TailWAG: Tail Latency Workload Analysis and Generation

Heng Zhuo
hzhuo2@wisc.com
University of Wisconsin-Madison
Madison, Wisconsin, USA

Mikko Lipasti
mikko@engr.wisc.edu
University of Wisconsin-Madison
Madison, Wisconsin, USA

Abstract

Server performance has always been an active field of research, spanning topics including databases, network applications, and cloud computing. Tail latency is one of the most important performance metrics for server performance. In this work, we study how different timing components affect server workload performance, including thread contention and interrupt handling. We introduce a workload analysis and generation tool: TailWAG. TailWAG relies on measured timing components from applications to generate a synthetic workload that mimics the original behavior. TailWAG can help server designers and application developers explore performance bottlenecks and optimization opportunities at early stages.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Modeling and simulation**.

Keywords: servers, workload generation, performance analysis

ACM Reference Format:
Heng Zhuo and Mikko Lipasti. 2023. TailWAG: Tail Latency Workload Analysis and Generation. In *Benchmarking in the Data Center (BID '23), February 25, 2023, Montreal, QC, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3583060.3583170

1 Introduction

With the current trend toward cloud computing, more and more focus has shifted to server performance. Companies and researchers pay increasing attention to cluster computing. Simply having a cluster with thousands of powerful cores does not guarantee optimal performance [6]; careful design and configuration, depending on the workload, are required [14][15]. But performance metrics are consistent throughout all cases: throughput and latency. In addition to mean latency, server applications also emphasize tail latency (95th or 99th percentile latency, worst 5% or 1% respectively of all requests), which is often used to specify the quality of service requirements for Service Level Agreements (SLA) [5].

Server systems are difficult to evaluate due to complex and fragile workload configurations. Tuning the system to run for the best performance is very difficult and time-consuming. Changes to one configuration parameter can lead to very different performances. As one example, prior work [2][21][18] showed that using a dedicated core for interrupt handling can isolate server threads from interrupts and reduce tail latency.

We verified this with a sample application (silo) from Tailbench [18] under various scenarios. As shown in Figure 1, assigning interrupt handling and server threads to different cores produces lower latency at low to moderate load, but higher loads (at the expense of latency) are possible if all cores are used for both tasks.

However, the follow-up questions are non-trivial: How many cores are required for dedicated interrupt handling? Are they dynamically allocated or statically determined? Also, more cores for interrupt handling means fewer cores for server threads. Using dedicated core (s) is an ad hoc solution that is not ideal for all scenarios.

Also, for any hardware features of configurations that are not available to the native machine, simulation becomes the only choice. Simulators provide flexibility and detailed exploration of architecture changes, but with the cost of extremely slow speed, such as gem5 [3][23], and are not ideal for early-stage design exploration. ZSim [26] on the other hand, trades off detailed microarchitectural implementation for faster speed. There are also other tools targeting RISC-V ISA such as firesim [17][16][25].

Server system hardware innovations and server applications are both evolving but at different paces. Hardware innovation usually goes through a detailed simulation and validation stage before implementation, which requires significant effort and a longer turnaround time; on the other hand, applications change more rapidly, and can evolve quickly via software patches.

In this paper, we perform a detailed analysis of Tailbench applications [18], break down timing components, abstract details from the individual application but provide high-level

BID '23, February 25, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0117-7/23/02…$15.00
https://doi.org/10.1145/3583060.3583170

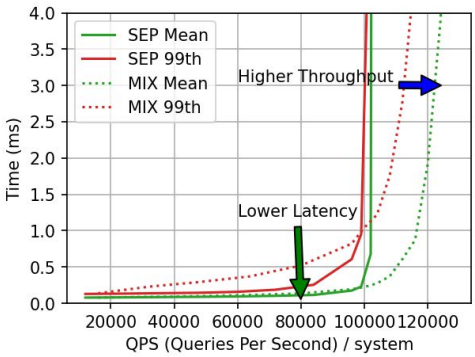


Figure 1. Latency vs Throughput. SEP: reserving core 1 for interrupt handling improves latency at low to moderate load, but lowers throughput; MIX: using all 4 cores for both interrupts and server threads yields higher throughput, with worse latency.

behavior analysis, and contribute insights on both latency and throughput. We introduce TailWAG, a single configurable synthetic workload, that enables swift parameter sensitivity analysis, and validate it against real hardware. The high-level workflow is shown in Figure 2. The key advantages of this synthetic workload are:

- TailWAG is simple, hardware independent, and easy to run natively or in simulation.
- TailWAG has repeatable behavior and measurements.
- TailWAG enables exploration of future/emerging/evolving workloads and hardware platforms before they exist.

