Pilot: A Framework that Understands How to Do Performance Benchmarks the Right Way

Yan Li, Yash Gupta, Ethan L. Miller, and Darrell D. E. Long
Storage Systems Research Center,
University of California, Santa Cruz, CA 95064, USA
Email:{yanli,ygupta,elm,darrell}@cs.ucsc.edu

***Abstract*—Carrying out even the simplest performance benchmark requires considerable knowledge of statistics and computer systems, and painstakingly following many error-prone steps, which are distinct skill sets yet essential for getting statistically valid results. As a result, many performance measurements in peer-reviewed publications are flawed. Among many problems, they fall short in one or more of the following requirements: accuracy, precision, comparability, repeatability, and control of overhead. This is a serious problem because poor performance measurements misguide system design and optimization. We propose a collection of algorithms and heuristics to automate these steps. They cover the collection, storing, analysis, and comparison of performance measurements. We also implement these methods as a readily-usable open source software framework called Pilot, which can help to reduce human error and shorten benchmark time. Evaluation of Pilot with various benchmarks show that it can reduce the cost and complexity of running benchmarks, and can produce better measurement results.**

***Index Terms*—computer performance; performance analysis; software performance; performance evaluation; system performance; measurement techniques; heuristic algorithms**

I. Introduction

Performance evaluation is a core task of computer systems research. In fact, everyone needs to do performance measurement from time to time: home users need to know their Internet speed, an engineer may need to figure out the bottleneck of a system, a researcher may need to calculate the performance improvement of a new algorithm. Yet not everyone has received rigorous training in statistics and computer performance evaluation. That is one of the reasons why we can find many incomplete or irreplicable benchmark results, even in peer-reviewed scientific publications. A widely reported study published in *Science* [11] found that 60% of the psychology experiments could not be replicated. Computer science cannot afford to be complacent. Hoefle and Belli analyzed 95 papers from HPC-related conferences and discovered that most papers are flawed in experimental design or analyses [7].

Basically, if any published number is derived from fewer than 20 samples, or is presented as a mean without variance or confidence interval (CI), the authors are likely doing it wrong. The following problems can often be found in published results:

- *Imprecise*: the result may not be a good approximation of the "real" value; often caused by failing to consider the width of CI and not collecting enough samples,
- *Inaccurate*: the result may not reflect what you need to measure; often caused by hidden bottleneck in the system,
- *Ignoring the overhead*: not measuring or documenting the measurement and instrumentation overhead,
- *Presenting improvements or comparisons without providing the p-value*, making it impossible to know how reliable the improvements are.

Time is another limiting factor. People usually do not allocate a lot of time to performance evaluation or tuning, yet few newly designed or deployed systems can meet the expected performance without a length tuning process. A large part of the tuning process is spent on running customer's benchmark workloads for model construction or testing candidate parameters. Designers are usually given only a few days for these tasks and have to rush the results by cutting corners, which often leads to unreliable benchmark results and sub-optimal system tuning decisions.

We want to improve this process by getting statistical valid results in a short time. We begin with a high level overview of what analytical methods are necessary to generate results that meet the statistical requirements, then design heuristic methods to automate and accelerate them. We cover methods to measure and correct the autocorrelation of samples, to use t-distribution to estimate the optimal test duration from existing samples, to detect the duration of the warm-up and cool-down phases, to detect shifts of mean in samples, and to use t-test to estimate the optimal test duration for comparing benchmark results. In addition to these heuristics, we also propose two new algorithms: a simple and easy-to-use linear performance model that makes allowance for warm-up and cool-down phases using only total work amount and workload duration, and a method to detect and measure the system's performance bottleneck while keeping the overhead within an acceptable range.

To encourage people to use these methods, we implement them in a software framework called *Pilot*. It can automate many performance evaluation tasks based on real-time time series analytical results, and can help people who have insufficient statistics knowledge to get good performance results. Pilot can also free experienced researchers from painstakingly executing benchmarks so they can get scientifically correct results using the shortest possible time. Pilot is a light-weight framework written in C++. We provide many interfaces for integrating Pilot into existing workload software. We release Pilot under the 3-clause BSD license and provide tools to foster a user community. We hope many researchers and engineers can find Pilot useful in their daily work.

II. Background

A. Performance measurement and benchmarking

Performance measurement is concerned with how to measure the performance of a running program or system, while *benchmarking* is more about issuing a certain workload on the system in order to measure the performance. High quality performance measurement and benchmarking are very important for almost everyone who uses an electronic or computer system, from researchers to consumers. For instance, performance evaluation is critical for computer science researchers to understand the performance of newly designed algorithms. System designers run benchmarks and measure application performance before design decisions can be made. System administrators need performance data to find the most suitable products to procure, to detect hardware and software configuration problems, and to verify if the performance meets the requirements of applications or Service-Level Agreement.

Performance measurement and benchmarking are similar but not identical. We usually can control how to run benchmarks, but measurement often needs to be done on applications or systems over which we have little control. Our following discussion applies to both benchmarking and measurement in general, and we use these two terms interchangeably unless otherwise stated.

Benchmarks must meet several requirements to be useful. The measurement results must reflect the performance property one plans to measure (accuracy). The measurement must be a good approximation of the real value with quantified error (precision). Multiple runs of the same benchmark under the same condition should generate reasonably similar results (repeatability). The overhead must be measured and documented. If the results are meant for publication, they also must include enough hardware and software information so that other people can compare the results from a similar environment (comparability) or replicate the result (replicability).

