****

**Kauno technologijos universitetas**

Informatikos fakultetas

Programų inžinerijos katedra

**Komanda: Grupė 1**

**Projektas: GymBuddy**

Programų sistemų testavimas

|  |  |
| --- | --- |
|  |  |
| **Aistis Jakutonis IFF-3/1**  **Tautrimas Ramančionis IFF-3/1**  **Nojus Birmanas IFF-3/1**  **Juozas Balčikonis IFF-3/1**  Studentai | (parašas) (data) |
|  |  |
| **Eligijus Kiudys**  Dėstytojai | (parašas) (data) |
|  |  |

Lab4 ataskaita

**Kaunas, 2025**

Content

[1. Introduction 3](#_Toc197537068)

[1.1. Purpose of performance testing (look through if this makes sense) 3](#_Toc197537069)

[1.2. Overview of the app (look through if this makes sense) 3](#_Toc197537070)

[2. Test environment setup 4](#_Toc197537071)

[2.1. Devices and emulator information (maybe add some more info) 4](#_Toc197537072)

[2.2. Testing tools (look down in the documentation and write the correct tools and write a little about those tools) 4](#_Toc197537073)

[3. Profiling test cases 5](#_Toc197537074)

[3.1. Manual test scenarios executed (what manual performance tests were done explain them a little) 5](#_Toc197537075)

[3.2. CPU usage analysis 5](#_Toc197537076)

[3.3. Memmory usage analysis 12](#_Toc197537077)

[4. Load/Stress testing 13](#_Toc197537078)

[4.1. Simulated load stategy 13](#_Toc197537079)

[4.2. Results 13](#_Toc197537080)

[4.3. Profiler during load 13](#_Toc197537081)

[5. Findings and insights 14](#_Toc197537082)

[6. Conclusion 15](#_Toc197537083)

# Introduction

## Purpose of performance testing (look through if this makes sense)

The primary goal of performance testing in this project is to evaluate how efficiently the application utilizes system resources, particularly CPU and memory. By monitoring resource consumption during key user interactions, we can assess whether the app performs smoothly and identify areas where performance may degrade over time or with increased usage. Another important aspect is to observe how the app behaves under stress, such as rapid or repeated database operations, to ensure it remains stable and responsive. Through this process, we aim to detect performance bottlenecks, such as memory leaks, inefficient methods, or excessive CPU usage, and make improvements that enhance the overall user experience. This testing is especially important as mobile devices vary in performance capabilities, and ensuring the app runs reliably across different environments is critical to its usability and quality.

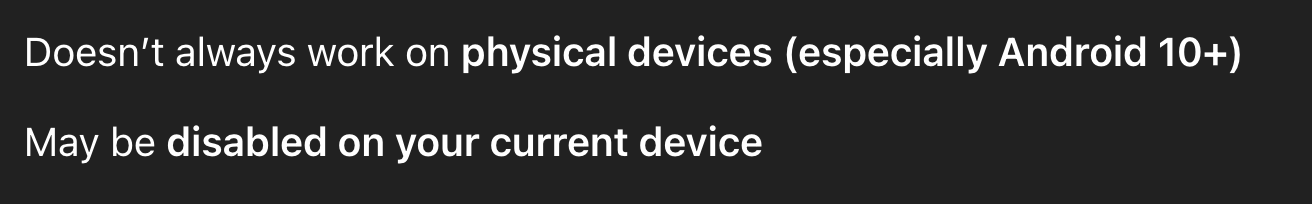
## Overview of the app (look through if this makes sense)

The application is a simple Android app developed using Android Studio, designed to manage and store user data locally on the device. It uses Room, a persistence library that provides an abstraction layer over SQLite, to handle all database operations such as creating, reading, updating, and deleting records. One of the key characteristics of the app is that it does not connect to the internet or use any form of cloud storage. All user data is stored locally on the device. This ensures data privacy and allows the app to function offline. Additionally, the app is designed for a single user, meaning there is no user authentication system or functionality to share data between users. This focus on simplicity and local functionality makes performance optimization of local resources especially important.