System setup configuration is explored in the context of NIC (Network Interface Controller) interrupt handling, and potential further performance improvement is predicted for a hypothetical interrupt accelerator.

2 Background

2.1 Tail Latency

Cloud computing relies on utilizing many compute nodes across multiple layers of abstraction. Often, single requests are broken down into multiple sub-requests, and overall performance can be determined by the most poorly-performing sub-requests, i.e., n-th percentile tail latency. Tail latency is becoming a more important factor as a measure of the quality of service[5]. Sriraman and Wenisch studied how different OS operations affect server latency by using the eBPF tool [7][29][28]. These operations include hardware interrupts, network software interrupts (both receive and transmit), and scheduling actions. Not surprisingly, Active-Exe (thread active servicing the requests) and Net (net mid-tier latency) contribute the most to service time. Tiny components contribute minimally to average latency but can cause a severe cascade tail latency problem [2]. Other work such

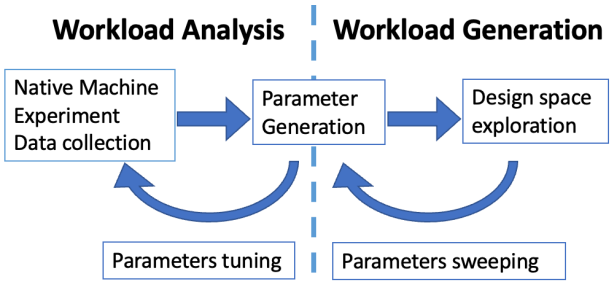


Figure 2. TailWAG workflow.

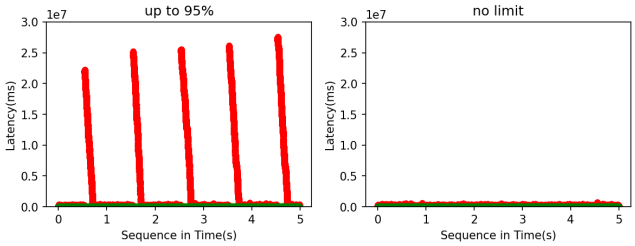


Figure 3. Linux real-time scheduling limit. The Linux kernel default limit for real-time tasks is 95% of total CPU time. When real-time tasks reach the limit, they are preempted for 50 ms, leading to the latency spikes shown above (left). Over-riding this setting to unlimited CPU utilization eliminates most of the latency spikes (right) but may cause starvation. Hardware setup in Table 1.

as Multilate [20] and Treadmill [35] are recent studies on load testers for evaluating server systems and looked into pitfalls including co-location applications.

From here, there are two orthogonal paths for improving performance: application developers can work to reduce service time, and service providers can reduce run time environmental overheads and noise. There are many system parameters that can affect performance dramatically [18][5]. Features including sleep states and voltage/frequency scaling are useful for energy management but can induce unexpected latency to requests, distorting the tail latency distribution. Also, Treadmill [35] pointed out client-side queuing can generate biased results and should be avoided by using multiple clients.

Setting a real-time priority for server threads avoids competition with lower-priority tasks, and can be used to ensure lower latency. But, for safety reasons, the kernel prevents real-time priority tasks from consuming 100% of the CPU, leading to the latency hiccups shown in Figure 3. Server maintainers can choose to set -1, which means there is no limit. However, this may lead to potential starvation as interrupt handling can be delayed indefinitely under heavy load.

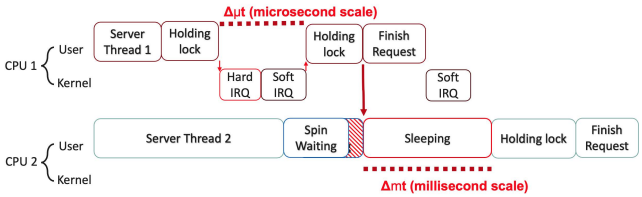


Figure 4. When hardIRQ occurs during the critical section on CPU1, CPU2 can potentially give up waiting due to spin limit, and be forced to sleep. Tuning the spin limit (the shaded portion of CPU2) to avoid this outcome is not trivial.

2.2 Network Interrupt Handling

Interrupt handling [9] is an important core-api layer and is commonly defined as synchronous (exception, software) and asynchronous (timer, driver hardware) [4]. The Linux kernel is equipped with a mixture of interrupt and polling design, named NAPI [8]. When a packet arrives (receiving process), the NIC hardware interface raises an asynchronous hardware interrupt request (IRQ); the system is notified, the CPU saves the current program state, switches context, and enters the interrupt handling routine (hereafter referred to as hardIRQ). During the hardIRQ, the CPU will disable preemption and interrupts, so only a minimal amount of work should be performed: (`napi_schedule`) function raises `NET_RX_SOFTIRQ`, schedules the software interrupt routine (hereafter softIRQ) and clears the interrupt line. On the other hand, softIRQ is preemptible and serves as the main function of polling network packets into specified user buffers depending on the protocol and applications.

