Treadmill: Attributing the Source of Server Tail Latency through Precise Load Testing and Statistical Inference

Yunqi Zhang
University of Michigan
yunqi@umich.edu

David Meisner
Facebook Inc.
meisner@fb.com

Jason Mars
University of Michigan
profmars@umich.edu

Lingjia Tang
University of Michigan
lingjia@umich.edu

Abstract—Managing tail latency of requests has become one of the primary challenges for large-scale Internet services. Data centers are quickly evolving and service operators frequently desire to make changes to deployed software and hardware features. Such changes demand a confident understanding of the impact on one’s service, in particular its effect on tail latency (e.g., 95th- or 99th-percentile response latency of the service). Evaluating the impact on the tail is challenging because of its inherent variability. Existing tools and methodologies for measuring these effects suffer from a number of deficiencies including poor load tester design, statistically inaccurate aggregation, and improper attribution of effects. As shown in the paper, these pitfalls can often result in misleading conclusions.

In this paper, we develop a methodology for statistically rigorous performance evaluation for server workloads. First, we find that the design of the server load tester is critical to ensuring quality results and empirically demonstrate the inaccuracy of load testers in previous work. Learning from these flaws, we design and develop a modular load tester platform, Treadmill, that overcomes many pitfalls of existing tools. Next, utilizing Treadmill, we then construct measurement and analysis procedures that can properly attribute performance factors. We build on prior research in statistically-sound performance evaluation and quantile regression, extending it to accommodate the idiosyncrasies of server systems. Finally, we use our augmented methodology to evaluate the utility of common server hardware features with Facebook production workloads on production hardware. We decompose the effects of these features on request tail latency and demonstrate that our evaluation methodology provides superior results, particularly in capturing complicated and counter-intuitive performance behaviors. By tuning the hardware features as suggested by the attribution, we reduce the 99th-percentile latency by 43% and its variance by 93%.

Keywords—tail latency; load testing; data center;

I. INTRODUCTION

Mitigating tail latency (i.e., high quantiles of the latency distribution) is critical for quality of service in large-scale Internet services [1]. High tail latency has been identified as one of the key challenges facing modern data center design as it results in poor user experiences, particularly for interactive services such as web search and social networks. These services are powered by clusters of machines wherein a single request is distributed among a large number of servers in a “fan-out” pattern. In such design, the overall performance of such systems depends on the slowest responding machine [2]. Recent work has sought to control and understand these tail requests both at the individual server and overall cluster level [3].

For data center operators, the capability of accurately measuring tail latency without disrupting production system is extremely important for a number of reasons. First, evaluating hardware configurations requires extensive testing and evaluation against existing workloads. Servers are typically acquired in large quantities (e.g., 1000s at a time), so it is important to choose the best design possible and accurately provision resources. Second, it is necessary to be able to faithfully measure performance effects without disrupting production systems. The high frequency of software and hardware changes makes it extremely hard, or impossible, to evaluate these changes in production, which can easily result in user-visible incidents. Instead, it is desirable to understand the impact of performance-critical decisions in a safe, but accurate load testing environment.

Building such load testing environment for accurate tail latency measurement is particularly challenging. This is primarily because there are significantly more systems and resources involved for large-scale Internet service workloads (e.g., the server-side software, the network connections etc.) than traditional single-server workloads (e.g., SPEC CPU2006, PARSEC). Although there have been several prior works [4,5,6,7,8,9,10] trying to bridge this gap recently, they have fundamental pitfalls as we will show later in the paper. This is concerning because these tools are commonly used in research publications and their pitfalls may result in misleading conclusions. In addition to academia, there is a lack of such accurate tail latency measurement test bed in industry, which results in unnecessary resource over-provisioning [11] and unrecognized performance regressions [12].

Furthermore, to be able to control the tail latency of these Internet services, a faithful understanding of the source of tail latency is required. These Internet services interact with a wide range of resources including operating system, network stack and server hardware that the ability of quantitatively attributing the source of tail latency to individual resources is critical yet challenging. Moreover, although a number of prior works [13,14,15,16,3,17,18,19,20,21,22,23,24,25] have studied the impact of individual resource on the tail latency, many resources have complex interacting behaviors (for example, TurboBoost and DVFS scaling governor may interact
indirectly through competing for the thermal headroom) that cannot be captured in isolated studies. Note that this capability of identifying the source of tail latency relies on the first aforementioned challenge. In other words, without an accurate measurement of the tail latency we will not be able to correctly attribute it to various sources.

In this paper, we first survey existing performance evaluation methodologies used in the research community. We have identified a set of common pitfalls across these tools:

- **Query inter-arrival generation** - Load testing software is often written for software simplicity. We find commonly used programming paradigms create an implicit queueing model that approximates a closed-loop system and systematically underestimates the tail latency. Instead, we demonstrate a precisely-timed open-loop load test is necessary to properly exercise the queueing behavior of the system.

- **Statistical aggregation** - Due to high request rates, sampling must be used to control the measurement overhead. Online aggregation of these latency samples must be performed carefully. We find that singular statistics (e.g., a point estimate of the 95th- or 99th-percentile latency) buries far too much detail; static histograms used in other load testers also exhibit bias.