# Test environment setup

## Devices and emulator information (maybe add some more info)

Hardware used for testing: (didn’t work)



* Samsung Galaxy S21 (Android 14) (API Level 34)

Emulators used for testing:

* Medium Phone (API Level 35)

These environments were used to install, run, and profile the app during performance testing to ensure consistent behavior across both physical and virtual devices.

## Testing tools (look down in the documentation and write the correct tools and write a little about those tools)

**Android Profiler** (built into Android Studio): Used to monitor CPU usage, memory consumption, and real-time performance data during app execution.

**Heap Analyzer** (via Android Studio's memory dump feature): Used to identify memory allocation patterns, object retention, and possible memory leaks.

**Custom Kotlin Script**: Used to simulate repeated database operations for local load testing in the absence of a REST API.

**JMeter**: Considered for simulating load through HTTP requests if a mock REST service was implemented.

# Profiling test cases

## Manual test scenarios executed (what manual performance tests were done explain them a little)

Add new item to Room DB (adding new item – repeating test three times)

Edit existing entry (editing existing entry – repeating test three times)

Delete entry (deleting entry – repeating test three times)

List/load all entries (opening a page with all entries – repeating test three times)

Search/filter operations (executing search operation – repeating test three times)

(repeating tests three times to ensure that they give the correct results but in the results only present one test example)

