Appears in the Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)

TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications

Harshad Kasture Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{harshad, sanchez}@csail.mit.edu

***Abstract*—Latency-critical applications, common in datacenters, must achieve small and predictable tail (e.g., 95^{th} or 99^{th} percentile) latencies. Their strict performance requirements limit utilization and efficiency in current datacenters. These problems have sparked research in hardware and software techniques that target tail latency. However, research in this area is hampered by the lack of a comprehensive suite of latency-critical benchmarks.**

We present TailBench, a benchmark suite and evaluation methodology that makes latency-critical workloads as easy to run and characterize as conventional, throughput-oriented ones. TailBench includes eight applications that span a wide range of latency requirements and domains, and a harness that implements a robust and statistically sound load-testing methodology. The modular design of the TailBench harness facilitates multiple load-testing scenarios, ranging from multi-node configurations that capture network overheads, to simplified single-node configurations that allow measuring tail latency in simulation. Validation results show that the simplified configurations are accurate for most applications. This flexibility enables rapid prototyping of hardware and software techniques for latency-critical workloads.

I. Introduction

Latency-critical applications are increasingly common in datacenters. These applications form the fabric of interactive, large-scale online services. *Tail latency*, not average latency, is the key performance metric for these applications. For example, web search leaf nodes must provide 99^{th} percentile latencies of a few milliseconds [17, 49]. The need for low tail latency presents new challenges and opportunities for system designers, as many hardware and software techniques in current systems seek to improve *long-term average performance*, but do not help or even hurt *short-term worst-case latency* [28, 32].

Unfortunately, the lack of a comprehensive suite of latency-critical benchmarks makes studying this emerging class of applications much harder than it should be. This difficulty causes two crucial problems. First, it hampers research that seeks to optimize systems for latency-critical applications. Latency-critical applications have a wide variety of latency requirements and microarchitectural characteristics. However, most recent work in this area uses one or a few latency-critical applications in their evaluations [25, 32, 33, 48], which do not stress a wide range of behaviors. Some prior work in this area even uses more readily-available sequential and parallel batch workloads (e.g., from SPEC CPU2006 or PARSEC [12]) and treats them as latency-critical applications [15, 57]. While this approach allows more diversity, it misses fundamental characteristics of latency-critical workloads (e.g., their request-response nature). Second, most new ideas in architecture and systems are evaluated with throughput-oriented applications only, not latency-critical ones, which constitutes a blind spot in the design of these techniques. For example, many cache partitioning techniques use coarse-grain, periodic reconfigurations to adapt to changing application behavior over time [10, 42]. While this helps long-term throughput, it can dramatically worsen tail latency [30]. Similarly, the profiling phases employed by many cache partitioning schemes [15] also hurt tail latency. Readily available latency-critical benchmarks can help researchers design techniques that do not inadvertently hurt tail latency, increasing their chances of adoption.

To tackle these problems, latency-critical workloads must be as easy to run and characterize as conventional, throughput-oriented ones. This is challenging for three reasons. First, since tail latency represents the few slowest requests (e.g., the slowest 1% requests when measuring the 99^{th} percentile latency), it is much more sensitive to small perturbations and requires a statistically robust methodology. Second, there are many methodological pitfalls that can skew latency measurements. As shown in recent work, even widely-used load testers suffer from some of these pitfalls, which often cause orders-of-magnitude measurement errors [44, 56]. Third, it is not enough for these workloads to run on real systems—to truly complement throughput-oriented benchmark suites, these workloads should also be easy to run in microarchitectural simulators, even those with limited system support.

We present *TailBench*, a new benchmark suite of latency-critical applications that addresses these challenges. TailBench includes a diverse set of latency-critical applications, as well as a robust, validated experimental methodology that makes it easy to run these benchmarks on real systems and in simulation. Specifically, we make the following key contributions:

- *We select eight representative latency-critical applications with a diverse set of characteristics (Sec. III).* TailBench applications span a wide range of domains, including web search, transactional databases, key-value stores, and real-time text, speech, and image processing. These applications cover a wide range of tail latencies (from microseconds to seconds), allowing designers to evaluate the impact of proposed techniques on tail latency at different timescales.
- *We integrate all workloads under a common harness that implements a robust, statistically sound methodology (Sec. IV).*

This methodology avoids the many pitfalls that afflict conventional load testers [56]. Additionally, we find that although network latency and kernel overheads are important contributors to tail latency in some applications, in many others tail latency is dominated by user-level application work. We use this insight to design multiple configurations of the TailBench harness that allow a range of load-testing scenarios: from full-blown multi-node configurations, to a simple single-node setup that can be easily simulated.

- *We validate the TailBench methodology in both real systems and simulation (Sec. VI).* We show that the simplified harness configurations faithfully measure tail latency for six of our eight benchmarks with significantly reduced measurement costs, and allow measuring tail latency directly in simulation.
- *We illustrate TailBench's benefits through a case study (Sec. VII).* We show that thread-level parallelism often accrues suboptimal tail latency benefits. We use a microarchitectural simulator to distinguish the effect of synchronization overheads from that of contention in the shared memory system.

II. Background

A. Anatomy of Latency-Critical Applications

Large-scale, interactive online services (e.g., web search) must mine through massive datasets to satisfy each request. These datasets are spread across hundreds or thousands of nodes, and are kept in DRAM or Flash to ensure fast response times. These workloads are architected in a high-fanout, multi-tiered configuration, with *root nodes* receiving user requests and farming them out to *leaf nodes* for processing. Thousands of leaf nodes may collaborate to serve each user request [9, 17, 32], and the latency perceived by the user is determined by the few slowest nodes, since the root node must wait for results from most or all leaf nodes to produce the final response. Thus, to ensure acceptable end-to-end latencies, the *tail latencies* (e.g., 95^{th} or 99^{th} percentile latencies) of leaf nodes should be small (e.g., a few milliseconds) and uniform across nodes.

The need for low, predictable tail latency limits the utilization and efficiency of conventional datacenter servers. Servers running latency-critical applications operate at low utilization to guard against queuing delays, long requests, and other sources of performance variability. Further, their spare capacity cannot be used by batch applications, as uncontrolled sharing of cores, caches, and power causes high and unpredictable tail latency degradation [30, 33, 36]. As a result, datacenters servers typically have utilizations of 5-30% [8, 9, 37]. This poor utilization wastes billions of dollars in equipment and terawatt-hours of energy annually [8].

Consequently, prior work has proposed a wide variety of software and hardware techniques to improve utilization and efficiency in systems running latency-critical applications without degrading latency. These techniques include new cluster managers that schedule and migrate applications across systems to reduce interference [18, 32, 36, 54], fast dynamic voltage-frequency scaling (DVFS) techniques to improve power efficiency [25, 29, 32, 48], hardware and software schemes to use low power idle states [37, 39, 53], and hardware resource partitioning schemes that allow batch workloads to run alongside latency-critical ones, improving utilization [29, 30, 33, 57].

