ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Avoiding the Ordering Trap in Systems Performance Measurement

Dmitry Duplyakin, *Nikhil Ramesh*, *Carina Imburgia*[*]
Hamza Fathallah Al Sheikh, *Semil Jain*, *Prikshit Tekta*
Aleksander Maricq, *Gary Wong*, *Robert Ricci*

University of Utah [*]University of Washington

Abstract

It is common for performance studies of computer systems to make the assumption—either explicitly or implicitly—that results from each trial are independent. One place this assumption manifests is in experiment design, specifically in the order in which trials are run: if trials do not affect each other, the order in which they are run is unimportant. If, however, the execution of one trial *does* affect system state in ways that alter the results of future trials, this assumption does not hold, and *ordering must be taken into account* in experiment design. In the simplest example, if all trials with system setting A are run before all trials with setting B, this can systematically bias experiment results leading to the *incorrect* conclusion that "A is better than B" or vice versa.

In this paper, we: (a) explore, via a literature and artifact survey, whether experiment ordering is taken in to consideration at top computer systems conferences; (b) devise a methodology for studying the effects of ordering on performance experiments, including statistical tests for order dependence; and (c) conduct the largest-scale empirical study to date on experiment ordering, using a dataset we collected over 9 months comprising nearly 2.3M measurements from over 1,700 servers. Our analysis shows that ordering effects are a hidden but dangerous trap that published performance experiments are not typically designed to avoid. We describe OrderSage, a tool that we have built to help detect and mitigate these effects, and use it on a number of case studies, including finding previously unknown ordering effects in an artifact from a published paper.

1 Introduction

Systems performance analysis typically involves running a series of trials and then calculating statistical measures (such as mean or median) from the performance data collected. These measures are used to conclude that one system is, on average $X\%$ faster than another, that the addition of a new feature does not have a statistically-significant impact on performance [12, 16], or that software scales well to large problem sizes. One of the most fundamental assumptions of this kind of analysis [36] is that trials are *independent*; in particular, that each trial is unaffected by prior trials in the series. If this assumption does not hold, it can systematically bias results and alter or even invalidate conclusions drawn from them.

Typical systems research work does not take ordering into consideration as part of experiment design. This can lead to violations of the independence assumption.

The problem is especially pernicious because there is not one, or even a few, root causes behind performance-affecting state that carries over between trials. In the highly complex environment of a modern computer system, there are a large number of hardware and software components whose state can be carried over from one trial to another [26]. These include caches [8], data layout in RAM and on disk [22], application and operating system tuning parameters [20, 41], and even temperature (with consequences such as thermal throttling [3, 13]). The systems under test themselves can, intentionally or unintentionally, make changes that persist between trials, such as changes to software packages, global system configuration, environment variables [26], or files.

Thus, while the question of *why* order matters is important, it is highly specific to the software being tested, the hardware it is run on, and the design of the experiment. Before "*why*" can be considered, there is the more fundamental question of *whether* the order matters for a specific experiment. In many cases, knowing that order-dependent performance *exists* can itself be an interesting result because it indicates some unexpected property of the software or system under test. Therefore, eliminating it entirely through experiment design is not always even desirable.

In this paper, we formulate a systematic approach to analyzing whether the order of trials within an experiment affects results. We use this method to collect and analyze a large new performance dataset that we collected on over 1,700 servers over a period of 9 months and show that experiment order is a factor that cannot be neglected. We find that for the selected benchmarks the order can bias performance by 50% or more and potentially alters conclusions in 72% of cases.

Order is acknowledged to have some level of impact in the literature [1, 26]. However, we show this acknowledgment has not translated into experiment design *in practice*. We conducted a survey of three major systems conferences and found that it is exceedingly uncommon to discuss experiment ordering in these papers. Furthermore, we examined the artifacts for the papers and find that they are not designed to detect or avoid ordering effects. To help relieve this situation, we contribute OrderSage, a tool that helps experimenters with both the orchestration and analysis aspects of experiment ordering.

In this paper, we make the following contributions:

- We perform a literature survey of top-tier systems conferences (Section 2), showing that experiment order is reported as part of experiment design in fewer than 10% of papers. We also analyze these paper's artifacts, and show that this neglect extends to the way experiments are run in practice: more than 94% of artifacts run experiments in either a single fixed order or do not specify an order.
- We develop a methodology (Section 3), using established statistical tests, for determining whether results are order-dependent and narrowing down specific experimental tests that are particularly affected.
- We collect and publicly release a large, first-of-its-kind dataset for studying the impact of ordering on performance experiment results (Section 4). This dataset contains the results of over 2.3M trials run in a variety of different orders.
- We analyze this dataset using our methodology (Section 5) and show that ordering can make a significant difference, even to the level of potentially changing conclusions. This provides strong evidence for the claim that systems researchers should consider order in their experiment design.
- We developed and release OrderSage, a tool that easily applies our methodology to performance experiments (Section 6). OrderSage embodies both a mechanism for randomizing experiment order and analyzing its effects. To demonstrate its use, we present case studies (Section 7) applying it to the performance test-suite for `memcached` [2, 7, 21], and to NPBench [43]. We also use OrderSage on one of the artifacts from our literature survey and find a previously-unknown ordering effect in it.

We cover related work in Section 8 and conclude in Section 9.

2 Literature and Artifact Survey

To evaluate the extent to which ordering effects are taken into account in practice in the systems research literature, we conducted a survey involving the OSDI '21, SOSP '21, and EuroSys '22 conferences. We selected these three conferences because they ran *Artifact Evaluation Committees* (AECs), meaning that we were able to look at both what *papers say* about ordering and what the artifacts (code, scripts, etc.) *actually do*.

We had two inclusion criteria for our survey. First, the papers need to have received all three AEC badges (Available, Functional, and Reproduced)—this lets us know that not only did the paper have an artifact submitted, but that the artifact is complete. Second, the papers need to base their main claims

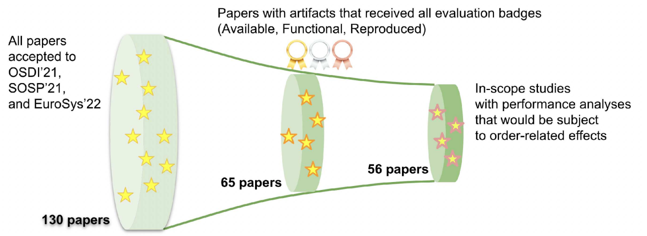
Papers with artifacts that received all evaluation badges (Available, Functional, Reproduced)
All papers accepted to OSDI'21, SOSP'21, and EuroSys'22
In-scope studies with performance analyses that would be subject to order-related effects
130 papers
65 papers
56 papers

Figure 1: Paper selection for our literature and artifact survey.

on a set of performance metrics (e.g., runtime, latency, bandwidth, etc.) executed on real (not simulated) systems. Under these criteria, we ended up with 56 papers out of the three conferences' 130 papers, as seen in Figure 1.

