	r_1	0	1
3	h_0	r_0	1	2
4	h_1	r_1	1	2

(d) Fully balanced

Table 2: Example schedules for the four experimental designs for experiments with two hosts ($H = 2$) in which two requests are executed per condition ($R = 2$). Each example shows four observations, each with a host ID, request ID (e.g., a request to an endpoint for a particular user), software version, and batch number.

1. The sharp null is true—that is, the experiment has no effects at all, so that δ is zero and all variance components are the same (e.g., $\sigma^2_{\alpha(0)} = \sigma^2_{\alpha(1)} = \sigma_{\alpha(0)\alpha(1)}$) [4], or
2. There are no treatment interactions, but there is a constant additive effect δ.[7]

Although these requirements are rather narrow, they correspond to a common scenario in which benchmarks are used for difference detection; by minimizing the standard error of the experiment, we are better able to detect situations in which there is a deviation from no change in performance.

In the four designs we discuss in the following sections, we constrain the design space to simplify presentation in a few ways. First, we assume symmetry with respect to the pattern of delivery of requests; we repeat each request the same number of times in each condition, so that $N = RT$. Second, because executing the same request on multiple hosts within the same condition would increase ν_R (and therefore inflate $V[\hat{\delta}]$), we only consider designs in which requests are executed on at most one machine per condition. And finally, since by (1) and (2), the variances of the error terms are equal, we drop the superscripts for each σ^2.

We consider two classes of designs: single-batch experiments, in which half of all hosts are assigned to the treatment, and the other half to the control, or two-batch experiments in which hosts run both versions of the software. Note that R requests are split among two batches in the latter case.[8]

In the single-batch design, each of the R requests is executed on either the first block of hosts or the second block, where each block is either assigned to the treatment or control. Therefore, when computing the host-level duplication coefficient, ν_H for a particular condition, we only sum across $H/2$ hosts:

$$\nu_H = \frac{1}{RT}\sum_{h=1}^{H/2}\left(\frac{RT}{H/2}\right)^2 = \frac{2RT}{H},$$

In the two-batch design, each of the R requests are executed in both the treatment and control across spread across all H hosts:

$$\nu_H = \frac{1}{RT}\sum_{h=1}^{H}\left(\frac{RT}{H}\right)^2 = \frac{RT}{H},$$

4.2.1 *Definitions*

Unbalanced design (“live benchmarking”). In the unbalanced design, each host only executes one version of the software, and each request is processed once. It can be carried out in one batch (see example layout in Table 2 (a)). This design is the simplest to implement since it does not require the ability to replay requests. It is often the design of choice when benchmarking live requests only, whether as a constraint or merely as a choice of convenience: it obviates the need for a possibly complex infrastructure to record and replay requests. The variance of the difference-in-means estimator for the unbalanced design is:

$$\mathrm{V}_{\mathrm{UB}}(\hat{\delta}) = 2T(\sigma^2_\alpha + \sigma^2_\gamma) + 2\frac{2RT}{H}(\sigma^2_\beta + \sigma^2_\eta) + 2\sigma^2_\epsilon$$

The standard error is $\sqrt{\frac{1}{RT}\mathrm{V}_{\mathrm{UB}}(\hat{\delta})}$. Expanding and simplifying this expression yields:

$$\mathrm{SE}_{\mathrm{UB}}(\hat{\delta}) = \sqrt{2(\frac{1}{R}(\sigma^2_\alpha + \sigma^2_\gamma) + \frac{2}{H}(\sigma^2_\beta + \sigma^2_\eta) + \frac{1}{RT}{\sigma_\epsilon}^2)}$$

The convenience of the unbalanced design comes at a price: the standard error term includes error components both from requests and hosts. We will show later that this design is the least powerful of the four: it achieves significantly lower accuracy (wider confidence intervals) for the same resource budget.