However, the lack of a readily-available, comprehensive benchmark suite continues to be a key stumbling block for work in this area. Many of these studies use workloads internal to datacenter operators like Google or Facebook [32, 33, 36, 38, 55, 56]. Academic studies use one or a few latency-critical benchmarks [25, 48, 54], which limits the range of behaviors and performance requirements across which their proposed techniques can be evaluated. Some work uses more readily-available sequential and parallel batch workloads (e.g., from SPEC CPU2006 or PARSEC) and treats them as latency-critical applications [15, 57]. However, these applications differ from latency-critical applications in important ways, e.g., in their activity profile (continuous activity vs request-response behavior characterized by short idle periods [30, 37]), as well as in their microarchitectural characteristics [21].

B. TailBench vs. Existing Benchmark Suites

While some existing benchmark suites include latency-critical applications, they form a small part of the suite, and often focus on specific domains (e.g., real-time analytics [1] or machine learning [23]). These benchmark suites suffer from four problems: they include a small number of latency-critical applications, have limited diversity, suffer from methodological issues, and are hard to simulate. We now compare TailBench with representative benchmark suites along these dimensions.

CloudSuite [21] is perhaps the closest to TailBench. CloudSuite includes open-source counterparts to many common datacenter applications. However, the main focus of CloudSuite is on the microarchitectural characteristics of cloud applications and their impact on throughput. CloudSuite includes only four latency-critical applications out of a total of eight: solr (search), memcached (data caching), cassandra (NoSQL database), and elgg (web serving), and includes no applications from important domains such as speech and image recognition. Further, these applications cover a limited range of tail latencies, either 100s of milliseconds (solr) or a few milliseconds (memcached and cassandra). By contrast, TailBench includes applications from a broad set of domains that cover a wide range of tail latencies, from tens of microseconds to seconds. Covering a wide spectrum is important because different software and hardware techniques impact tail latency at different timescales. For example, DVFS techniques can react in microseconds, deep sleep states have transition latencies of hundreds of microseconds, and on-chip caches take tens of milliseconds to warm up.

Additionally, CloudSuite workloads use load testers such as YCSB [16] and Faban [3] that suffer from methodological problems resulting in large errors in latency measurement. These load testers model a *closed-loop system*, where a few client threads issue requests and block waiting for responses [56]. However, latency-critical applications receive requests from a large pool of users, and thus behave as *open-loop systems*, where the application receives requests

TABLE I
TAILBENCH APPLICATIONS.

	xapian	masstree	moses	sphinx	img-dnn	specjbb	silo	shore
Domain	Online Search	Key-Value Store	Real-Time Translation	Speech Recognition	Image Recognition	Java Middleware	OLTP (in-memory)	OLTP (disk/SSD)
Configuration and Input Set	English Wikipedia, zipfian query popularity	mycsb-a (50% GETs/PUTs), 1.1 GB table	opensubtitles.org corpora, phrase mode	CMU AN4 corpus	MNIST corpus	Standard	TPC-C, 1 warehouse	TPC-C, 10 warehouses
Language	C++	C++	C++	C++	C++	Java	C++	C++
L1I MPKI	1.14	0.23	1.79	0.06	0.32	8.87	1.2	22.68
L1D MPKI	13.69	11.41	26.82	23.83	87.49	15.62	2.88	23.83
L2 MPKI	8.94	9.32	24.77	20.22	16.64	14.91	1.92	20.22
L3 MPKI	0.02	5.41	19.95	3.51	15.05	3.49	0.56	3.51
Branch MPKI	7.22	5.66	2.24	6.94	0.35	4.99	5.58	6.94
95^{th} %ile 20%	2.67 ms	428 μs	3.06 ms	2.08 s	2.51 ms	293 μs	191 μs	1.99 ms
latency 50%	4.88 ms	688 μs	5.41 ms	2.78 s	3.94 ms	507 μs	374 μs	2.80 ms
at load 70%	9.48 ms	1.18 ms	11.42 ms	3.82 s	6.91 ms	739 μs	1.33 ms	4.20 ms

at a rate independent of its throughput. Prior work has shown that inadvertently introducing closed loops, known as the coordinated omission problem [44], can significantly underestimate tail latency. Treadmill [56] identifies this and several other issues with CloudSuite's load testers, such as client-side queuing and insufficient sampling. By contrast, TailBench's harness (Sec. IV) accounts for these factors to produce robust, unbiased measurements.

Finally, CloudSuite applications use a multi-machine configuration. While this setup mimics the architecture of scale-out applications, it makes them hard to run in simulation for long enough to accurately measure tail latency. By contrast, TailBench's harness includes different implementations: from full-blown multi-node configurations, to a simple single-node setup that can be easily simulated. This setup allows us to identify the minimum level of simulation fidelity required to faithfully measure tail latency for each application. We find that, in many cases, a simple user-level simulator is sufficient to study these workloads (Sec. VI).

BigDataBench [51] includes several big data applications as well as representative datasets. Like CloudSuite, BigDataBench focuses on microarchitectural characterization, and suffers from the same limitations: only three of its nineteen benchmarks are latency-critical, it lacks a rigorous methodology for measuring latencies, and employs multi-node measurement setups.

Other recent benchmark suites target specific application domains within datacenters. For example, DCBench [26] and the AMPLab Big Data Benchmark [1] focus on data analytics applications, while Sirius [24] targets applications for intelligent personal assistants like Apple Siri. Besides being domain-specific, these suites include applications with higher latencies than the interactive services TailBench focuses on. Other domain-specific suites include Tonic [23] for deep learning and YCSB [16] for NoSQL databases.

III. TAILBENCH APPLICATIONS

We now briefly describe the applications included in TailBench. Table I reports the input set, tail latency, and microarchitectural characteristics of each application.

xapian [6] is an open-source search engine written in C++ and widely used both in popular websites (e.g., the Debian wiki) and software frameworks (e.g., Catalyst). Online search engines handle petabytes of index data, which is split into shards spread across thousands of leaf nodes. The bulk of the processing happens at the leaf nodes, with each node independently searching its portion of the index. We configure `xapian` to represent a leaf node. In our experiments, the search index is built from a dump of the English version of Wikipedia from July 2013. Query terms are chosen randomly, following a Zipfian distribution, which has been shown to model online search query distributions well [7, 20].

masstree [35] is a fast, scalable in-memory key-value store written in C++. In-memory key-value stores serve as data storage backends for a wide variety of services. Key-value stores handle large amounts of data, which is split up into memory-resident shards spread across hundreds of servers. Each user request often involves many tens or hundreds of requests to the key-value store; these applications therefore have very short latency requirements, e.g., about 100 μs [32, 35]. While there are many open-source key-value stores, we chose `masstree` since it is highly optimized to make efficient use of the memory hierarchy of modern multicores. We drive `masstree` using a modified version of the Yahoo Cloud Serving Benchmark [16] that has 50% `get` and 50% `put` queries.

moses [31] is a state-of-the-art statistical machine translation (SMT) system written in C++. SMT systems underpin online translation services such as Google Translate, and also form an important component of speech-based interfaces such as Apple Siri. We use the phrase-based decoder included in `moses`; `moses` also supports tree-based decoding. We drive `moses` using randomly-chosen dialogue snippets from the opensubtitles.org English-Spanish corpus [45].