- **Client-side queueing bias** - Due to high throughput rates (100k - 1M requests per second) in many commercial systems, we demonstrate that multiple client machines must be used to test even a single server. Without lightly utilized clients, latency measurements quickly begin to incorporate client-side queueing, therefore biasing measurements.

- **Performance “hysteresis”** - We observe a phenomenon in which the estimated latency converges after collecting a sufficient amount of samples, but upon running the load test again, the test converges to a different value. This is caused by changes in underlying system states such as the mapping of logical memory, threads, and connections to physical resources. We refer to this as hysteresis because no reasonable amount of additional samples can make the two runs converge to the same point. Instead we find experiments must be restarted multiple times and the converged values should be aggregated.

Based on these insights, this paper proposes a systematic procedure for accurate tail latency measurement, and details the design choices that allow us to overcome the pitfalls of existing methodologies. The proposed procedure leverages multiple lightly-utilized instances of Treadmill, a modular software load tester, to avoid client-side queueing bias. The software architecture of Treadmill preserves proper request inter-arrival timings, and allows easy addition of new workloads without complicated software changes. Importantly, it properly aggregates the distributions across clients, and performs multiple independent experiments to mitigate the effects of performance hysteresis.

The precise measurement achieved by Treadmill enables the capability of identifying “where” the latency is coming from. We build upon recent research in quantile regression [26], and attribute portions of the tail latency to various advanced hardware features. This allows us to unveil the system “black box” and better understand the impact of tuning hardware configurations on tail latency. We perform this evaluation using Facebook production hardware running two critical Facebook workloads: the pervasive key-value server Memcached and a recently-disclosed software routing system mcrouter [27]. Using our tail latency attribution procedure, we are able to identify many counter-intuitive performance behaviors, including complex interactions among different hardware resources that cannot be captured by prior studies of individual hardware features.

Finally, we demonstrate that we successfully capture 90% of performance variation in the system for mcrouter and over 95% for Memcached. By carefully tuning the hardware features as suggested by the attribution, we reduce the 99th-percentile latency by 43% and the variance of 99th-percentile by 93%.

To summarize, this paper has three main contributions:

- **Survey of common pitfalls in existing load testing methodology** – We conduct a survey of existing methodologies in related work, and present an empirical demonstration of their shortcomings. We classify these flaws into four major principles for future practitioners.

- **Accurate cluster-based performance evaluation methodology** – We present the design of a robust experimental methodology, and a software load testing tool Treadmill, which we release as open-source software\(^1\). Both properly fulfill the requirements of our principles and are easily extensible for adoption.

- **Attributing the source of tail latency** – The high precision measurements achieved by our methodology enables the possibility of understanding the source of tail latency variance using quantile regression. We successfully attribute the majority of the variance to several advanced hardware features and the interactions among them. By carefully tuning the hardware configurations recommended by the attribution results, we significantly reduce the tail latency and its variance.

### II. Pitfalls in the State-of-the-Art Methodologies

To understand the requirements of an evaluation test bed, we first survey existing methodologies and tools in prior work. Many tools are available to study the performance of server-side software, including YCSB [4], Faban [28], Mutillate [29] and CloudSuite [6]. These tools have been widely used in standard benchmark suites, including SPEC2010 jEnterprise [30], SPEC2011 SIP-Infrastructure [31], CloudSuite [6] and BigDataBench [8], thereby many recently published research projects.

\(^1\)https://github.com/facebook/treadmill
Through studying these existing tools, we found several common pitfalls. We categorize them into four major themes provided below.

<table>
<thead>
<tr>
<th>Theme</th>
<th>YCSB</th>
<th>Faban</th>
<th>CloudSuite</th>
<th>Mutilate</th>
<th>Treadmill</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Inter-arrival Generation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statistical Aggregation</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Client-side Queuing Bias</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Hysteresis</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A. Query Inter-arrival Generation

A performance evaluation test bed requires a load tester, a piece of software that issues requests to the server in a controlled manner. A client machine will run the load tester which periodically constructs, sends and receives requests to achieve a desired throughput. There are 2 types of control loops that are often employed to create these timings: open-loop control and closed-loop control [32]. The closed-loop controller has a feedback loop, where it only tries to send another request after the response to the previous request has already been received. In contrast, the open-loop controller sends requests at defined times regardless of the status of the responses. Almost all the modern data center server-side softwares are built to handle open-loop setup, so that each server thread does not reject requests from clients while busy processing previous ones.

However, many load testers are implemented as closed-loop controller because of software simplicity, including Faban, YCSB and Mutilate as we shown in Table I. Often, load is generated by using worker threads that block when issuing network requests. The amount of load can then be controlled by increasing or decreasing the amount of threads in the load generator. Unfortunately, this pattern exactly resembles a closed-loop controller; each thread represents exactly one potentially outstanding request.