Also, as part of the generic interrupt handling layer, before the hardIRQ exits with the function call `irq_exit`, it always tries to `invoke_softirq` if there is a softIRQ waiting and the current time quantum has not elapsed.

These two mechanisms are designed to serve different goals for the Linux kernel system and did not receive much attention for immediate improvement when NAPI was introduced. With either design, the asynchronous hardIRQ can lead to unpredictable performance hiccups. As shown in Figure 4, a microsecond-scale hardIRQ can lead to millisecond-scale latency.

2.3 Existing benchmarks and tools

For research purposes, benchmarks can be executed in a simulated environment to examine system behavior under different combinations of user-specified disk image, kernel, and modification to the hardware system, and any improvement or innovation can be observed and evaluated. However, there is a noticeable gap between performance on real hardware and simulation. One of the reasons is that real machines have more unexpected background noise than the simulated environment. It is almost impossible to replicate the same performance over different setups, core frequencies, cache sizes, and Ethernet hardware.

Moreover, hardware simulation is time-consuming. For simulation data points, one second of hardware machine time costs hours to days of simulation time on gem5 full system mode [23]. ZSim can provide improved speed but with less microarchitectural detail, and still requires environmental build and setup. Before proposing techniques to accelerate interrupt handling, we need to demonstrate the effects brought by these timing disturbances in a controlled fashion.

A synthetic workload can be representative, easy to tune, and quick to converge, making it ideal for early-stage design exploration. Similar work in other domains includes SynFull [1] for evaluating NoC traffic and STMBench7 [10] for software transactional memory.

Also, in addition to targeting application execution itself, this work also includes insights into runtime policy and network activities. TailWAG is easy to run in simulation, but in this work, all workload analysis and generation are performed on a native server machine.

Table 1. Server System Configuration

Model	Supermicro SYS-2029GP-TR
Cores	6 Cores Intel Xeon Gold 6128, 3.5 GHz
Caches	32KB L1, 6MB L2, 19.25MB L3
Main Memory	96GB, 1333 MHz
Operating System	Ubuntu 22.04 , Linux kernel 5.15.0

3 Workload Analysis

First, we describe the applications and the setup from Tailbench [18]. There are eight applications with varying attributes: **silo** [30], which is an in-memory transaction database; **specjbb** [27], which is a Java middleware warehouse server benchmark; **masstree** [24], which is a key-value store database with mycsb-a dataset (50% get and 50% set); **img-dnn**, image recognition with MNIST dataset [22]; **xapian** [33], an online search engine with Wikipedia English version; **shore** [11], on disk transaction database with TPC-C; **sphinx** is speech recognition with CMU AN4 alphanumeric database [32]; **moses** [19] is real-time translation, fed with opensubtitles.org English-Spanish corpus snippets.

The Tailbench harness provides three configurations: integrated (single process handles both server and client, communicating via memory mapped buffers); loopback (separate server and client process on same node communicating via TCP using localhost); and networked (server and client are on different nodes, communicating via Ethernet). All of the analysis experiments are conducted on a networked setup, and TailWAG uses the integrated setup for simplicity. Server system hardware setup is listed in Table 1.

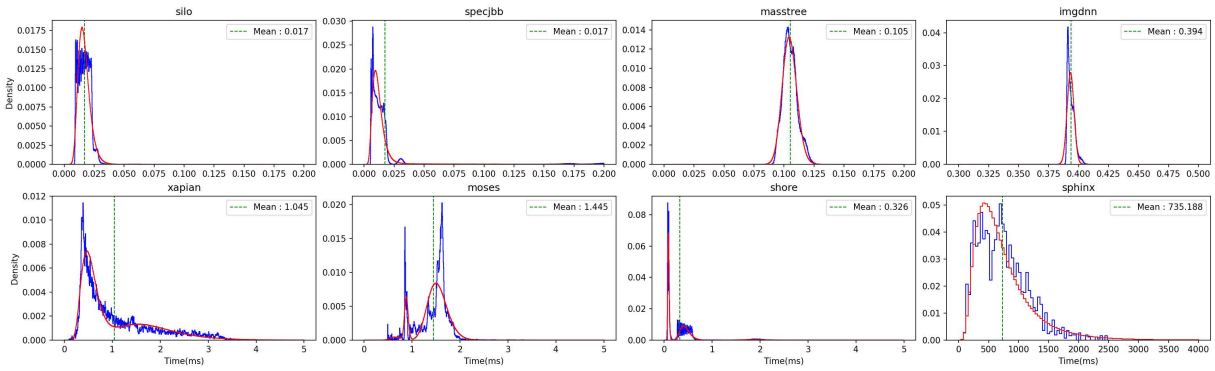


Figure 5. Service Time Distribution: Blue line shows the measured service distribution; Red line shows distribution coverage used in TailWAG workload generation. Python Scipy is used for distribution fitting.