Request balanced design (“parallel benchmarking”). The request balanced design executes the same request in parallel on different hosts using different versions of the software. This requires that one has the ability to split or replay traffic, and can be done in one batch (see example layout in Table 2 (b)). Its standard error is defined as:

$$\mathrm{SE}_{\mathrm{RB}}(\hat{\delta}) = \sqrt{2(\frac{1}{R}\sigma^2_\gamma + \frac{2}{H}(\sigma^2_\beta + \sigma^2_\eta) + \frac{1}{RT}{\sigma_\epsilon}^2)}$$

The request balanced design cancels out the effect of the request, but noise due to each host is amplified by the average number of requests per host, $\frac{R}{H}$. Compared to live benchmarking, this design offers higher accuracy, with similar resources. Compared to sequential benchmarking, this design can be run in half the wall-clock time, but with twice as many hosts. It is therefore suitable for benchmarking when request replaying is available and time budget is more important than host budget.

Host balanced design (“sequential benchmarking”). The host balanced design executes requests using different versions of the software on the same host, and thus requires two batches. Each request is again only executed once, so a request replaying ability is not required (see Table 2 (c)).

Compared to live benchmarking, sequential benchmarking takes twice as long to run, but can use just half the number of hosts. It may therefore be useful in situations with no replay ability and a

[7]When the outcome variable is log-transformed, this δ corresponds to a multiplicative effect.

[8]Alternatively, one can think of the experiment as involving subjects and items [3] (which correspond to hosts and requests). The two-batch experiments correspond to within-subjects designs, and single-batch experiments correspond to between-subjects designs.

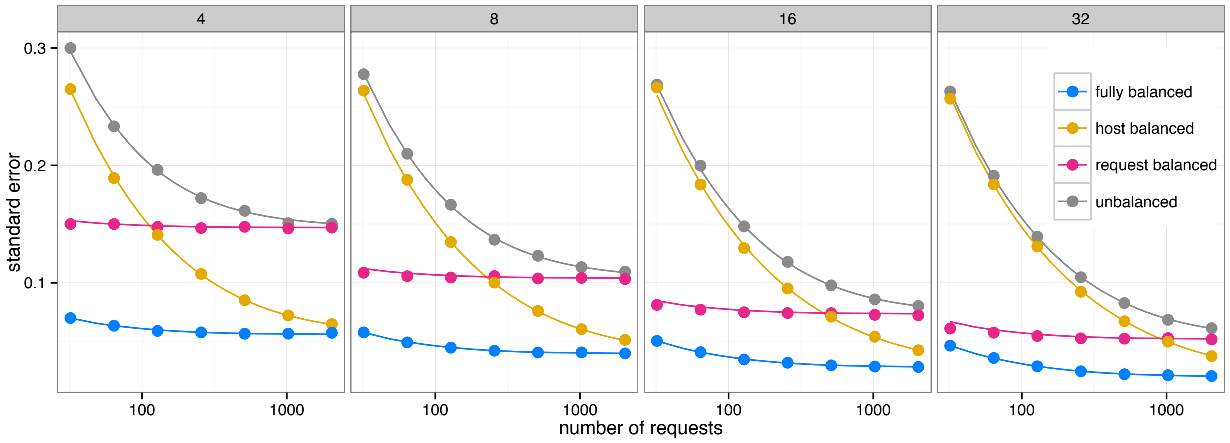

Figure 4: Standard errors for each of the four experimental designs as a function of the number of requests and hosts. Lines are theoretical standard errors from Section 4.2.1 and points are empirical standard errors from 10,000 simulations. Panels indicate the number of hosts used in the benchmark, and the number of requests are on a log scale.

limited number of hosts for benchmarking. It is also more accurate, for a given number of observations, as shown by its standard error:

$$\mathrm{SE}_{\mathrm{HB}}(\hat{\delta}) = \sqrt{2\big(\frac{1}{R}(\sigma_\alpha^2 + \sigma_\gamma^2) + \frac{1}{H}\sigma_\eta^2 + \frac{1}{RT}\sigma_\epsilon{}^2\big)}$$

The host balanced design cancels out the effect of the host, but one is left with noise due to the request. Note that the number of hosts does not affect the precision of the SEs in this design.