Figure 1 demonstrates the impact of closed-loop and open-loop controllers. The “Open-Loop” line shows the cumulative-distribution of the number of outstanding requests in an open-loop controlled system when running at 80% utilization. The “Closed-Loop” lines show the distribution of the number of outstanding requests in a closed-loop controlled system with 4, 8 and 12 concurrent connections respectively. The closed-loop controller significantly underestimates the number of outstanding requests in the system and therefore queueing latency, which creates bias in tail latency measurement.

A. Query Inter-arrival Generation

Figure 2. Different latency distributions measured from multiple servers in the a multi-client performance evaluation procedure, where the y-axis shows the decomposition among servers as what percent of samples is contributed by each server. We can see from the figure that Client 1 dominates the high quantiles of the combined distribution thus bias the measurement, because it resides on a different rack than the other clients and the server.

Figure 1 demonstrates the impact of closed-loop and open-loop design. For an open-loop design, the number of outstanding requests varies over time and follows the shown distribution. The high-quantiles of the distribution exhibit far more outstanding requests (and therefore queueing latency) than a closed-loop design. Using a closed-loop design can significantly underestimate the request latency especially at high quantiles. Therefore, we conclude that a open-loop design is required to properly exercise the queueing behavior of the system.

B. Statistical Aggregation

Due to high request rates, load tester software needs to perform at least some statistical aggregation of latency samples to avoid the overhead of keeping large number of samples. We find that care must be taken in this process and two types of errors can occur.

First, it is important for load testers to keep an internal histogram of latency that adapts over time. Those load testers that do maintain a histogram often make the mistake of statically setting the histogram buckets. Non-adaptive histogram binning will break when the server is highly utilized, because the latency will keep increasing before reaching the steady state thus exceeds the upper bound of the histogram.

Moreover, if the requests have distinct characteristics (e.g., different request types, sent by different machines, etc.), we observe that bias can occur due to improper statistical aggregation. For example, in Figure 2 we demonstrate a scenario where four clients are used to send requests to the same Memcached server, and “Client 1” is on a different rack than the other clients and the server. At each latency point on the x-axis, the shaded regions represent the proportion of samples that come from one of the four clients. As the quantile gets higher, one can clearly see that most of the samples are coming from “Client 1”. This bias is problematic because the performance estimate of the system becomes a function of one single client. Instead, one should extract the interested metrics (e.g., 99th-percentile latency) at each client individually, and aggregate them properly.
**A. Treadmill**

Given the existing pitfalls in state-of-the-art load tester tools, we decide to build our own load tester Treadmill. To demonstrate the effectiveness of our methodology, we then evaluate it on real hardware (Section III-C).

**B. CloudSuite**

A modular software load tester (Section III-A), and a robust procedure for tail latency measurement (Section III-B). To guarantee the performance of the load tester tools, we decide to build our own load tester Treadmill. Specifically, the problems in existing tools are addressed by the following design decisions.

- **Query inter-arrival generation**: To guarantee the query inter-arrival generation can properly exercise queueing behavior of the system, we implement an open-loop controller. The control loop is precisely timed to generate requests at an exponentially distributed inter-arrival rate, which is consistent with the measurements obtained from Google production clusters [1].

**C. Client-side Queueing Bias**

While operating a load tester, it is important to purely measure the effects of server-side latency. For workloads with long service times (e.g., complex MySQL queries), clients do not have to issue many requests to saturate the server. However, for workloads like Memcached the request rates on the clients and network themselves are quite high. This can lead to queueing effects in the clients and network themselves, thereby bias the latency measurements. YCSB and CloudSuite suffer from such bias due to their single client configuration as shown in Table I.

Figure 3 shows an example of how client and network utilizations can bias latency measurements. In “Single-Client Setup”, the client and the network have the same utilization as the server, which results in increasing queueing delay when the server utilization increases. This will bias the latency measurement, whereas in the multi-client setup the utilizations of the network and the client are kept low enough that they only add an approximately constant latency.

**D. Performance “Hysteresis”**

Through experimentation we find an unusual behavior in how estimates converge that we refer to as performance “hysteresis”. Figure 4 demonstrates that as more samples are collected, the estimate of 99th-percentile latency begins to converge to a singular value. However, if the server was restarted and another run was performed, the estimate can converge to a different value.

In this case, the estimates have a large sample size and we would have expected the “confidence” in each estimate to be high, but clearly there still exists variance across runs. In fact, the difference between these estimates and the average is as high as 67%. This phenomenon means that one cannot achieve higher statistical accuracy simply by “running for longer” and is similar to effects observed in STABILIZER [33]. Instead, it is necessary to restart the entire procedure many times and aggregate the results.

In this case, the estimates have a large sample size and we would have expected the “confidence” in each estimate to be high, but clearly there still exists variance across runs. In fact, the difference between these estimates and the average is as high as 67%. This phenomenon means that one cannot achieve higher statistical accuracy simply by “running for longer” and is similar to effects observed in STABILIZER [33]. Instead, it is necessary to restart the entire procedure many times and aggregate the results. However, none of the existing load testers is robust enough to handle this scenario as shown in Table I.

**III. METHODOLOGY**

To overcome the four common pitfalls we find in existing methodologies, we design and develop Treadmill, a modular software load tester (Section III-A), and a robust procedure for tail latency measurement (Section III-B). To demonstrate the effectiveness of our methodology, we then evaluate it on real hardware (Section III-C).