For all experiments, both the round trip latency and service time of every single query are collected from the server system. Afterward, analysis and generation can be performed on any machine. Python Scipy [31] library was used for analysis.

3.1 Service Time Distribution

First, we break down application service time, which consists of active CPU time spent on computation, memory access such as caches, and I/O. CDF (Cumulative Distribution Function) was reported in the Tailbench paper [18]. To better understand and analyze the difference statically, we use PDF (Probability Density Function). This also helps us abstract characteristics into TailWAG parameters. As shown in Figure 5, we can build PDF from:

- one centralized (low variance) continuous sample distribution: silo, over 99% are under 40 µs; masstree, over 99.9% are within 75 – 150 µs; img-dnn, all service times are within 0.35 – 0.45 ms.
- one wide continuous sample distribution: xapian, spanning across 0 – 4 ms; moses, over 99% ranging from around 0.6 – 3 ms; sphinx, ranging up to 3 s.
- two distributions: specjbb, around 94% are under 25 µs, 3% are between 25 – 40 µs; shore, in which around 50% are less than 200 µs, 45% are within 300 – 800 µs.
- additional distributions, to give 100% coverage, for instance for specjbb: around 3% of requests are within 150 – 250 µs. Details are listed in Table 2.

1/(Service time) indicates the theoretical upper bound for how many requests the server can handle, i.e. shorter service time leads to a higher achievable load QPS (queries per second).

In addition to service time, round trip latency in networked configurations includes TCP stack and NIC (Network Interface controller) latency. The round-trip latency starts when the query is generated on the client side and ends when the client receives the response from the server.

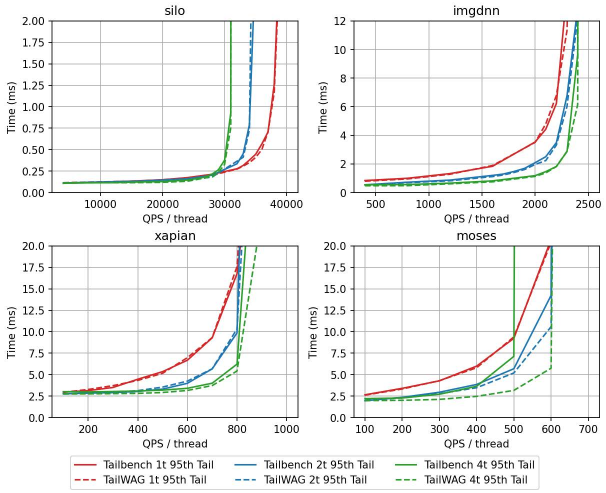


Figure 6. Thread Scaling (95th tail) for 1/2/4 server threads. Dashed lines show TailWAG, and solid lines are measured. TailWAG matches behavior well in all cases except moses, where it reflects an upper bound without its memory bottleneck. TCP contention impacts scaling for silo.

Networking activities affect the server as load increases and can lead to cascading effects.

3.2 Thread Scaling

Server applications are written as multithreaded applications for higher throughput, utilizing multiple cores or even multi-node server systems. Even with well-designed multithreaded applications, it is difficult to achieve linear scaling on performance due to resource contention. There are unavoidable minimum penalties that can be as small as microseconds.

Not all Tailbench applications scale up well for different reasons. Half of the applications have close to linear scaling: masstree, img-dnn, xapian and sphinx, since they have

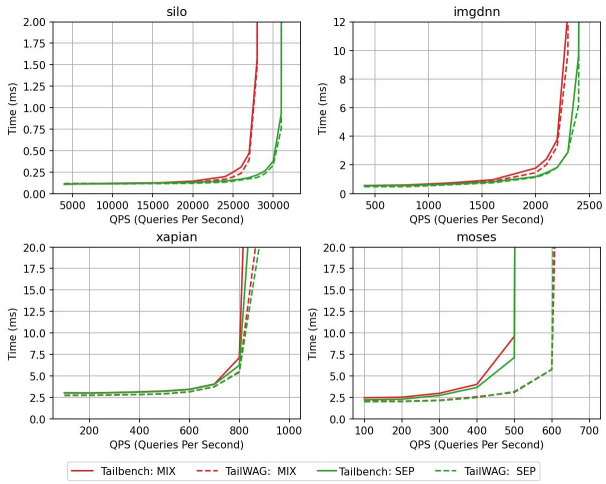


Figure 7. NIC disturbance (95th tail): Effect of IRQ Handling Noise on Tail Latency using TailWAG. TailWAG can capture effects from different applications, and also predict a memory bottleneck for moses.

longer service and less critical section sharing within the application. Due to open loop configuration (multiple clients to multiple server threads), using more threads can hide some longer service requests. Figure 6 shows selected examples.

