Lecture 17: Dynamic Analysis and Testing 3

CS 5150, Spring 2025

Administrative Reminders

- Sample Questions (for in-class exams) are available on Canvas. Solutions will be shared early next week.

- Teams with external client: Remind your client to submit scores right after meeting/presentation.
  - Course staff will not send any reminders.

Lecture goals

- Leverage continuous integration to boost productivity by "shifting left"
- Leverage dynamic analysis tools to find bugs

Continuous integration ("CI")

- Build and test whole systems regularly
  - Discover issues earlier
  - Reduce integration pain through automation and isolation of issues
  - Test beyond single developer's resources
  - Eliminate reliance on developers' discipline
  - Continuously monitor readiness of code
- Applies to both development and release
  - Continuous Build + test
  - Continuous Delivery

CI/CD Terms

- **Continuous Build (CB)** integrates the latest code changes at head and runs an automated build and test.

- **Continuous Delivery (CD)**: a continuous assembling of release candidates, followed by the promotion and testing of those candidates throughout a series of environments—sometimes reaching production and sometimes not.

- **Release candidate (RC):** A cohesive, deployable unit created by an automated process, assembled of code, configuration, and other dependencies that have passed the continuous build.

Read how Google manages CI: https://xgwang.me/google-ci

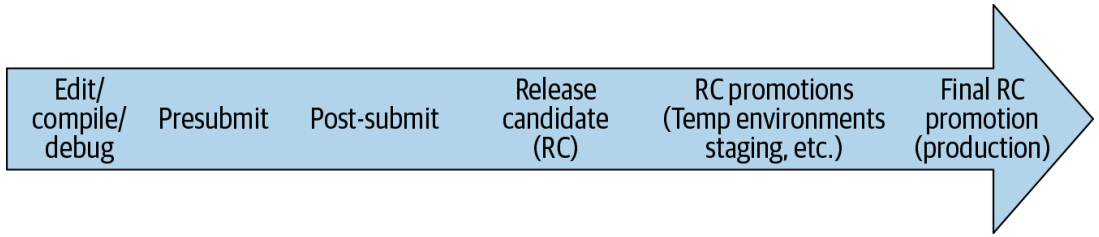
CI Decisions

- *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, and may require dedicated resources
- **Mid-air Collision**: Two changes touching different files causing a test to fail

Shift left

Heavyweight

Lightweight



Advantages of Lightweight: Fast Feedback Loops!

Poll: pre-submit vs. post-submit tests

PollEv.com/cs5150sp25

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

Multi-system CI

- Without monorepo, need to assemble system from several asynchronously-versioned repositories
- Large integration tests can't check every revision/combination
- **Objective**: identify "configurations" (revision combinations) suitable for promotion (larger-scale testing, release)

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

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 datasets, intrusive tooling
  - Average case vs. worst case, tail metrics
  - Tension between latency and bandwidth

