LE/EECS 4443: Mobile User Interfaces (LAB)

Week 7: Performance, Profiling, Monitoring, & Benchmarking

Shogo Toyonaga[1]

[1]Lassonde School of Engineering
York University

January 31, 2025

Table of Contents

Introduction

By the end of this tutorial, you will be able to...

1. Understand the necessity of Software Performance Engineering (SPE)
2. Apply Performance Testing in the wild
3. Write and apply proper benchmarking strategies
4. Interpret the results...

Performance: Introduction

- Performance testing can have many meanings, metrics, and implementations (e.g., static, dynamic) depending on the circumstance.
- In the case of LE/EECS 4443, we will interpret it as how efficient your application runs with respect to the following metrics:
  1. Response Time (e.g., "Jank")
  2. Throughput
  3. Utilization (Power, Network, etc.,)
- It is important to consider the scope of your performance testing. Are you measuring performance of a component or the entire system?

Introduction

The metrics we will consider are...

1. **Static:** No need to execute the system.
2. **Dynamic:** Requiring the system to execute in order to capture relevant information.
3. **Internal:** Requiring special code which is injected into the system to identify events of interest and capture relevant information.
4. **External:** Using third-party software that is not part of your application to capture relevant information.

Performance: Introduction

Accounting for performance metrics allows you to...

1. Ensure the expected behaviour(s) of system transactions
2. Decrease the workload and cost required to refactor non-optimal components
3. Eliminate bottlenecks (e.g., memory)
4. Easily scale the system as workloads increase by identifying its limitations (e.g., CPU, Memory)

Types of Performance Testing

1. **Load Testing:** Performance under an expected workload
2. **Stress Testing:** Performance under very high workloads
3. **Endurance Testing:** Performance under an expected workload for a, “long” time.
4. **Spike Testing:** Performance under varying levels of increasing or decreasing workloads.
5. **Capacity Testing:** Determining the upper limit of the applications performance.
6. **Configuration Testing:** Performance under varying configurations of the system.

Performance Testing: Considerations

Depending on how you measure performance, you must account for confounding variables:

1. **System Perturbation**
2. **Capture Rate**
3. **System Overload**
4. **Network Speeds**

You should also consider the **generalizability** of your workloads & time intervals to compromise between internal & external validity.

Performance Testing: Practical Advice

- Understand the **goal** of your performance tests. What are your research questions? What about your hypotheses?
- For **instrumentation testing** (with probes), be reasonable with the granularity. Ensure that the action you're investigating is not too short or not too long.
- If you are using **external testing**, ensure a reasonable sampling rate to ensure statistical significance.
- Do your best to account for **confounding variables**; your excellent performance may be due to caching rather than your implementation itself.

Supplemental Resources (Homework)

1. **Android Documentation: Optimize Your Build Speed**
2. **Android Documentation: Profile Your App Performance**
3. **YouTube: App Performance Analysis with the Android Studio Profiler**
4. **YouTube: Performance Debugging - MAD Skills**

Profiling: Introduction

Recall

"An app has poor performance if it responds slowly, shows choppy animations, freezes, or consumes too much power.

Fixing performance problems involves profiling your app, or identifying areas in which your app makes inefficient use of resources such as the CPU, memory, graphics, or the device battery. This topic describes the Android Studio tools and techniques to use to fix common performance problems."

https://developer.android.com/studio/profile

Profiling: Internal vs External

A quick recap...

1. **Internal (Instrumentation)**
   - Allows for collection of granular metrics at the cost of high overhead if the recorded operation are too short or too fast.
   - Introduces, "heisenbugs"; bugs whose presence depends on the measurement process.
2. **External (Sampling)**
   - Interrupts the CPU in frequent intervals to map operation(s) to the clock cycle.
   - Sampling must be sufficiently high to ensure statistical significance and the reduced likelihood of, "missing" important event(s).
   - Less accurate but has insignificant overhead

Profiling: Heuristic

Remark

If the operations are sufficiently slow, it is preferable to use instrumentation.

This is because the additional overhead due to invasive probing becomes insignificant.

Introduction

Remark

“Benchmarking is a way to inspect and monitor the performance of your app. You can regularly run benchmarks to analyze and debug performance problems and help ensure that you don’t introduce regressions in recent changes.

Android offers two benchmarking libraries and approaches for analyzing and testing different kinds of situations in your app: Macrobenchmark and Microbenchmark.”

https://developer.android.com/topic/performance/benchmarking/benchmarking-overview

Introduction

Just to make the differentiation very clear:

- **Benchmarking** is about empirically comparing performance between applications.
- **Profiling** is about understanding why your performance is good or bad.

Types of Benchmarks I

1. **Specification-Based** (≈ Java Interface)
2. **Kit-Based (Implementation Provided, No Overriding)**
3. **Hybrid (≈ Java Abstract Class)**
4. **Synthetic (UI Automator)**
5. **Microbenchmark:** Analyzes performance of situations specific to the app, not ones that might relate to overall system issues.

