Lecture 16: Dynamic Analysis and Testing 3

CS 5150, Spring 2026

Announcements

- April 14: In-class activity: Bug Bounty
- April 16: In-class exam 2: Syllabus Lecture 9 – 16 (until Apr 9, Thu)
- Report #3: Apr 25 (Same structure as Report #2)

Previously

- Styles of testing: Exploratory, Smoke Tests, Black/White box, …
  - Unit, Integration, Regression Testing, …
- Flaky vs Brittle Tests
- Behavior Driven Development
- Test Doubles/Stubbing/Mocking
- Chaos Engineering
- CI/CD Pipelines

Some important terms

- **CI (Continuous Integration):** A development practice where developers frequently merge code changes into a shared repository, and each change is automatically built and tested.
- **CD (Continuous Delivery/Deployment):** Code is always in a deployable state (but deployment is manual) or every change that passes tests is automatically deployed to production
- **CB (Continuous Build):** The process of compiling and packaging the software automatically after changes.
- **RC (Release Candidate):** A version of software that is potentially ready for final release, pending final validation (can be multiple)
- **Final RC/Production Release**: Version that is actually released in production

CI decisions

- *How* to compose systems along release workflow
- *Which* tests to run *when* along release workflow
- Typical setup
  - **Pre-submit** test suite gates all merges
    - Compilation and fast tests relevant to affected code
  - **Post-submit** test suite verifies subset of commits on trunk
    - Contains larger, more integrated tests
    - Blesses commits that pass as "green"
  - **Release promotion** pipeline verifies candidates for release
    - Contains even larger tests, may require dedicated resources

Automation, speed, & infrastructure

- Builds, tests, and deployment must be automated and reliable
  - Ideally completely **reproducible**
- Most steps must be fast to avoid impeding productivity
  - **Cache** build products
  - **Skip** unaffected tests
  - **Parallelize** & invest in compute resources
- Benefits from tooling
  - Integration with version control and code review
    - Pre-merge and pre-release gates
    - "Last-known-good" branch (new work should branch from here, not trunk)
  - **Bisect** breakages
  - Log all results
  - Automatically **rerun flaky tests**

Dynamic analysis

Common dynamic analysis tools

- Coverage
- Debuggers
- Memory checkers
- Sanitizers
- Profilers

Fuzz testing

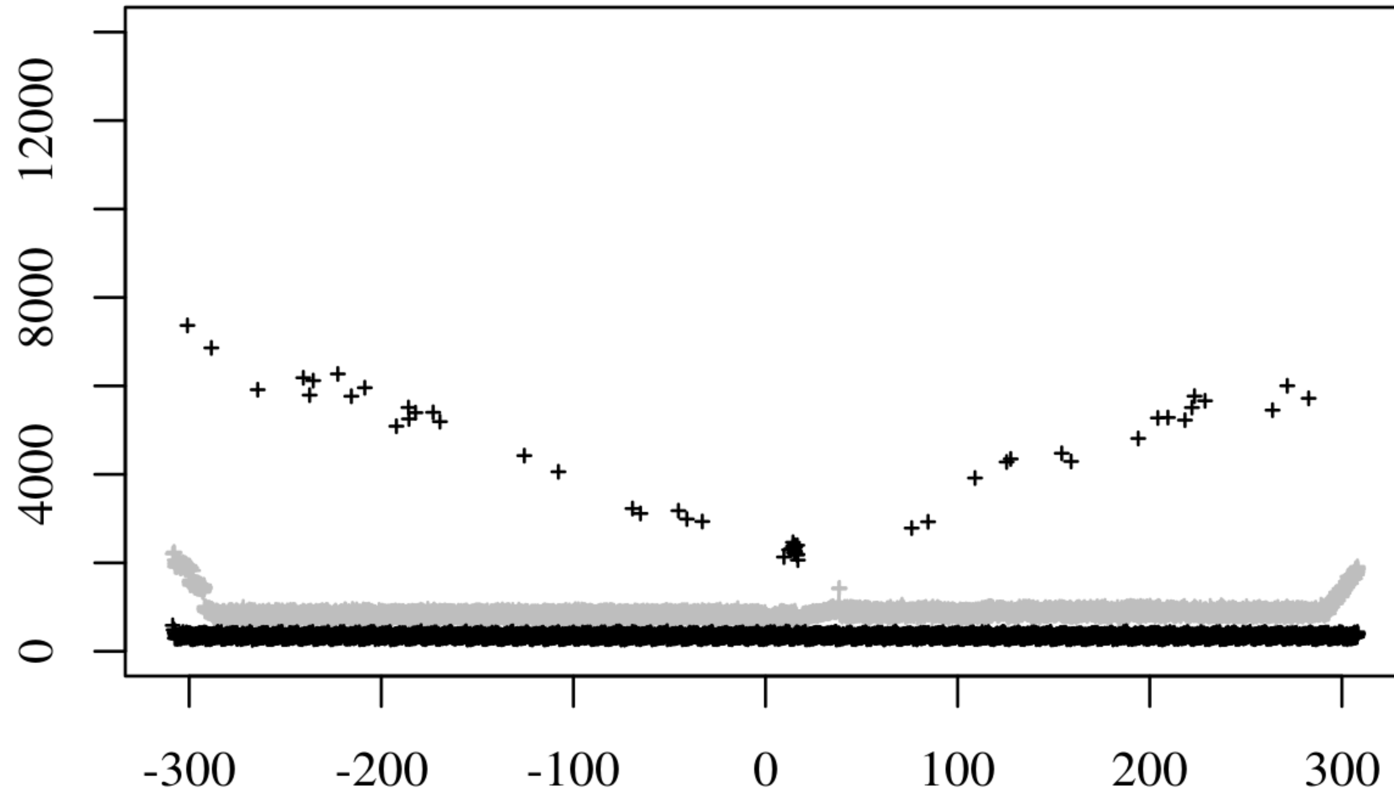
- Give program random input, look for crashes, assertion violations
- Increased in popularity in 2010s; very effective at finding security vulnerabilities
- Can be enhanced with coverage feedback
  - Use genetic algorithms, neural networks to construct input that exercises particular branches