People usually use these terms, especially accuracy and precision, to mean many different things. Here we give our definition of these terms. It is assumed that the readers have basic statistics knowledge and understand basic concepts such as mean, variance, and sample size. Boudec wrote a good reference book for readers who want to know more about these statistical concepts [1].

a) Accuracy: reflects whether the results actually measure what the user wants to measure. A benchmark usually involves many components of the system. When we need to measure a certain property of the system, such as I/O bandwidth, the benchmark needs to be designed in such a way that no other components, such as CPU or RAM, are limiting the measured performance. This requires carefully designing the experiment [5], measuring the usage of related components while the benchmark is running, and checking which component is limiting the overall performance.

b) Precision: is related to accuracy but is a different concept. Precision is the difference between the measured value and the real value the user needs to measure. In statistical terms, precision is the difference between a sample parameter and its corresponding population parameter. Precision can be described by confidence interval (CI). The CI of a sample parameter describes the range of possible population parameter at certain likelihood. For instance, if the CI of a throughput mean (μ) is C at the 95% confidence level, we know that there is a 95% chance that the real system's throughput is within interval $[\mu - \frac{C}{2}, \mu + \frac{C}{2}]$. In practice, CIs are typically stated at the 90% or 95% confidence level. We can see that the common practice of presenting a certain performance parameter using only one number, such as saying the write performance of a disk drive is 100 MB/s, is misleading.

c) Repeatability: is critical to a valid performance measurement because the goal of most performance benchmark is to predict the performance of future workloads, which exactly means that we want the measurement results to be repeatable. In addition to the accuracy and precision, errors in the measurement can have a negative impact on repeatability. There are two kinds of errors: the systematic error and random error. Systematic error means that the user is not measuring what he or she plans to measure. For instance, the method of benchmarking is wrong or the system has some hidden bottleneck that prevents the property to be measured from reaching its maximum value. In these cases, even though the results may look correct, it may well be not repeatable in another environment. Random errors are affected by "noise" outside our control, and can result in non-repeatable measurements if the sample size is not large enough or samples are not independent and identically distributed (*i.i.d.*).

We will discuss how to achieve these properties in the following sections, and one can see that scientifically performing a benchmark demands significant knowledge of the computer system and statistics. We cannot expect all administrators, engineers, and consumers to have received rigorous training in both computer science and statistics. It is not news that many vendors publish misleading, if not utterly wrong, benchmark results to promote their products. Many peer-reviewed research publications also suffer from poor understanding or execution of performance measurement [7].

B. Getting results fast

The old wisdom for running benchmark is to run it for as long as one can tolerate and hope the law of large numbers can win over all errors. This method is no longer suitable for today's fast changing world, where we simply no longer have a lot of time to run benchmark. We have heard field support engineers complaining[1] that they are usually only given one or two days after the installation of a new computer cluster or distributed storage system to prove that the system can deliver whatever performance promised by the salesperson, very often using the customer's own benchmark programs. Modern distributed systems can have hundreds, if not thousands, of parameters to tune, and the performance engineer needs to run an unfamiliar benchmark repeatedly and try different parameters. Apparently the shorter the benchmark is the more parameters can be tested, thus resulting in better system tuning results.

[1] Private conversation with an engineer from a major HPC provider.

Existing analytical software packages, such as R [12], are either too big or slow for run-time analysis, are hard to integrate with applications (for instance, R is a large GPL-licensed software package), or require the user to write complex scripts to use all its functions.

In all, we realize that we need an easy-to-use software tool that can guide the user and help to automate most of the analytical tasks of computer performance evaluation. We are not introducing new statistical method in this paper. Instead we focus on two tasks: finding the most suitable and practical methods for computer performance evaluation, and design heuristics methods to automate and accelerate them.

III. Algorithms

We define the following terms. Also see a sample in Fig. 1.

Performance index (PI) is one property we want to measure. For instance, the throughput of a storage device is a PI, and its latency is another PI.

Session is the context for doing one measurement. We can measure multiple PIs in one session. One session can include multiple **rounds** of benchmarks, and each round can have a different length.

Work amount is the amount of work involved in one round of benchmark. The work amount is related to the length of the workload. For instance, in a sequential write I/O workload round, we write 500 MB data using 1 MB writes, the work amount of this round is 500.

Work unit is a smallest unit of work amount that we can get a measurement from. Using the above sample, if the I/O size is 1 MB, we can measure the time of each I/O syscall and calculate the throughput of each I/O operation. Here the work unit is 1 MB, and we have 500 work units in that workload round. Not all workloads should be divided into units. Pilot expects the work unit to be reasonably homogeneous. So, for instance, reading one 1 MB from different locations of a device can be thought as homogeneous because the difference in performance is small and mostly normally distributed. But shifting from sequential I/O to random is not homogeneous because that would result in huge difference in I/O performance. In general, the user should only divide the workload into units when one *expects* them to have similar performance. If not the user should not use the work unit-related analytical methods of Pilot and should stick with readings (see next term) only. We leave heterogeneous work units as a future work.

Reading is a measurement of a PI of a round. Each benchmark round generates one reading for each PI at the end of the round. In the sample above, when PI is the throughput of the device, we can get one throughput reading for each round.

Unit reading (UR) is a measurement of a PI of a work unit. In the sample above, we would have 500 throughput Unit readings for Round 1 because it contains 500 work units, and these 500 unit readings would be the throughput of each 1 MB I/O operation.

Work-per-second (WPS) is the calculated speed at which the workload consumes work. This is usually the desired PI for many simple workloads.

Some workload cannot be meaningfully separated into homogeneous work units, such as booting up a system and randomly reading files of different sizes. We get only readings from this kind of workloads.

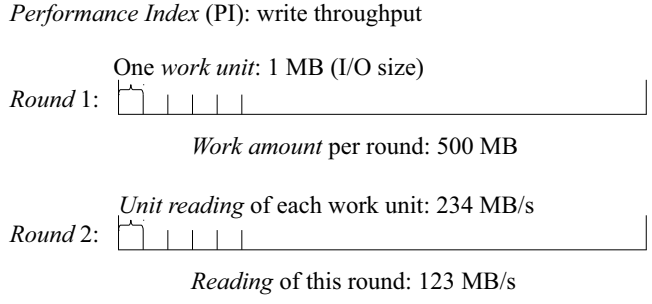

Fig. 1. A sample write workload to illustrate the terms. This workload consists of two rounds. Each round has a work amount of 500 MB.

