2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Treadmill: Attributing the Source of 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 the deployed software and production hardware configurations. 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 and performance factor attribution for server workloads. First, we find that careful design of the server load tester can ensure high quality performance evaluation, and empirically demonstrate the inaccuracy of load testers in previous work. Learning from the design flaws in prior work, we design and develop a modular load tester platform, `Treadmill`, that overcomes pitfalls of existing tools. Next, utilizing `Treadmill`, we construct measurement and analysis procedures that can properly attribute performance factors. We rely on 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 impact 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) improves the 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 the production system is important for a number of reasons. First, servers are typically acquired in large quantities (*e.g.*, 1000s at a time), so it is important to choose the best design possible and carefully provision resources. Evaluating hardware configurations requires extensive and accurate measurements against exising workloads. 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, because it 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.*, distributed server-side software, 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 several pitfalls in their load test design as we will show later in the paper. Unfortunately, these tools are commonly used in research publications for evaluation and the pitfalls may result in misleading conclusions. Similar to the academic community, there is also a lack of an accurate tail latency measurement test bed in industry, causing unnecessary resource over-provisioning [11] and unexplained performance regressions [12].

Furthermore, to be able to control the tail latency of these Internet services, a thorough and correct understanding of the source of tail latency is required. These Internet services interact with a wide range of systems and resources including operating system, network stack and server hardware thus the ability of quantitatively attributing the source of tail latency to individual resources is critical yet challenging. 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 resources on the tail latency, many resources have complex interacting behaviors that cannot be captured in isolated studies. For example, Turbo Boost and DVFS governor may interact indirectly through competing for the thermal headroom. Note that the capability of identifying the source of tail latency relies on the first aforementioned challenge. In other words, without

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.47

IEEE computer society

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 tester 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) fails to capture detailed information; static histograms used in other load testers also exhibit bias.
- **Client-side queueing bias** - Due to the 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 be impacted by client-side queueing, generating biased results.
- **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 tail latency to various hardware features that cause the tail. 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 in isolation.

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 systems 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], Mutilate [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 I
Summary of load tester features.

	YCSB	Faban	CloudSuite	Mutilate	Treadmill
Query Interarrival Generation			✓		✓
Statistical Aggregation				✓	✓
Client-side Queueing Bias		✓		✓	✓
Performance Hysteresis					✓
Generality	✓	✓			✓

A. Query Inter-arrival Generation

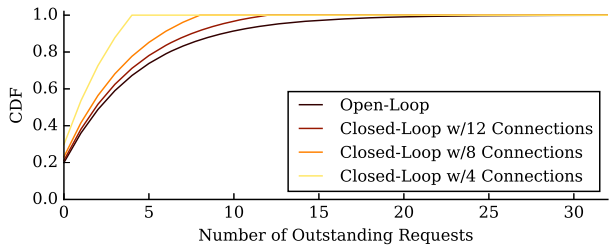

Figure 1. Comparison of the number of outstanding requests between closed-loop and open-loop controllers. The "Open-Loop" line shows the cumulative-distribution of the number of outstanding requests in an open-loop controlled system when running at 80% utilization. The "Closed-Loop" lines show the distribution 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 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 two 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.

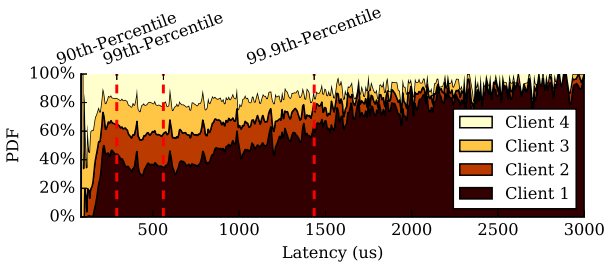

Figure 2. Different latency distributions measured from multiple clients in the a multi-client performance evaluation procedure, where the y-axis shows the decomposition among clients as what percent of samples is contributed by each client. We can see from the figure that Client 1 dominates the high 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 aggregations 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, each shaded region represents 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.