Fully balanced design ("controlled benchmarking"). The fully balanced design achieves the most accuracy with the most resources. It executes the same request on the same host using different versions of the software. This requires that one has the ability to split or replay traffic, and takes two batches to complete (see example layout in Table 2 (d)). This design has by far the least variance of the four designs, and is the best choice for benchmarking experiments in terms of accuracy, as shown by the standard error:

$$\mathrm{SE}_{\mathrm{FB}}(\hat{\delta}) = \sqrt{2(\frac{1}{R}\sigma_\gamma^2 + \frac{1}{H}\sigma_\eta^2 + \frac{1}{RT}\sigma_\epsilon{}^2)}$$

This design requires replay ability, as well as twice the machines of sequential benchmarking and twice the wall-clock time (batches) of live and parallel benchmarking. However, it is so much more accurate than the other designs that it requires far fewer observations (requests) to reach the same level of accuracy. Depending on the tradeoffs between request running time costs and batch setup cost, as well as the desired accuracy, this design may end up taking less computational resources than the other designs.

4.2.2 *Analysis*

We analyze each design via simulation and visualization. We obtained realistic simulation parameters from our model fits to the top endpoint (Table 1). Our simulation then simply draws random $\hat{y}$ values from normal distributions using these parameters. For any given design, the simulations differ in that host effects, request effects, and random noise are redrawn from a normal distribution with σ_α, σ_β, and σ_ε, respectively.

We first consider simulations for each experimental design with a fixed number of hosts and requests and zero average effects. Figure 5 shows the distribution of $\hat{\delta}$s generated from 10,000 simulations. We can see that the fully balanced design has by far the least

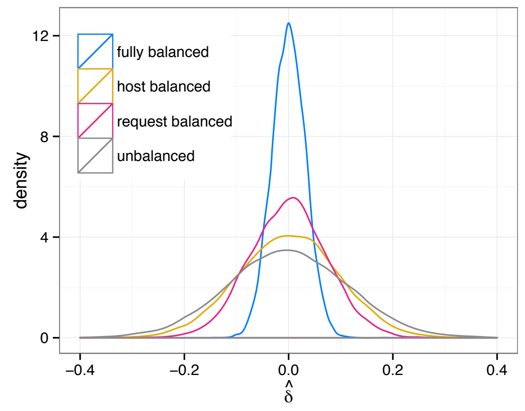

Figure 5: Distribution of $\hat{\delta}$s for each of the four experimental designs. The data was generated by simulating 10,000 hypothetical experiments from the random effects model in on Eq. 3 using parameter estimates from endpoint 1 in Table 1 with 16 hosts and 256 requests, for each design.

variance in $\hat{\delta}$, followed by the request balanced, host balanced, and unbalanced designs.

Next, we explore the parameter space of varying hosts and requests in Figure 4. In all cases, adding hosts or adding requests narrows the SE. For the unbalanced and host balanced designs, the effect of the number of requests on the SE is much more pronounced than that of the number of hosts: in the former because variability due to requests is much higher than that due to hosts, as shown in Figure 1; and in the latter because we control for the hosts. Similarly, the request balanced design controls for requests, and therefore shows little effect from varying the number of requests. And finally, the fully balanced design exhibits both the smallest SE in absolute terms, as well as the least sensitivity to the number of hosts.

5. BOOTSTRAPPING

So far we have discussed simple models that help us understand the main levers that can improve statistical precision in distributed, user-based benchmarks. Our theoretical results, however, assume that the model is correct, and that parameters are known or are easily estimated from data. This is generally not the case, and in fact, estimating models from the data may require many more observations than is necessary to estimate an average treatment effect.[9]

In this section we will review a simple non-parametric method for performing statistical inference—the bootstrap. We then evaluate how well it does at reconstructing known standard errors based on simulations from the random effects model. Finally, we demonstrate the performance of the bootstrap on real production benchmarks from Perflab.

5.1 Overview of the bootstrap

Often times we wish to generate confidence intervals for an average without making strong assumptions about how the data was generated. The bootstrap [12] is one such technique for doing this. The *bootstrap distribution* of a sample statistic (e.g., the difference in means between two experimental conditions) is the distribution of that statistic when observations are resampled [12] or reweighted [23, 25]. We describe the latter method because it is easiest to implement in a computationally efficient manner.