**A. Treadmill**

Given the existing pitfalls in state-of-the-art load tester tools, we decide to build our own load tester Treadmill. Specifically, the problems in existing tools are addressed by the following design decisions.

- **Query inter-arrival generation**: To guarantee the query inter-arrival generation can properly exercise queueing behavior of the system, we implement an open-loop controller. The control loop is precisely timed to generate requests at an exponentially distributed inter-arrival rate, which is consistent with the measurements obtained from Google production clusters [1].
- **Statistical aggregation:** To provide high precision statistical aggregation, Treadmill goes through three phases during one execution: warm-up, calibration and measurement. During the warm-up phase, all measured samples are discarded. Next, we determine the lower and upper bounds of the sample histogram bins in the calibration phase. The calibration phase is useful to reduce the amount of information lost from transforming detailed latency samples into a histogram. Finally, Treadmill begins to collect samples until the end of execution. Histograms are used to reduce the storage and performance overhead, and are re-binned when sufficient amount of values exceed the histogram limits.

- **Client-side queueing bias:** To avoid client-side queueing bias, we use wangle [34], which provides inline execution of the callback function, to ensure that the response callback logic is executed immediately when the response is available. In addition, we highly optimize for performance (e.g., lock-free implementation), which indirectly reduces the client-side queueing bias by keeping the clients at low utilization.

Furthermore, we also optimize for generality, making it easy to extend Treadmill to new workloads. Moreover, Treadmill is also able to reproduce configurable workload characteristics.

- **Generality:** We also optimize for generality, to minimize the amount of effort required to extend Treadmill to a new workload. So far, we have successfully integrated Treadmill with Memcached [35], TAO [36], mcrouter [27], Thrift [37] and many other internal services at Facebook. Each integration takes less than 200 lines of code.

- **Configurable workload:** It has been demonstrated in prior work [38] that workload characteristics (e.g., the ratio between GET and SET requests in Memcached) can have a big impact on the system performance. Therefore, being able to evaluate the system against various workload characteristics can improve the accuracy of measurement. To do so, a JSON formatted configuration file can be used to describe the workload characteristics and feed into Treadmill.

### B. Tail Latency Measurement Procedure

Treadmill provides highly accurate measurement even at high quantiles. However, as we illustrated in Figure 3, multiple clients are needed to avoid client-side queueing bias for testing high throughput workloads like Memcached. Therefore, we developed a methodology leverages multiple Treadmill instances to perform load testing for such workloads.

To measure the tail latency, multiple instances of Treadmill are used to send requests to the same server, where each instance sends a fraction of the desired throughput. Then the same experiment is repeated multiple times, and the measurements from each experiment is aggregated together to get a converged estimate. Particularly, We make the following design decisions when designing this procedure.

- **Statistical aggregation:** First, we need to aggregate the statistics reported by multiple instances of Treadmill in each experiment. In this process, the common practice that combines distributions obtained from all Treadmill instances to a holistic distribution and then extracts interested metrics (e.g., 99th percentile latency) could be heavily biased by outliers as we illustrated in Figure 2. Instead, we first compute the interested metrics from each individual Treadmill instance, and then combine them by applying aggregation functions (e.g., mean, median) on these metrics.

- **Client-side queueing bias:** By leveraging multiple instances of Treadmill, that each of them is highly performing, we can keep all the clients under low utilization thus prevent the measurement from client-side queueing bias.

- **Performance hysteresis:** To avoid performance hysteresis, Gym takes multiple measurements by repeating the same experiment multiple times. After each experiment, we record the collected measurement and repeat this procedure until the mean of the collected the measurements has already converged.

### C. Evaluation

In this section, we evaluate the accuracy of the tail latency measurement obtained from Treadmill. We focus on Memcached [39] due to its popularity in both industry [40,35] and academia [41,42,43,44,45,46], as well as its stringent performance needs. Besides Treadmill, we also deploy two other recently published load testers CloudSuite [6] and Mutilate [29] for comparison.

To set them up, we explicitly follow the instructions they publish online. Specifically, 1 machine is used for running Memcached server, and 1 machine is used for the load tester from CloudSuite. Mutilate runs on 8 “agent” clients and 1 “master” client as suggested in the instructions, and also sends requests to 1 Memcached server. For Treadmill, we also use 8 clients in order to compare against Mutilate with the same amount of resource usage. Table II shows the hardware specifications of the system under test, which is used for all experiments in this paper.

<table>
<thead>
<tr>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>DRAM</td>
</tr>
<tr>
<td>Ethernet</td>
</tr>
<tr>
<td>Kernel Version</td>
</tr>
</tbody>
</table>

When setting up these load testers, we also start a tcpdump process on the load test machines to measure the
ground truth latency distribution. Tcprof provides a good ground truth measurement because it measures the latency at network-level, thus eliminates potential client-side queueing delay. The tcprof process is pinned on an idle physical core to avoid possible probe effect.