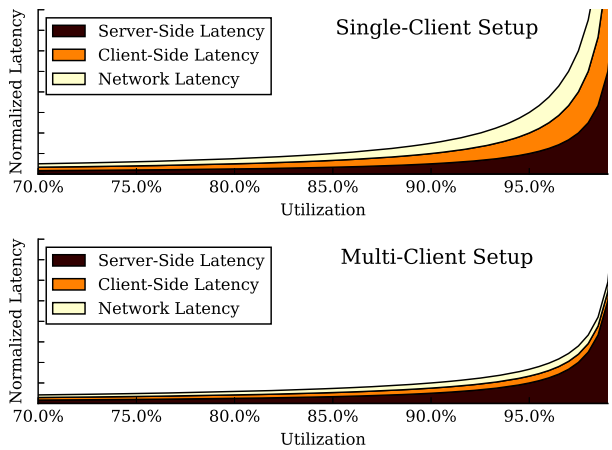

Figure 3. Comparison between single-client and multi-client setups for measuring request latency. In the single-client setup, the network and the client have the same utilization as the server, which results in increasing queueing delay when the server utilization increases. This will bias the latency 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.

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 time (*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. As one can see, the client-side latency and the network latency grow as the server utilization increases, and represent a significant fraction of the end-to-end latency.

We find that it is extremely challenging, if not impossible, to design a single-client load tester that can fully saturate the server without incurring significant client-side queueing for modern data center workloads that operate at microsecond-level latency. Instead, it is necessary to have a multi-client setup and have a sufficient number of machines such that the client-side and network latency is kept low. In "Multi-Client Setup", we increase the number of client machines to minimize these biases. After this adjustment the majority of measured latency comes from the server.

D. Performance "Hysteresis"

Through experimentation we find an usual 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 is restarted and another run is performed, the estimate can converge to a different value.

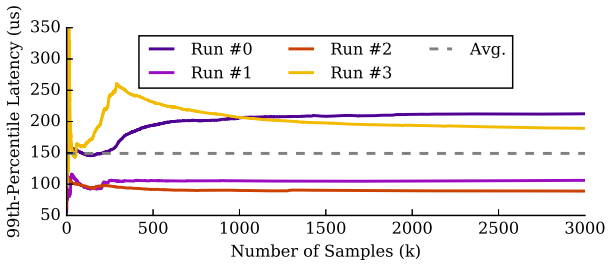

Figure 4. Variance exists regardless of a single run's sample size. A single run exhibits a large variance for a small sample size (*i.e.*, early in the run). With a sufficiently large sample size the estimate of 99th-percentile latency converges. However, empirically we find that each run can converge to a *different* value. Although the testing procedure of each run would yield a tight confidence interval, the result of each run clearly varies significantly (15-67% variation from the average).

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 the 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 minimizes the amount of effort required to extend `Treadmill` to new workloads. So far, we have successfully integrated `Treadmill` with several services including Memcached [35] and mcrouter [27]. Each integration takes less than 200 lines of code.
- **Configurable workload:** It has been demonstrated in prior work [36] 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 (*e.g.*, request size distribution) and fed 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 develop 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, multiple measurements are taken by repeating the same experiment multiple times. After each experiment, we record the collected measurements 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 [37] due to its popularity in both industry [38,35] and academia [39,40,41,42,43,44], 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 the experiments in this paper.

Table II
Hardware specification of the system under test.

	Specification
Processor	Intel Xeon E5-2660 v2
DRAM	144GB @ 1333MHz
Ethernet	10GbE Mellanox MT27500 ConnectX-3
Kernel Version	3.10

When setting up these load testers, we also start a tcpdump process on the load test machines to measure the