The most basic way of getting a confidence interval for an average treatment effect for iid data is to reweight observations independently, and repeat this process R times. We assign each observation i a weight $w_{r,i}$ from a mean-one random variable, (e.g., Uniform(0,2) or Pois(1)) [23, 25] and use them to average the data, using each replicate number r and observation number i as a random seed:

$$\hat{\delta}_r^* = \frac{1}{N_r^*}\sum_i w_{r,i} y_i I(D_i = 1) - \frac{1}{M_r^*}\sum_i w_{r,i} y_i I(D_i = 0)$$

Here, N_r^* and M_r^* denote the sum of the bootstrap weights (e.g., $\sum_i w_{r,i} I(D_i = 1)$) under the treatment and control, respectively. This process produces a distribution of our statistic, the sample difference in means, $\hat{\delta}^*_{r=1\ldots,R}$. One can then summarize this distribution to obtain confidence intervals. For example, to compute the 95% confidence interval for $\hat{\delta}$ by taking the 2.5th and 97.5th quantiles of the bootstrap distribution of $\hat{\delta}$. Another method is to use the central limit theorem (CLT). The distribution of our statistic is expected to be asymptotically normal, so that one can compute the 95% interval using the quantiles of the normal distribution with a mean and standard deviation set to the sample mean and standard deviation of the $\hat{\delta}_r^*$s. The CLT intervals are generally more stable than directly computing the quantiles of the bootstrap distribution, so we use this method for all bootstrap confidence intervals given in this paper.

Similar to how the iid standard errors in Section 3.1.1 underestimate the variability in $\bar{Y}$, we expect the iid bootstrap to underestimate the variance of $\hat{\delta}$ when observations are clustered, yielding overly narrow ("anti-conservative") confidence intervals and high false positive rates [21]. The solution to this problem is to use a *clustered bootstrap*. In the clustered bootstrap, weights are assigned for each factor level, rather than observation number. For example, if we wish to use a clustered bootstrap based on the request ID, as to capture variability due to the request, we can assign all observations for a particular host to a weight. In the case of requests, we would instead use host IDs and replicate numbers as our random number seed, and for each replicate, compute:

[9]For example, identifying host-level effects requires executing the same request on multiple machines, and identifying request-level effects requires multiple repetitions of the same request. Both increase request-level duplication, which reduces efficiency when request-level effects are large relative to the noise, σ_ε (Sec. 4.2).

$$\hat{\delta}_r^* = \frac{1}{N_r^*}\sum_i w_{r,h[i]} y_i I(D_i = 1) - \frac{1}{M_r^*}\sum_i w_{r,h[i]} y_i I(D_i = 0)$$

5.2 Bootstrapping experimental differences

Before examining the behavior of the bootstrap on production behavior, we validate its performance based on how well it approximates known standard errors, as generated by our idealized benchmark models from Section 3. To illustrate how this works, we can plot the true distribution of $\hat{\delta}$s drawn from the 10,000 simulations for the fully balanced design shown in Figure 6, along with the $\hat{\delta}^*$s under the host and request-clustered bootstrap for a single experiment. The host-clustered bootstrap, which accounts for the variation induced by repeated observations of the same host, tends to produce estimates of the standard errors that are similar to the true standard error, while the request-clustered bootstrap tends to be too narrow, in that it underestimates the variability of $\hat{\delta}$.

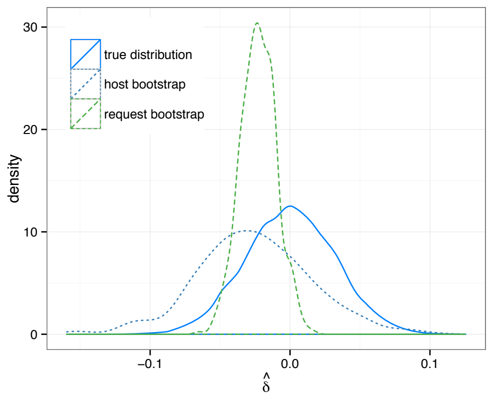

Figure 6: Comparison of the distribution of $\hat{\delta}$s generated by 10,000 simulations (solid line) with the distribution of bootstrapped $\hat{\delta}^*$s from a single experiment (dashed lines).