Tcprof records the timestamps that request and response packets flow through the network interface card (NIC). By matching the sequence IDs of the packets, we can map each request to its corresponding response and calculate the time difference between the two timestamps as the ground truth latency in our evaluation. However, this ground truth latency is expected to be lower than the one that the load testers measure, because the timestamps tcprof reports are taken when the network packets arrive the NIC. Certain amount of time is spent in kernel space to handle the network interrupts before the packets reach the user code, where these load testers reside.

1) Measurement under 10% Utilization: In the first experiment, we use the three load testers to send 100k requests per second (RPS) to the Memcached server. This translates to 10% CPU utilization on the server. We modified all three load testers to report the entire latency distribution at the end of execution as shown in Figure 5, in which the tcprof curve shows the latency distribution measured by tcprof as ground truth. The 99th-percentile latency measured by each tool is plotted in the figure with the dashed line.

As shown in Figure 5, CloudSuite measures a drastically higher tail latency (99th-percentile latency is higher than 250µs thus not shown in the figure), because of heavy client-side queueing bias. This is due to the fact that it runs a single client to generate the load. Although Mutilate leverages 8 clients, it still fails to capture the shape of the ground truth latency distribution, and overestimates the tail latency. Due to the improper query inter-arrival generation, we notice that the ground truth latency distribution measured in Mutilate experiment is different from the ones measured in CloudSuite and Treadmill experiments. In contrast, Treadmill precisely captures the shape of the ground truth latency measurement, thus achieves highly accurate measurements. There is a fixed offset between the tcprof and Treadmill curves, due to the expected computation spent in kernel space for interrupt handling.

2) Measurement under 80% Utilization: Similarly, we construct another experiment with these three load testers to send 800k requests per second (RPS) to one Memcached server, which runs at 80% CPU utilization. In this experiment, we find that CloudSuite is not efficient enough to saturate the server at such high utilization because it runs a single client, thus not shown in the figure. As we illustrated in Figure 1, Mutilate, which runs a closed-loop controller, limits the maximum number of outstanding requests thus underestimates the ground truth tail latency. However, Treadmill is still able to precisely measure even the high percentile latency, and the expected gap between Treadmill and tcprof remains the same (30µs) as during low utilization shown in Figure 5.
Open-loop controller does a much better job properly exercising the queueing behavior of the system, because the number of outstanding requests is not limited, which reassembles a realistic setting in production environment. Although the implementation of Mutilate overestimates the tail latency from the “ground truth”, the 99th-percentile latency measured by Mutilate is still underestimated. Note that Treadmill still maintains a fixed offset to the ground truth latency distribution measured by tcpdump, and the offset is exactly the same (30μs) as during low utilization shown in Figure 5.

In conclusion, CloudSuite suffers from heavy client-side queueing bias, and Mutilate cannot properly exercise the queueing behavior of the system due to the closed-loop controller, whereas Treadmill precisely measures the tail latency even at high utilization.

IV. TAIL LATENCY ATTRIBUTION

With sufficient amount of samples, Treadmill is able to obtain accurate latency measurements even at high quantiles. However, we sometimes find the measured latency from each run converges to a different value as shown in Figure 4. This suggests that the systems or the states of the system we measure is changing across runs, which may also happen in production environment if we do not have a technique to carefully control it. In this section, we analyze this variance of tail latency using a recently developed statistical inference technique, quantile regression [26], and attribute the source of variance to various factors.

A. Quantile Regression

To analyze the observed variance in a response variable, analysis of variance (ANOVA) is often used to partition the variance and attribute it to different explanatory variables. However, the classic ANOVA technique assumes normally distributed residuals and equality of variances, which has been demonstrated unsuitable by prior work [47] for many computer system problems due to the common presence of non-normally distributed data. In addition, ANOVA can only attribute the variance of the sample means. In contrast, quantile regression [26] is a technique proposed to attribute the impact of various factors on any given quantiles, which does not make any assumption on the distribution of the underlying data. Therefore, quantile regression is particularly suitable for our purpose of analyzing the sources that contribute to the tail latency.

Similarly to ANOVA, quantile regression takes a number of samples as its input, where each sample includes a set of explanatory variables $x_i$ and a response variable $y$. The response variable is expected to vary depending on the explanatory variables. Quantile regression produces estimates of coefficients $c_i$ that minimizes the prediction error on a particular quantile $\tau$ for given $X$ as shown in Equation 1. In addition to individual explanatory variables, it can also model the interactions among them by including their products (e.g., $c_{12}(\tau)x_1x_2$ in Equation 1). It uses numerical optimization algorithm to minimize a loss function, which assigns a weight $\tau$ to underestimated errors and $(1-\tau)$ to overestimated ones for $\tau$-th quantile, instead of minimizing the squared error in ANOVA.

$$Q_y(\tau|X) = c_0(\tau) + c_1(\tau)x_1 + c_2(\tau)x_2 + \cdots + c_{12}(\tau)x_1x_2 + c_{13}(\tau)x_1x_3 + \cdots + (1)$$

In this case, we design the response variable to be a particular quantile (e.g., 99th-percentile) of the latency distribution and the explanatory variables to be a set of factors that we suspect to have impact on the latency distribution.

