Treadmill: Tail Latency Measurement at Microsecond-level Precision

Yunqi Zhang
Johann Hauswald
David Meisner
Jason Mars
Lingjia Tang
Welcome

Section 1: Tail latency
08:40 ~ 09:00 — Overview of data centers and tail latency
09:00 ~ 09:30 — Common pitfalls for measuring tail latency

Coffee break

Section 2: Treadmill
10:00 ~ 10:20 — Overview of Treadmill
10:20 ~ 10:40 — Tail latency measurements with Treadmill

Section 3: Beyond just Treadmill
10:40 ~ 11:00 — Extending Treadmill to support your applications
11:00 ~ 11:20 — New applications enabled by Treadmill
Overview
Data centers

• Built in massive sizes
  • Several football fields

• Expensive to construct
  • 100s of millions capital cost¹
  • Consume huge amount of energy
    • 3% of global energy supply²

¹The Datacenter as a Computer
²Global warming: Data centres to consume three times as much energy in next decade, experts warn
Internet services

- User interactive applications
- Powered by large-scale distributed systems
- Millions of user queries hitting the servers
Latency

waiting time + eating time

eating time

latency distribution
Tail latency

- Often orders of magnitudes higher than average
- Directly affects user experience
- Resources are often provisioned based on tail latency
“It is challenging for service providers to keep the tail of latency distribution short for interactive services as the size and complexity of the system scales up.”

— Jeffrey Dean, Luiz Barroso
“The Tail at Scale”
Google
Challenges

• Tail latency is sensitive to any variance in the system

• Many services operate at latency as low as microseconds

• Many architectural components are involved
Tail latency measurements
Tail latency measurements

PARSEC
[pär-"sek] A unit of measure

MiBench

CloudSuite

Yahoo Cloud Serving Benchmark

TPC Transaction Processing Performance Council

BigDataBench
Are we doing this right?
Common Pitfalls in Tail Latency Measurements
Tail latency measurements
Tail latency measurements

load tester

network

server
Tail latency measurements

load tester → network → server
Tail latency measurements

load tester → network → server
Tail latency measurements

load tester → network → server
Tail latency measurements
Tail latency measurements

load tester  network  server
Tail latency measurements

- load tester
- network
- server

 latency distribution

**tail latency**
Client-side queueing bias

load tester  network  server
Client-side queueing bias

![Diagram showing client-side queueing bias with load tester, network, and server components.](image-url)
Client-side queueing bias

Multi-Client Setup

Normalized Latency

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Server-Side Latency</th>
<th>Client-Side Latency</th>
<th>Network Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>70.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>75.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>85.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Treadmill: Tail Latency Measurement at Microsecond-Level Precision
Multiple clients are needed to avoid client-side queueing bias.

Client-side queueing bias

Multi-Client Setup

- Server-Side Latency
- Client-Side Latency
- Network Latency

Utilization

Normalized Latency

70.0%  75.0%  80.0%  85.0%  90.0%  95.0%
Query inter-arrival generation

[Closed v.s. Open System Models. 'NSDI 2006]
Query inter-arrival generation

[Closed v.s. Open System Models. ‘NSDI 2006]

closed-loop

open-loop
Query inter-arrival generation

[Closed v.s. Open System Models. ‘NSDI 2006]

closed-loop

open-loop
Query inter-arrival generation

[Closed v.s. Open System Models. ‘NSDI 2006]

closed-loop

send()

open-loop

async_send()
Query inter-arrival generation
Query inter-arrival generation
Query inter-arrival generation
Query inter-arrival generation

Open-Loop

Closed-Loop

underestimate the tail
Query inter-arrival generation

Open-Loop is necessary to properly exercise the system queueing behavior of the system.

Open-loop is necessary to properly exercise the system queueing behavior.
Statistical aggregation
Statistical aggregation
Statistical aggregation
Statistical aggregation

Client 2  Client 3  Client 4  Client 1

Probability Density Function (PDF)

Latency (us)
0.0%  20.0%  40.0%  60.0%  80.0%  100.0%

90th Percentile  99th Percentile  99.9th Percentile

Client 4  Client 3  Client 2  Client 1
Statistical aggregation
Statistical aggregation

GET(key)

SET(key, value)
Statistical aggregation

GET(key)

SET(key, value)
Statistical aggregation

GET(key)

SET(key, value)

GET(key)

SET(key, value)

tail latency
Statistical aggregation must be performed **carefully**.
Performance hysteresis
Performance hysteresis

More samples are probably going to help
Performance hysteresis

![Graph showing the 99th percentile latency over the number of samples.](image)
Performance hysteresis

Sufficient amount of samples are needed to get to the steady state
Performance hysteresis