Following the selection and filtering phase, we performed the survey in two passes; each paper/artifact was reviewed by a different reviewer in each pass, with the goal of countering individual reviewers' biases. Table 1 presents the results from both passes as well as the agreement between the two passes. It is worth noting that the spread is larger for the artifact analyses because they required more investigation than reading the evaluation sections of the surveyed papers. Regardless, we consider all observed relative agreement numbers to be high enough to serve as a convincing basis for our conclusions. As detailed below, our artifact review provided much more insight into how the studies were run compared to the information in the papers alone.

Do papers specify the ordering of their experiment design? Only 4 out of 56 papers (7%) clearly stated the order in which the corresponding performance experiments were run. This percentage is not surprising, because space constraints lead authors to focus on describing the factors that are key to their work instead of latent factors such as the order of execution. Of special note is that EuroSys '22 allowed artifact description appendices, which we considered as part of the paper rather than part of the artifact. This is where we found most of the ordering-related information; these appendices allowed authors to detail steps in their evaluation workflows, leaving no ambiguity about the orderings.

Do papers describe their inter-experiment reset procedures? Between 4 and 10 papers, or 7–18%, described reset procedures for ensuring that subsequent trials are not potentially impacted by the preceding tests. Such procedures included clearing caches, running warmup tests, rebooting hardware, and launching new cloud instances, among others. Similar to our conclusion about the order information, the reset specification was scarce in the studied papers.

What order do artifacts execute experiments in? Because papers do not tell us much about what order is used for experiments, we examined the artifacts themselves. 36 to 37 artifacts (64–66%) use a fixed-order experiment design. This was typically implemented by providing a "run all" script

Table 1: Results of studying 56 papers and the corresponding artifacts.

Attribute being tested	**1st pass**	**2nd pass**	**Match b/w passes**
Paper explicitly describes an order of experiment execution	4 (7%)	4 (7%)	93%
Paper describes a reset procedure to be run between experiments	4 (7%)	10 (18%)	82%
Artifact's primary experiment execution order:			63%
fixed	36 (64%)	37 (66%)	
undefined	17 (30%)	17 (30%)	
parallel	3 (5%)	2 (4%)	
Artifact runs a reset procedure between experiments	27 (48%)	16 (29%)	73%

that iterates through the studied algorithms or configuration options in sequence with no randomization. In other cases, a specific order was documented in the repository's README files. Many other artifacts (17, or 30%) provided instructions on how to run individual groups of tests (e.g., for specific figures and sections in the papers) but did not specify any sequence between them—we categorized their orderings as undefined. Another small class of studies (2–3 artifacts) used parallel execution, where tests were run concurrently on multiple worker machines or cloud instances and therefore can be considered to run each test in its own clean environment. We have not identified any artifacts that implemented a randomized ordering, or which clearly showed explicit attention to ordering concerns. To summarize, 53–54 artifacts out of 56 (94–96%) used undefined or fixed orderings, both of which can be questioned from the presentation and experiment design perspectives. Expanding on the latter case, we show in this study that fixed-order experiment designs have potential to introduce adverse bias in performance analysis.

Do artifacts use inter-experiment reset procedures? Between 16 and 27 artifacts (29–48%) ran identifiable procedures to reset the system to a known state between experiments. Finding these procedures in the code is a non-trivial and time-consuming process, which explains the spread between the results in the two survey passes. While we did find reset procedures in up to half of the artifacts, it is concerning that the other half of the artifacts did not manifest any reset procedures. While it may not matter for some of the studies because of the nature of their performance analysis, there is a chance that for a subset of them it may be an oversight causing undesirable effects on their conclusions.

From the survey, we learn that the literature does not make order an explicit part of experiment design, and we do not see evidence that ordering issues are explicitly addressed. In the remainder of this paper, we show how this can constitute a trap for experimenters and discuss how to avoid this trap.

3 Analyzing Order Dependence

In this section, we detail the procedure we have designed to find order dependence in performance experiments. There are two primary outputs from this procedure: first, it reports whether the statistical distribution of performance results differs when run using *fixed-order* and *random-order* experiment designs—that is, whether the order has an effect on the experiment results. Second, it reports whether these differences are potentially large enough to *change inferences*—that is, whether it is possible for ordering effects to be large enough that the conclusions drawn from an experiment could change based on the execution order.

For consistency, we use the following terminology in this section and throughout the remainder of the paper:

Test: A test is an individual unit of the system under evaluation. A test typically represents an individual benchmark or an application with a specific configuration or input.

Trial: A trial is an execution of a test. The outcome of a trial is a single-metric performance assessment, such as runtime, throughput, latency, etc. Multiple trials of the same test typically exhibit variations in performance stemming from the nondeterminism intrinsic to the test itself or the system used for benchmarking.

Run: A run is a set of trials, in a particular order, of *all* tests in series. Conceptually, one could (and often does) report the results from a single run in a single order.

Experiment: An experiment is a collection of one or more runs done for the purpose of reaching a conclusion about the system(s) under evaluation; typically, such a conclusion will be reached by comparing results of the trials associated with different tests. The order of trials within the runs of an experiment is part of the *experiment design* and is referred to as the *experiment order*.

An outline of the method is shown in Algorithm 1; below, we go through each step in detail.

❶ Select a "Baseline" Order Select an ordering of trials that will be used for "fixed order" runs. The order itself is not important; the order in which trials have been run in the past, or a "natural" order (such as by increasing parameter value) is sufficient. This does not need to be a "correct" order: it will act as the control against which we test random orderings.

❷ Define a "Reset to Clean State" Procedure Each run (series of trials) should start from a clean state, such that

Algorithm 1 Order-Dependence Test

```
Input T: List of trials in baseline order                          ▷ ❶
Input R: Reset procedure                                           ▷ ❷
Input N: Number of repetitions
Input α: Desired family-wise error rate (commonly 0.05)
 1: for $n = 1, \ldots, N$ do                                       ▷ ❸
 2:     Execute $R$
 3:     for all $t \in T$ do                 ▷ Run trials in baseline order
 4:         fixedOrderResults$[t][n] \leftarrow$ Execute $t$
 5:     end for
 6:     Execute $R$
 7:     for all $t \in$ RandomlyPermute$(T)$ do  ▷ Run trials in random order
 8:         randomOrderResults$[t][n] \leftarrow$ Execute $t$
 9:     end for
10: end for
                                                                   ▷ ❹
11: for all $t \in T$ do          ▷ Calculate p-values for distribution comparison
12:     $p_{KW}[t] \leftarrow$ KruskalWallis(fixedOrderResults$[t]$, randomOrderResults$[t]$)
13: end for
14: $\alpha_{BC} \leftarrow \alpha / length|T|$   ▷ Use Bonferroni corr. for multiple comparisons
15: if $\exists t \in T \mid p_{KW}[t] < \alpha_{BC}$ then
16:     return true       ▷ Order matters for 1 test → it matters for the experiment
17: else
18:     return false
19: end if
```

Algorithm 2 CI Overlap Test

```
Input fixedOrderResults, randomOrderResults from Algorithm 1
Input t: test to check
1: (fLow, fMedian, fHigh) $\leftarrow$ RankBasedCI(fixedOrderResults$[t]$)    ▷ ❺
2: (rLow, rMedian, rHigh) $\leftarrow$ RankBasedCI(randomOrderResults$[t]$)
3: if (fLow > rHigh) $\vee$ (fHigh < rLow) then
4:     return Case 1                                ▷ Inference does change
5: else if (fLow < rMedian < fHigh) $\wedge$ (rLow < fMedian < rHigh) then
6:     return Case 2                     ▷ Inference likely does not change
7: else
8:     return Case 3                  ▷ Inference may or may not change
9: end if
```

state left over from earlier runs will not affect performance results of the next run. In many cases, this will be much more expensive than running the benchmarks themselves: for the purposes of the experiments in this paper, this procedure is a reboot of the server on which the benchmarks are executed. This might instead consist of restarting a server process, provisioning fresh VMs, clearing storage devices, etc.