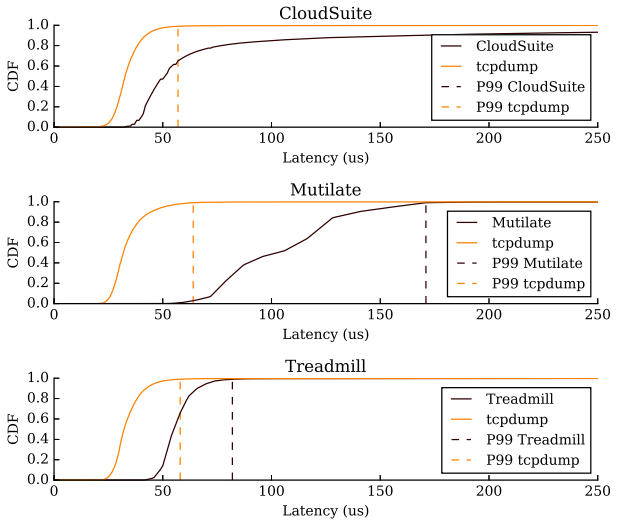

Figure 5. Latency distributions measured by CloudSuite, Mutilate and `Treadmill` at 10% server utilization, in which only `Treadmill` accurately captures the ground truth distribution measured by tcpdump. CloudSuite heavily overestimates the tail latency due to client-side queueing delay, and Mutilate also overestimates the tail latency and fail to capture the shape of the ground truth distribution. In contrast, `Treadmill` precisely captures the shape of ground truth distribution and maintains a constant gap to the tcpdump curve even at high quantiles. Note the gap between tcpdump measurement and load tester measurement is *expected* due to kernel space interrupt handling as we explain in the experimental setup.

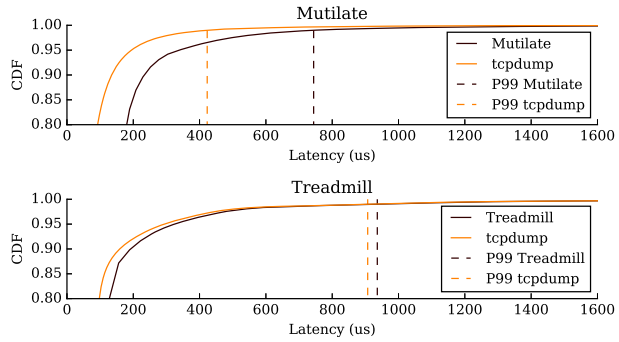

Figure 6. Latency distributions measured by Mutilate and `Treadmill` at 80% CPU utilization, where the ground truth tail latency measured by tcpdump is underestimated in the Mutilate experiment due to closed-loop controller. CloudSuite is not efficient enough to saturate the server at such high utilization 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 tcpdump remains the same (30μs) as during low utilization shown in Figure 5.

ground truth latency distribution. Tcpdump provides a good ground truth measurement because it measures the latency at network-level, thus eliminates potential client-side queueing delay. The tcpdump process is pinned on an idle physical core to avoid possible probe effect.

Tcpdump 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 tcpdump 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 tcpdump curve shows the latency distribution measured by tcpdump 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 distribution, thus achieves highly accurate measurements. There is a fixed offset between the tcpdump 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 send this many requests because of the performance limitation of a single client; we only report the measurements from Mutilate and `Treadmill` in Figure 6.

Similar to the previous experiment, the measured ground truth latency distributions from Mutilate experiment and `Treadmill` experiment are drastically different, especially at high quantile. This is due to the fact that tcpdump measures the ground truth driven by the control loop of the load tester. Mutilate runs a closed-loop controller, which artificially limits the maximum number of outstanding requests as we illustrated in Figure 1, therefore heavily underestimates the 99th-percentile latency by more than

2×. 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 are 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 the 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 have been demonstrated unsuitable by prior work [45] 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 τ 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 τ to underestimated errors and $(1-\tau)$ to overestimated ones for τ-th quantile, instead of minimizing the squared error in ANOVA.

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

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 an 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 an 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) 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

Table III
Quantile regression factors.