A. Warm-up and Cool-down Phase Detection

Performance results are often used to predict the run time of future workloads, and it is a common practice to use one number to express the performance of a PI. For example, people usually say "the write throughput of this device is X". Using only one number assumes that the device's performance follows a linear model. Linear models (*work amount* = *duration* × *speed*) are simple, but using only one number can only state the device's stable performance and is not adequate when the performance of the PI can be significantly affected by a long warm-up or cool-down phase.

Most computer devices require a setup or warm-up phase before its performance can reach a stable level, like shown in Fig. 2. If not properly accounted for, these warm-up phases can have a negative impact on the precision of the measurement. A common practice is to run the workload for a long time and hope the effect of the warm-up phase can be amortized. However, when the duration of the warm-up phase is not known, there is no way to know the actual impact on the precision (see the samples in § V). We describe two methods to address different kinds of workloads.

We consider the following phases of a workload:

1) The setup phase, including the steps that do not consume work amount, such as allocating memory, initializing variables, and opening files, etc.
2) The warm-up phase when the system starts to perform work but has not yet reached stable performance;
3) The stable phase where the work amount is being consumed at a stable rate;
4) The cool-down phase when the system's performance starts to drop before finishing all the work (this is usually observed in multi-threaded workloads when some but not all threads finish the allocated work and the number of active threads starts to drop at the end of the workload).

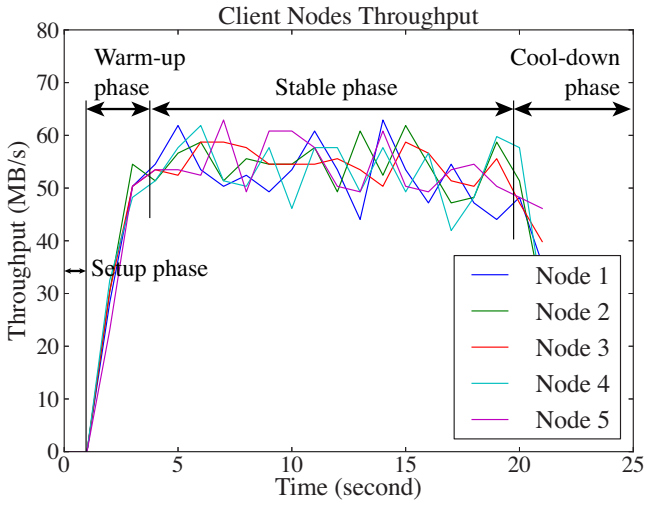

Fig. 2. Throughput of a multi-node random read write workload. It shows the setup phase, the warm-up phase caused by caching effect, and the cool-down phase caused by shutting down of I/O threads.

We call them collectively the non-stable phases. When the workload has multiple rounds, each round may or may not have its own non-stable phases, and when they have, the duration can be different. We consider two cases, the first is when the benchmark can provide unit readings, the second is for workloads that cannot provide unit readings.

a) Workloads that can provide unit reading: If the benchmark workload can provide unit readings, which is the measurement of each work unit, we can calculate the shift in UR mean and use these change-points to separate the URs into phases. Multiple change-point detection is a challenging research question, especially when we cannot make any assumption about the distribution of the error or the underlying process. The method we use also has to be fast to calculate and should support online update.

After evaluating many change-point detection methods, we found that the E-Divisive with Medians (EDM) [10], which is a new method published by Matteson and James in 2014, best fits our requirements. EDM is non-parametric (works on mean and variance) and robust (performs well for data drawn from a wide range of distributions, especially non-normal distributions). EDM's initial calculation is $O(n \log n)$ and can do update in $O(\log n)$ time.

EDM outputs a list of all the change-points in the time series. It is common to see many change-points at the start and end of the workload. These change-points divide the test data into multiple segments. Pilot uses a heuristic method to determine which segment is the stable segment: it has to be the longest segment and also dominate the test data (containing more than 50% of the samples). This method can effectively remove any number of non-stable phases at the beginning and the end.

b) Workloads that cannot provide unit reading: Some workload cannot be meaningfully separated into units. In these cases, we designed the following Work-per-second (WPS) Linear Regression Method to detect and remove the non-stable phases from the results of these workloads. A linear regression model works best when:

1) The work amount of the workload is adjustable,
2) There is a linear relationship between the work amount and the duration of the workload,
3) The duration of the setup, warm-up, and cool-down phases are relatively stable across rounds.

It is not necessary to check these conditions beforehand. We will know that one or more of them are false if the result of the WPS method has a very wide CI or a high prediction error. The WPS method also applies autocorrelation detection and subsession analysis, which make it more tolerant of the inconsistency in measurements.

Let w be the work amount, t be the total duration of the workload, t_{su} be the duration of the setup phase, t_{wu} be the duration of the warm-up phase, t_s be the duration of the stable phase, t_{cd} be the duration of the cool-down phase, w_{wu} be the work amount consumed by the warm-up phase, w_s be the work amount consumed by the stable phase, and w_{cd} be the work amount consumed by the cool-down phase. We have (note that the setup phase of a workload does not consume work amount)

$$t = t_{su} + t_{wu} + t_s + t_{cd} \tag{1}$$

$$w = w_{wu} + w_s + w_{cd} \tag{2}$$

v_s is the stable system performance we need to measure. By definition, it can be calculated from the work amount of the stable phase divided by the duration of the stable phase:

$$v_s = \frac{w_s}{t_s} \tag{3}$$

Combining equation (1), (2), and (3), we can have

$$\begin{aligned} t &= t_{su} + t_{wu} + \frac{w - w_{wu} - w_{cd}}{v_s} + t_{cd} \\ &= \left(t_{su} + t_{wu} + t_{cd} - \frac{w_{wu} + w_{cd}}{v_s}\right) + \frac{1}{v_s} w \\ t &= \alpha + \frac{1}{v_s} w \end{aligned} \tag{4}$$

Equation (4) is the model we use in Pilot. Given enough number of (w, t) pairs, we can use regression to estimate the value of α and v_s. The current implementation of Pilot uses the Ordinary Least Square estimator [6] for its simplicity, and other estimators can be added when necessary. We need the samples to be i.i.d. in order to calculate the CI of v_s using the t-distribution. We use subsession analysis, which calculates the autocorrelation coefficient of input samples and merges adjacent correlated samples to create fewer but less correlated samples, before running the regression estimator (see § III-D).