❸ Run in Both Fixed and Random Orders We execute a series of runs. Each run consists of the same set of trials, and each trial may comprise of multiple invocations of the system under evaluation in order to increase statistical significance. For half of our runs, trials are run in the fixed baseline order; in the other half, the order is randomly permuted (separately for each run). Between runs, the environment is reset to a clean state using ❷. Since the evaluation might take long enough that time-varying effects (such as hardware degradation) could be observed, fixed (Lines 3–5) and random order (Lines 7–9) runs are interleaved to avoid bias. The outcome of each run is a set of performance results, one from each trial, with the units being the "natural" units for the tests, e.g., seconds for runtime, MB/s for bandwidth, etc. The experimenter should complete a sufficiently sized set of runs (Line 1) to provide the desired statistical significance in the subsequent steps.

❹ Compare Distributions The next step is to compare the samples obtained from the fixed- and random-order runs. The intuition behind this step is that if the two sets of samples come from the same statistical distribution, it can be said that the order does not change the distribution, and thus does not matter. If they come from different distributions, then the order does indeed matter.

To avoid assumptions of normality (which have been shown to rarely hold for computer systems performance results [22]), we use the non-parametric Kruskal-Wallis test (Lines 11–13). This distribution-comparison test produces a p-value indicating the likelihood of observation assuming the null hypothesis (i.e., that both samples come from the same distribution). This should be performed *for each test*, longitudinally across all runs: the two populations are (a) the outcomes for all trials (executions) of the test from fixed-order runs, and (b) the outcomes of all trials for the same test from random-order runs. Thus, we are looking at whether a particular test's performance differs based on where its trials occur in the runs that are differently ordered.

For each test, compare the p-value produced by Kruskal-Wallis with a threshold chosen to provide the confidence level desired; we aim for a family-wise error rate of $\alpha = 0.05$ (95% confidence) as is common in such tests. Because we perform a potentially large number of comparisons, the problem of multiple comparisons [25] arises; we apply the Bonferroni correction [9] (Line 14) to obtain the per-test thresholds (α_{BC}) required for multiple comparisons to reach the target family-wise confidence level. This correction scales the thresholds down (making them stricter) in proportion to the number of comparisons made.

If the p-value is above the threshold, we cannot reject the null hypothesis, and therefore conclude that both samples could have come from the same distribution—the order *likely does not matter*. If the p-value is below the threshold, we reject the null hypothesis and conclude that a single distribution would be highly unlikely to yield the observed samples—*the order of the tests* ***does*** *matter*.

We note that it is possible, and in our experience common, that different tests within an experiment produce different results at this step. This could indicate that some tests are affected by what runs before them and others are more robust in this respect. Overall, however, as long as *any* test shows order-dependence, this indicates that the experiment design as a whole needs to be aware of ordering (Lines 15–19).

❺ Compare Confidence Intervals A typical experiment setup in performance analysis is to ask whether there is a difference in performance between two systems. A situation particularly important to avoid is one in which inferences from the experiment could *change* depending on the ordering, in turn leading to a change in the *conclusions drawn*. We look for such situations by comparing confidence intervals [12] (CIs) as shown in Algorithm 2. CIs can be compared *within* tests (e.g., comparing fixed and random orders, to determine whether order changes the median computed), or *across* tests (e.g., comparing two or more tests and checking whether different performance is observed.)

The outcome of this test tells us something related to, but distinct from, the Kruskal-Wallis test. Kruskal-Wallis tells us whether the distributions differ, but not directly whether they differ enough to change conclusions in a significant way. Looking at the effect size (detailed in Section 5.1.1) gives us a sense of the latter, but the CI test answers it directly. Recall that a CI is an estimated interval we expect to include the true value of a population measure [12]. For instance, for the 95% CI of the median (the interval we use), we expect that in a collection of many such intervals, 95% of our estimates would contain the true population median.

We use rank-based CIs estimating the population median [17] to avoid assumptions of normality. This comparison results in three possible cases (visualized in Figure 5):

Case 1: The CIs for the fixed- and random-order runs do not overlap. In this case, we can have high confidence that we would expect to compute different medians depending on the order. This is a **red flag**, and indicates that we could come to different conclusions based on the order.

Case 2: The median for at least one of the two samples lies within the CI for the other population. If this is the case, given one population, we could have potentially arrived at the *other* observed median, and we conclude that our conclusions likely would not change based on order.

Case 3: In the final case, the CIs overlap, but both medians are outside the other group's interval. This case is inconclusive, and requires more careful analysis to determine if it could change conclusions. Still, it is a potential sign that more careful experiment design is needed.

4 Dataset and Data Collection

To study performance effects at a large scale, we have collected a dataset covering nearly 2.3 million executions (*trials*) of 25 benchmarks on 1,700 machines over a period of nine months. Many benchmarks were run in multiple configurations, such as on different sockets or with different CPU frequency settings, resulting in multiple *tests* per benchmark application. This data was collected across more than 9,000 *runs*. We released this dataset as part of this paper's artifact: https://github.com/ordersage/paper-artifact. Collection of the dataset covers the first three steps of the method described in Section 3; we cover the rest of the steps in the this section.

This dataset focuses on *low-level measurements* of CPU and memory performance through the use of standard benchmarks, in particular STREAM [23], the NASA Parallel Benchmarks [27] (NPB), and Reece's memory benchmarks [30, 31]. We have additional benchmarks of disk and network performance, but leave analysis of them to future work. Our case studies in Section 7 have examples of our methodology applied to higher-level applications.

4.1 Environment

We collected our data by running experiments in CloudLab [5], a public testbed for research use. CloudLab has a variety of different types of server hardware [37], and we ran our experiment across 13 different server types. We considered each configuration of each benchmark on each node type as constituting its own test for the purposes of this analysis: thus, we have 1,880 different collections of corresponding trials to compare. CloudLab is an attractive platform for this work, as it has previously undergone study to quantify and calibrate the level of variability across different hardware in the platform [22].

Servers in CloudLab are allocated at a *bare-metal* level to one user at a time. Disks used are all local to the server, and for this paper, we do not consider the network or other shared resources. Thus, our benchmarks were not affected by any other simultaneous users of the servers in question or the CloudLab system as a whole, and did not have any artifacts due to virtualization. We believe our dataset to be robust with respect to time-varying, location-dependent (e.g., environmental/temperature), and micro-architectural factors: we gathered this data over a period of months; CloudLab nodes are in three different geographically distant data centers; and they encompass a variety of processor and memory technologies.