sphinx [50] is an accurate speech recognition system written in C++. Speech recognition systems are an important component of speech-based interfaces and applications such as Apple Siri, Google Now, and IBM Speech to Text. Speech recognition is a compute-intensive activity, involving probabilistically pruning a large search tree. `sphinx` uses sophisticated acoustic, phonetic,

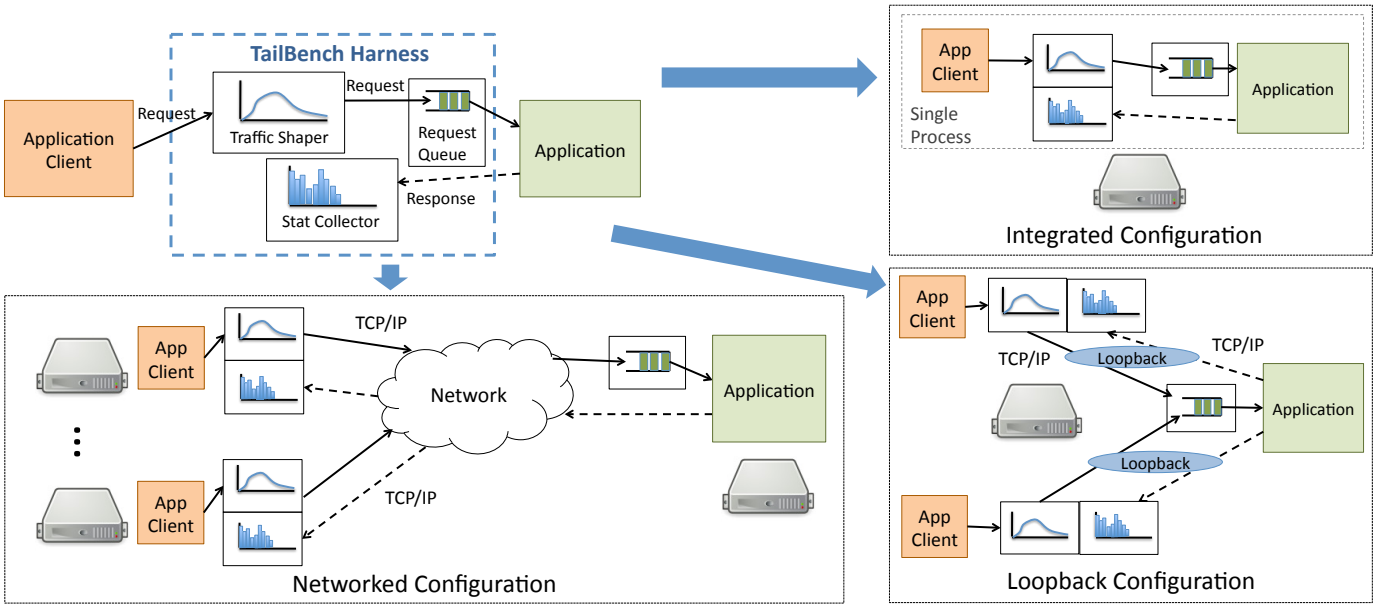

Fig. 1. TailBench harness components and its three configurations. The traffic shaper controls the arrival rate of requests from the clients to the application, while the statistics collector maintains request latency statistics. In the multi-node *networked configuration*, clients run on separate machines and communicate with the application over the network using TCP/IP. In the single-node *loopback configuration*, clients run on the same machine as the application and communicate using TCP/IP over the loopback interface. The single-node *integrated configuration* integrates the client and the application in a single process.

and language models to improve efficiency and accuracy. We drive `sphinx` using randomly-chosen utterances from the CMU AN4 alphanumeric database.

`img-dnn` [2] is a handwriting recognition application based on OpenCV [14]. Handwriting recognition is an example of the broader class of image recognition applications, widely used today for optical character recognition, image-based search (e.g., Google Goggles), automatic image tagging, and a variety of other online applications. `img-dnn` uses a deep neural network-based autoencoder coupled with softmax regression to identify handwritten characters. We drive the application using randomly-chosen samples from the MNIST database [19].

`specjbb` [5] is an industry-standard Java middleware benchmark. Java middleware is widely used in business services and must often satisfy strict latency constraints. `specjbb` emulates a 3-tier system, typical of many server-side Java applications. The modeled system is a wholesale company that handles different types of client requests (e.g., processing payments and deliveries). We run `specjbb` using HotSpot v1.8.

`silo` [47] is a fast in-memory transactional database. `silo` is designed to scale well on modern multicores, avoiding centralized contention points and making efficient use of the memory hierarchy. Databases like `silo` are widely used in online transaction processing systems (OLTP). We drive `silo` using TPC-C, an industry-standard OLTP benchmark [46].

`shore` [27] is a transactional database. Unlike `silo`, it is an on-disk database, and differs significantly in how it stores and accesses data. We drive `shore` using TPC-C. For the results in this paper, database and logs are both stored in a solid state drive to avoid having `shore` be bottlenecked on disk I/O.

Table I reports the measured tail latency for each application at various loads, as well as the application's microarchitectural characteristics. All results reported in Table I were collected using a multi-node configuration (our experimental methodology is described in detail in Sec. VI-A). We perform a detailed latency characterization of each application in Sec. V.

IV. TailBench Harness

The TailBench harness controls the end-to-end execution of each latency-critical application, and integrates the functionality for input load generation and statistics collection. Fig. 1 shows the three components of the TailBench harness: the *traffic shaper*, which controls the timing characteristics of the request stream; the *request queue*, which holds incoming requests and measures service and queuing times; and the *statistics collector*, which aggregates timing statistics. The TailBench harness has a modular design that allows multiple implementations to suit the needs of specific measurement scenarios. We first discuss the multi-node *networked* configuration, which faithfully captures all sources of latency, and then discuss two simplified configurations, which reduce measurement complexity without sacrificing accuracy for most applications.

A. Networked Configuration

The networked configuration (Fig. 1, lower left) employs one or more client machines to drive the application. Each client machine hosts an application-specific client module integrated with the traffic shaper and statistics collector. The client module continuously generates requests and hands them to the traffic shaper, which simulates the desired load by inserting delays between requests before sending them to the application over the network. The traffic shaper uses an *open-loop* design, i.e., it sends requests according to their desired timing characteristics without waiting for responses to previous requests. Prior work has shown that open-loop setups are representative of datacenter

traffic patterns [56] and accurately capture the queuing delays that form a significant portion of tail latency [29]. The harness generates queries with exponentially-distributed interarrival times with a configurable rate, which have been shown to accurately model datacenter traffic [38].

The request queue is shared among application threads. The request queue stores incoming requests, and measures *queuing time* (time spent waiting in the queue) as well as *service time* (execution time starting from when the request is handed to an application thread) for each. Upon completion of the request, this timing data is sent back over the network to the appropriate statistics collector module.