In addition to the requirements we talked about earlier, linear regression requires that the following conditions be met:

1) The differences between the work amounts of rounds are sufficiently large,
2) The sample size is sufficiently large.

We designed Pilot to keep running the workload at various length and for many rounds until the desired width of the CI is reached. Because we cannot know the total number of rounds

that are needed at the beginning, we designed the following algorithm to generate different work amount for each round: let (a, b) be the valid range for the work amount, we pick the midpoint of the interval as the work amount for the first round $(a + \frac{b-a}{2})$. This midpoint divides the interval into two smaller intervals of equal length. We then use the midpoints of these intervals for future rounds. Repeating this process can give us a sequence of unequal numbers that can be used as the work amounts. Fig. 3 gives a the first few numbers in this sequence as a sample.

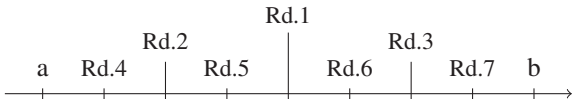

Fig. 3. Sample sequence of work amounts for the first 7 rounds. Rd.1 is the midpoint of a and b; Rd.2 is the midpoint of a and Rd.1; Rd.3 is the midpoint of Rd.1 and b; Rd.8 would be at the midpoint of a and Rd.4.

Pilot takes a and b from user input. In practice the user usually will likely set a to 0. This could cause the problem that some rounds are too short. Very short rounds are usually meaningless because they could be dominated by the non-stable phases. Pilot checks the duration after running each round, and if it finds that the previous round is shorter than a preset lower bound, the result will be stored but not used in analysis. Pilot doubles the work amount of the previous round until the round duration is longer than the lower bound, and will update a to that work amount.

In practice, the algorithm as described above has another drawback that the work amount of the first few rounds may be very large if b is a large number. For instance, if the user wants to understand the throughput of a device and uses (0, device size) for the valid parameter range, the first few rounds can be very long, and it would take a long time before the user can see the benchmark result. It is important for Pilot to give the user a quick (albeit rough) estimation of the result before spending a long time refining it. We use the following heuristic method in Pilot to solve this problem. Say that we know in round 1 that the time needed for finishing work amount a is $t_1 = s$ seconds, and for each new round we want it to be k seconds longer than the previous round. This means that the nth round would be $t_n = s + (n-1)k$ seconds long. Therefore, the total duration (t) of the n rounds would be:

$$t = \sum_{i=1}^{n} t_n = \frac{1}{2}k(n-1)n + ns.$$

Now if we want to get the initial result in t seconds, we can calculate k:

$$k = \frac{2t - 2sn}{n^2 - n} \tag{5}$$

Pilot uses equation (5) to calculate the initial slice size where t is a tunable parameter with a preset value 60 seconds. The number of rounds, n, should be greater than 50 in most cases [2] for the central limit theorem to take effect.

Another problem is that the work amount derived from this algorithm may be shorter than α (sum of the work amount of all non-stable phases). The method we use in Pilot to handle this issue is that we calculate the value of α after each round, and use the new value of α to update a. We also remove all results from previous rounds whose work amount is smaller than the newly calculated α.

B. Bottleneck detection

The accuracy requirement states that the benchmark should be stressing only the component that you want to measure, not anything else. This requires that the performance of other potential bottlenecks be monitored during the benchmark and that they are not limiting the performance of the benchmark. Many factors [5] can affect a benchmark in practice, and it is very hard to take all of them into account even for experienced people. The following automated approach cannot replace rigorous experimental design for scientific research and should be used to assist regular users and help to reduce the workload of scientists.

We use a systematic approach to bottleneck detection. First we identify the devices and data path between the devices. They are represented using a multi-source, multi-sink directed network. Each block is a device, which can have multiple inputs and one output. The data links describe how the pressure of the workload is generated. Fig. 4 is a sample that represents a workload that generates writes into multiple network attached storage systems from multiple clients, the data links start from the disk drives of the client machines (or memory if the data are generated on-the-fly) and end at the remote server's storage devices. It is possible to have more than one disjoint data path in a workload.

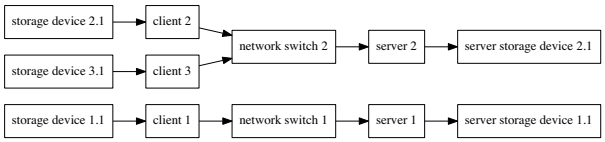

Fig. 4. Devices and data flows as represented by using a directed network

This device-level graph network provided a basis for understanding the related devices at a higher level, but is not detailed enough for actually carrying out the analysis. The second step is to discover components along the data paths. This is done by expanding a device into components. Using the sample above, we can expand the device link graph into a more detailed component link graph as shown in Fig. 5. We always measure the bi-directional data flow between components even when the workload only generates data in one direction because a congestion in any direction can negatively affect the performance of both directions.

The third step is to measure and monitor the utilization rate of each of these components while the workload is running. The results are invalid if any component's utilization rate passes a certain threshold. The monitoring is done throughout the workload because the utilization rates can change over time.