Applications that are not scaling well include silo, specjbb, shore, and moses. Moses was reported to have a memory bottleneck in Tailbench [18]. Shore was only provided with a single warehouse database, which cannot scale as intended. Silo and specjbb share the same root cause as *TCP send()/recv()* behave as a high-contention critical section due to very short service times.

Executing TCP stack code as part of the harness can be easily overlooked when deployed. Multiple threads share a server port in the Tailbench harness, requiring serialization. Measurements show that networking system calls including recv() and send() cost from 3 µs to over 10 µs, depending on the size of the request and response. This latency is harmless for applications that have mean latency at a millisecond scale but can be crucial for small service time applications, such as specjbb and silo, where mean service latency is under 100 µs.

3.3 NIC Interrupts

As shown in Figure 4, server threads are perturbed by asynchronous interrupts, which we refer to as NIC noise. However, the effects are heavily workload-dependent. Here, we expand the analysis to include three more applications running on four cores, but with different allocation strategies: MIX, where interrupt handling is bound to the same core as server threads; and SEP, where interrupt handling is bound to a dedicated core. Results are shown in Figure 7. Applications like silo and img-dnn, with low variance service time, are more sensitive to interrupts; on the other hand, xapian and moses, with higher variance service time, are less sensitive to interrupts.

```
void doRun(){
  //setup parameters
  ...
  //each iteration is one query
  while (true) {
    recvRequest();

    lock(recvLock);
    spin(recvLockCycles);
    unlock(recvLock);

    p=uniform_random(0.001%,100%);
    if p < prob1
      distCycles=log_random(mean1,var1);
    else if p < prob1+prob2
      distCycles=log_random(mean2,var2);
    else
      distCycles=log_random(mean3,var3);
    spin(distCycles);

    //can be multiple noise injection
    spin(noiseCycles);

    lock(sentLock);
    spin(sentLockCycles);
    unlock(sentLock);

    sendResponse();
  }
}
```

Listing 1. Generated Workload Pseudocode. Each loop represents a request sequential timing behavior.

4 TailWAG Generation and Validation

TailWAG does not try to capture all the details of each application from Tailbench, including data movement, cache accesses, loop constructs, or other control flow. Instead, we use the timing distributions that are learned as discussed in Section 3. Each target application run with the networked setup is treated as a black box, and only service time and total time are captured as metrics.

The generated workload is written as an individual application that occupies the CPU and exhibits the same characteristics as the real workload in terms of CDF (cumulative distribution function) of round trip service time, as shown in Listing 1. For each query, it spins for cycles specified by different components, representing occupying CPU. It also acquires and releases a lock to represent a critical section. Key parameters are:

Main execution distributions: these are the main parameters that determine the active service time for a request.

Table 2. Parameters used for workload generation

Application	Distribution 1 Mean (Variance)	Distribution 2 Probability, Mean (Variance)	Distribution 3 Probability, Mean (Variance)	Critical Section 1 Mean (Variance)	Critical Section 2 Mean (Variance)
silo	16.14 µs (0.288)	0.1%, 59.6 µs (0.085)	NA	3.5 µs (0.15)	3.5 µs (0.15)
specjbb	11.1 µs (0.39)	3%, 30.73 µs (0.066)	3%, 200 µs (0.049)	3.9 µs (0.1)	3.9 µs (0.1)
masstree	105 µs (0.06)	0.001%, 1400 µs (0.05)	NA	3.5 µs (0.2)	3.5 µs (0.2)
img − dnn	395 µs (0.007)	NA	NA	6.4 µs (0.5)	6.4 µs (0.5)
shore	92.8 µs (0.163)	45%, 432 µs (0.24)	4.2%, 1900 µs (0.124)	3.9 µs (0.1)	3.9 µs (0.1)
xapian	532 µs (0.3)	37.8%, 1800 µs (0.35)	NA	4.2 µs (0.1)	4.2 µs (0.1)
moses	840 µs (0.133)	85.7%, 1525 µs (0.14)	0.05%, 11450 µs (0.54)	4.9 µs (0.1)	4 µs (0.1)
sphinx	624.28 ms (0.510)	3%, NA	NA	50 µs (0.6)	50 µs (0.6)