B. Simplified Harness Configurations

The networked configuration captures all sources of latency, including network link and switch delays as well as network stack overheads. However, this setup is complex: one must ensure that the networking infrastructure matches those found in modern datacenters, both in hardware capabilities (e.g., high-bandwidth, low-latency network interface cards and switches), and in the interference patterns from other applications sharing the network. In addition, networking hardware must be carefully configured to achieve low latency. For example, prior work has shown that interrupt-to-core mapping (via receive side scaling or flow steering) and interrupt coalescing can have a significant impact on request latency [11, 41]. Indeed, in setting up the networked configuration for our experiments, we spent several days tuning the networking setup (Sec. VI), which reduced round-trip network latencies from 200 to 50 μs.

Additionally, while network delays are an important consideration in datacenters, operators often treat network latencies separately from processing latencies by, for example, assigning different time budgets to each [48, 52].

The *loopback* configuration (Fig. 1, lower right) focuses purely on request processing in the application while ignoring network delays. In this configuration, application and client reside on the same machine and communicate over TCP/IP using the loopback interface. This captures most of the overheads introduced by the network stack.

While the loopback configuration is significantly easier to set up than the multi-node configuration, it is still too complex for some use cases. In particular, evaluating the impact of proposed hardware changes on tail latency requires simulating enough requests to meaningfully measure tail latency. The loopback configuration would require simulating a multiprogrammed configuration in a full-system simulator. Unfortunately, typical simulation speeds for full-system simulators are only about 200 KIPS [13], which makes long simulations impractical.

To facilitate faster simulation, we implement the *integrated* configuration (Fig. 1, upper right). The integrated configuration combines client, harness, and application into a single process, with modules communicating via shared memory. While this approach ignores network stack overheads, we show that these constitute a small fraction of total processing time for many applications, and ignoring them does not significantly impact observed latency characteristics (Sec. VI). Since the integrated configuration employs userspace communication, it can be simulated with faster user-level simulators [40, 43].

C. Statistics Collection and Latency Measurement

The TailBench harness collects detailed request-level latency statistics that can be used to derive mean and percentile latencies, as well as to construct full service and sojourn time distributions. For short runs, the harness maintains latency measurements for each individual request to maximize accuracy. For longer runs, it uses high dynamic range (HDR) histograms [4] to minimize space overheads while still maintaining high accuracy. HDR histograms can capture statistics over a wide range of values (e.g., latencies ranging from 1 µs to 1000 s) with logarithmic space overheads while maintaining high precision (e.g., recorded value within 1% of the actual). For instance, in the above example, the HDR histogram only needs to maintain 100 buckets between any two subsequent powers of 10 (e.g., for latencies between 1 ms and 10 ms), allowing the entire range to be covered with only 900 buckets.

We carefully design our methodology to avoid the pitfalls that afflict prior testbeds [56]. Each measurement run is preceded by a warmup period of sufficient length to ensure that we measure steady-state execution only. In the networked and loopback configurations, we ensure that there are sufficient clients so that client-side queuing is not a concern.

Accurately measuring tail latency requires collecting a large number of measurement samples. Since tail latency inherently measures "outliers" (e.g., the slowest 5% requests when measuring the 95^{th} percentile latency), even small changes, such as reordering of a few requests, can have a large impact on the value measured. It is therefore necessary to collect enough samples to ensure that the measurement run is representative. However, individual runs, even if they are sufficiently long, can yield wrong results due to *performance hysteresis* [56], i.e., systematic bias introduced due to factors like memory layout that change from run to run. We counter this by performing repeated runs, randomizing requests as well as interarrival times in each run to ensure that we measure a representative distribution across runs. The harness performs enough runs to achieve 95% confidence intervals of at most 1% for each latency metric reported.

V. Application Characterization

We now study the latency characteristics of each application, including request *service times* and *sojourn times*. The service time of a request measures the time the application takes to process that request. The sojourn time, by contrast, is the *end-to-end* request latency, from the time the request was issued to the time a response is received. Sojourn time includes, in addition to the service time, the time spent queued while the application is busy servicing previous requests as well as network delays. We also study how multithreading affects tail latency in these applications. All measurements in this section were obtained using the networked harness configuration (Sec. IV). We explain our experimental methodology in detail in Sec. VI-A.

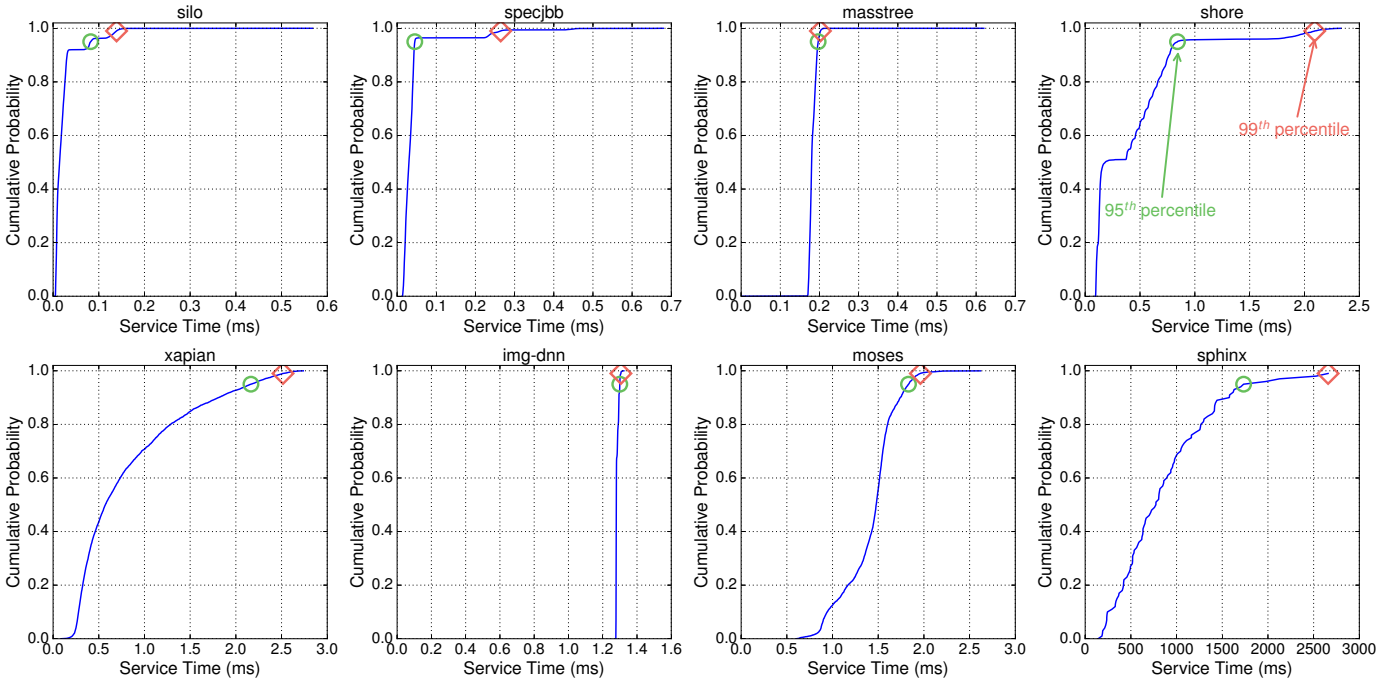

Fig. 2. Cumulative distribution function (CDF) of service times for each application, with service times on the x-axis and cumulative probability on the y-axis.