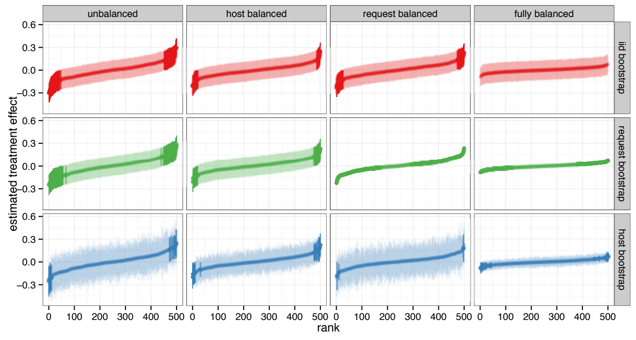

Figure 7: Visualization of bootstrap confidence intervals from 500 simulated A/A tests, run with different experimental designs and bootstrap methods. Experiments are ranked by estimated effect size. Shaded error bars indicate false positives. The request-level and iid bootstraps yield overly narrow confidence intervals.

Next, we evaluate three bootstrapping procedures—iid, host-clustered, and request-clustered—with each of the four designs for a single endpoint. To do this, we use the same model parameters from endpoint 1, as in previous plots. To get an intuitive picture for

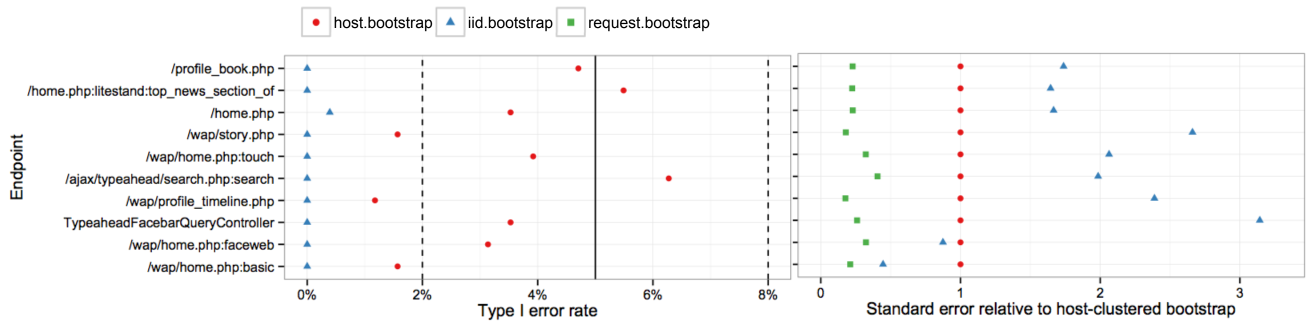

Figure 8: Empirical validation of bootstrap estimators for the top 10 endpoints using production data from Perflab. Left: Type I error rates for host-clustered and IID bootstrap (request-clustered bootstrap has a Type I error rate of $> 20\%$ for all endpoints and is not shown); Solid line indicates the desired 5% Type I error rate, and the dashed lines indicate the range of possible observed Type I error rates that would be consistent with a true error rate of 5%. Right: Standard error of host-clustered, request-clustered, and iid bootstrap relative to the host-clustered standard error.

how the confidence intervals are distributed, we run 500 simulated A/A tests, and for each configuration, we rank-order experiments by the point estimates of $\hat{\delta}$, and visualize their confidence intervals (Figure 7). Shaded regions represent false positives (i.e., their confidence intervals do not cross 0). The iid bootstrap consistently produces the widest CIs for all designs. This happens because the iid bootstrap doesn't preserve the balance across hosts or requests across conditions when resampling. There is also a clear relationship between the width of CIs and the Type I error rate, in that there are a higher proportion of type I error rates when the CIs are too narrow.