Latency vs. throughput

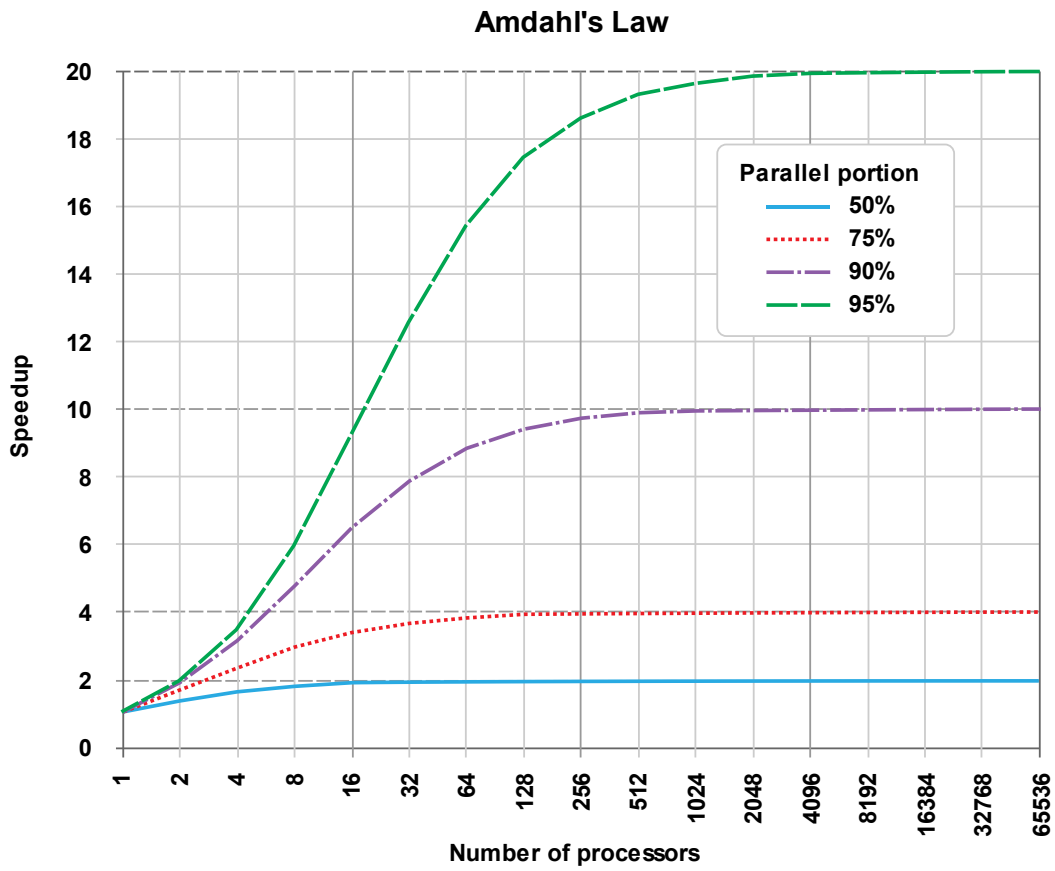
- **Latency**: Duration between a single trigger and the system's response
  - "Tail latency" (e.g., 95th percentile under a specified load) is more important than average

- **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 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 $s \rightarrow \infty$
- T_after = $p*T/s + (1 - p)*T$
- $S(s) = 1/(1 - p + p/s)$
- **$S \rightarrow 1/(1 - p)$**

Amdahl's Law implications



Poll: PollEv.com/cs5150sp25

You use a text search application to look for all occurrences of a keyword in all the files of a large source code repository.

Using a single core, half of the time is spent reading files and looking for the keyword, and half the time is spent formatting and printing a sorted summary of the results to the console.

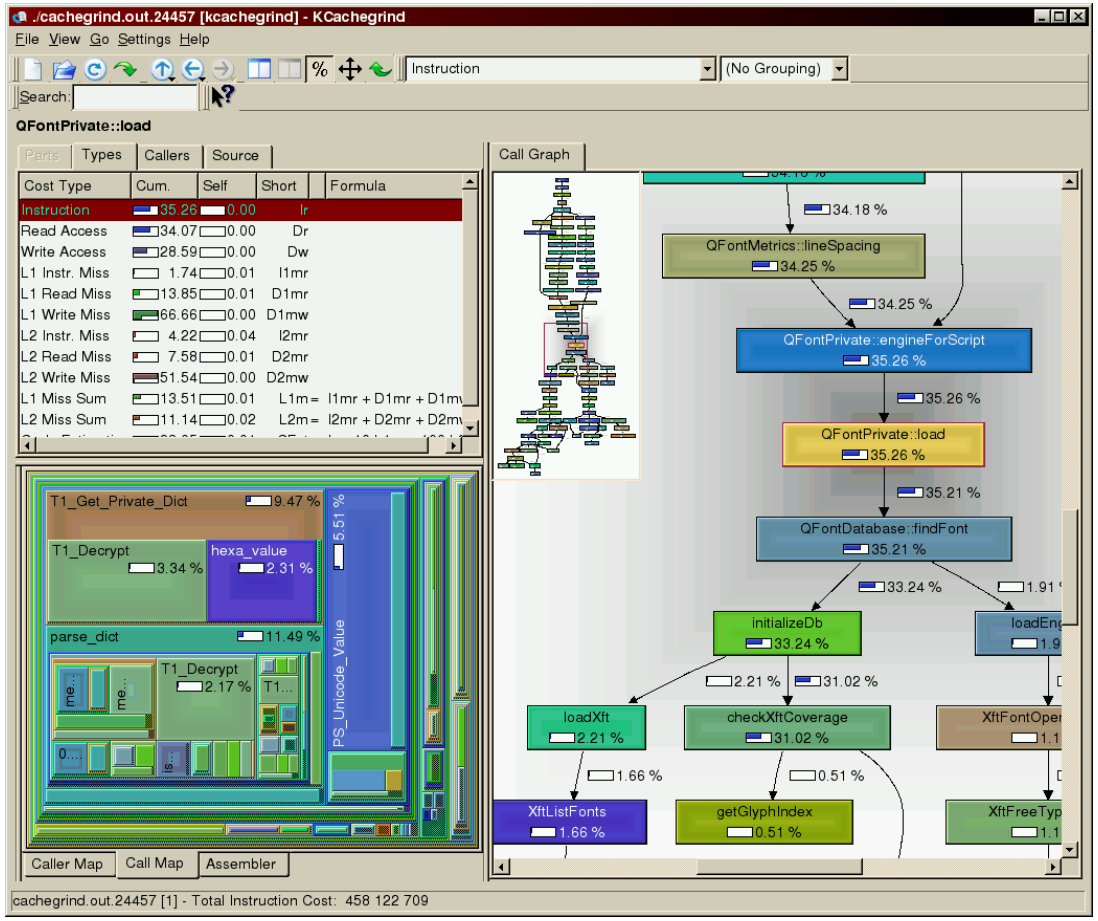
What is the maximum speedup that could be achieved by distributing the *embarrassingly parallel* work across multiple cores/nodes?