Factor	Low-Level	High-Level
NUMA Control (numa)	same-node	interleave
Turbo Boost (turbo)	off	on
DVFS Governor (dvfs)	ondemand	performance
NIC Affinity (nic)	same-node	all-nodes

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.

$$pseudo\text{–}R^2_\tau = 1 - \frac{\sum_{i=0}^{N} w(\tau, err^\tau_{qr}(i))|err^\tau_{qr}(i)|}{\sum_{i=0}^{N} w(\tau, err^\tau_{const}(i))|err^\tau_{const}(i)|} \quad (2)$$

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 τ-th quantile is defined in Equation 3 as the difference between empirically measured quantile y^τ_i and the predicted quantile $model^\tau(X_i)$ conditional on explanatory variables X_i.

$$err^\tau_{model}(i) = y^\tau_i - model^\tau(X_i) \quad (3)$$

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

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

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 measurements 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 the 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 prevent 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 percentile, *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 = 12μs), and their interaction "numa:turbo" (5μs), and the intercept (155μs), therefore the estimated 95th-percentile latency is 12 + 5 + 155 = 172μs. *Std. Err* is the estimated standard error for the coefficient estimation at 95% confidence interval. *P-value* is the standard p-value for null hypothesis testing, 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.

Table IV

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" to 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 + 29 + 40 + 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 to highlight these values in bold.

	50th-Percentile			95th-Percentile			99th-Percentile		
Factor	Est.	Std. Err	p-value	Est.	Std. Err.	p-value	Est.	Std. Err	p-value
(Intercept)	65 μs	<1 μs	<1e-06	155 μs	<1 μs	<1e-06	355 μs	5 μs	<1e-06
numa	2 μs	<1 μs	<1e-06	24 μs	<1 μs	<1e-06	56 μs	8 μs	<1e-06
turbo	-2 μs	<1 μs	<1e-06	-12 μs	<1 μs	<1e-06	-29 μs	7 μs	1.00e-04
dvfs	1 μs	<1 μs	<1e-06	<1 μs	<1 μs	**2.60e-01**	-8 μs	8 μs	**3.54e-01**
nic	<1 μs	<1 μs	<1e-06	2 μs	<1 μs	2.07e-03	29 μs	8 μs	1.10e-04
numa:turbo	3 μs	<1 μs	<1e-06	5 μs	1 μs	2.40e-04	21 μs	11 μs	**6.37e-02**
numa:dvfs	-3 μs	<1 μs	<1e-06	-29 μs	1 μs	<1e-06	-57 μs	11 μs	<1e-06
numa:nic	<1 μs	<1 μs	<1e-06	-6 μs	1 μs	4.00e-05	-20 μs	11 μs	**6.91e-02**
turbo:dvfs	<1 μs	<1 μs	<1e-06	14 μs	1 μs	<1e-06	40 μs	11 μs	3.30e-04
turbo:nic	3 μs	<1 μs	<1e-06	23 μs	1 μs	<1e-06	23 μs	10 μs	2.90e-02
dvfs:nic	-1 μs	<1 μs	<1e-06	-15 μs	1 μs	<1e-06	-58 μs	11 μs	<1e-06
numa:turbo:dvfs	2 μs	<1 μs	<1e-06	12 μs	2 μs	<1e-06	3 μs	16 μs	**8.70e-01**
numa:turbo:nic	<1 μs	<1 μs	**6.25e-01**	-7 μs	2 μs	8.00e-05	-14 μs	15 μs	**3.59e-01**
numa:dvfs:nic	3 μs	<1 μs	<1e-06	34 μs	2 μs	<1e-06	79 μs	15 μs	<1e-06
turbo:dvfs:nic	<1 μs	<1 μs	**7.66e-01**	-9 μs	2 μs	<1e-06	4 μs	14 μs	**7.96e-01**
numa:turbo:dvfs:nic	-8 μs	<1 μs	<1e-06	-43 μs	3 μs	<1e-06	-83 μs	23 μs	2.50e-04