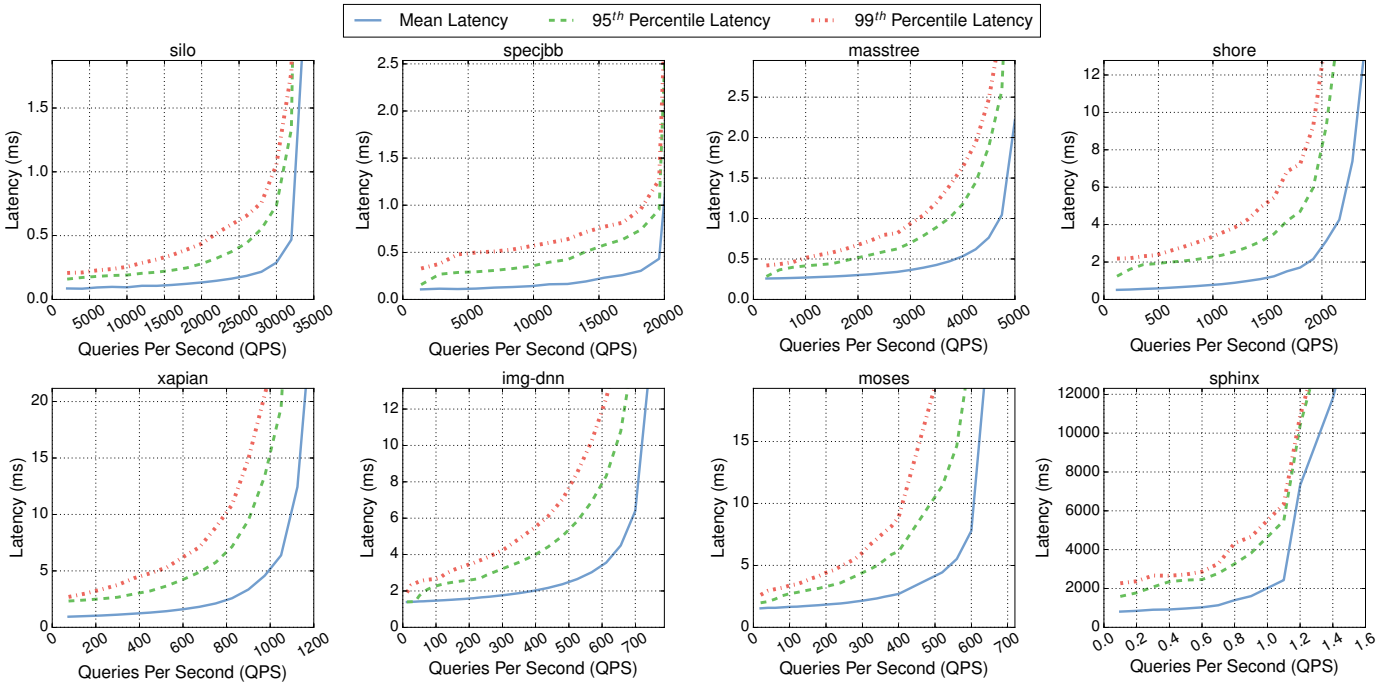

Fig. 3. Mean, 95^{th} percentile, and 99^{th} percentile latencies for each application across a range of request rates. All applications use a single worker thread.

Application service times: Fig. 2 shows the cumulative distribution function (CDF) of request service times for each TailBench application. Service times vary widely across applications: while most `specjbb` and `silo` requests finish in under 100 μs, `sphinx` requests can take more than a second each. Applications also vary widely in how tightly their request service times are distributed. For some applications request service times are distributed fairly evenly across a large range; `xapian` requests, for example, take anywhere from 200 μs to 2.7 ms. Other applications, such as `specjbb` and `shore`, have most of their request times distributed in a fairly narrow range, but have a "long tail" of requests that take much longer than others. Finally, `masstree` and `img-dnn` have nearly constant request service times.

Sojourn times vs. load: Fig. 3 shows the mean, 95^{th} percentile, and 99^{th} percentile tail latencies for each application at various request rates (queries per second, or QPS). In these experiments, applications use a single worker thread. At very low request

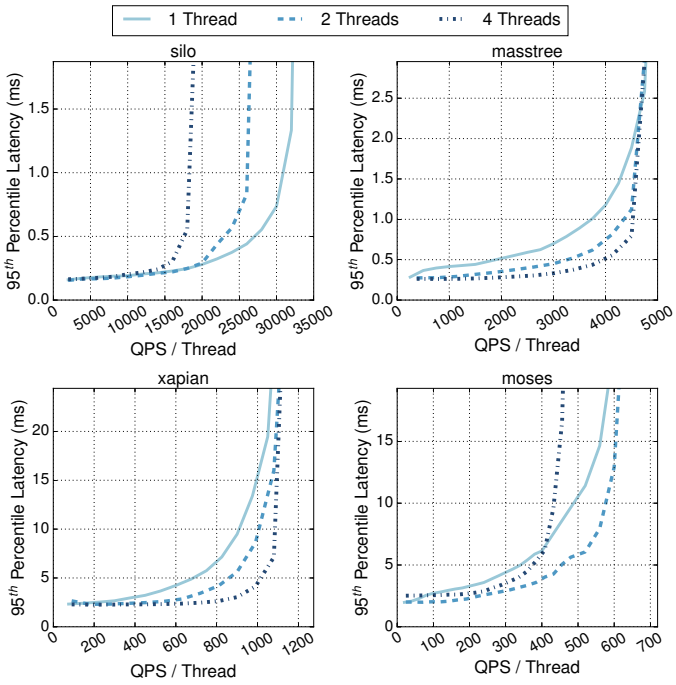

Fig. 4. 95^{th} percentile latency for representative applications across a range of request rates, as the number of worker threads increases from 1 to 4.

rates, the difference between mean and tail latencies depends mostly on the distribution of request service times (Fig. 2). As request rates increase, both mean and tail latencies increase, since incoming requests are more likely to experience queuing delay as they wait for previous requests to finish. However, tail latencies increase much more rapidly than the mean. This rapid increase often limits the utilization of servers running latency-critical applications: since datacenter operators must account for occasional load fluctuations, latency-critical applications operate at request rates well below saturation. The gap between tail and mean latencies is higher for applications with more variable service times: not only do long requests contribute to the tail themselves, they are more likely to cause other requests to be queued up behind them. There is, however, no general way to determine the exact relationship between tail and mean latencies. Determining the impact of a design decision on tail latency thus requires measuring tail latency directly—throughput metrics (e.g., mean latency or instructions per cycle) do not suffice.