4.2 Baseline Order ❶

The baseline order that we use is a "natural" one that groups benchmarks from the same suite (e.g., NPB [27]) together, and reflects the order in which we added the suites to our experiment setup. This reflects the type of order that a systems experimenter would be likely to arrive at in the process of developing scripts to run their experiments.

4.3 Reset Procedure ❷

The reset procedure we use is a fresh load of the operating system and clean boot of the host on which the experiments are run. This means that each run sees, as much as possible, the "pristine" state of a just-booted machine, not affected by any software or configuration changes made by prior users.

It is important to note that we do not claim this clean state to be *correct*: we do not claim that the results of a trial gathered under these conditions are more “valid” than results after the machine has been running for some time. It is possible for boot-time effects to alter results, and for some tests, a scenario in which a machine has been booted and active for a long period of time may be more *realistic*. What we *do* claim about this procedure is that we can be confident that all runs *started* from the same state. Therefore, it can tell us if the order of trials within the run affected results.

4.4 Running in Fixed and Random Orders ❸

Our data collection framework allocates machines in CloudLab on which to perform runs. For each run, it randomly chooses—with equal probability—to execute trials in our *fixed baseline* order or a *random order*. This procedure ensures that we *interleave* baseline and random runs, running them in approximately equal proportion throughout the entire time period to avoid a systematic bias in one direction or the other due to potential changes in the facility over time. If the random order is chosen, the framework shuffles the list of all trials for that run. The framework records this order information for use in future analysis.

5 Analysis Applied to Our Dataset

We now describe how we analyze the order-dependence of the performance results gathered in Section 4. This section covers steps ❹ and ❺ from the method described in Section 3.

The nature of our data collection adds another dimension to our *tests*, and thus we adopt terminology used elsewhere in the literature [22] for clarity. Because CloudLab contains servers of many different types, each of the *tests* we define will be executed on 13 different hardware types—each different hardware type may have a different processor, different amount of RAM, etc. We refer to a combination of *{test, hardware type}* as a *configuration*, where the *test* itself is a combination of *{benchmark, settings}*. For example, the STREAM benchmark run in its `COPY` mode on a server of type `m400` represents one configuration; STREAM in `COPY` mode on a server of type `c6520` is another; and STREAM in `SCALE` mode on an `m400` would be a third. In total, we have 1,880 such configurations. Results from *trials* executed under a particular *configuration* across all *runs* are grouped together: our primary comparison of interest is whether the same configuration produces different results when run as part of differently-ordered runs. It is worth noting that we do not expect results from different configurations to be independent, and do not analyze them as such: there is strong likelihood, for example, that STREAM in `COPY` mode exhibits similar order-dependent performance effects to STREAM in `SCALE` mode. The value derived from running so many configurations is that it helps to make our results robust with respect to many different programs, settings for those programs, and types of hardware.

We analyze data from our memory and CPU benchmarks as separate experiments: this avoids mixing results from performance tests with very different goals, and offers interesting insight into how the effects of ordering can differ depending on the main resource being exercised.

5.1 Comparing Distributions ❹

The next step in our method is to compare the distributions of performance results for each configuration when run in fixed vs. random orders.

5.1.1 Memory Benchmarks

The left side of Figure 2 plots the p-values for all 1,198 configurations of memory benchmarks. For this test, the Bonferroni-corrected α_{BC} ($n = 1,198$) is 4.2×10^{-5}. Configurations are sorted on the x-axis according to the effect size (discussed below). As can be seen from the figure, most (1,042, or 87%) of the configurations fall well below the α_{BC} threshold, showing clear evidence of performance effects due to ordering.

To strengthen our analysis, we calculated the *effect size* for each pair of compared samples. This measure is not meant to replace the p-values but rather should complement them [40]. While the statistical tests indicate that the probability of the sampling error causing the observed performance difference may be low, measuring the effect size helps us understand the scale of the difference between the groups.

We calculate the effect size for each statistical test. The larger the effect size, the larger the estimated difference between the populations being compared; a small effect size can indicate that even when there is a statistically significant difference revealed by a p-value, it may be small enough not to be of *practical* importance. To align with the Kruskal-Wallis test, we use the non-parametric formulation of the effect size η^2 that is defined using the H-statistic [4]. In statistics terms, η^2, which yields values between 0 and 1, estimates the fraction of variance in the dependent variable that can explained by the independent variable. The review article [40] provides additional context and includes the formula for η^2 calculation.

η^2 values are plotted in the right side of Figure 2. Past the first few hundred configurations, η^2 becomes larger indicating that the difference between the fixed-order and random-order results becomes larger. This is also the exact region in which $p < \alpha_{BC}$, which indicates significance. It is worth noting that we do not compare η^2 with arbitrary thresholds but rather observe its growth across the range of the tested configurations for comparison purposes; from this standpoint, it is assessed similarly to how we interpret percentage differences in Section 5.1.3.

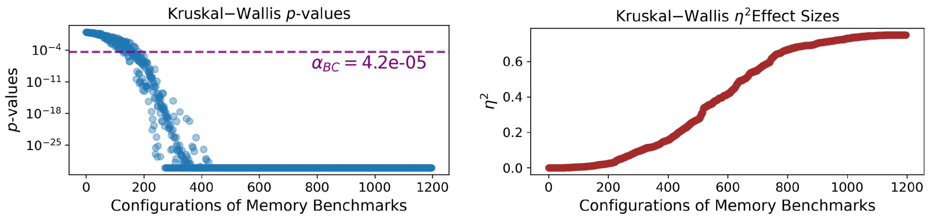

Figure 2: Kruskal-Wallis p-values and effect sizes for memory benchmarks, sorted in order of increasing Kruskal-Wallis η^2 effect size. The horizontal line at the bottom of the left plot comes from rounding small values up to 10^{-30} for display.

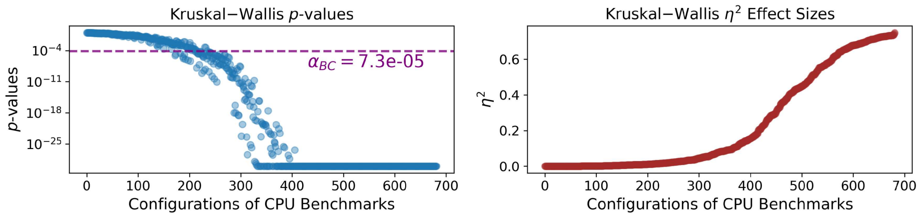

Figure 3: Kruskal-Wallis p-values and η^2 effect sizes for CPU data. The plotting is as described for Figure 2.

5.1.2 CPU Benchmarks

Figure 3 shows the Kruskal-Wallis p-values and effect sizes for our CPU benchmarks. For these comparisons, α_{BC} ($n = 682$) is calculated as 7.3×10^{-5}. As with the memory benchmarks, most configurations fall well below the α_{BC} threshold, indicating that the order in which they are run makes a difference. The most observable distinction between the CPU tests and the memory tests is the shape of the effect size curve: while there are still some configurations that have large effect sizes, there are fewer of them. There are a larger number of configurations that reach statistical significance in the p-value test but have an effect size small enough that it may not have a practical impact. This demonstrates the need to look at both significance tests *and* effect sizes.