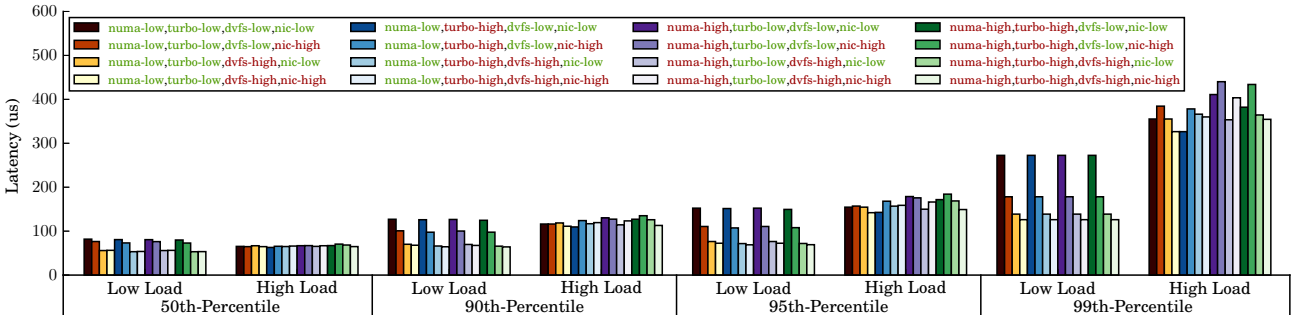

Figure 7. Estimated latency of Memcached at various percentiles under low utilization and high utilization using the result from quantile regression.

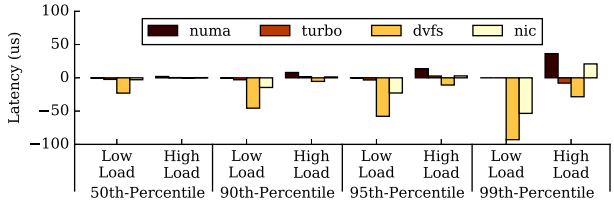

Figure 8. The average impact in latency of turning each individual factor to high-level for Memcached, assuming each of the other factors have equal probability of being low-level and high-level. Negative latency means latency reduction, and positive number means latency increase.

To summarize the result, Figure 7 shows the estimated latency of all factor permutations at various percentiles under low and high server utilization correspondingly. From the result, we have several findings as follows:

- **Finding 1.** *The variance of latency increases from lower to higher server utilization, because of the increasing variance in number of outstanding requests.* This is similar to what we observe in a M/M/1 queueing model [46], that the variance of number of outstanding requests $\frac{\rho}{(1-\rho)^2}$, where ρ 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 [47].* This also explains the reason why we observe many statistically insignificant cases (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.

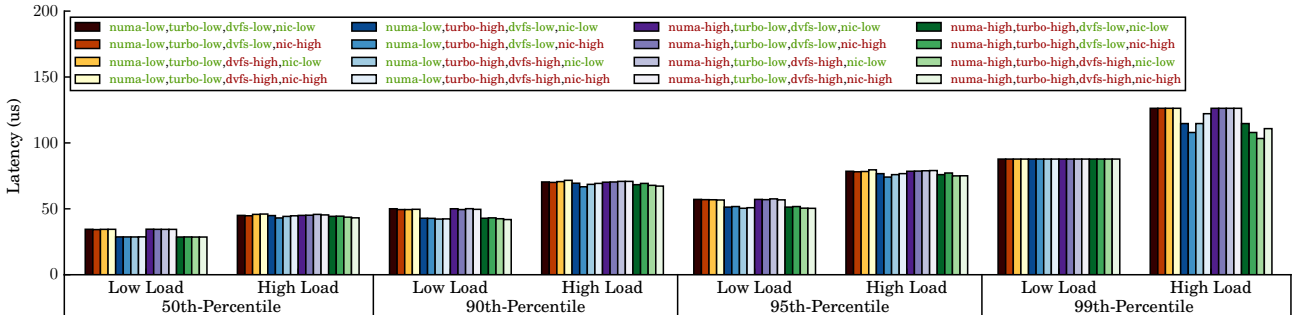

Figure 9. Estimated latency of mcrouter at various percentiles under low utilization and high utilization using the result from quantile regression.

- **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 the 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 this behavior during high load because the high queueing delay magnifies the overhead of accessing remote memory node.