What is a performance bug?

Avoid premature optimization!

- Does not meet deadlines / satisfy SLA
- Responsiveness, smoothness do not meet requirements
  - 100 ms: GUI
  - 15-30 ms: Animation (30-60 fps)
  - 10 ms: MIDI, VR
- Unexpected slowdown for certain inputs / DoS vulnerability
- Performance regression (gradual and acute degradation)
- Performance variability across platforms
- Sub-optimal throughput for HPC

Example of non-uniform performance



Performance testing challenges

- How much room for improvement is there?
  - **Amdahl's law**: Limits to speedup from parallelization, local optimization
  - **Roofline analysis**: Do you expect to be limited by bandwidth or compute?
- Is slowdown localized, dispersed, or emergent?
- Getting reliable measurements is difficult
  - Inconsistency, load dependency, **JIT compilation**, non-representative workloads, intrusive tooling
  - Average case vs. worst case, tail metrics
  - Tension between **latency** and **bandwidth**

Latency vs. throughput

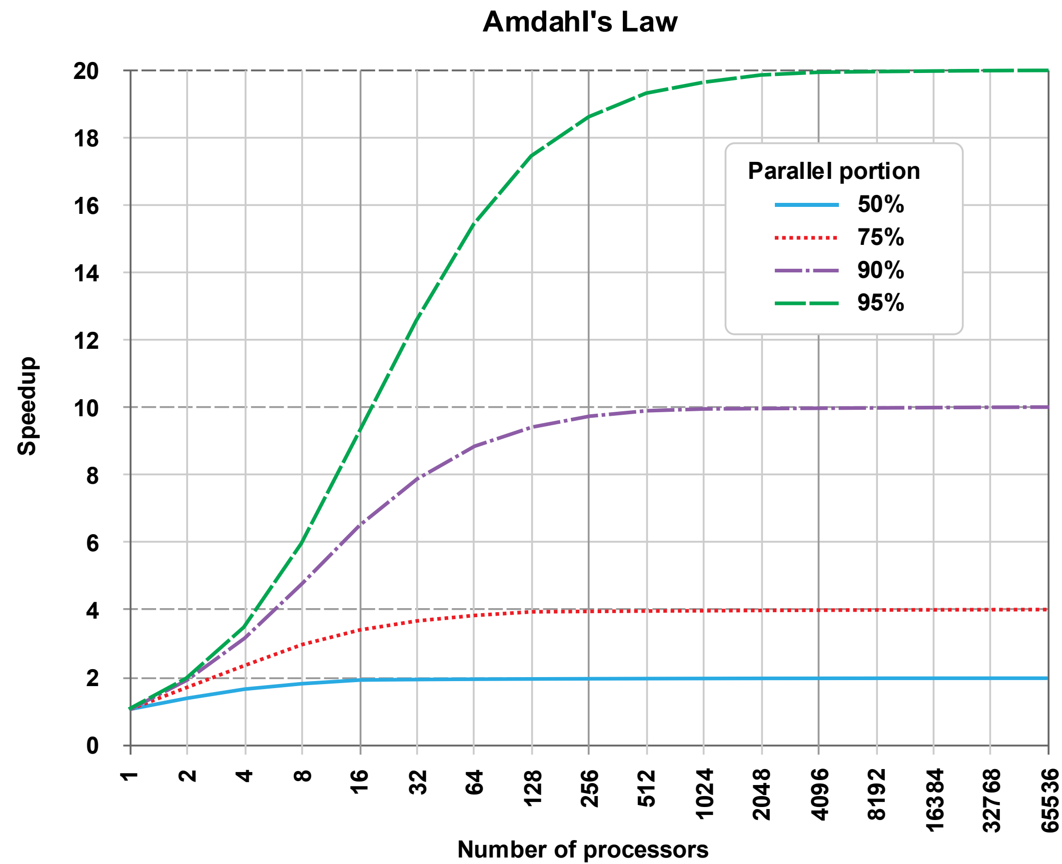
- **Latency**: Duration between a single trigger and the system's response
  - "Tail latency" (e.g., 95th percentile under a specified load) is more important than average
  - A single slow response can leave a negative impression on user!
  - SLA (Service Level Agreement) compliance: “99.9% of requests must complete within 200ms”