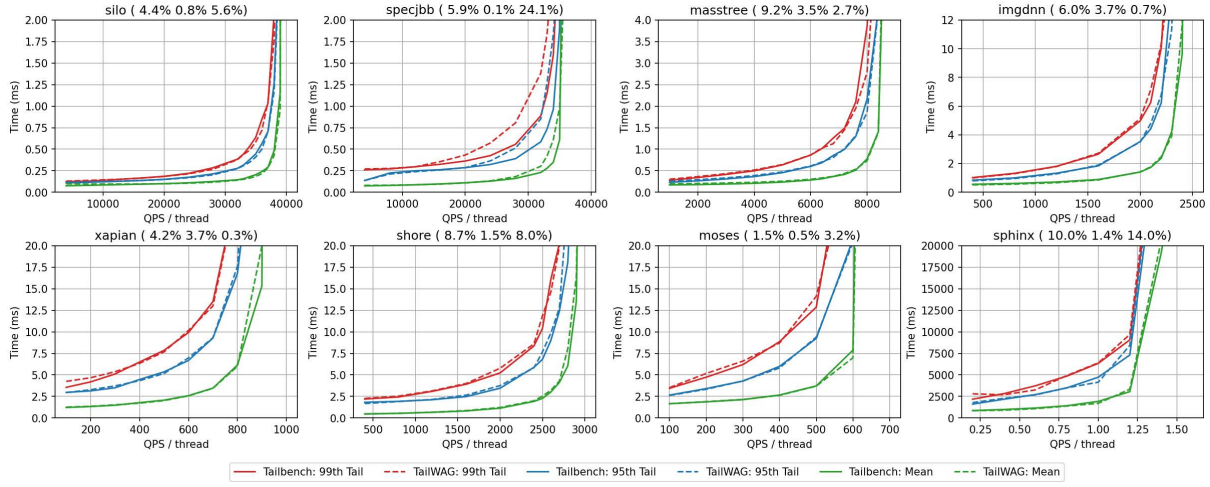


Figure 8. Single-threaded TailWAG Validation: Dashed lines show TailWAG. Numbers next to the application report latency error at 20%/50%/70% load, with an average error of 6.2%/1.9%/7.3% respectively.

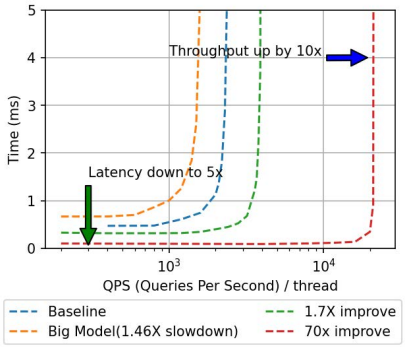
During the study, we are using three distribution components derived from characterizing the applications. These are represented as a vector of distributions, with corresponding means, variances, and probabilities (summing to one). We found three are enough for these applications, but this can easily be expanded to more if necessary. Statistical functions from the Python Scipy package provide continuous probability function fitting [31], giving the best estimation of mean and variance for a set of data.

Threads: using multiple worker threads helps with system utilization, keeping fewer resources idle, and increasing overall application throughput in the ideal case [18]. However, more threads also increase the possibility of lock contention.

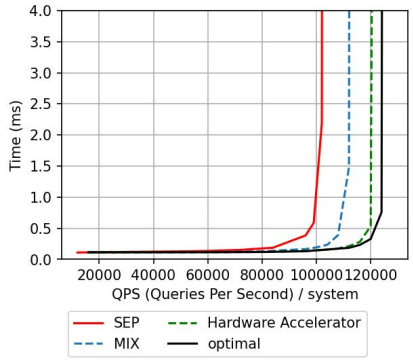
Critical execution time: There are shared resources between threads, corresponding to critical execution time as we discovered (tcp send and recv time). After the non-critical section, every request will try to acquire the lock. A hybrid lock was added that will spin for a fixed time, then go to sleep. After successfully grabbing the lock, the thread will remain busy until it expends the critical section execution time.

Timing disturbance: a timing disturbance happens within the service section. There can be multiple timing disturbances. Small timing disturbance range from tens to hundreds of microseconds, emulating hardware interrupts as the example illustrated in Figure 4. It can also be over milliseconds to seconds, emulating rare events such as I/O, runtime policies (example shown in Figure 3), and garbage collection, discussed more in 5.3.

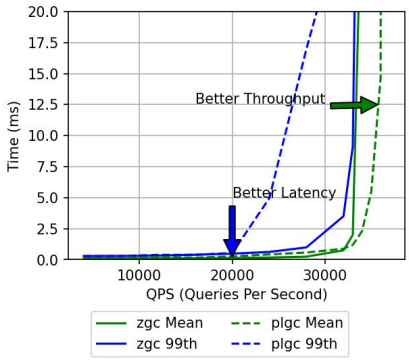
With proper selection and tuning of parameters, TailWAG can mimic the latency distribution from real applications. Parameter values are listed in Table 2, and behavior for single-thread configurations is plotted against measured results in Figure 8 with dash lines. Overall, TailWAG closely matches measured results for both latency (mean, 95th, 99th) and throughput, with the exception of specjbb 99th tail latency, which is caused by background threads that are not modeled by TailWAG. Further validation results are shown in Figure 6 and Figure 7 (dashed vs. solid lines).