Types of Benchmarks II

6 **Macrobenchmark:** Measures larger end-user interactions, such as startup, interacting with the UI, and animations. The library provides direct control over the performance environment you're testing. It lets you control compiling and lets you start and stop your app to directly measure actual app startup or scrolling.

7 **Kernel:** A broader, more generalizable, inexpensive version of the Microbenchmark. It may miss bottlenecks or fail to sufficiently stress the system resources.

8 **Application:** A real world, "simulation" of the expected user(s) and their associated behaviour(s).

Good Benchmarks

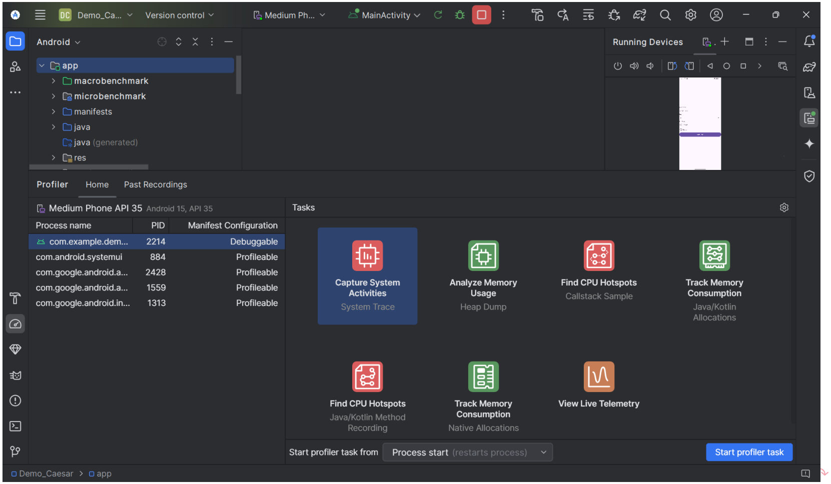
Good benchmarks are...

1. **Relevant (Scope)**
2. **Reproducible**
3. **Fair (Control of Confounding Variable(s))**
4. **Verifiable (Strategies & Metrics)**
5. **Usable (Show me the code!!!!)**

Instructions I

1. Download & import Demo_Caesar
2. Click ⋮ which is beside the debugging button to expand the menu.
3. Click and run the activity as profileable or debuggable.
4. The emulator will start. Select a task that interests you and start the profiler task.
5. Play around with the UI & use the Android Documentation for clarification. You will only learn by getting your hands dirty!

Instructions II



Medium Phone API 35 Android 15, API 35

Process name	PID	Manifest Configuration
com.example.dem...	2214	Debuggable
com.android.systemui	884	Profileable
com.google.android.a...	2428	Profileable
com.google.android.a...	1559	Profileable
com.google.android.in...	1313	Profileable

Macrobenchmark I

```
@RunWith(AndroidJUnit4.class)
public class ExampleMacro {

    @Rule
    public MacrobenchmarkRule mBenchmarkRule = new MacrobenchmarkRule();

    @Test
    /**
     * A benchmark to test ROT-13 on a short piece of plaintext
     */
    @Test
    public void ROT13_Encryption() {
        mBenchmarkRule.measureRepeated(
                "com.example.demo_caesar",
                // Put your list of metrics here....
                Arrays.asList(new StartupTimingMetric(), new FrameTimingMetric()
                    ),
                CompilationMode.DEFAULT,
                StartupMode.COLD,
                5,
                // Your benchmarking actions go here using UiAutomator to
                    control the device
                scope -> {
                    UiDevice device = scope.getDevice();
```

Macrobenchmark II

```
                    // Start the Activity and Wait
                    scope.startActivityAndWait();
                    // Attach Views to Variables (Controllers)
                    UiObject2 spinner, user_input, encrypt, decrypt, button, solution;
                    spinner = device.findObject(By.res(scope.getPackageName(), "spinner_key"));
                    user_input = device.findObject(By.res(scope.getPackageName(), "user_input"));
                    encrypt = device.findObject(By.res(scope.getPackageName(), "encrypt"));
                    decrypt = device.findObject(By.res(scope.getPackageName(), "decrypt"));
                    button = device.findObject(By.res(scope.getPackageName(), "compute_button"));
                    solution = device.findObject(By.res(scope.getPackageName(), "computed_results"));
                    // Perform Actions
                    user_input.setText("Hello world!");
                    encrypt.click();
                    button.click();
                    return null;
                });
    }
}
```

Macrobenchmark III

1. Write your own macrobenchmarks:
   app > macrobenchmark > java > ExampleMacro.java

Conclusion: Benchmarking

1. **Android Documentation: Writing a MacroBenchmark**
2. **Android Documentation: Writing a Microbenchmark**
3. **Android Documnentation: Measure Performance**
4. **Demo_Caesar**

Conclusion

Remark

Thank you for your attention!
Questions?