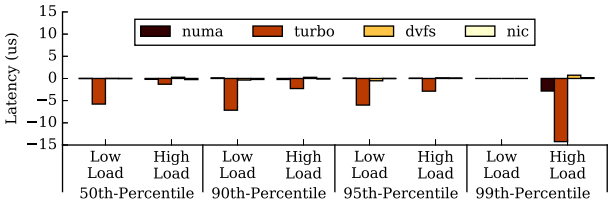

Figure 10. The average impact in latency of turning each individual factor to high-level for mcrouter, assuming each of the other factors have equal probability of being low-level and high-level. Negative latency means latency reduction, and positive number means latency increase.

- **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.

C. Mcrouter Results

Similarly, we also construct experiments with mcrouter workload as shown in Figure 9, which is a configurable protocol router that turns individual cache servers into massive-scale 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 packets, 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 compared to low load.

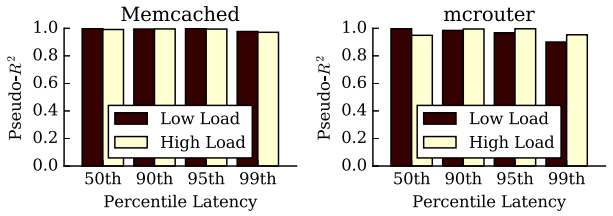

Figure 11. Pseudo-R^2 (shown in Equation 2) of the quantile regression results at various load levels and percentiles, which demonstrates good coverage of sources of variance. Pseudo-R^2 quantifies the goodness-of-fit of the model, which ranges in [0, 1] that higher value indicates better model fit. Our regression models show high pseudo-R^2 values (>0.90), which suggests that they are able to explain majority of the variance.

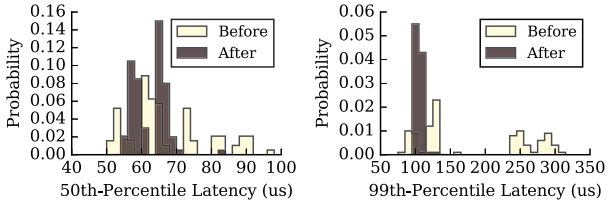

Figure 12. Using the knowledge we gain from quantile regression, both the latency and the variance of latency are significantly reduced after carefully controlling the factors contributed to the variance. The average 99th-percentile latency in 100 experiments is reduced from 181μs to 103μs, and the standard deviation is reduced from 78μs to 5μs.

D. Quantifying Goodness-of-fit

Although the low p-values obtained from quantile regression suggests 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 the 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 the same experiment 100 times using randomly selected configurations as "before", and compare against the 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 69μs to 62μs, and the standard deviation has been reduced from 13μs to 5μs. The expected 99th-percentile latency has been reduced from 181μs to 103μs, and the standard deviation has been reduced from 78μs to 5μs. 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 configuration.

VI. Related Work

There has been large amount of work on developing statistically sound performance evaluation methodology. Mytkowicz et al. [48] 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. [45] 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. Curtsinger 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 [49] leverage confidence interval and hypothesis testing to compensate the variability they discover in architectural simulations for multi-threaded workloads. Tsafrir et al. [50,51] develop input shaking technique to address the environmental sensitivities they observe in parallel job scheduling simulations. Georges et al. [52] point out a list of pitfalls in existing Java performance evaluation methodologies, and propose JavaStats to perform rigorous Java performance analysis. Breughe and Eeckhout [53] point out benchmark inputs are critical for rigorous evaluation on microprocessor designs.

There are also a number of prior works reducing the variance of query latency, and improving the tail latency. Shen [54] models the request-level behavior variation caused by resource contention, and proposes a contention-aware OS scheduling algorithm to reduce the tail latency.

Others have developed benchmark suites that capture the representative workloads in modern data centers. 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 workloads 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. [55] present a data center simulation infrastructure, BigHouse, that can be used to model data center workloads.

VII. Conclusion

In this paper, we identify four common pitfalls through an in-depth survey of existing tail latency measurement methodologies. To overcome these pitfalls, we design a robust procedure for accurate tail latency measurement, which uses `Treadmill`, a modular software load tester we develop. With 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 and Thomas Wenisch for their valuable feedback on earlier drafts of this paper. This work was supported by Facebook and by the National Science Foundation under grants CCF-SHF-1302682 and CNS-CSR-1321047.