B. Factor Selection

First of all, we need to identify the potential factors that may affect the tail latency. We list all the factors we suspect to have an impact on the tail latency. Then we use null hypothesis testing on a large number of samples collected from repeated experiments under random permutations of all the factors, to identify the factors that actually have impact on the tail latency. Although the factors may vary depending on the workload and the experimental environment, we find a list of factors consistently affecting the tail latency across various workloads we have experimented with.

- **NUMA Control**: The control policy for non-uniform memory access (NUMA), which determines the memory node(s) to allocate for data. The same-node policy prefers to allocate memory on the same node until it cannot be allocated anymore, whereas the interleave policy uses round robin among all nodes.
- **Turbo Boost**: Frequency up-scaling feature is implemented on many modern processors, where the frequency headroom heavily depends on the dynamic power and thermal status. The scaling management algorithm is implemented in processor’s hardware power control unit, and it is not clear how it will impact the tail latency quantitatively.
- **DVFS Governor**: Dynamic voltage frequency scaling allows the operating system to up-scale the CPU frequency to boost performance and down-scale to save power dynamically. In this section, we study two commonly used governors including performance (always operating at the highest frequency) and ondemand (scaling up the frequency only when utilization is high).
- **NIC Affinity**: The hardware network interface card (NIC) uses receive side scaling (RSS) to route the network packets to different cores for interrupt handling. The routing algorithm is usually implemented through a hashing function computed from the packet header. For example, the NIC on our machine (shown in Table II) provides a 4-bit hashing value, which limits the number of interrupt queues to $2^4 = 16$. We study the impact of
mapping all the interrupt queues to cores on the same
CPU socket, and evenly spread across the two sockets.

Therefore, we use a 2-level full factorial experiment
design with the 4 factors listed above as shown in Table III.

In addition to the 4 factors listed above in isolation, we
also model the interactions among combinations of them,
because they might not necessarily be independent from
each other. For example, the impact of DVFS governor may
depend on Turbo Boost because they can indirectly interact
with each other due to the contention in thermal headroom.

C. Quantifying Goodness-of-fit

In ANOVA, coefficient of determination $R^2$ is often used
to quantify the fraction of variance that the model is able
to explain. However, an equivalent of $R^2$ does not exist for
quantile regression. Therefore, we define a pseudo $R^2$ metric
in Equation 2 using the same idea. The metric falls in the
range of $[0, 1]$, where 1 means the model perfectly predicts
the given quantile and 0 means its accuracy is the same as
the best constant model that always predicts the same value
regardless of the explanatory variables. In the equation, the
numerator represents the sum of the prediction errors of the
quantile regression model, and the denominator is the error
of the best constant model.

\[
\text{pseudo-} R^2 = 1 - \frac{\sum_{i=0}^{N} w(\tau, err^\tau(i))|err^\tau(i)|}{\sum_{i=0}^{N} w(\tau, err^\tau_{\text{const}}(i))|err^\tau_{\text{const}}(i)|} \tag{2}
\]

For each sample, the prediction error is computed as the
product of the absolute prediction error and a weight. The
prediction error for sample $i$ at $\tau$-th quantile is defined in
Equation 3 as the difference between empirically measured
quantile $y^\tau_i$ and the predicted quantile $model^\tau(X_i)$
conditional on explanatory variables $X_i$.

\[
\text{err}_{\text{model}}(i) = y^\tau_i - model^\tau(X_i) \tag{3}
\]

The weight assigned to each error is defined in Equation 4
as $(1 - \tau)$ for overestimation and $\tau$ for underestimation,
which is the same as the loss function in quantile regression.

\[
w(\tau, err) = \begin{cases} (1 - \tau) & : err < 0 \\ \tau & : err \geq 0 \end{cases} \tag{4}
\]

V. Evaluation of Hardware Features

The precision of tail latency measurements achieved by
Treadmill enables the possibility of understanding and
attributing the sources of tail latency variance. In this
section, we present the complex and counter-intuitive performance
behaviors identified through attributing the source of
tail latency, and demonstrate the effectiveness of our
methodology in improving tail latency.

A. Experimental Setup

We leverage quantile regression to analyze the measure-
ments obtained under various configurations presented in
Table III to understand the sources of tail latency variance.

To perform quantile regression, we first obtain latency samples under various configurations. We randomly choose
one permutation of the configurations for each experiment to
preserve independence among experiments, until we have at
least 30 experiments for each permutation of configurations.
Given we are studying 4 factors, we will need at least $2^4 \times
30 = 480$ experiments. For each experiment, we randomly
sub-sample 20k latency samples during the time when the
latency distribution has already converged. We make sure
this sub-sampling does not hurt the precision of the analysis
by comparing against a model obtained using more samples,
and we observe no significant difference.

Each factor is coded as 0 at low-level, and 1 at high-level
in the samples. Before feeding the data into the quantile
regression model, we perturb the data using a symmetric
variance at 0.01 standard deviation. This is useful to pre-
vent the numerical optimizer from getting trapped in local
optimal, because all explanatory variables are discrete values
(i.e., dummy variables). The perturbation is small enough
that it does not affect the quality of the regression itself.