To more closely examine the precision of each bootstrap method with each design, we run 10,000 simulated A/A tests for each configuration and summarize their results in Table 3. For the request-balanced design, we also include an additional bootstrap strategy, which we call the host-block bootstrap, where pairs of hosts that execute the same requests are bootstrapped. This ensures that when whole hosts are bootstrapped, the balance of requests is not broken. This method turns out to produce standard errors with good coverage for the request-balanced design, and so we will henceforth refer to the host–block bootstrap strategy as the host-clustered bootstrap in further analyses. We can see that the host-clustered bootstrap appears to estimate the true SE most accurately for all designs.

Design	Bootstrap	Type I err.	$\widehat{\text{SE}}$	SE
unbalanced	request	21.4%	0.07	0.10
unbalanced	iid	17.8%	0.07	0.10
unbalanced	host	3.6%	0.11	0.10
request balanced	request	71.6%	0.01	0.07
request balanced	iid	10.8%	0.07	0.07
request balanced	host	0.8%	0.11	0.07
request balanced	host-block	5.7%	0.08	0.07
host balanced	request	8.4%	0.07	0.07
host balanced	iid	6.8%	0.07	0.07
host balanced	host	5.0%	0.08	0.07
fully balanced	request	46.8%	0.01	0.03
fully balanced	host	4.8%	0.03	0.03
fully balanced	iid	0.0%	0.07	0.03

Table 3: Comparison of Type I error rates and standard errors for each experimental design and bootstrap strategy. $\widehat{\text{SE}}$ indicates the average estimated standard error from the bootstrap, while SE indicates the true standard error from the analytical formulae. Anti-conservative bootstrap estimates of the standard errors produce high false positive rates (e.g., >5%).

5.3 Evaluation with production data

Finally, having verified that the bootstrap method provides a conservative estimate of the standard error when the true standard error is known (because it was generated by our statistical model), we turn toward testing the bootstrap on raw production data from Perflab, which uses the fully balanced design. To do this, we conduct 256 A/A tests using identical binaries of the Facebook WWW codebase. Figure 8 summarizes the results from these tests for the top 10 most visited endpoints that are benchmarked by Perflab.

Consistent with the results of our simulations, we find that the request-level bootstrap is massively anti-conservative, and produces confidence intervals that are far too narrow, resulting in a high false positive rate (i.e., $> 20\%$) across all endpoints. Similarly, we find that our empirical results echo that of the simulations: the iid bootstrap produces estimates of the standard error that are far wider than they should be. The host-clustered bootstrap, however, produces Type I error rates that are not significantly different from 5% for all but 3 of the endpoints; in these cases, the confidence intervals are slightly conservative, as is desired in our use case. For these reasons, all production benchmarks conducted at Facebook with Perflab use the host-clustered bootstrap.

6. RELATED WORK

Many researchers recognized the obstreperous nature of performance measurement tools, and addressed it piecemeal. Some focus on controlling architectural performance variability, which is still a very active research field. Statistical inference tools can be applied to reduce the effort of repeated experimentation [11, 19]. These studies focus primarily on managing host-level variability (and even intra-host-level variance), and do not spend much attention on the variability of software.

Variability in software performance has been examined in many other studies that attempt to quantify the efficacy of techniques such as: allowing for a warm-up period [8]; reducing random performance fluctuations using regression benchmarking [18]; randomizing multi-threaded simulations [1]; and application-specific benchmarking [27]. In addition, there is an increasing interest specifically in the performance of online systems and in the modeling of large-scale workloads and dynamic content [2, 6].

Several studies addressed the holistic performance evaluation of hardware, software, and users. Cheng et al. described a system called *Monkey* that captures and replays TCP workloads, allowing the repeated measuring of the system under test without generating synthetic workloads [9]. Gupta et al., in their *Diecast* system, addressed the challenge of scaling down massive-scale distributed systems into representative benchmarks [15]. In addition, representative characteristics of user-generated load is critical for benchmarking large-scale online systems, as discussed by Manley et al. [20]. Their system, *hbench:Web*, tries to capture user-level variation in terms of *user sessions* and user *equivalence classes*. Request-level variation is modeled statistically, as opposed to measuring it directly.