![Graph showing 99th percentile latency over number of samples for different runs. The graph includes lines for Run #0, Run #1, Run #2, Run #3, and the average (Avg.).]
Performance hysteresis

More samples **do not help** when underlying system state is changing
Treadmill
How we started

- Precise tail latency measurements for Memcached
  - How will my change impact the performance?
  - How many servers should we buy next year?
- Collected extensive statistics on production traffic
- None of existing software delivered the precision
Building Treadmill

- Started in December 2013
- Deployed for Memcached in June 2014
- Extend to more services
Project status

- Open sourced in June 2016
- Paper published at ISCA 2016
- Maintained by Facebook
- Several research projects
Tail Latency Measurement Procedure
Overview

4. Statistical aggregation

Server 1

Clients (Treadmill)

Server 2

1. Queueing Bias

3. Inter-arrival Generation

2. Performance Hysteresis

Server (Hosting Service)

Stats

Stats
Overview

4. Statistical aggregation

Server 1

Clients (Treadmill)

Server 2

3. Inter-arrival Generation

2. Performance Hysteresis

1. Queueing Bias

Stats

Server (Hosting Service)

Stats
Avoid client-side queueing bias

• Setup
  • Target total QPS
  • Multiple Treadmill instances
  • Monitor Treadmill utilization

• Analysis
  • >20% utilization -> increase Treadmill instances
Avoid client-side queueing bias

- Implementation details
  - Lock-free
  - Inline execution for immediate response processing
Avoid performance hysteresis

• Setup:
  • Launch Treadmill
  • Monitor P99s for convergence

• Analysis:
  • Underlying architectural state could be changing if no convergence across runs

• Solutions:
  1. Pinpoint architectural changes
  2. Average across multiple runs
Inter-arrival Query Generation

- Setup:
  - Exponentially distributed
  - Included in Treadmill!

- Analysis:
  - Validated

---

Figure 12: Distribution of time between query arrivals.

---

Inter-arrival Query Generation

Scheduler.cpp

```cpp
#include <omitted>

namespace omitted

double Scheduler::randomExponentialInterval(double mean) {
    static std::mt19937* rng = new std::mt19937();
    std::uniform_real_distribution<double> dist(0, 1.0);
    /* Cap the lower end so that we don't return infinity */
    return -log(std::max(dist(*rng), 1e-9)) * mean;
}

void Scheduler::waitNs(int64_t ns) {
    /* We need to have *precise* timing, and it's not achievable with any other means like 'nanosleep' or EventBase.
     * "pause" instruction would hint processor that this is a spin-loop, it will burn as much CPU as possible. The processor will use this hint to avoid memory order violation, which greatly improves its performance. */
    for (auto start = nowNs(); nowNs() - start < ns;) {
        asm volatile("pause");
    }
}
```

Statistical aggregation

• Setup:
  • Each TM instance generates its own distribution
  • Dynamically binned histograms

• Analysis:
  • Confirm shapes
Collecting Latency

- Collect all samples (naive approach)
- Static histograms (better)
  - Bins for latencies, increment counter for each
  - May be too coarse since bins are fixed

Increasingly large memory footprint!

How to choose bins?
Collecting Latency

- Dynamic histograms (Treadmill) using 3 states
  1. Warmup: steady-state queueing
  2. Calibration: set min/max latency as bounds of histogram
  3. Measurement: increment bin counter

1024 bins (default)

Adjust min/max as needed
Interpreting Results

Workload:
Sentiment Analysis Web Service
(Natural Language Processing)

**Throughput**
N Samples: 977
Average: 2495.86 +/- 22.5995
Min: 0
Max: 3306.88
50% Percentile: 2507.18
90% Percentile: 2757.9
95% Percentile: 2877.82
99% Percentile: 3196.9
Min Bin 1168.43
Max Bin 5697

**Request_latency**
N Samples: 249541
Average: 14117.3 +/- 171.81
Min: 2815.05
Max: 617634
50% Percentile: 5514.8
90% Percentile: 8074.41
95% Percentile: 35034.9
99% Percentile: 246403
Min Bin 4064.53
Max Bin 524288

• Stat types:
  • Count (samples, exceptions)
  • Continuous (latency, throughput)

• Statistics generated

• Class to extend stats collected
Precise measurements

![Diagram showing the process of load testing with tcpdump and Memcached]

- Load tester
- tcpdump
- Network
- Memcached
- Server

Precise measurements
Measurement

- 100,000 QPS

- 800,000 QPS
Measurement

- 100,000 QPS

- 800,000 QPS
Extending Treadmill
Structure

• Top-level files shared across workloads
  • Stat collection
  • Query generation
  • Treadmill configuration
• Guarantees against the 4 pitfalls described
Structure