B. Memcached Result

Table IV shows the result from quantile regression for
different percentiles, including 50th-, 95th- and 99th,- for Memcached workload at 70% server utilization. For each
centile, $Est.$ shows the estimated coefficient (i.e., $c_i(\tau)$
in Equation 1) of each factor, where negative value means
turning the factor to high-level reduces the corresponding
quantile latency. To estimate the quantile latency for a
given hardware configuration, one needs to add up all
the qualified estimated coefficients ($Est.$ in the table) and
the intercept. For example, to estimate the 95th-percentile
latency for a configuration that only “numa” and “turbo”
are turned to high-level, one needs to add their coefficients
of them in isolation $(24 + 12 = 12\mu s)$, and their interaction
“numa:turbo” $(5\mu s)$, and the intercept $(155\mu s)$, therefore
the estimated 95th-percentile latency is $12 + 5 + 155 = 172\mu s$. Std. Err is the estimated standard error for the
coefficient estimation at 95% confidence interval. \textit{P-value}
is the standard p-value for null hypothesis test, which is the
probability of obtaining a result equal or more extreme than
the observed one. A p-value smaller than 0.05 is usually
considered as strong presumption against null hypothesis,
which suggests the corresponding factor has a significant
impact on the percentile latency.
Results of quantile regression for Memcached at high utilization, which detail the contribution of each feature on the latency. The first several rows show the base latency ( Intercept ) and the latency of each feature enabled in isolation. The other rows provide the interaction latency effect of multiple features. For example, “ turbo ” is the best single feature ( -29µs ) in isolation to turn on to reduce 99th-percentile latency. Turning “ nic ” high-level is only beneficial if “ dvfs ” is set to high-level ( 29 + 8 + 58 = -37µs ), otherwise the net effect would be an latency increase ( 29µs ). Surprisingly, setting “ turbo ” on, which is beneficial in isolation, in addition to “ nic ” and “ dvfs ” would actually increase the 99th-percentile latency ( -29 + 8 + 58 + 23 + 58 + 4 = 1µs ) due to the negative interaction among them. Note that for some rows, the uncertainty in the data is significant, and we choose a p-value of 0.05 and highlight these values in bold.

To summarize the result, Figure 7 shows the estimated latency of Memcached at various percentiles under low utilization and high utilization using the result from quantile regression. This is similar to what we observe in a M/M/1 queueing model [48], that the variance of number of outstanding requests \( \rho (1-\rho) \), where \( \rho \) is the server utilization, grows as the utilization increases.

- **Finding 2.** The variance of latency increases from lower to higher quantile as suggested by the growing standard error shown in Table IV, because the variance of a quantile is inversely proportional to the density [49]. This also explains the reason why we observe many statistically insignificant impact ( p-value > 0.05 ) and the uncertainty is high at high quantiles.

- **Finding 3.** The latency could be higher at lower utilization when the DVFS governor is turned to ondemand policy, because of frequent transitions among frequency steps. The 50th- and 90th-percentile latencies are higher during low load than high load under ondemand DVFS governor. This is because requests have a higher probability of experiencing the overhead of transitioning from lower to higher frequency steps, whereas the CPU is kept at high frequency during high load and does not need many transitions.
Finding 4. Turning NIC affinity policy from same-node to all-nodes during low load can significantly reduce the latency when DVFS governor is set to ondemand. The cores have larger utilization range under same-node policy than all-nodes, which leads to higher probability of experiencing frequency step transitions. This does not occur at high load because the utilization is already high enough that the number of frequency transitions is negligible. Prior study performed on the same hardware factors in isolation fails to capture such interacting behaviors among multiple factors due to the limitation of isolated study.

Finding 5. As shown in Table IV, the interactions among factors are demonstrated to have statistically significant impact on the latency as many of them have a p-value smaller than 0.05. In addition, the estimated coefficients of interactions are sometimes larger than individual factors, which means the interactions among factors can have higher impact on the latency. For example, turning NUMA control policy to interleave increases the 99th-percentile latency by 56µs as shown in the table, but its positive interaction with performance DVFS governor results in a 9µs improvement. These interacting behaviors are complicated and sometimes counter-intuitive, and cannot be captured by isolated studies of individual factors. Therefore, it is necessary to use statistical techniques like quantile regression to model the interactions.

Due to the interactions among factors, we cannot simply decompose the variance of tail to each individual factor. However, by assuming all the other factors are randomly selected (i.e., each factor has equal probability of being low-level and high-level), we can quantify the impact of each factor on average case as shown in Figure 8.

Finding 6. Interleaved NUMA control policy increases the latency by up to 44µs especially during high load. This is caused by bad connection buffers allocation that majority of the server threads have their connection buffers allocated on the remote memory node, while same-node policy guarantees half of the threads get their buffers allocated on the local node. We only observe behavior during high load because the high queueing delay magnifies the overhead of accessing remote memory node.

Finding 7. The amount of impact each factor contributes varies depending on the load levels. For example, DVFS governor has the highest impact at low load, whereas NUMA policy is the biggest contributor to the variance at high load. This is caused by the complex interacting behavior among different features, which again, is not captured by isolated studies in prior works [17].