Table 2 summarizes the observed Kruskal-Wallis p-values. From this table, we can clearly conclude that the order of experiments matters for the selected microbenchmarks. This effect appears to be more pronounced for memory benchmarks which have a higher ratio of configurations with $p < \alpha_{BC}$.

5.1.3 Relative Difference

Most computer systems studies report their results not in terms of effect sizes but in terms of absolute or relative differences between several alternatives. To align our analysis with our research community, we also looked at the order effects in terms of percentage differences (representing the difference between the mean fixed-order result and mean random-order result, divided by the mean-fixed order result):

$$\Delta_{\%} = \frac{\mu_{fixed} - \mu_{random}}{\mu_{fixed}} \times 100\%$$

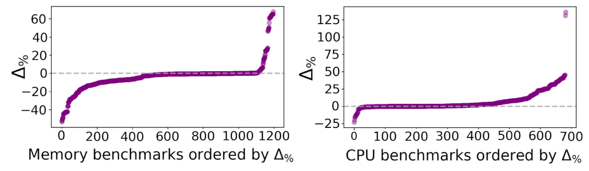

Figure 4: Percent Difference for Memory & CPU benchmarks.

Figure 4 shows the observed $\Delta_{\%}$ values for memory and CPU benchmarks. The configurations are sorted on the x-axis by $\Delta_{\%}$ values, and the y-axis is $\Delta_{\%}$ for a particular configuration. The range of $\Delta_{\%}$ values gives a sense of the magnitude of the studied order-related effects.

Our memory data is measured in throughput, so the higher the value, the better the performance—since we calculate $\Delta_{\%}$ as fixed order performance minus random order performance, a positive $\Delta_{\%}$ indicates that the *fixed* order had better performance than the randomized order. A negative $\Delta_{\%}$ means the *randomized* order performed better. Conversely, our CPU data is measured as execution times, so *lower* values mean better performance: for these, *positive* values mean that the *randomized* experiment design results in better performance.

From these figures, we can see that both memory and CPU benchmarks have some effects that would be considered large enough to affect results. Though they have similar absolute average $\Delta_{\%}$ values (8% for memory, and 7.3% for CPU), the details of their curves are quite different. Both have some configurations that are faster in random order and some that are slower, but the magnitudes and shapes of the curves differ.

Table 2: Configuration classification showing whether there is difference between fixed and random orders or not. The comparisons used Kruskal-Wallis test with Bonferroni-corrected $\alpha_{BC} = 4.17 \times 10^{-5}$ for memory and $\alpha_{BC} = 7.33 \times 10^{-5}$ for CPU comparisons.

Benchmark Type	**Kruskal-Wallis $p < \alpha_{BC}$**	**Kruskal-Wallis $p > \alpha_{BC}$**	**Total**
Memory	1042 (86.97%)	156 (13.02%)	1198
CPU	475 (69.65%)	207 (30.35%)	682

5.2 Comparing Confidence Intervals ❺

In Section 5.1, we showed that experiment order does impact performance using statistical tests and percentage difference. In this section, we look at whether the bias caused by experiment order can result in incorrect conclusions. Answering this question will allow us to establish whether a researcher should consider experiment order while analyzing performance.

Figure 5 shows each of the three cases from the confidence interval overlap test, using examples drawn from our memory benchmarks. The x-axis represents the experiment order. The y-axis is the rank-based 95% confidence interval of the median performance for each order, with the shaded region representing the interval and dashed lines extending the interval limits to the full width of each figure for comparison. The diamond represents the median value. Case 1 indicates that the conclusion *would* change based on order, Case 2 indicates that it is *unlikely* to do so, and Case 3 is *inconclusive*.

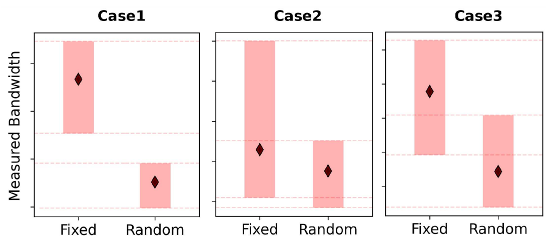

Figure 5: Examples of CI arrangements. The plots are created based on actual measurements for three memory tests; scales are different for them. Red vertical bars are rank-based non-parametric 95% CIs for medians, and ♦—median estimates.

For both memory and CPU benchmarks, we found that most configurations fell into Case 1, meaning that order could change conclusions. This can be seen in Figure 6. The effect is more pronounced for memory benchmarks, where 81% of the configurations fall into Case 1, than for CPU benchmarks, for which only 56% fall into Case 1. Overall, 72% of all configurations are in Case 1. From these results, it becomes amply clear that one can arrive at an incorrect conclusion by merely modifying the experiment order. A performance analysis needs to consider order to ensure accurate results.

6 Automating Experiment Order Testing

We have shown that order can be important in experiment designs. However, it can be difficult for experimenters to rethink their performance experiments to account for this factor. To this end, we have developed OrderSage, a tool that enables experimenters to follow the methodology from Section 3 in their experiments with minimal effort.

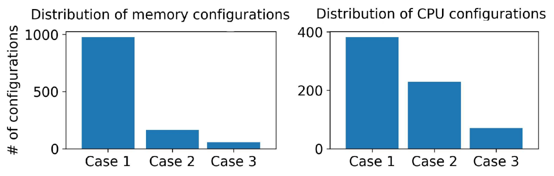

Figure 6: Three CI cases for memory and CPU tests.

6.1 Motivation

The data collection efforts described in Section 4 can be characterized as *long-term*, *extensive*, and *requiring robust infrastructure*. The first refers to the fact that we collected the studied measurements over the period of 9 months. The second indicates that we benchmarked a large pool of hardware types, used numerous tests, and studied many unique permutations of commonly tuned benchmark and system parameters. The third stresses that experimentation of this kind requires reliable computing resources and software for orchestrating test execution, gathering and storing results, etc. We met these requirements using CloudLab hardware, the testbed's programmatic interface built upon `geni-lib` Python package [38], and a set of custom scripts developed for orchestration [39].

This is in contrast to most studies, which describe *short-term* and *focused* analysis efforts, where experimenters thoroughly study subsets from many combinations of tunable parameters and gather the needed results in a limited timeframe. In such settings, the emphasis often is on demonstrating that one algorithm or hardware implementation is better than the alternatives and on characterizing its observed gains. To arrive at such conclusions, experimenters need to make sure that their measurements are not significantly impacted by the order-related effects. In the simplest cases, this means that if they were to run the same sets of tests in different orders, their conclusions would remain valid. We note that in many cases in which order-dependent performance is discovered, this may indicate unexpected behaviors in the system, and is itself an interesting finding.