These steps are finicky and error-prone when manually executed. To minimize the effort and the possibility of missing important components, we combine several heuristics to automatically generate the data path and suggest components

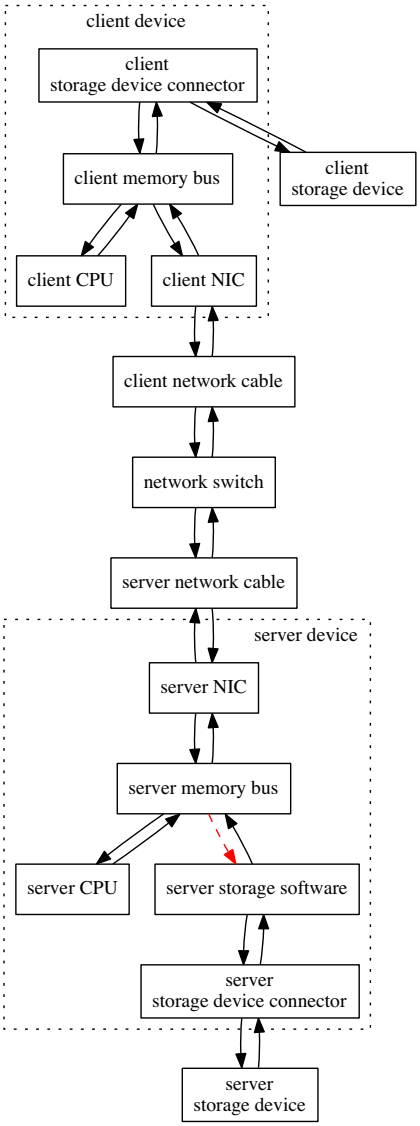

Fig. 5. Component data link graph expanded from one data link (Fig. 4). The dashed red line shows the expected bottleneck. No other link or node should have a more than 80% utilization rate. Even for cases when the data mainly flows in one direction, we still measure bi-directional flows because a congestion in any direction would slow down both data flows. We use two links between components in the graph to represent the send and receive data flows, enabling us to measure and visualize them respectively.

for the analysis. Pilot scans the list of parameters to identify file system path and check them one-by-one to see if they are locally or remotely mounted. For locally mounted file systems the related storage devices are added to the device list; for remotely mounted file systems the network subsystem and the remote machines are added. Pilot needs access to remote machines to scan for hardware and utilization information. This information is then used to generate the device and component data link graph. This scanning process is done by using customizable plug-ins. Pilot also supports importing a user designated component graph. There are two types of plug-ins that are used in the bottleneck detection. The first are static plug-ins that detect the hardware property and return a theoretical bandwidth limit. The second are dynamic plug-ins that can monitor the utilization rate for one or more PIs.

Finding the optimal sampling rate remains challenging because measuring the utilization rates of many components inevitably incurs overhead on the system. If the rate is too slow, we may miss spikes in component usage; if the rate is too high the overhead may shift the bottleneck of the system. In the fourth step we run the workload again with the bottleneck detection disabled to measure the results of the benchmark, and compare the results with and without the bottleneck detection. If there is a big difference between these two results, we can know that the bottleneck detection function has incurred a non-trivial impact on the result, and there is no guarantee that the bottleneck would remain the same when the bottleneck detection function is disabled. We need to find the highest sampling rate that generates an overhead no larger than a threshold. Pilot uses the Newton's method to find out this optimal sampling rate.

C. Overhead measurement

Measuring the performance of an application usually incurs running extra instructions for taking the measurement, doing some calculation, and storing the data. The overhead can be high when unit readings are being measured because that could require storing a large amount of information. Pilot executes a benchmark with the measurement instructions enabled and later disabled, and compare the results to calculate the overhead of the measurement. Since with the measurement instructions disabled Pilot could no longer acquire unit readings, the measurement of overhead will be done on the readings only.

D. Auto-correlation Detection and Mitigation

A benchmark session needs to be long enough so that we can collect enough samples to calculate the CI at the desired confidence level. The more samples we have, the narrower the CI can be made. However, a crucial issue that is often overlooked in many published benchmark results is the autocorrelation among samples. Autocorrelation is the cross-correlation of a sequence of measurements with itself at different points in time. Conceptually, a high autocorrelation means that previous data points can be used to predict future data points, and that would invalid the calculation of CI no matter how large the sample size is. Most measurements in computer systems are autocorrelated because of the stateful nature of computer systems. For instance, most computer systems have one or more schedulers, which allocate time slice to jobs. The measured performance of such jobs would be highly correlated when they are taken within a single time slice, and would change significantly between time slices if the duration of a measurement unit is not significantly longer than the size of a time slice. The autocorrelation in the samples must be properly handled before we can go on to the next step to calculate the sample's CI.

Autocorrelation is measured by the autocorrelation coefficient of a sequence, which is calculated as the covariance between measurements from the same sequence as

$$R(\tau) = \frac{\mathrm{E}[(X_t - \mu)(X_{t+\tau} - \mu)]}{\sigma^2},$$

where τ is the time-lag. The autocorrelation coefficient is a number in range $[-1, 1]$, where -1 means the sample data are reversely correlated and 1 means the data is autocorrelated. In statistics, $[-0.1, 0.1]$ is deemed to be a valid range for declaring the sample data has negligible autocorrelation [3].

Subsession analysis [3] is a statistical method for handling autocorrelation in sample data. n-subsession analysis models the test data and combines every n samples into a new sample. Pilot calculates the autocorrelation coefficient of measurement data after performing data sanitizing, such as non-stable phases removal, and gradually increases n until the autocorrelation coefficient is reduced to within the desired range.

E. Deciding Optimal Session Length

On a high level, a benchmark session comprises many rounds. We calculate the CI after collecting new data from each round. The session ends when the CIs of all PIs reach the target. Because each round can have non-stable phases that are not contributing samples, we should maximize the length of each round and minimize the number of rounds. This also has the extra benefit of including those work units that are far from the beginning of the initial work amount.

But we cannot begin the first round using the maximum work amount, because in many cases the maximum work amount can be very large. This is typical in storage benchmarks where the limit can be the total size of a storage device, and it would take several dozens of hours to finish one round that fully writes a device. In network benchmarks, we can even set the maximum work amount to unlimited because these workloads can keep running forever. If we start the first round of workload with the full work amount, we risk letting the user wait too long a time before showing the first result. Therefore, we begin the benchmark session with a few short trial rounds to learn the duration to work unit ratio.

a) Workloads that provide unit readings: We treat each unit reading as one measurement. One workload round can usually provide hundreds or thousands of measurements, making it faster to reach the required sample size. For instance, if a sequential write workload writes 500 MB data using 1 MB I/O, we can get 500 throughput measurements if the workload saves the duration of each write. Pilot sends the unit reading results through the non-stable phases removal, performs autocorrelation reduction, and uses the rest of the unit readings to calculate the CI. Pilot keeps running new rounds of the workload until the desired width of the CI is reached.

b) Workloads that cannot provide unit reading: These workloads are handled using the WPS method (§ III-A), which also performs non-stable phases detection and removal, subsession analysis, and decides the optimal number of rounds to achieve the desired width of CI.