- Only need to implement workload specific code
- Web service (RPC) framework agnostic
- Encapsulated in 3 files
• Request types

```c
enum Operation {
    GET,
    SET,
    DELETE
};
```
Connection.h

```cpp
#include<hostname>
DECLARE_int32(port);

namespace omitted

class Connection<MemcachedService> {  
public:
    explicit Connection<MemcachedService>(folly::EventBase& event_base) {
        std::string host = nsLookUp(FLAGS_hostname);
        ConnectionOptions opts(host, FLAGS_port, mc_ascii_protocol);
        client_ = std::make_unique<AsyncMcClient>(event_base, opts);
        ...
    }

    // Implement each request type
    folly::Future<MemcachedService::Reply> sendRequest(std::unique_ptr<typename MemcachedService::Request> request) {
        ...
        if (request->which() == MemcachedRequest::GET) {
            // send synchronous GET request
            //
            auto req = std::make_shared<McGetRequest>(request->key());
            fm_->addTask([this, req, p] () mutable {
                client_->sendSync(*req, std::chrono::milliseconds::zero());
                p->setValue(MemcachedService::Reply());
            });
        } else if (request->which() == MemcachedRequest::SET) {
            // send SET request
            //
        } else {
            // send DELETE request
            //
        }
        return f;
    }
```
• Determines the state (e.g., warm-up, or measurements)

• Determines the workload mix (e.g., 80% GET, 20% SET) by passing in a JSON config file

```cpp
#include <omitted>

// define warm-up samples
DECLARE_int64(number_of_keys);

// define warm-up samples
DECLARE_int64(number_of_keys);

namespace omitted

template <>
class Workload<MemcachedService> {
public:
  enum State {
    MEMCACHED_WARMUP,
    GET
  };
...

std::tuple<std::unique_ptr<MemcachedService::Request>,
           Promise<MemcachedService::Reply>,
           Future<MemcachedService::Reply>>
getNextRequest() {
  ...;

  std::unique_ptr<MemcachedService::Request> request;
  if (state_ == State::MEMCACHED_WARMUP) {
    request =
      std::make_unique<MemcachedRequest>(MemcachedRequest::SET,
                                         std::move(key));
    request->setValue(std::to_string(index_));
    if (index_ == FLAGS_number_of_keys - 1) {
      LOG(INFO) << "WARMUP complete";
      state_ = State::GET;
    }
  } else if (state_ == State::GET) {
    request =
      std::make_unique<MemcachedRequest>(MemcachedRequest::GET,
                                         std::move(key));
  }
  ...;

  return request;
}
```
Boilerplate Code

- Treadmill.cpp
- ExampleService.h
- Makefile.am
New Applications
Research — Complex Interactions

• Understand the impact of complex architectural factors and their interactions

• NUMA [NUMA Experience ’HPCA 2013, Tales of the Tail ’SoCC 2014]

• NIC [Architecting to Achieve a Billion Request per Second Throughput ’ISCA 2015, Tales of the Tail ’SoCC 2014, Chronos ’SoCC 2012]


Architectural components have complex interactions
Tail latency attribution

- Architectural components
  - NUMA allocation policy
  - DVFS frequency governor
  - TurboBoost
  - NIC interrupt handling
Complex interactions

- NIC IRQ policy
  - same-node
  - all-nodes
- DVFS Governor
  - ondemand
  - performance

NIC IRQ same-node

more frequency transition overhead

NIC IRQ all-nodes

more frequency transition overhead
Complex interactions

- NIC IRQ policy
  - same-node
  - all-nodes
- DVFS Governor
  - ondemand
  - performance

Tail latency attribution enables understanding of complex interactions
Tail latency reduction

43% reduction in 99th-percentile latency
93% reduction in its variance
Research — Benchmarking Web Services

• Steps:

  1. Encapsulate application into Thrift RPC as web service

  2. Implement Treadmill requirements and define dataset used

• Multiple projects using Treadmill in our group

  • Investigating DNN based NLP applications
Industry - Give access to HW vendor

- HW vendors can test and provide reliable measurements to customers
- Perform similar analysis as previously outlined to find complex interactions
Industry - Performance A/B test

- Code change -> run A/B test to infer the performance impact (part of code review)
- Catch performance regressions before production
- Lower overhead than testing in production
Industry - Capacity Planning

- Find the maximum throughput each server can sustain given a tail latency requirement
Industry - Replaying production trace

• Record production traffic

• Store into a JSON workload configuration file

• Use treadmill to replay the production traffic

• Realistic inter-arrival rate to measure performance or debug hard-to-reproduce bugs
Q&A
Thank you!

• Treadmill: https://github.com/facebook/treadmill

• Tutorial: http://treadmill.eecs.umich.edu