Aiming to support such focused experimentation, we design OrderSage with the target user in mind who is an experimenter collecting performance data for publication (such

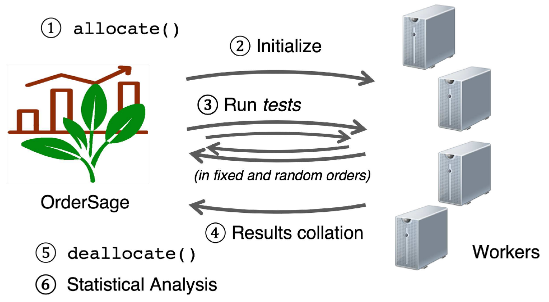

Figure 7: The main operation of OrderSage.

as in a research venue) or for decision-making (such as in a production computing environment.) We assume that such a user can script the execution of each test (e.g., in *shell* or *Python*), and thus OrderSage's primary responsibilities are to execute a number of runs with the scripted tests in different orders, collect and store test results, and analyze the results through the lens of possible order effects. Below, we describe the key components of OrderSage's experiment orchestration.

6.2 Implementation

OrderSage is implemented as a set of Python modules and is available at `https://github.com/ordersage/ordersage`. Its operation can be described using the following terms:

Controller: A machine that facilitates experimentation on a remote node or nodes and performs the statistical analyses. This is the primary place where OrderSage's code runs.

Worker: A machine that executes an experiment that consists of several fixed and random runs. Workers are accessed (through `ssh`) and controlled by a single controller node. One controller can make use of one or more workers.

Results: Each trial produces one performance result in the form of a floating-point number. Multiple performance metrics should be treated as separate results. OrderSage collects these results from the worker(s) and stores them on the controller for analysis.

Figure 7 shows a high-level system diagram that includes the key processes being orchestrated by OrderSage:

① **allocate():** Support is provided for use of worker nodes that are either pre-allocated by the user or reserved on CloudLab testbed. Outside of CloudLab, any node—local or remote—into which the user can `ssh`, run the tests (including installing software, if necessary), and execute the reset procedure can be used. Analogous `allocate()` routines can be implemented for commercial clouds using the APIs they provide.

② **Initialize:** Worker nodes are readied via an initialization script provided by the user. This typically includes installing and configuring the software under evaluation. During initialization, a baseline order is selected (Step ❶ from the method described in Section 3) by running a user-provided script that produces the commands for each test. OrderSage defaults to rebooting a worker node as the "clean state" reset procedure (Step ❷ from the method). However, this behavior can easily be redefined by the user.

③ **Run Tests:** After initialization, tests are run according to Step ❸ of the method. Each run consists of an execution of all trials in some order. Half of the runs are performed in the baseline order, and half in an order that is randomly shuffled (separately for each run); these orders are interleaved. The reset procedure is executed in between each run.

④ **Collect Results:** Performance measurements are collected as outlined in Step ❸ of our method and saved as raw data from each trial. Additionally, metadata such as random orderings, machine environment information, execution times, and `stdout` outputs are saved for each run.

⑤ **deallocate():** If the worker(s) were `allocated()`ed in ①, (such as with our CloudLab integration), OrderSage will handle the deallocation of nodes at the end of the experiment.

⑥ **Statistical Analysis:** Results are analyzed according to Steps ❹ and ❺ from the method described in Section 3. The results of all statistical analyses are saved on the controller node. If multiple worker nodes were used, the necessary result aggregation will take place. A final comparative step will provide a high-level overview of the combined vs. individual experiment analyses.

Running OrderSage is detailed in Appendix A.

7 Case Studies

We used OrderSage to facilitate the methodology proposed in Section 3 for three case studies. Our goals were to demonstrate its use on common benchmarks and application test suites and investigate the impact of test order on the results of these cases. All case studies were executed on CloudLab servers of the `xl170` hardware type [37]. For these experiments, which occurred over 24-48 hours, each experiment was run on a single worker. All data from these experiments are included in the released artifact: `https://github.com/ordersage/paper-artifact`.

7.1 `memcached` Benchmark Suite

This case study uses `memcached`'s own benchmark suite, which is designed to mimic the process of reporting perfor-

Table 3: Test results for the memcached experiment. We use Bonferroni correction with $n = 3$ and $\alpha_{BC} = 0.0167$ (providing a family-wise error rate of 0.05). The Kruskal-Wallis p-values are shown, as are their interpretation relative to α_{BC}: the column contains • if the null hypothesis can be dismissed or ○ if it cannot. $\Delta_{\%}$ is calculated as in Section 5.1.3 and the CI cases are as defined in Section 3.

Test	KW p-value	KW test	$\Delta_{\%}$	CI case
cmd_set	0.49	○	0.3	2
cmd_get	0.74	○	−0.2	2
get_hits	0.00009	•	5.3	3

mance numbers for this application. memcached [2] is an efficient and widely used in-memory key-value store, and its associated mc-crusher [24] benchmark suite includes a variety of scripts designed to exercise a server instance and measure its performance.

The mc-crusher documentation specifies "You should start a fresh memcached", and includes a series of three tests (cmd_set, cmd_get, and get_hits) in its included sample configuration file, executed serially in that order; accordingly, we start memcached after the reboot in our standard Step ❷ reset procedure, and perform those same three tests in each of our runs. We follow the same ordering of trials in the fixed case, with a single instance of memcached for all trials (following the mc-crusher distribution exactly). We increase the sample duration to 60 seconds per test (to reduce the influence of noise on each sample) and permute the order of the trials in our random runs to check our hypothesis that ordering affects the observed results, but otherwise do not modify the sample mc-crusher parameters. From inspection of the mc-crusher source, we expect the three benchmarks to operate on generally disjoint data, and therefore do not anticipate any direct connection between the execution of one and the output of the next. However, it is difficult to predict the presence or magnitude of *indirect* ordering artifacts, where the side effects of previous computation might influence the efficiency of subsequent operations, which is what our analysis aims to measure.

Table 3 presents the results we obtained by running OrderSage with memcached version 1.5.22, with 50 fixed and 50 random runs, each including the three tests described.

Overall, we conclude that the order of trials within a run *does* affect the measurements obtained for the mc-crusher environment under test, at the 95% significance level. This coincides with the get_hits's median performance changing by over 5% based on whether a fixed-order or random-order experiment design is used.

7.2 NPBench & NPB

NPBench is a "a set of NumPy code samples representing a large variety of HPC applications" [43]. The authors use it to test a variety of Python HPC frameworks and compilers that aim to accelerate NumPy code; they also expect the results to be useful to end-users of such frameworks. NAS Parallel Benchmarks (NPB) is an open source benchmarking suite which includes "a small set of programs designed to help evaluate the performance of parallel supercomputers." [27] We select two tests from NPBench that exercise operations used in data analytics and machine learning: sparse matrix-vector multiplication (SPMV) and the normalized exponential function (softmax) used in neural networks. In addition, we select integer sort (IS) from NPB, which is used to benchmark random access memory. SPMV and softmax are generally CPU-bound, while IS generally has its performance limited by memory speed. Using OrderSage, we did 100 runs in each of fixed and random orders. We set problem sizes to large enough values to get meaningful results on CloudLab machines: flag L in NBbench is expected to take about 1000ms to run whereas class D in NPB is the largest test problem for IS, and the median runtime was 36 seconds.

Table 4: Test results for the NPBench & NPB experiment. Columns are as described for Table 3.