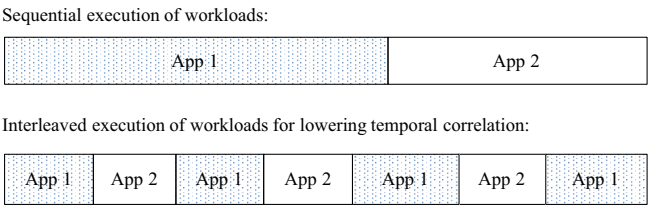

Fig. 6. Interleaved execution of benchmarks can help to lower temporal correlation between rounds.

F. Comparing Results

The need to compare benchmark results using the shortest possible time is our very first motivation for designing Pilot. Not only useful for system design and tuning, a handy tool for comparing benchmark results can help many tasks in systems research, such as choosing the fastest data structure for storing certain data, finding performance regressions in software development, and deciding the best parameters for storage or network communication. Without using the correct statistics method at runtime, most of these decisions were done in an ad hoc manner, either prematurely with too little data, or blindly wasting time to gather more than necessary data.

Suppose we have n workloads and the results of them are comparable, which means that they are using the same unit of measurement and are of the same scale. Now we need to rank (order) these workloads according to their results. In some cases we need to run these workloads to get new results, in other cases the comparison involves old benchmark results. For old results we need three values: the mean, subsession sample size, and subsession variance (we need to use the subsession analysis, as shown in § III-D, to reduce autocorrelation between samples because i.i.d. is a hard requirement for all the analyses we use in this section). For the rest of this section all samples are subsession samples that have low autocorrelation.

Unlike the algorithm in § III-E, which executes multiple rounds of a workload guided by heuristics until a desired width of CI is reached, the algorithm for comparing results interleaves the execution of workload rounds to reduce temporal correlation between workload rounds, as shown in Fig. 6. For instance, Pilot executes the workloads in the following order: Workload 1 Round 1, Workload 2 Round 1, Workload 3 Round 1, . . ., Workload n Round 1; then go back to run Workload 1 Round 2, Workload 2 Round 2, etc.

There are two cases when we consider comparing two benchmark results. The first case is when their CIs are not overlapped. In this case we can be sure that one is greater than the other at the confidence level used to calculate the CIs. The second is when the CIs overlap. In this case we use the Welch's unequal variance t-test [15] (an adaptation of Student's t-test [13]) to compare the benchmark results, A and B. Welch's t-test is more reliable when the two samples have unequal variance and unequal sample sizes, which are true for most system benchmarks. This test can effectively tell us the probability of rejecting a hypothesis and the required

sample size. Here the null hypothesis (the hypothesis we want to reject) is that there is no statistical significant difference between result A and B (A = B). We compute the probability (p-value) of getting results A and B if the null hypothesis is true. Let $\overline{x}$ be the mean of the result, σ^2 be the variance, and n be the sample size. We can calculate the p-value:

$$t = \frac{\overline{x_A} - \overline{x_B}}{\sqrt{\frac{\sigma_A^2}{n_A} + \frac{\sigma_B^2}{n_B}}} \tag{6}$$

$$\nu = \left\lfloor \frac{\left(\frac{\sigma_A^2}{n_A} + \frac{\sigma_B^2}{n_B}\right)^2}{\frac{\sigma_A^4}{n_A^2(n_A-1)} + \frac{\sigma_B^4}{n_B^2(n_B-1)}} \right\rfloor \tag{7}$$

$$p = 2\,\mathrm{cdf}(t, \nu) \tag{8}$$

Equation (7) is the Welch-Satterthwaite equation for calculating the degree of freedom (ν), and $\mathrm{cdf}(t, n)$ is the Student's t-distribution with ν degrees of freedom. We multiply it by 2 to calculate the two-tailed distribution.

The comparing result algorithm runs until:

1) There are enough data for calculating the CIs,
2) Each adjacent CI pair is either non-overlap or their p-value of the null hypothesis ($A = B$) is less than the predefined threshold (usually 0.01),
3) Every CI is tighter than the required width (this step is optional but recommended because a narrower CI makes it easier to compare with new results in the future).

Pilot needs to decide the work amount for running each round of the workload. The value has to be chosen in such a way that we can minimize the number of rounds needed to reduce the impact of the overhead of starting a round. Mathematically the optimal subsession sample size can be calculated using a variation of equation (6).

IV. The Pilot Framework

Our analysis of recent publications in the systems research field shows three common types of benchmarking. The first is to evaluate the performance of a piece of source code, which is usually relatively short, requires little to no setup, and does not have a dedicated supporting benchmark framework involved. Samples of such cases are comparing the performance of two hash algorithms and comparing the performance of two ways of iterating over a large matrix. The second kind of benchmarks includes most workloads that are specifically designed for performance measurement. They are usually complex, requires a relatively long setup process, needs a pre-configured benchmark framework, and can have a long duration (hours to days). The third kind of benchmark includes those that are done quickly and on spot, often when the user needs to have a quick estimate of the performance of a certain piece of hardware or software, such as doing a short benchmark to decide which external hard drive is faster or downloading a file from the Internet to test the speed of the Wi-Fi. In these cases, the precision requirement of the measurement is not paramount, but they have to be done quickly and usually involve using a command line program (like dd [4] or cURL [14]).