References

[1] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011.

[2] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, 2013.

[3] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Proceedings of the Symposium on Cloud Computing (SoCC)*, 2014.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the Symposium on Cloud Computing (SoCC)*, 2010.

[5] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2007.

[6] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[7] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2008.

[8] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[9] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[10] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[11] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, 2013.

[12] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, 2013.

[13] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[14] ——, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[15] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks, "Tradeoffs between power management and tail latency in warehouse-scale applications," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2014.

[16] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict qos for latency-critical workloads," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[17] D. Lo and C. Kozyrakis, "Dynamic management of turbomode in modern multi-core chips," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[18] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceedings of the International Symposium on Computer Architecuture (ISCA)*, 2014.

[19] S. M. Zahedi and B. C. Lee, "Ref: Resource elasticity fairness with sharing incentives for multiprocessors," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[20] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.

[21] C. H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[22] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.

[23] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing google's warehouse scale computers: The numa experience," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[24] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2011.

[25] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011.

[26] R. Koenker, *Quantile regression*, 2005.

[27] A. Likhtarov, R. Nishtala, R. McElroy, H. Fugal, A. Grynenko, and V. Venkataramani, "Introducing mcrouter: A memcached protocol router for scaling memcached deployments," 2014.

[28] "Faban performance workload creation and execution framework," http://faban.org/.

[29] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2014.

[30] "Spec jenterprise," https://www.spec.org/jEnterprise2010/.

[31] "Spec sip-infrastructure 2011," https://www.spec.org/specsip/.

[32] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: A cautionary tale." in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2006.

[33] C. Curtsinger and E. D. Berger, "Stabilizer: Statistically sound performance evaluation," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[34] H. Fugal, "Wangle," https://github.com/facebook/wangle.

[35] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[36] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2012.

[37] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, 2004.

[38] C. Aniszczyk, "Caching with twemcache," https://blog.twitter.com/2012/caching-with-twemcache, 2012.

[39] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing." in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[40] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing soc accelerators for memcached," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.

[41] T. H. Hetherington, M. O'Connor, and T. M. Aamodt, "Memcachedgpu: Scaling-up scale-out key-value stores," in *Proceedings of the Symposium on Cloud Computing (SoCC)*, 2015.

[42] T. Hetherington, T. Rogers, L. Hsu, M. O'Connor, and T. Aamodt, "Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012.

[43] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[44] A. Gutierrez, M. Cieslak, B. Giridhar, R. G. Dreslinski, L. Ceze, and T. Mudge, "Integrated 3d-stacked server designs for increasing physical density of key-value stores," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[45] A. B. de Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Why you should care about quantile regression," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[46] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*, 2013.

[47] R. W. Keener, "Theoretical statistics: Topics for a core course," *International Statistical Review*, 2012.

[48] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[49] A. Alameldeen and D. Wood, "Variability in architectural simulations of multi-threaded workloads," in *High-Performance Computer Architecture (HPCA). Proceedings. The Ninth International Symposium on*, 2003.

[50] D. Tsafrir and D. Feitelson, "Instability in parallel job scheduling simulation: the role of workload flurries," in *Parallel and Distributed Processing Symposium (IPDPS). 20th International*, 2006.

[51] D. Tsafrir, K. Ouaknine, and D. Feitelson, "Reducing performance evaluation sensitivity and variability by input shaking," in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). 15th International Symposium on*, 2007.

[52] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the Annual Conference on Object-oriented Programming Systems and Applications (OOPSLA)*, 2007.

[53] M. B. Breughe and L. Eeckhout, "Selecting representative benchmark inputs for exploring microprocessor design spaces," *ACM Trans. Archit. Code Optim.*

[54] K. Shen, "Request behavior variations," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[55] D. Meisner, J. Wu, and T. F. Wenisch, "Bighouse: A simulation infrastructure for data center systems," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012.