A few studies also proposed statistical models for benchmarking experiments. Kalibera et al. showed that simply averaged multiple repetitions of a benchmark without accounting for sources of variation can produce unrepresentative results [17, 18]. They developed models focusing on minimizing experimentation time under software/environment random effects, which can be generalized to other software-induced variability. These models are similar to the model developed here, but they do not take into account host-level or user-level effects, which are central to the experimental design and statistical inference problem we wish to address. To the best of our knowledge, this is the first work to rigorously address distributed benchmarking in the context of user data.

7. CONCLUSIONS AND FUTURE WORK

Benchmarking for performance differences in large Internet software systems poses many challenges. On top of the many well-studied requirements for accurate performance benchmarking, we have the additional dimension of user requests. Because of the potentially large performance variation from one user request to another, it is crucial to take the clustering of user requests into account. Yet another complicating dimension, which is nevertheless critical to scale large testing systems, is distributed benchmarking across multiple hosts. It too introduces non-trivial complications with respect to how hosts interact with request-level effects and experimental design to affect the standard error of the benchmark.

In this paper we developed a statistical model to understand and quantify these effects, and explored their practical impact on benchmarking. This model enables the analytical development of experimental designs with different engineering and efficiency tradeoffs. Our results from these models show that a fully balanced design—accounting for both request variability and host variability—is optimal in minimizing the benchmark's standard error given a fixed number of requests and machines. Although this design may require more computational resources than the other three, it is ideal for Facebook's rapid development and deployment mode because it minimizes developer resources.

Design effects due to repeated observations from the same host also show how residual error terms that cannot be canceled out via balancing, such as batch-level host effects due to JIT optimization and caching, can also be an important lever for further increasing precision, especially when there are few hosts relative to the number of requests.

From a practical point of view, estimating the model parameters to compute the standard errors for these experiments (especially in a live system) can be costly and complex. We showed how the clustered online bootstrap can be used to estimate the standard error for a variety of experimental designs. Using empirical data from Facebook's largest differential benchmarking system, Perflab, we confirm that this technique can reliably capture the true standard error with good accuracy. Consequently, all production Perflab experiments use a fully-balanced design, and the host-level bootstrap to evaluate changes in key metrics, including CPU time, instructions, memory usage, etc. Our hope is that this paper provides a simple and actionable understanding of the procedures involved in benchmarking for performance changes in other contexts as well. With a more quantitatively informed approach, practitioners can select the most suitable experimental design to minimize the benchmark's run time for any desired level of accuracy.

Our results focus on measuring the performance of a single endpoint or service. Often times one wishes to benchmark multiple such endpoints or services. Pooling across these endpoints to obtain a composite standard error is trivial if one could reasonably regard the performance of each endpoint as independent.[10] If there is a strong correlation between endpoints, however, the models here must be extended to take into account covariances between observations from different endpoints.

Finally, another area of development for future work is a more extensive evaluation of the costs associated with each of the four basic designs. For example, one might devise a cost model which takes as inputs the desired accuracy and parameters such as batch setup time, fixed and marginal costs per request, etc. These cost functions' formulae would depend on the particulars of the benchmarking platform. For example, in some systems the number of requests needed for stable measurements for may depend on some maximum load per host. How much "warm up" time different subsystems require is nonlinear in the number of different requests, and might also depend on how services are distributed across machines.

8. ACKNOWLEDGEMENTS

Perflab is developed by many engineers on the Site Efficiency Team at Facebook. We are especially thankful for all of the help from David Harrington and Eunchang Lee, who worked closely with us on implementing and deploying the testing routines discussed in this paper. This work would not be possible without their help and collaboration. We also thank Dean Eckles and Alex Deng for their suggestions and feedback on this work.

[10]If each endpoint i constitutes some fraction w_i of the traffic, and the outcomes for each endpoint are uncorrelated, the variance of δ pooled across all endpoints is simply $\mathrm{Var}[\hat{\delta}_{\text{pool}}] = \sum_i w_i^2 \mathrm{Var}[\hat{\delta}_i]$. This independence assumption can be made more plausible by loading endpoints with disjoint sets of users.