Test	KW p-value	KW test	$\Delta_{\%}$	CI case
IS	0.83	○	0.00	2
SPMV	0.69	○	−0.60	2
softmax	0.03	•	0.46	3

The results from these experiments are in Table 4. IS and SPMV show no order-dependence. While softmax does show a statistically-significant change in distribution when run in a random order, the effect size of 0.46% is small enough that it is unlikely to make a difference in practice: these three tests can be safely run in any order. This demonstrates the need to look at effect sizes as well as statistical significance: a positive result from the Kruskal-Wallis test does not, by itself, guarantee that the effect is large enough to matter.

7.3 uFS Paper Artifact Reproduction

Our final case study looks at the uFS filesystem presented at SOSP 2021 [19]. This paper submitted an artifact and was awarded the Available, Functional, and Reproduced badges; it was part of our survey in Section 2. uFS is a user-level filesystem "semi-microkernel" [19] that claims good base performance and better scalability than the ext4 filesystem in the Linux kernel. This is demonstrated with benchmarks at various scales and under various threading conditions. Using OrderSage, we find that some experiments run for this paper *are* order-dependent with large effects (up to 17%), though not large enough to change the conclusions of the paper.

The evaluation scripts supplied with the artifact run multiple benchmarks, of which we selected the Microbenchmarks with single-threaded uFS and ext4 (both without journaling.) Their scripts run all 32 workloads in sequence; we modified

Table 5: Test results for the uFS experiment. In the original paper, `ufs` results are compared with corresponding `ext4nj` experiments. Columns are as described for Table 3.

Test	KW p-value	KW test	$\Delta_{\%}$	CI case
`ufs.ADSS`	0.028	●	16.8%	2
`ext4nj.ADSS`	0.364	○	-4.0%	2
`ufs.ADPS`	0.013	●	6.7%	2
`ext4nj.ADPS`	0.406	○	4.7%	2
`ufs.RDSR`	0.112	○	0.2%	2
`ext4nj.RDSR`	0.406	○	-0.8%	2
`ufs.RMS`	0.940	○	0.8%	2
`ext4nj.RMS`	0.650	○	0.1%	2
`ufs.LsMS`	0.650	○	-0.6%	2
`ext4nj.LsMS`	0.940	○	0.0%	2
`ufs.RMP`	0.545	○	0.0%	2
`ext4nj.RMP`	0.545	○	0.3%	2
`ufs.CMP`	0.496	○	-0.2%	2
`ext4nj.CMP`	0.256	○	-0.1%	2
`ufs.LsMP`	0.151	○	3.6%	2
`ext4nj.LsMP`	0.406	○	0.1%	2
`ufs.CMS`	0.019	●	-1.3%	2
`ext4nj.CMS`	0.705	○	0.3%	2
`ufs.RDPR`	0.112	○	0.2%	2
`ext4nj.RDPR`	0.226	○	1.9%	2

them to run one workload at a time as individual tests. We use the leftmost data point for evaluation as described in the paper's Section 4.2 and Figure 5a—these are used to evaluate the claim that uFS performs as well as or better than ext4 under baseline, single-threaded conditions. We used OrderSage and a `c6525-100g` node in CloudLab (which has a dedicated NVMe drive as does the original authors' machine) to run these tests in fixed and random orders (10 times each).

Our results (Table 5) show that order does not matter to most tests, but it does matter to three: `ufs.ADSS`, `ext4nj.ADPS`, and `ufs.CMS`, with the `ufs.ADSS` test changing the most: in the fixed order, its median is 119K with a tight CI of $[117K, 120K]$. In random order, its median drops by 16.8% with a much wider CI of $[74K, 121K]$. The conclusion from the uFS paper still holds: the random-order `ufs.ADSS` median of $98K$ is still greater than the $41K$ random-order result for baseline system it is compared to, `ext4nj.ADSS`, and the CIs do not overlap. This effect may be due to hardware differences: the original uFS paper was evaluated on an NMVe drive using Intel Optane memory, while the drive we used on CloudLab has traditional flash memory. As a result, latencies and flush strategies differ between the environments. However, this demonstrates the necessity of avoiding the ordering trap, as such order-dependent results are probably "hidden" in many published results, and likely indicate system effects that the authors may not be fully aware of.

8 Related Work

There is much scientific literature focused on experimental design and analysis of computer systems performance experiments [12, 16, 18, 32, 34]. Among recent work in related areas are studies of presentation flaws specific to performance results [11] and analysis of performance variability in computer systems [22]. In a separate but relevant context, some research and development efforts are focused on *testbeds*, i.e. computer infrastructure, designed for reproducible experiments [28, 42], and how they can facilitate trustworthy experimental evaluations. Studies of computer benchmarking [10, 15] consider both the nuances of benchmark design and interpretation of results. However, the aforementioned sources do not help conclusively answer the question: "Does the order of tests matter, and if so, how much?" Our study aims to bridge this gap.

One recent study related to our work focuses on repeatable experiments in highly variable cloud environments [1]. The authors study the following designs: 1) Single Trial, 2) Multiple Consecutive Trials, 3) Multiple Interleaved Trials (MIT), and 4) Randomized Multiple Interleaved Trials (RMIT). Another study of the RMIT execution plan led to the development of WPBench, a web serving benchmark suite that bundles a set of micro and application benchmarks [35]. The fixed and random orders we study correspond to MIT and RMIT, respectively. While those studies argue for using RMIT, our investigation extends previous work with a large-scale evaluation of both approaches and shows where the differences between the two are most significant. We also consider environments without "background noise" from other tenants.

The idea of turning a proposed methodology into a reusable tool was inspired by the recent work on Lancet, a self-correcting tool for latency analysis [14]. TraceSplitter [33] applies an analogous statistical approach to traffic traces. Similarly, Hyperfine [29] facilitates many tasks involved in the benchmarking process and common subsequent analyses. In turn, experimenters can focus more on creating interesting experiments with increased confidence that their conclusions are unbiased by factors such as test ordering. We implement OrderSage with this vision in mind and describe in this paper the results it collects in several use cases.

Another related study considers performance changepoints [6]. The data collection in our work is similar to the process described in that paper. However, rather than characterizing temporal patterns broadly, we focus on the order-related effects and the methodology for studying them.

9 Conclusion and Future Work

The order in which tests are run is a significant, but often neglected, part of experiment design—as shown in our survey, it is rarely mentioned in papers, and the artifacts that support them show little sign of being designed with ordering in mind. Our findings show that order can indeed make a difference: sometimes quite a large one. Systems experimenters should take this into account in their experiment designs, and test for order dependence when feasible. The response to discovered order-dependence will vary depending on the system, the experiment, and its goals. In some cases, there may be aspects of the system under test, test environment, or test procedure that need to be "fixed" to make runs more consistent and less dependent on order. In other cases, some amount of variability is simply to be expected, and experiments should be run in several randomized orders to avoid systematically biasing results with a single order. Finally, in some cases, it may be that a "clean" environment is not the most realistic one in which to run the experiment, and more effort needs to be taken to get the environment into a suitably realistic state.

Our work thus far has left out of scope a deep analysis of *why* order matters. This would be an interesting subject for follow-up research, and we expect that the reasons will be as varied as the tests that are run and the environments they are run in. One way to do such an investigation would be to analyze which tests *cause* changes in the following trials, and which ones see the largest *effects*. We hope that our open dataset and tool will help to enable such explorations.