- **Throughput**: Time it takes to process a fixed amount of work
  - Often a function of workload
    - Typically, throughput increases with workload size up to a saturation point
  - Reduce overhead with batching
    - Typically at expense of latency

Amdahl's Law

- **Speedup**: $S = T_before / T_after$
- Identify portion $\boldsymbol{p}$ of runtime cost amenable to optimization
  - $T_before = p*T + (1 - p)*T$
- Let s be speedup of optimization on this portion
  - **Example**: $s = 10$ for parallelizing on a 10-core machine
  - Often interested in limit as $s \to \infty$
- $T_after = p*T/s + (1 - p)*T$
- $S(s) = 1/(1 - p + p/s)$
- **$S \to 1/(1 - p)$**

Amdahl's Law implications



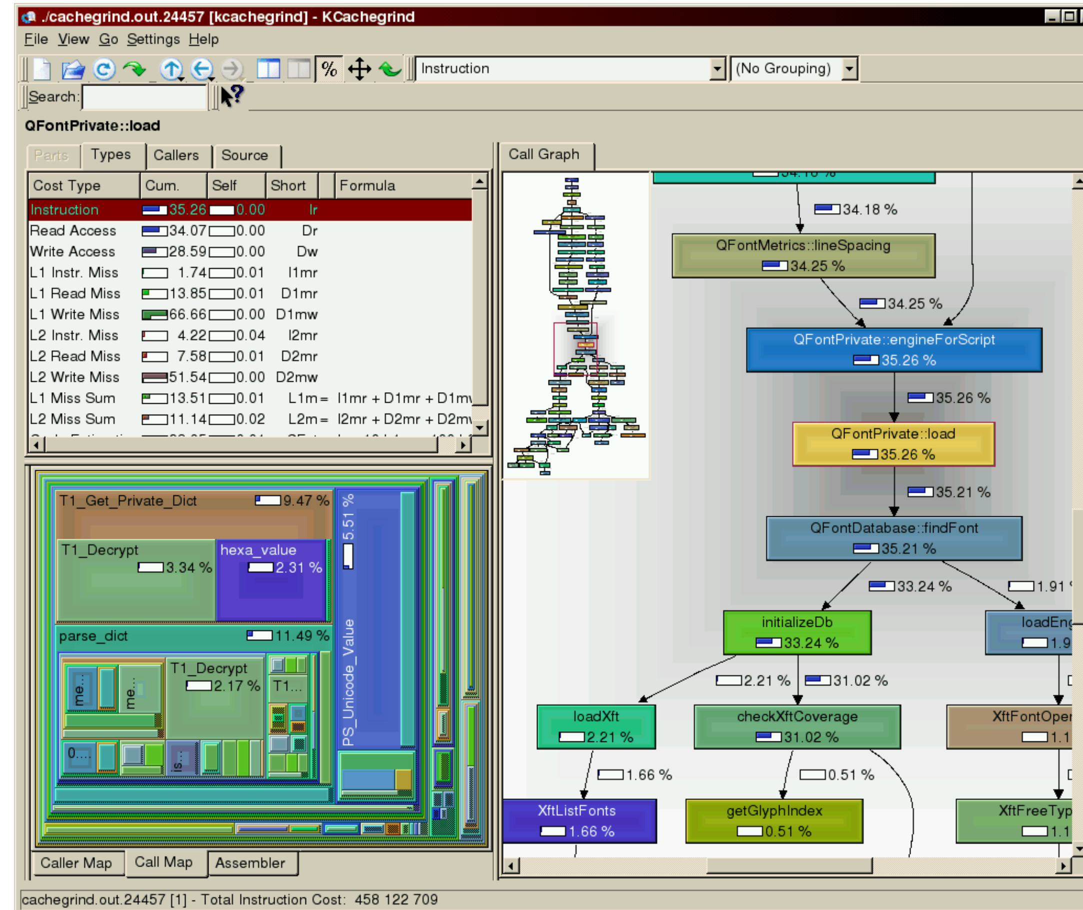