Impact of multithreading: Fig. 4 shows how tail latency changes with the number of worker threads for four representative applications. Each graph reports 95^{th} percentile latency (y-axis) as a function of *request rate per thread* (x-axis) and the number of threads (different lines). As the number of threads grows, the probability of a request finding all threads busy decreases, reducing the contribution of queuing time to tail latency. `masstree` and `xapian` behave as expected: with more threads, their tail latencies grow more slowly with load, while their per-thread saturation rates stay relatively constant. However, `silo` and `moses` do not behave as expected. In `silo`, adding threads causes each thread to saturate at a lower QPS; i.e., the overall application throughput at the saturation point improves *sublinearly*. Finally, `moses` behaves like `xapian` and `masstree` when the number of threads increases from one to two, but increasing the thread count to four *degrades* the saturation QPS for each thread to below the value for a single thread. This degradation can be caused by synchronization overheads among threads, or by contention among threads for shared memory resources (e.g., cache and memory bandwidth). In Sec. VII, we use microarchitectural simulation to separate both effects for each application.

TABLE II
CONFIGURATION OF THE EXPERIMENTAL SYSTEM.

Cores	8 Xeon E5-2670 cores (SandyBridge), 2.4 GHz nominal frequency
L1 caches	32 KB, 8-way set-associative, split D/I
L2 caches	256 KB private per-core, 8-way set-associative
L3 cache	20 MB total, 20-way set-associative, DRRIP, inclusive
Memory	32 GB, DDR3 1333 MHz
OS	Ubuntu 14.04, Linux kernel version 4.2.3

VI. VALIDATING SIMPLIFIED HARNESS CONFIGURATIONS

In this section, we compare the tail latency measured using the three harness configurations discussed in Sec. IV, in order to understand when it might be acceptable to use the simplified configurations. We also compare the tail latency measurements obtained on a real system with those obtained in simulation.

A. Experimental Methodology

Real system: All real-system measurements reported in this paper were performed on an Intel Xeon E5-2670 processor with 8 SandyBridge cores (Table II). We run applications on dedicated servers to avoid interference from colocated applications, and use real-time priority to prevent interference from background daemons. We disable TurboBoost and deep sleep states to avoid unpredictable performance fluctuations [28], and fix CPU frequency at the nominal value using the *cpufreq userspace* governor. When running in the networked or loopback configuration, we run multiple client processes to avoid client-side queuing [56]. The server and client machines used in the networked configuration each have an Intel I350 Gigabit Ethernet NIC and are connected via a Dell PowerConnect J-EX4200-48T switch. Even with this relatively simple setup, it required several days of tuning before we settled on the configuration that worked best for our applications. For all our measurements using the networked harness configuration, we use RSS to map interrupts to cores that are *not* running application threads, since we found that interrupt processing can significantly hurt latency, especially at high loads. We also disable interrupt coalescing, and use the TCP_NODELAY option to disable coalescing outbound packets using Nagle's algorithm. While these options improve latency, they may hurt throughput for some applications. We found this not to be the case for our applications.

Simulation: For simulation results, we use zsim [43], an execution-driven x86-64 simulator based on Pin [34]. ZSim

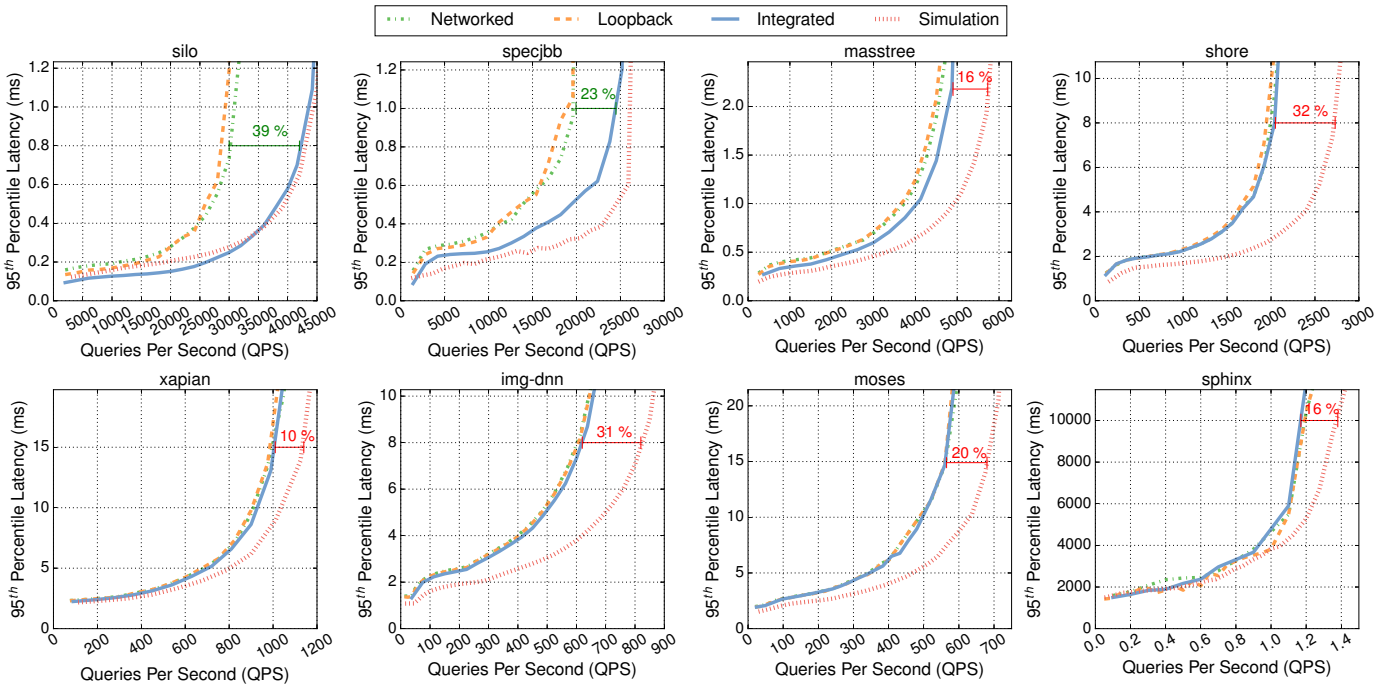

Fig. 5. 95^{th} percentile tail latency for single-threaded instances of each application. Each figure compares tail latency over four setups: the real system under the three harness configurations, and the simulated system under the integrated configuration. Differences in saturation QPS are shown between the integrated and networked configurations (in green) for `silo` and `specjbb`, and between the integrated configuration and simulation (in red) for the other applications.

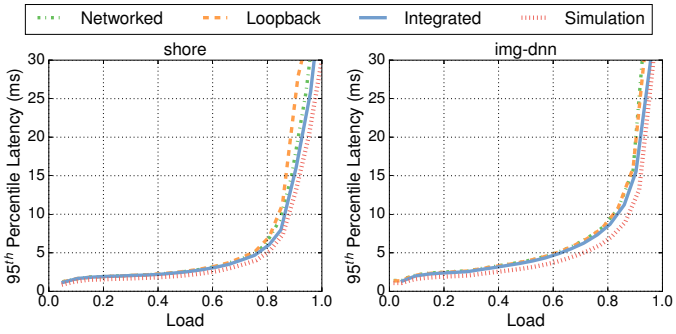

Fig. 6. 95^{th} percentile tail latency for single-threaded instances of `shore` and `img-dnn` as a function of system load instead of QPS.

achieves simulation speeds of several MIPS per simulated core, allowing us to perform the long simulations needed to measure tail latency accurately. ZSim is also accurate, with IPC errors of 2%-24% on SPEC CPU2006 benchmarks over a Nehalem system (average error of 9.7%). The simulated system has cores similar to our experimental system (Table II) and an identical memory system. All our simulation results use the integrated harness configuration (Sec. IV).