References

[1] Ali Abedi and Tim Brecht. Conducting repeatable experiments in highly variable cloud computing environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 287–292, 2017.

[2] Damiano Carra and Pietro Michiardi. Memory partitioning in memcached: An experimental performance analysis. In *Proceedings of the IEEE International Conference on Communications (ICC)*, June 2014.

[3] A. Cohen, F. Finkelstein, A. Mendelson, R. Ronen, and D. Rudoy. On estimating optimal performance of CPU dynamic thermal management. *IEEE Computer Architecture Letters*, 2(1), 2003.

[4] Barry H. Cohen. *Explaining Psychological Statistics*. John Wiley & Sons, 2008.

[5] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, July 2019.

[6] Dmitry Duplyakin, Alexandru Uta, Aleksander Maricq, and Robert Ricci. In datacenter performance, the only constant is change. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 370–379. IEEE, 2020.

[7] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2021.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[9] Yosef Hochberg and Ajit C Tamhane. *Multiple Comparison Procedures*. Wiley, 1987.

[10] Roger W Hockney. *The science of Computer Benchmarking*. SIAM, 1996.

[11] Torsten Hoefler and Roberto Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, page 73. ACM, 2015.

[12] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley- Interscience, April 1991.

[13] Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 2008.

[14] Marios Kogias, Stephen Mallon, and Edouard Bugnion. Lancet: A self-correcting latency measuring tool. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 881–896, 2019.

[15] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Systems Benchmarking*. Springer, 2020.

[16] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowskiu. *Systems Benchmarking For Scientists and Engineers*. Springer Nature Switzerland AG, Cham, Switzerland, 2020.

[17] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, 2011.

[18] Dennis KJ Lin, Timothy W Simpson, and Wei Chen. Sampling strategies for computer experiments: design and analysis. *International Journal of Reliability and applications*, 2(3):209–240, 2001.

[19] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the*

ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, 2016.

[21] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: bringing legacy code to byte-addressable persistent memory. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, July 2017.

[22] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[23] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[24] Memcached community. mc-crusher. https://github.com/memcached/mc-crusher, 2011–2020.

[25] R. G. Miller. *Simultaneous Statistical Inference (2nd Ed.)*. Springer Verlag, New York, 1981.

[26] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 265–276, New York, NY, USA, 2009. ACM.

[27] National Aeronautics and Space Administration, Advanced Supercomputing Division. Nasa parallel benchmarks (npb). https://www.nas.nasa.gov/software/npb.html, 2022.

[28] Lucas Nussbaum. Testbeds support for reproducible research. In *Proceedings of the Reproducibility Workshop*, Reproducibility '17, pages 24–26, New York, NY, USA, 2017. ACM.

[29] David Peter. Hyperfine: a command-line benchmarking tool. https://github.com/sharkdp/hyperfine, 2023.

[30] Alex W. Reece. Achieving maximum memory bandwidth. http://codearcana.com/posts/2013/05/18/achieving-maximum-memory-bandwidth.html, May 18 2013.

[31] Alex W. Reece. Memory bandwidth demo. https://github.com/awreece/memory-bandwidth-demo, May 19 2013.

[32] Jerome Sacks, William J Welch, Toby J Mitchell, and Henry P Wynn. Design and analysis of computer experiments. *Statistical science*, 4(4):409–423, 1989.

[33] Sultan Mahmud Sajal, Rubaba Hasan, Timothy Zhu, Bhuvan Urgaonkar, and Siddhartha Sen. Tracesplitter: A new paradigm for downscaling traces. In *16th European Conference on Computer Systems (EuroSys)*, April 2021.

[34] Thomas J Santner, Brian J Williams, William I Notz, and Brain J Williams. *The Design and Analysis of Computer Experiments*, volume 1. Springer, 2003.

[35] Joel Scheuner and Philipp Leitner. A cloud benchmark suite combining micro and applications benchmarks. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 161–166, 2018.

[36] David J Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. CRC Press, Boca Raton, Florida, 2000.

[37] The CloudLab Team. CloudLab hardware. https://www.cloudlab.us/hardware.php, 2018.

[38] The CloudLab Team. Describing a profile with python and geni-lib. http://docs.cloudlab.us/geni-lib.html, August 2018.

[39] The CloudLab Team. Collection script for cloudlab benchmarks. https://gitlab.flux.utah.edu/emulab/cloudlab-orchestration, 2021. Accessed: 2021-10-17.

[40] Maciej Tomczak and Ewa Tomczak. The need to report effect size estimates revisited. an overview of some recommended measures of effect size. *Trends in Sports Sciences*, 1(21):19–25, 2014.

[41] E. Weigle and Wu chun Feng. A comparison of TCP automatic tuning techniques for distributed computing. In *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*, 2002.

[42] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In

Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI). USENIX, December 2002.

[43] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. Npbench: A benchmarking suite for high-performance numpy. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, New York, NY, USA, 2021. Association for Computing Machinery.

Appendix A Using OrderSage

Using OrderSage is straightforward, and requires little beyond that used in a typical experiment.

- **Experiment Environment:** Users must have a controller node and at least one worker node that has remote-access capabilities. The controller is separate from the worker so that the latter can be rebooted as part of the reset procedure.
- **Experiment Repository:** The tests to run and their associated scripts are stored in a `git` repository created by the user, which makes them natively version-controlled. In addition to the system(s) under evaluation, the repository contains the following:
  - **Test Configuration Script:** Called during the initialization phase by the controller, this script prints a list of commands to `stdout`. The commands will be executed in-order for the fixed runs and shuffled for the random runs. Each command represents a single test and all commands must be unique. It is up to users to implement this script as they wish as long as these requirements are met. In the simplest case, it can be a series of print statements of varying test commands or it can be more complex and include methods to iterate through complex sets of parameters, producing a command for each one.
  - **Initialization Script:** The controller calls an initialization script to ready all workers for experimentation as defined by the user. This script can install packages, set machine states, etc. Its only requirement is that it creates a "results" directory in a location on the worker.
- **Configuration File:** To run OrderSage, the user creates a configuration file. This configuration contains the URL of the experiment repository, the location of the configuration and initialization scripts within the repository, and other parameters. These parameters include paths to results files, the number of runs, etc. If the set of worker node(s) is pre-allocated, the `workers` parameter of this file must contain a list of all worker node hostnames.
- **Define Reset Protocol:** Our default implementation of OrderSage calls `reset()`, which is implemented to reboot the worker node(s) and reconnect between runs. However, if users prefer a different reset procedure, they can override this method.
- **Results:** Results are collected in a single text file on each worker node. Each test run (trial) must provide a single, floating-point number on a new line of the file. It is important that this results file is presented in-order (i.e., the first trial produces the first number and the n^{th} trial produces the n^{th} number). In total, the number of lines in the result file must equal the *number of tests* $\times$ *the number of runs* $\times$ 2 (for fixed and random runs).

Once the aforementioned configuration is complete, a user can run OrderSage by executing the following command:

```
# python controller.py
```

The artifact with the code and data we released, https://github.com/ordersage/paper-artifact, has more information on running OrderSage and reproducing the results presented in this paper.