Profiling

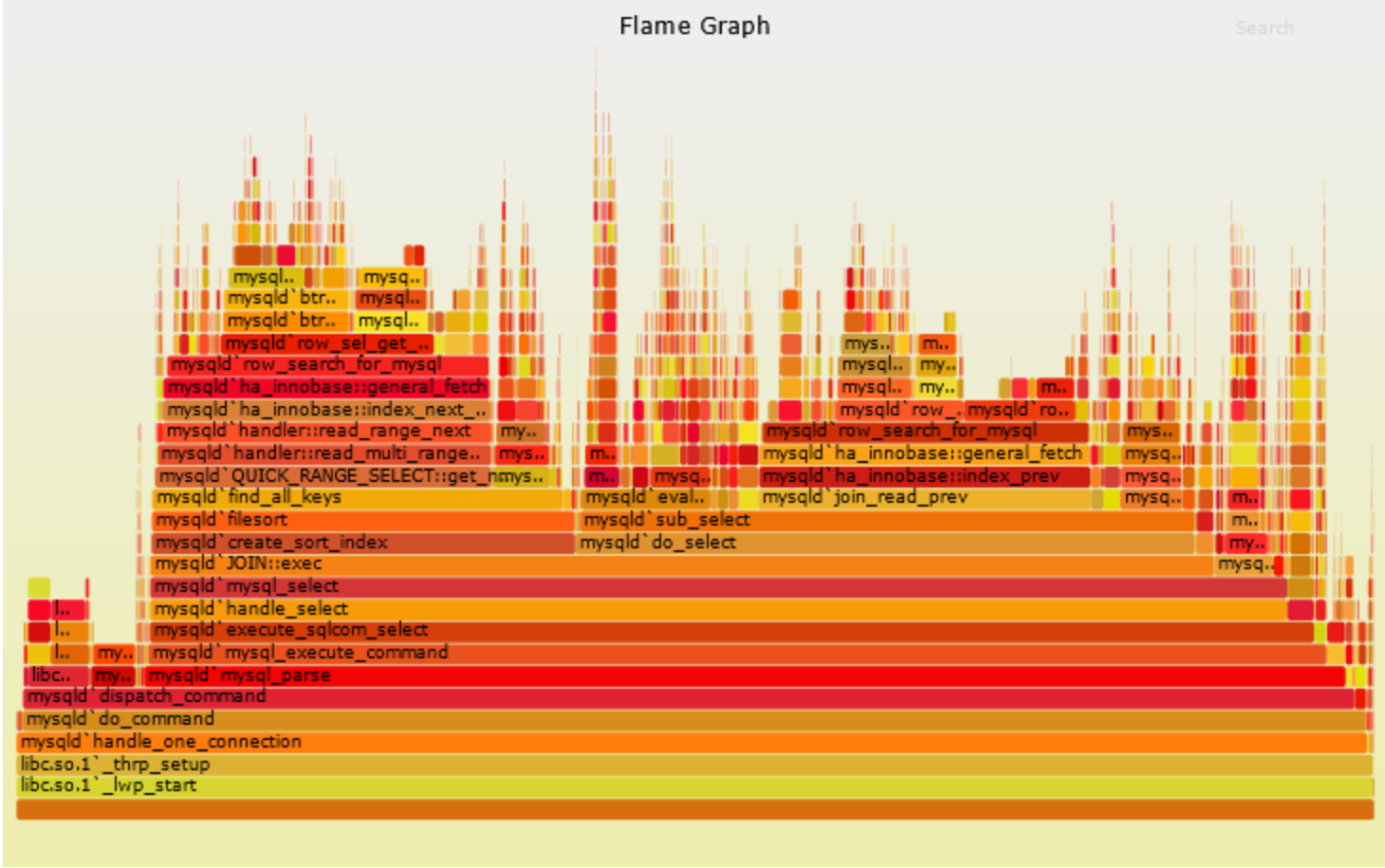
- How can we estimate *p*?
- Where should our optimization efforts be focused?