CC BY-SA Daniels220 @ English Wikipedia

Profiling

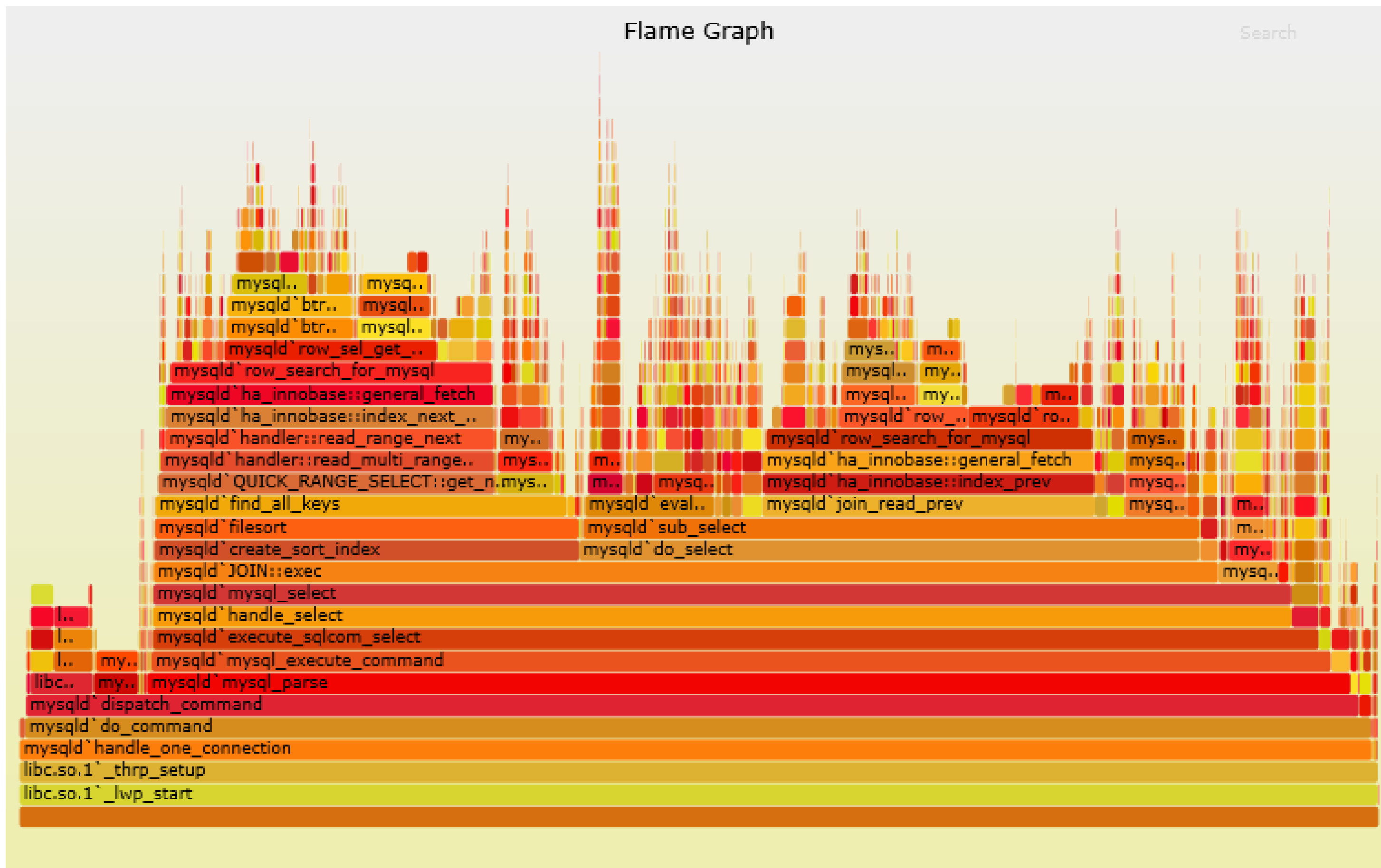
- How can we estimate *p*?
- Where should our optimization efforts be focused?

- Profiling techniques
  - Sampling: Periodically interrupt process and examine stack trace
    - Low overhead
    - Incomplete data
  - Tracing: Record whenever a function is called or returns
    - High overhead
    - Complete function counts
    - Timing may be distorted
  - Instruction-level: Estimate cost of each statement
    - Requires CPU model

callgrind/kcachegrind: tracing & instruction-level



Flame graphs



Example: CPU flame graph, showing MySQL codepaths that are consuming CPU cycles, and by how much.

https://www.brendangregg.com/flamegraphs.html

Browser profilers