Pilot is designed to work with all three kinds of benchmarks. Pilot takes the requirements of the benchmark as the input, applies the algorithms as described in the previous section to execute the benchmark, and generates a detailed report of the benchmark results. The default workload benchmark declaration includes the following tasks:

1) The number of PIs, and for each PI:
   a) name and unit,
   b) desired width of CI (default to 10% of mean),
   c) confidence level (default to 95%),
   d) desired autocorrelation coefficient (default to $[-0.1, 0.1]$).
2) The valid range of the work amount,
3) Non-stable phases removal (default: EDM and WPS),
4) Bottleneck detection (default: enabled),
5) Overhead detection (default: enabled),

For the first kind of benchmark, Pilot can be easily linked into an existing code base by adding a few lines of code, very much like the way how a unit test framework is used. We support C/C++ first because they are the language in which most performance critical code is written. For the second kind of benchmark, Pilot provides an extensive list of library functions for integrating with the existing benchmark framework. The developer of the benchmark can choose to either let Pilot decide the number of rounds and the work amount for each round (the simplest way) or to manually control the executing of the workload and use Pilot's analytical functions as a guide to the execution (more flexible). For the third kind of benchmark, Pilot can run quick and short benchmark jobs by controlling a workload program through a command-line interface.

V. Evaluation

Pilot is designed to make running benchmarks easier. That cannot be measured unless we do a user study. Unfortunately, we do not have the resource for that, nor could we measure how much time can be saved by using Pilot, because that depends on how people run benchmarks before using Pilot. Instead, we evaluate the characteristics of the algorithms in order to understand them better. We present two evaluations: time to reach the desired width of CI and prediction error of the results.

A. Time to reach desired CI

Reaching the desired width of CI means that the precision requirement is achieved and is often the major goal of a measurement. Understanding the time to reach (TTR) desired CI can help us plan and design benchmark tasks, and identify problems in current benchmarking practices. We compare the TTR of three methods: *UR data without non-stable phase removal*, *UR with EDM non-stable phase removal*, and *WPS method*. The desired width of CI is set to 10% of mean. All methods include data sanitization, such as short-round detection and using subsession analysis to mitigate autocorrelation.

The result is shown in Table I. We can see that the TTR varies greatly in different workloads. TTR is not only affected by the required sample size but also the autocorrelation in the samples (high autocorrelation requires merging more samples),

TABLE I
THE TIME NEEDED TO REACH (TIME-TO-REACH, TTR) DESIRED WIDTH OF CI (10% OF RESULT MEAN) USING VARIOUS METHODS.

Workload	UR w/o non-stable removal			UR with EDM non-stable removal			WPS method	
	Result CI, σ[a]	TTR CI[b], σ	SS CI[c], σ	Result CI, σ	TTR CI, σ	SS CI, σ	Result CI, σ	TTR CI, σ
Seq. write (hard disk)	116–116, 3	1687–3738, 829	605–1000, 342	106–108, 0.3	127–229, 300	1–2, 1	99–108, –	730–8950, 1655
Seq. write (flash)	707–741, 6	12–15, 10	6–9, 14	719–719, 6	13–15, 5	12–16, 14	664–681, –	267–389, 58
Send data over Wi–Fi	26–27, 0.3	33.79–45.47, 20	1–2, 1	20–22, 0.4	50–106, 289	1–1, 1	21–26, –	8120–14345, 3012

[a] Result CI is the average measured CI of the results from respective method. σ is the average measured standard deviation. The unit for Result CI and σ is MB/s. All CIs are calculated at 95% confidence level.
[b] TTR CI shows the CI of time-to-reach. The unit for TTR is a second.
[c] SS is the subsession size (how many adjacent samples are merged to reduce autocorrelation). All methods include autocorrelation removal.

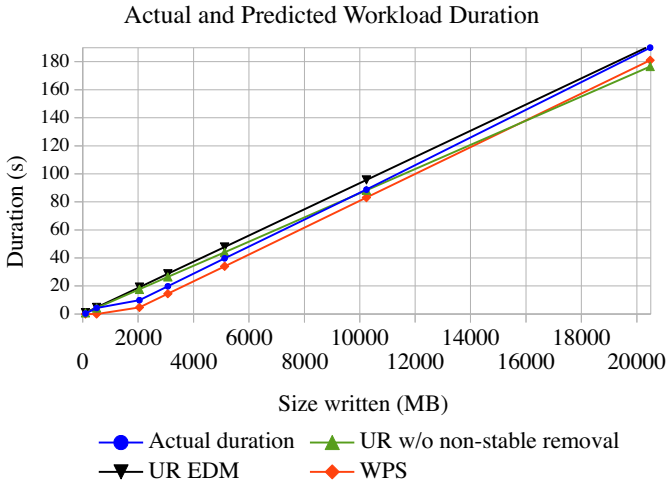

Fig. 7. The actual and predicted time for writing certain amount of data sequentially. Three methods are used to predict the run time: UR without non-stable phases removal, UR with EDM non-stable phase removal, and the WPS method. Closer to the actual duration is better. The slope of the actual duration line changes at around 2000 MB, after which the effect of write cache starts to become less significant. The CI for each data point is very tight so we omit them in the figure for clarity.

which in turn can be affected by different non-stable phases removal methods. TTR varies greatly even when using the same method. This highlights the importance of using runtime analysis to decide the optimal sample size on-the-fly and the inadequacy of using a fixed benchmark duration. Using a fixed duration of benchmark leads to imprecise results for workloads that require a long duration and a waste of time for workloads that only require a short duration.

We can also see that the EDM method converges to the desired CI using the shortest time. Without non-stable phases removal, the URs from the hard disk have very high autocorrelation, and Pilot needs to merge almost a thousand adjacent samples to reduce the autocorrelation to below 0.1. If we remove the non-stable phases first as in the EDM method, the autocorrelation becomes much lower and also the time needed to reach the desired sample size is shorter; EDM can be seen as a way to sanitize the samples. The WPS method does not require UR data and can only get one sample per round, thus it requires a longer time to reach the required sample size. We can see that sometimes the WPS method needs more than 8000 seconds. Therefore, it should only be used when UR data is not available. The results of all three methods also demonstrate the importance of detecting and handling autocorrelation. The high numbers of SS mean that many workloads have inherently very high autocorrelation between samples and it is wrong to apply almost any statistical method if the results are not i.i.d.

The result matches our expectation. The *UR with EDM method* should be the first choice for workloads that can provide UR data, and the *WPS method* can be used for other workloads. Whenever possible, benchmark workloads should be divide into measurable small units to increase the sample size and reduce the TTR.