B. Single-Threaded Applications

Fig. 5 presents the 95^{th} percentile latency observed using the various harness configurations on a real system, as well as the 95^{th} percentile latency measured in simulation.

Harness configurations: Focusing first on the real-system results, we note that the measured tail latency using the three harness configurations is very similar for six of the eight applications. This is not surprising: in our system, the Linux networking stack introduces an overhead of about 25 μs at each end (application and client) for the networked configuration, and about 20 μs for the loopback configuration. This is a small fraction of typical request service times for most applications, even for applications like `masstree` and `shore` where the typical request takes a few 100 μs.

Network stack overheads are more pronounced for `specjbb` and `silo`, which have much shorter requests (95^{th} percentile service times of under 100 μs). At low loads, this causes a small difference in measured tail latency relative to the integrated configuration. As load increases, however, the effect of the longer service times for the networked and loopback configurations becomes more pronounced as slower request processing leads to higher queuing. Eventually, the two configurations saturate before the integrated configuration does, with saturation request rates being 23% lower than the integrated configuration for `specjbb`, and 39% lower for `silo`. Thus, while the qualitative behavior of the latency profile remains the same for the three configurations, latency increases more rapidly with load for the networked and loopback configurations.

Note that network latency depends heavily on the characteristics of the networking hardware (NICs, switches, and topology) and on network contention. While we find network delays to not be significant for most of our benchmarks, they may be significant in other network setups. Such cases would require using the multi-node configuration or enhancing the simplified configurations with a network simulator.

Simulation: Latency profiles in simulation are similar to the real-system ones for all applications. However, since simulation introduces some performance error, the measured tail latency at each request rate is somewhat different from the real-system measurements. This is as expected; since the simulated system

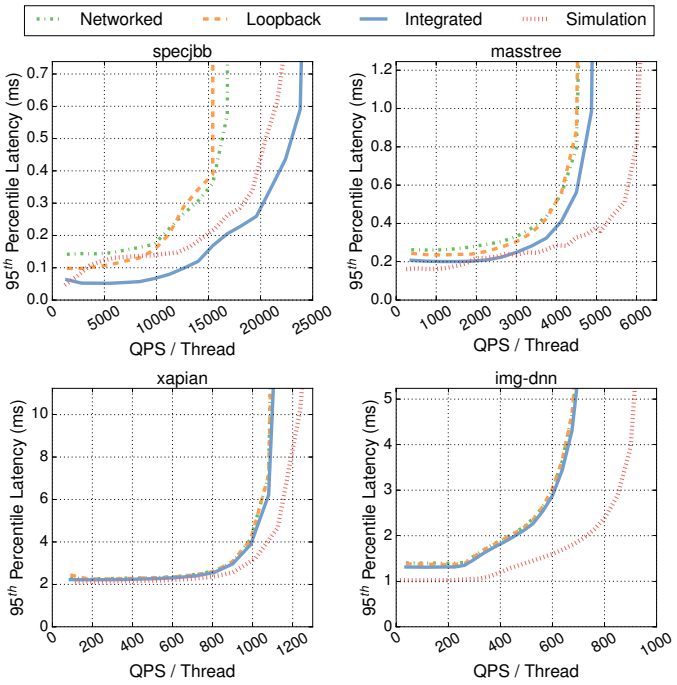

Fig. 7. 95^{th} percentile tail latency for multi-threaded instances of representative applications, each with 4 threads. Each figure compares tail latency over four setups: the real system under the three different harness configurations, and the simulated system under the integrated configuration.

is faster than the real system for most of our applications, it experiences less load at any given request rate, which in turn results in lower queuing delays and thus lower tail latency. Another way to see this is to note that for each application, the request rates at which the real and simulated systems reach a given tail latency level, as well as the request rates at which they saturate, differ by a constant factor, which is the performance error introduced by the simulator.

To illustrate this effect further, Fig. 6 shows the 95^{th} percentile latency against *system load* for the two applications with the largest simulation error, `shore` and `img-dnn`. We see that the real-system and simulated latency profiles are nearly identical for both applications. Since the simulated and real systems have different performance, they reach a given system load at slightly different request rates (Fig. 5), but their behavior at each load level is very similar (Fig. 6). We observe similar behavior for other applications, but omit those results in the interest of space. We conclude that simulation can yield accurate insights into an application's tail latency behavior.

C. Multithreaded Applications

Fig. 7 presents the 95^{th} percentile latency for four of our applications in various configurations, where each application is multithreaded (four worker threads). We see similar behavior as in the single-threaded case: the three real-system configurations are almost identical for applications with relatively long service times (`xapian`, `img-dnn`, `masstree`), while the networked and loopback configurations experience higher latencies for applications with short requests (`specjbb`). As in the single-threaded case, simulation results agree with real-system measurements, with small deviations introduced by simulation error. We observe similar behavior for the remaining applications, but omit these results in the interest of space.

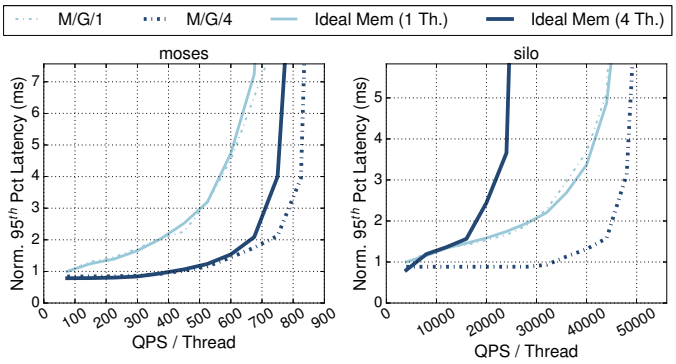

Fig. 8. Normalized 95^{th} percentile latency for `moses` and `silo` with 1 and 4 threads, using an M/G/n queuing model (dashed lines), and simulating an idealized memory system (solid lines). Results reveal that `moses`'s suboptimal scaling in Fig. 4 is due to memory system contention, while `silo`'s scaling problems are due to synchronization overheads.

VII. Case Study

One of the key benefits of TailBench is to make latency-critical workloads as easy to simulate as conventional throughput workloads. We demonstrate this benefit through a simple case study, where we use simulation to find why `moses` and `silo` scale poorly with thread count. Specifically, we will determine the relative importance of two factors: synchronization overheads and contention in shared memory resources.