9. REFERENCES

[1] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating non-deterministic multi-threaded commercial workloads. In *In Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, 2002.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th Joint Conference On Measurement And Modeling of Computer Systems (SIGMETRICS/Performance'12)*, London, UK, June 2012.

[3] R. H. Baayen, D. J. Davidson, and D. M. Bates. Mixed-effects modeling with crossed random effects for subjects and items. *Journal of Memory and Language*, 59(4):390–412, 2008.

[4] E. Bakshy and D. Eckles. Uncertainty in online experiments with dependent data: An evaluation of bootstrap methods. In *Proceedings of the 19th ACM SIGKDD conference on knowledge discovery and data mining*. ACM, 2013.

[5] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA'05, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.

[6] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 ACM SIGMETRICS joint International Conference on Measurement and modeling of Computer Systems*, SIGMETRICS '98/PERFORMANCE '98, pages 151–160, New York, NY, USA, 1998. ACM.

[7] G. E. Box, J. S. Hunter, and W. G. Hunter. *Statistics for Experimenters: Design, Innovation, and Discovery*, volume 13. Wiley Online Library, 2005.

[8] A. Buble, L. Bulej, and P. Tuma. Corba benchmarking: A course with hidden obstacles. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 6 pp.–, 2003.

[9] Y. chung Cheng, U. Hölzle, N. Cardwell, S. Savage, and G. M. Voelker. Monkey see, monkey do: A tool for tcp tracing and replaying. In *In USENIX Annual Technical Conference*, pages 87–98, 2004.

[10] W. G. Cochran. *Sampling techniques*. John Wiley & Sons, 2007.

[11] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Designing computer architecture research workloads. *Computer*, 36(2):65–71, 2003.

[12] B. Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26, 1979.

[13] D. G. Feitelson. Workload modeling for computer systems performance evaluation. Unpublished manuscript, v. 0.42. **www.cs.huji.ac.il/~feit/wlmod/wlmod.pdf**.

[14] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and Deployment at Facebook. *IEEE Internet Computing*, 17(4), July 2013.

[15] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Trans. Comput. Syst.*, 29(2):4:1–4:48, May 2011.

[16] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for experimental design, measurement, simulation, and modeling*. Wiley, 1991.

[17] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on International Symposium on Memory Management*, ISMM'13, pages 63–74, New York, NY, USA, 2013. ACM.

[18] T. Kalibera and P. Tuma. Precise regression benchmarking with random effects: Improving mono benchmark results. In *Formal Methods and Stochastic Models for Performance Evaluation*, volume 4054 of *Lecture Notes in Computer Science*, pages 63–77. Springer Berlin Heidelberg, 2006.

[19] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 185–194, New York, NY, USA, 2006. ACM.

[20] S. Manley, M. Seltzer, and M. Courage. A self-scaling and self-configuring benchmark for web servers (extended abstract). In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'98/PERFORMANCE'98, pages 270–291, New York, NY, USA, 1998. ACM.

[21] P. McCullagh. Resampling and exchangeable arrays. *Bernoulli*, 6(2):285–301, 2000.

[22] A. B. Owen. The pigeonhole bootstrap. *The Annals of Applied Statistics*, 1(2):386–411, 2007.

[23] A. B. Owen and D. Eckles. Bootstrapping data arrays of arbitrary order. *The Annals of Applied Statistics*, 6(3):895–927, 2012.

[24] D. B. Rubin. Estimating causal effects of treatments in randomized and nonrandomized studies. *Journal of Educational Psychology*, 66(5):688–701, 1974.

[25] D. B. Rubin. The Bayesian bootstrap. *The Annals of Statistics*, 9(1):130–134, 1981.

[26] S. R. Searle, G. Casella, C. E. McCulloch, et al. *Variance Components*. Wiley New York, 1992.

[27] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang. The case for application-specific benchmarking. In *In Workshop on Hot Topics in Operating Systems (HOTOS'99)*, pages 102–107, 1999.

[28] T. A. Snijders. *Multilevel analysis*. Springer, 2011.