B. Predicting future workloads

One major purpose for measuring performance is to use it to predict the time needed for running future workloads. It can be seen from Table I that different methods generate slightly different results. This evaluation is designed to see which results can best predict future durations. We measure the actual durations of sequentially writing different amounts of data and compare them with the predicted durations that are calculated using the values from Table I. We use the center of the CIs in the calculation. The result is shown in Fig. 7.

The slope of the actual duration line becomes steeper after around 2000 MB. Our test machine's Linux OS caches most of the writes before 2000 MB in memory so the time needed is short. After that the writes become slower because the system has to actually write the data to the disk. Therefore, the write performance after the first 2000 MB better reflects the real disk's write performance. It can be seen that the EDM line's slope best matches the slope of the post-2000 MB segment of the actual duration line. This means that UR with EDM non-stable phases removal is best for calculating the stable performance. Without non-stable phases removal, the slope of the green line is negatively affected by the huge write cache of the system. The WPS method also gets the correct slope before and after 2000 MB so it can also find the correct stable performance when UR is not available.

It should be noted that due to limited space we are only presenting the results of one HDD, and the prediction errors also depend on how the workload works. Our current conclusion is that the EDM method is best for getting the stable performance

and also has the shortest TTR. The WPS method can effectively detect the non-stable phases even when only readings are available.

VI. Related Work

Boudec's book [1] extensively covers the statistics knowledge that is needed for computer performance evaluation. Hoefle and Belli analyzed 95 papers from HPC-related conferences with a focus on experimental design and how to present the results to make them more interpretable [7]. Jimenez et al.'s work on using OS-level container can be adopted along with Pilot to get reproducible performance evaluation results [8].

Auto-pilot [16] provides a script language for controlling the execution of benchmarks along with other mechanisms for timing and analyzing results. Auto-pilot offers functions to calculate CI and stop the execution when the required width is reached but lacks other functions such as handling unit readings separately, reducing auto-correlation, detecting non-stable phases, or generating work amount dynamically. Some parts of it are written in Perl and cannot be easily integrated into C programs for doing high performance run-time analysis. *ministat* [9] is a tool for comparing benchmark results at a certain confidence level and drawing the results using ASCII art.

VII. Conclusions and Future Work

Manually monitoring and performing benchmark analysis are demanding and error prone. We propose a series of algorithms and heuristics to automate this process and generate reports that are scientifically and statistically valid using as short time as possible. Evaluation shows that these methods are non-parametric and robust, and can shorten the time needed for running benchmark. We have implemented the proposed methods in an easy-to-use open source framework called Pilot, which actually contains more heuristics methods for running benchmarks than can be described in this paper. We sincerely hope it can increase the quality of performance evaluation in computer systems research and reduce people's effort. We are trying our best to make Pilot easy to learn and use. We provide Pilot's source code, precompiled packages, tutorials, API documentation, mailing list, issue tracker, and wiki at **https://ascar.io/pilot**.

Pilot is an ongoing project, and we are developing more functions, such as a distributed mode that can measure and make decisions based on measurements from many nodes and analytical functions for detecting slow shift for long term monitoring. The development will be handled in a community-friendly manner and we welcome outside contributions.

Acknowledgments

This research was supported in part by the National Science Foundation under awards IIP-1266400, CCF-1219163, CNS-1018928, CNS-1528179, by the Department of Energy under award DE-FC02-10ER26017/DESC0005417, by a Symantec Graduate Fellowship, by a grant from Intel Corporation, and by industrial members of the Center for Research in Storage Systems.

References

[1] Jean-Yves Le Boudec. *Performance Evaluation Of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010.

[2] Tianshi Chen, Yunji Chen, Qi Guo, Olivier Temam, Yue Wu, and Weiwu Hu. Statistical performance comparisons of computers. In *Proceedings of the 18th International Symposium on High-Performance Computer Architecture (HPCA-18)*. IEEE, 2012.

[3] Domenico Ferrari. *Computer Systems Performance Evaluation*. Prentice-Hall, 1978. ISBN 9780131651265.

[4] The Open Group. dd, convert and copy a file. The Single UNIX® Specification, 2013.

[5] Tim Harris. Do not believe everything you read in the papers. Fourth National Information Communications Technology Australia (NICTA) Research Centre Software Systems Summer School, https://ssrg.nicta.com.au/Events/summer/16/harris.pdf, February 2016.

[6] Fumio Hayashi. *Econometrics*. Princeton University Press, 2000.

[7] Torsten Hoefle and Roberto Belli. Scientific benchmarking of parallel computing systems. In *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC15)*, 2015.

[8] Ivo Jimenez, Carlos Maltzahn, Jay Lofstead, Adam Moody, Kathryn Mohror, Remzi Arpaci-Dusseau, and Andrea Arpaci-Dusseau. Characterizing and reducing cross-platform performance variability using OS-level virtualization. In *The First IEEE International Workshop on Variability in Parallel and Distributed Systems (VarSys 2016)*, Chicago, USA, May 2016.

[9] Poul-Henning Kamp. ministat, a statistics utility. https://www.freebsd.org/cgi/man.cgi?ministat, November 2012.

[10] David S. Matteson and Nicholas A. James. A nonparametric approach for multiple change point analysis of multivariate data. *Journal of the American Statistical Association*, 109(505), 2014.

[11] Open Science Collaboration. Estimating the reproducibility of psychological science. *Science*, 349(6251), 2015.

[12] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. URL http://www.R-project.org/.

[13] Student. The probable error of a mean. *Biometrika*, 6(1): 1–25, 1908.

[14] Contributors to the cURL project. The cURL command line tool. https://curl.haxx.se/, May 2016.

[15] B. L. Welch. The generalization of Student's problem when several different population variances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[16] Charles P. Wright, Nikolai Joukov, Devaki Kulkarni, Yevgeniy Miretskiy, and Erez Zadok. Auto-pilot: A platform for system software benchmarking. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 175–188, Anaheim, CA, April 2005.