(a) DNN evolution: Increased model complexity (orange), vectorization (green), hardware accelerator (red) affect mean latency.



(b) NIC IRQ: SEP/MIX with hardIRQ Accelerator affect 95th tail latency, with same application (silo). The optimal case needs an additional (5th) core.



(c) Java GC: Parallel (plgc, throughput oriented) and Zero (zgc, latency oriented) affect mean and 99th tail latency.

Figure 9. Three case studies. Dashed lines from TailWAG.

5 Case Study and Discussion

TailWAG can now represent a variety of applications using different sets of parameters, and the workflow is shown in Figure 2. Moreover, these parameters can be dependent or independent, upon user specification. Workload generation by changing one of the parameters allows designers and programmers to explore potential changes without fully implementing them. Below, we showcase how to use TailWAG to explore: changes in service time due to algorithm complexity or hardware innovation for img-dnn; NIC IRQ handler acceleration for silo; and infrequent timing hiccups (garbage collection) for specjbb.

5.1 DNN evolution

Machine learning evolves rapidly with algorithm, model, and hardware changes appearing regularly. **img-dnn** within this suite would be affected by such changes. The baseline **img-dnn** runs on a Xeon CPU. TailWAG abstracts hardware details, replacing them with timing components (main distribution of mean 395 µs, from Table 2) reflecting performance. Here we evaluate two changes to the model: Increasing complexity of the ML model for better accuracy and robustness, with more burden on computation; [12] reported there is an annual increase of 1.46X on FLOPs in CNN-based image recognition within Google. Second, hardware innovations that speed up the ML workload; [13] measured there is 1.6x to 70x speedup (compare to CPU) when GPU or TPU are used for running CNN models. TailWAG can capture these trends: GPU/TPU innovation reduces the mean service time by 1.7x to 320 µs or 70x to 7.75 µs. Here, TailWAG only predicts the impact of this particular change, providing insight on other important behaviors (contention, tail latency distribution, etc.). Using our model, we can predict performance benefits as in Figure 9a. As performance increases, img-dnn moves into a realm where microsecond-scale timing disturbances are important, such as thread contention, lock implementation and interrupt handling, similar to silo and specjbb.

5.2 NIC hardIRQ Accelerator

As previously discussed and shown in Figure 1, reserving a core for interrupt handling (SEP vs. MIX) can improve tail latency at the cost of lowered throughput. Here we use TailWAG to examine the SEP and MIX cases and compare them to a hypothetical IDEAL case where interrupts do not perturb the server threads at all, as well as a hypothetical hardIRQ accelerator that services the interrupt with minimal overhead (around 0.5 µs) using a tightly-coupled accelerator (TCA) attached to the CPU core. The TailWAG results in Figure 9b show that such an accelerator has the potential to bridge much of the performance gap to the optimal case, while likely requiring much less hardware than a dedicated core. Detailed exploration of a hardIRQ accelerator is left to future work.

5.3 Garbage Collection

Many applications are written in high-level programming languages that provide garbage collection, such as specjbb in our test suite. Garbage collection itself is a non-trivial problem: there is no single garbage collector that is best for every application, and it should be chosen with trade-offs in mind [34]. There are several garbage collectors used in java; here we picked plgc (parallel, throughput-oriented) and zgc (latency-oriented) to showcase TailWAG. Here, single thread specjbb is modeled in Figure 9c: plgc case has better service latency, but worse "stop the world" events, around 50 ms every 10 seconds; zgc has worse mean service time (around 10%), since garbage collection runs concurrently, but has better "stop the world" events, less than 1ms each, leading to better tail latency distribution.

6 Conclusions and Future work

Tail latency is a crucial performance metric for server workloads. Benchmarks exist, but suffer from limitations and require a significant amount of effort to run as they suffer from run-to-run variation and experimental noise. In this work, we propose TailWAG, a workflow that analyzes tail latency-sensitive workloads, extracts a set of parameters that describe timing behavior; then generates a synthetic workload that closely mimics real application performance, both single load cumulative distribution, and trends under different loads. TailWAG performs well overall, with average 6.2% errors on mean, 1.9% on 95th tail and 7.3% on 99th tail latency respectively. Moreover, using the generated workload and parameter sweeping, we can explore the design space of hardware/software innovations and assess runtime configuration impacts with minimal implementation effort.