To distinguish between these factors, we simulate each application with an idealized memory system with zero-cycle latency to DRAM and infinite DRAM bandwidth, eliminating memory contention and the impact of increased shared cache misses. Fig. 8 shows the 95^{th} percentile latency for `moses` and `silo` in this idealized memory system with one and four threads, normalized to the 95^{th} percentile latency at low load with one thread. Fig. 8 also shows the predicted 95^{th} percentile latency using an M/G/n queuing model [22] (where n = number of threads). The latencies predicted by the queuing model would be realized if there were no overhead to adding threads (i.e., if service times stayed constant).

Comparing the simulation and M/G/n results reveals different trends for `moses` and `silo`. Simulation and M/G/n results are in agreement for `moses`, revealing that `moses`'s performance degradation at four threads in the real system (Fig. 4) is largely due to contention in the memory system, and could be alleviated by adding memory resources (e.g., larger shared caches). By contrast, the simulated idealized memory system does not improve `silo`'s performance with four threads, suggesting that synchronization overheads are the culprit.

VIII. Conclusions

We have presented TailBench, a benchmark suite and evaluation methodology for latency-critical applications. TailBench seeks to make latency-critical applications as easy to run and characterize as throughput-oriented benchmarks. TailBench includes representative applications from a diverse set of domains

that exhibit a wide range of tail-latency behaviors, and a harness that implements a robust and statistically sound load testing methodology and allows several measurement configurations. Our validation results show that while a multi-node, networked harness configuration offers maximum measurement fidelity, a simplified single-node, integrated setup captures tail latency accurately for most benchmarks. The integrated configuration significantly reduces measurement costs, facilitating studying tail latency in simulation. Finally, we have used simulation to identify the causes of sublinear scaling for two of our applications. TailBench is open-source and publicly available at http://tailbench.csail.mit.edu.

Acknowledgments

We thank Maleen Abeydeera, Nathan Beckmann, Mark Jeffrey, Xiaosong Ma, Nosayba El-Sayed, Suvinay Subramanian, Po-An Tsai, and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grant CCF-1318384, a grant from the Qatar Computing Research Institute, and a Google Research Award.

References

[1] "AMPLab big data benchmark," https://amplab.cs.berkeley.edu/benchmark/.

[2] "A deep network handwriting classifier," https://github.com/xingdi-eric-yuan/multi-layer-convnet.

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

[4] "HdrHistogram: A high dynamic range histogram," https://github.com/HdrHistogram/HdrHistogram.

[5] "SPECjbb," https://www.spec.org/jbb2015/.

[6] "Xapian project," https://github.com/xapian/xapian.

[7] R. Baeza-Yates, "Applications of web query mining," in *ECIR*, 2005.

[8] L. Barroso and U. Hölzle, "The case for energy-proportional computing," *IEEE Computer*, vol. 40, no. 12, 2007.

[9] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer*, 2nd ed. Morgan & Claypool, 2013.

[10] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *PACT-22*, 2013.

[11] A. Belay, G. Prekas, A. Klimovic, S. Grossman *et al.*, "IX: A protected dataplane operating system for high throughput and low latency," in *OSDI-11*, 2014.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT-17*, 2008.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 2011.

[14] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly, 2008.

[15] H. Cook, M. Moreto, S. Bird, K. Dao *et al.*, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *ISCA-40*, 2013.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC-1*, 2010.

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

[18] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *ASPLOS-XIX*, 2014.

[19] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Process. Mag.*, vol. 29, no. 6, 2012.

[20] D. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, 2015.

[21] M. Ferdman, A. Adileh, O. Kocberber, S. Volos *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *ASPLOS-XVII*, 2012.

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

[23] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen *et al.*, "DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers," in *ISCA-42*, 2015.

[24] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li *et al.*, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *ASPLOS-XX*, 2015.

[25] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner *et al.*, "Adrenaline: Pinpointing and reigning in tail queries with quick voltage boosting," in *HPCA-21*, 2015.

[26] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," in *IISWC*, 2013.

[27] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-MT: A scalable storage manager for the multicore era," in *EDBT*, 2009.

[28] S. Kanev, K. Hazelwood, G. Wei, and D. Brooks, "Tradeoffs between power management and tail latency in warehouse-scale applications," in *IISWC*, 2014.

[29] H. Kasture, D. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *MICRO-48*, 2015.

[30] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict QoS for latency-critical workloads," in *ASPLOS-XIX*, 2014.

[31] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch *et al.*, "Moses: Open source toolkit for statistical machine translation," in *ACL-45*, 2007.

[32] D. Lo, L. Cheng, R. Govindaraju, L. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *ISCA-41*, 2014.

[33] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *ISCA-42*, 2015.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[35] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *EuroSys*, 2012.

[36] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. Soffa, "Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *MICRO-44*, 2011.

[37] D. Meisner, B. Gold, and T. Wenisch, "PowerNap: eliminating server idle power," in *ASPLOS-XIV*, 2009.

[38] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *ISCA-38*, 2011.

[39] D. Meisner and T. Wenisch, "DreamWeaver: Architectural support for deep sleep," in *ASPLOS-XVII*, 2012.

[40] J. Miller, H. Kasture, G. Kurian, C. Gruenwald *et al.*, "Graphite: A distributed parallel simulator for multicores," in *HPCA-16*, 2010.

[41] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris, "Improving network connection locality on multicore systems," in *EuroSys*, 2012.

[42] M. Qureshi and Y. Patt, "Utility-based cache partitioning," in *MICRO-39*, 2006.

[43] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA-40*, 2013.

[44] G. Tene, "How not to measure latency," in *Low Latency Summit*, 2013.

[45] J. Tiedemann, "Parallel Data, Tools and Interfaces in OPUS." in *LREC*, 2012.

[46] TPC Council, "TPC-C benchmark, revision 5.11," 2010.

[47] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *SOSP-24*, 2013.

[48] B. Vamanan, H. Sohail, J. Hasan, and T. N. Vijaykumar, "TimeTrader: Exploiting latency tail to save datacenter energy for online search," in *MICRO-48*, 2015.

[49] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry *et al.*, "Low latency via redundancy," in *CoNEXT*, 2013.

[50] W. Walker, P. Lamere, P. Kwok, B. Raj *et al.*, "Sphinx-4: A flexible open source framework for speech recognition," Sun Microsystems, Tech. Rep., 2004.

[51] L. Wang, J. Zhan, C. Luo, Y. Zhu *et al.*, "BigDataBench: A big data benchmark suite from internet services," in *HPCA-20*, 2014.

[52] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *SIGCOMM*, 2011.

[53] D. Wong and M. Annavaram, "Knightshift: Scaling the energy proportionality wall through server-level heterogeneity," in *MICRO-45*, 2012.

[54] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-Flux: Precise online QoS management for increased utilization in warehouse scale computers," in *ISCA-40*, 2013.

[55] X. Zhang, E. Tune, R. Hagmann, R. Jnagal *et al.*, "CPI2: CPU performance isolation for shared compute clusters," in *EuroSys*, 2013.

[56] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *ISCA-43*, 2016.

[57] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems," in *ASPLOS-XXI*, 2016.