C. Mcrouter Results

Similarly, we also construct experiments with mcrouter workload as shown in Figure 7, which is a configurable protocol router that turns individual cache servers into massivescale distributed systems. Figure 10 shows the average impact of the 4 factors assuming other factors are selected randomly with equal probability.

Finding 8. Turbo Boost significantly improves the latency especially during low load for mcrouter. This is because a large fraction of the computation mcrouter needs to do is to deserialize the request structure from network packet, which is CPU-intensive and can easily be accelerated by frequency up-scaling. However, this difference is much smaller, sometimes statistically insignificant, during high load, because the available thermal headroom is smaller comparing to low load.
D. Quantifying Goodness-of-fit

Although the low p-values obtained from quantile regression suggest high confidence that the studied factors have significant impact on the tail latency, it is also possible that they only contribute to a small fraction of the total variance. Therefore, we quantify the goodness-of-fit in this section, which demonstrates that our model covers the majority of observed variance.

Figure 11 shows the pseudo-$R^2$ values, we previously defined in Equation 2, of the quantile regression models at various percentiles, which quantifies the variance that can be explained. Our models have consistently high pseudo-$R^2$ values (the lowest one is 0.9), which suggests that they are able to explain the majority of the observed variance.

E. Improving Tail Latency

We further evaluate the quantile regression results in Figure 12, in which we perform same experiment 100 times using randomly selected configurations as “before”, and compare against best configuration for 99th-percentile latency recommended by our quantile regression model as “after”. As we can see from the figure, both latency and the variance of latency have been significantly reduced. Specifically, the expected 50th-percentile latency has been reduced from 69\(\mu\)s to 62\(\mu\)s, and the standard deviation has been reduced from 13\(\mu\)s to 5\(\mu\)s. The expected 99th-percentile latency has been reduced from 181\(\mu\)s to 103\(\mu\)s, and the standard deviation has been reduced from 78\(\mu\)s to 5\(\mu\)s. The reductions we achieve on 99th-percentile latency are much larger than on 50th-percentile, because we optimize for 99th-percentile when choosing the best configurations.

VI. Related Work

There has been large amount of work on developing statistically sound performance evaluation methodology. Mytkowicz et al. [50] show the significance of measurement bias commonly existing in computer system research, and present a list of experimental techniques to avoid the bias. Oliveira et al. [47] present a study on two Linux Schedulers using statistical methods, which demonstrates that ANOVA can sometimes be insufficient especially for non-normally distributed data whereas quantile regression can provide more conclusive insights. Curtisinger and Berger propose STABILIZER [33], which randomizes the layouts of code, stack and heap objects at runtime to eliminate the measurement bias caused by layout effects in performance evaluation. Alameldeen and Wood [51] leverage confidence interval and hypothesis testing to compensate the variability they discover in architectural simulations for multi-threaded workloads. Tsafir et al. [52,53] develop input shaming technique to address the environmental sensitivities they observe in parallel job scheduling simulations. Georges et al. [54] point out a list of pitfalls in existing Java performance evaluation methodologies, and propose JavaStats to perform rigorous Java performance analysis. Breughe and Eeckhout [55] point out benchmark inputs are critical for rigorous evaluation on microprocessor designs.

Others have developed benchmark suites that capture the representative workloads in modern warehouse scale computers. Cooper et al. [4] present the Yahoo! Cloud Serving Benchmark (YCSB) framework for benchmarking large-scale distributed data serving applications. Fan et al. [5] present and characterize 3 types of representative workload in Google data centers, including web search, web mail and MapReduce. In addition, Lim et al. [7] further characterize the video streaming workload at Google and benchmark them to evaluate new server architectures. Ferdman et al. [6] introduce the CloudSuite benchmark suite, which represents the emerging scale-out workloads running in modern data centers. Wang et al. [8] enrich the data center workload benchmark by presenting BigDataBench, which covers diverse cloud applications together with representative input data sets. Hauswald et al. [9,10] introduce benchmarks of emerging machine learning data center applications. Meisner et al. [56] present a data center simulation infrastructure, BigHouse, that can be used to study data center workloads.

VII. Conclusion

In this paper, we identify four common pitfalls through in-depth survey of existing tail latency measurement methodologies. To overcome these pitfalls, we design a robust procedure for accurate tail latency measurement, which leverages Treadmill, a modular software load tester we
develop. Using the superior measurements achieved by this procedure, we leverage quantile regression to analyze, and attribute the sources of variance in tail latency to various hardware features of interest. Using the knowledge we gain from the attribution, we reduce the 99th-percentile latency by 43% and its variance by 93%.

ACKNOWLEDGMENT

We would like to thank Andrii Grynenko, Anton Likhtarov, Pavlo Kushnir, Stepan Palamarchuk, Dave Harrington, Sara Golemon, Dave Watson, Adam Simpkins, Hans Fugal, Tony Savor, and Edward D. Rothman for their contributions. We would also like to thank the anonymous reviewers, Thomas Wenisch for their valuable feedback on earlier drafts of this paper. This work was supported by Facebook and XXX.

REFERENCES