Future directions for this work include adding and evaluating additional parameters that may lead to better matches to application behavior and automating the parameter search process by using machine learning techniques. One of the potential additional parameters is QPS-dependent variance, leading to more performance hiccups when the system is at a higher load: longer latency for higher load; higher variance for higher load; higher chance of noise for higher load. Moreover, in addition to the current log distributions, more distributions can be added for generating all latency-related randomization, including uniform and geometric distributions. Shared resource access is limited to two locks (recv and sent) in this paper. In more complicated and detailed cases, lock contention within the application may not be adequately captured. Different threads are not always accessing the same critical section, and an array of locks could represent different shared resources. Both the length of the lock array and the probability of accessing the same lock can be tuned.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This research was supported in part by NSF award CCF-2010830 and AFRL Award FA9550-18-1-0166.

References

[1] Mario Badr and Natalie Enright Jerger. 2014. SynFull: Synthetic traffic models capturing cache coherent behaviour. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, Minneapolis, MN, USA, 109–120.

[2] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, et al. 2017. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Transactions on Computer Systems* 34, 4 (Jan. 2017), 1–39.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (May 2011), 1–7.

[4] Daniel P. Bovet and Marco Cesati. 2006. *Understanding the Linux kernel* (3rd ed ed.). O'Reilly, Beijing ; Sebastopol, CA.

[5] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80.

[6] Christina Delimitrou and Christos Kozyrakis. 2018. Amdahl's law for tail latency. *Commun. ACM* 61, 8 (July 2018), 65–72.

[7] The Linux Kernel documentation. 2022. BPF Documentation. https://www.kernel.org/doc/html/latest/bpf/index.html

[8] The Linux Foundation. 2016. wiki:networking:napi. https://wiki.linuxfoundation.org/networking/napi

[9] Thomas Gleixner and Ingo Molnar. 2010. linux/genericirq.rst at v5.15 · torvalds/linux · GitHub. https://github.com/torvalds/linux/blob/v5.15/Documentation/core-api/genericirq.rst

[10] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. 2007. STMBench7: a benchmark for software transactional memory. *ACM SIGOPS Operating Systems Review* 41, 3 (June 2007), 315–324.

[11] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, et al. 2009. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology - EDBT '09*. ACM Press, Saint Petersburg, Russia, 24.

[12] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, et al. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 1–14.

[13] Norman P. Jouppi, Cliff Young, Nishant Patil, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, Toronto ON Canada, 1–12.

[14] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, et al. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, Portland Oregon, 158–169.

[15] Svilen Kanev, Kim Hazelwood, Gu-Yeon Wei, and David Brooks. 2014. Tradeoffs between power management and tail latency in warehouse-scale applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Raleigh, NC, USA, 31–40.

[16] Sagar Karandikar, Howard Mao, Donggyu Kim, et al. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Los Angeles, CA, 29–42.

[17] Sagar Karandikar, Albert Ou, Alon Amid, et al. 2020. FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne Switzerland, 715–731.

[18] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Providence, RI, USA, 1–10.

[19] Philipp Koehn, Richard Zens, Chris Dyer, et al. 2007. Moses: open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions - ACL '07*. Association for Computational Linguistics, Prague, Czech Republic, 177.

[20] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*. ACM Press, Amsterdam, The Netherlands, 1–14.

[21] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, Seattle WA USA, 1–14.

[22] Li Deng. 2012. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Processing Magazine* 29, 6 (Nov. 2012), 141–142.

[23] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, et al. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs].

[24] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12*. ACM Press, Bern, Switzerland, 183.

[25] Nathan Pemberton and Alon Amid. 2021. FireMarshal: Making HW/SW Co-Design Reproducible and Reliable. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Stony Brook, NY, USA, 299–309.

[26] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. (June 2013), 12.

[27] spec. 2015. specjbb. https://www.spec.org/jbb2015/

[28] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. 2019. SoftSKU: optimizing server architectures for microservice diversity @scale. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, Phoenix Arizona, 513–526.

[29] Akshitha Sriraman and Thomas F. Wenisch. 2018. μSuite: A Benchmark Suite for Microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Raleigh, NC, 1–12.

[30] Stephen Tu, Wenting Zheng, Eddie Kohler, et al. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, Farminton Pennsylvania, 18–32.

[31] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* 17, 3 (March 2020), 261–272.

[32] Willie Walker, Paul Lamere, Philip Kwok, et al. 2004. Sphinx-4: a flexible open source framework for speech recognition. (2004), 18.

[33] xapian. 2022. https://xapian.org

[34] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Transactions on Programming Languages and Systems* 44, 4 (Dec. 2022), 1–34.

[35] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. 2016. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Seoul, South Korea, 456–468.