- 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

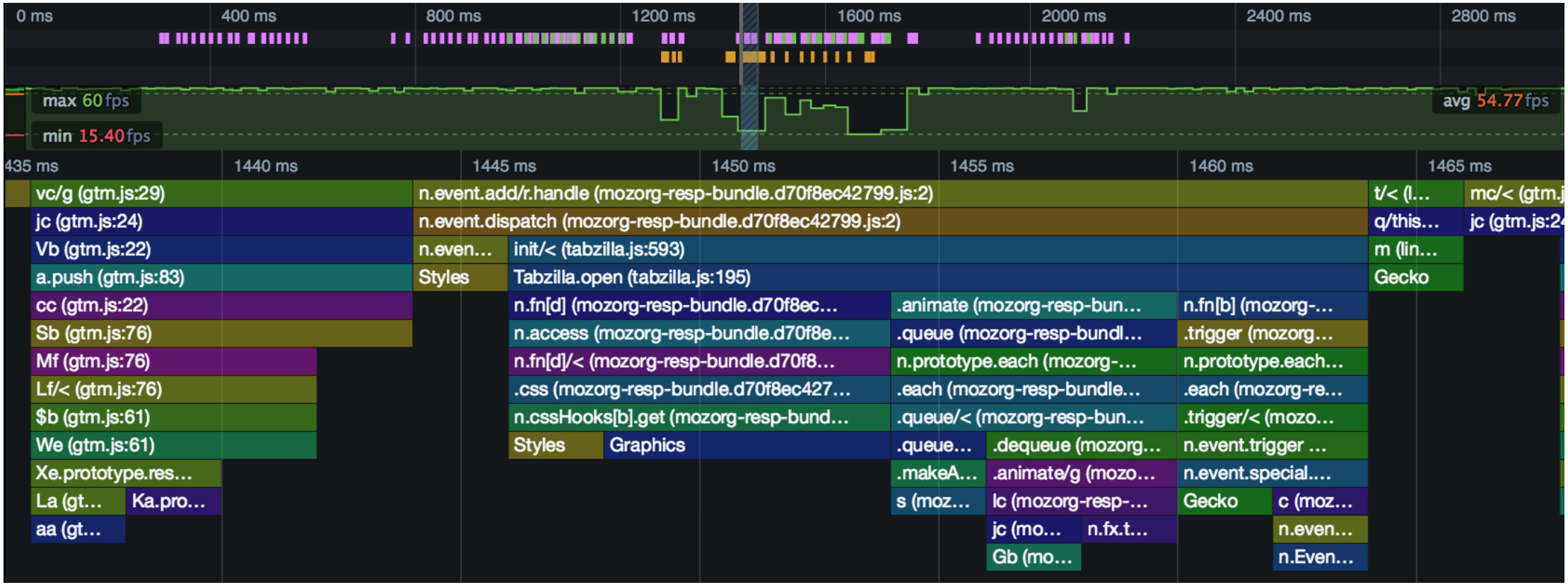


Flame graphs



https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html

Browser profilers



Monitoring

- To detect degradation and catch regressions, need to log and monitor performance metrics
  - Can measure duration of tests in CI, but benefits from unloaded servers
- For services, also need to monitor performance in production
  - Network conditions, load are dynamic
  - With scalable microservice architectures, counterintuitive bottlenecks may appear
    - Scaling the wrong components can remove beneficial backpressure

Soak testing

- Tests often execute for less time than a production system
  - Many production systems never turn off (e.g., embedded controllers)
  - Some defects (e.g. memory leaks, fragmentation) are innocuous for short runs
- **Soak testing**: Subject system to significant load for extended period of time (days, months, years)
  - Be sure to log key performance metrics (cycle time, memory usage)
  - Not particularly compatible with a rapid CI pipeline
    - Still good to run periodically to catch issues sooner