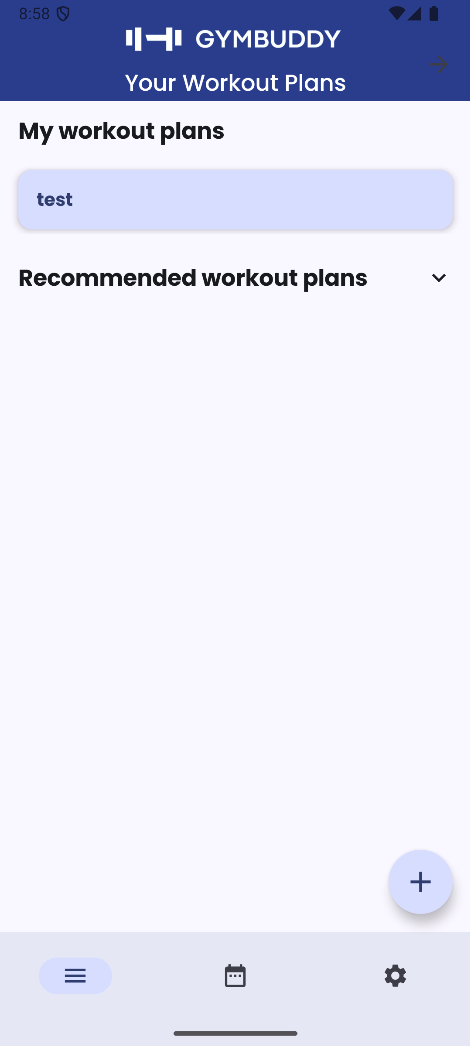
## CPU usage analysis

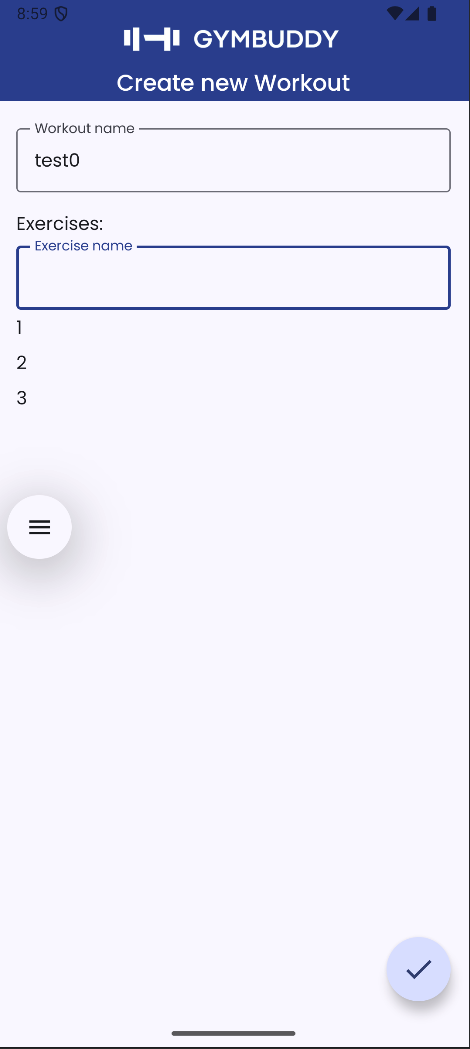
Method used: Android Profiler

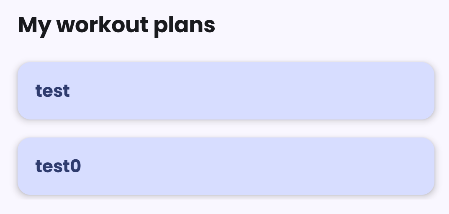
Observations: (photos below)

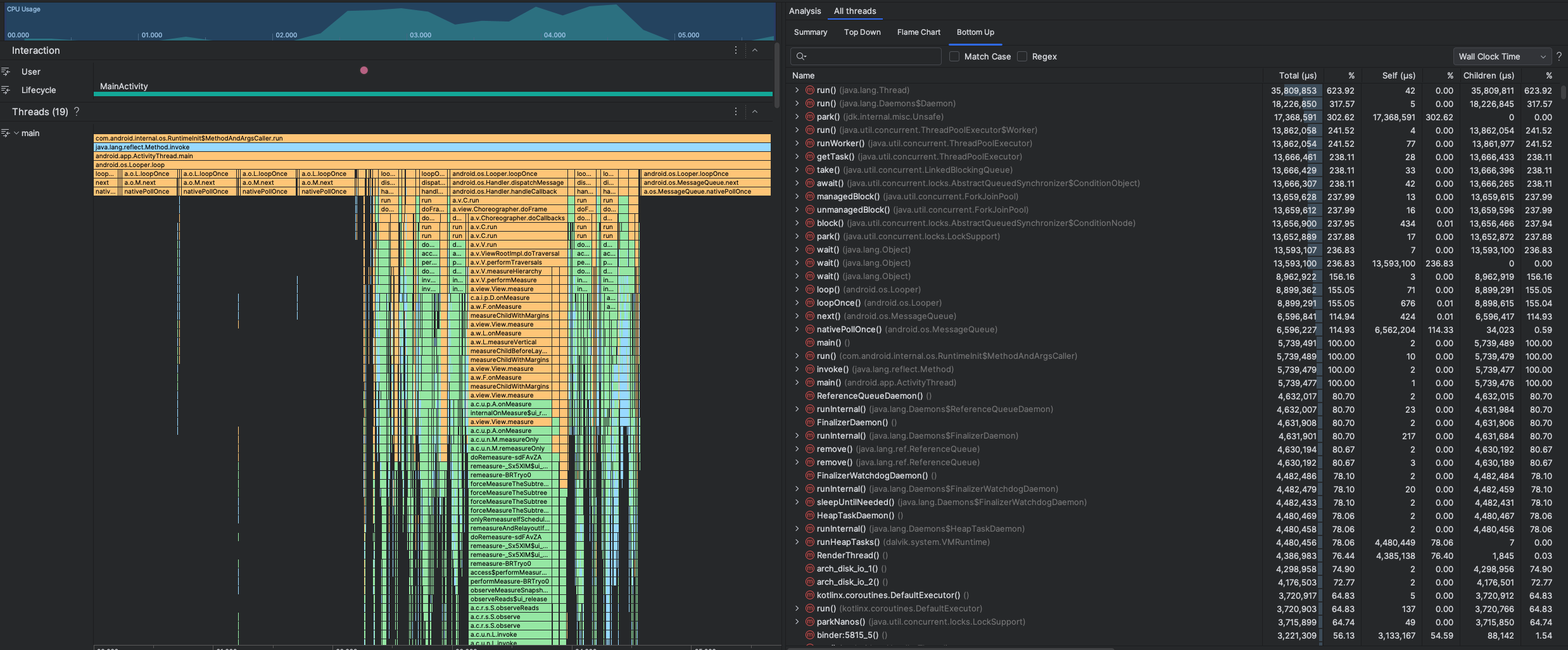
* Total CPU time during test
* Top methods/classes consuming CPU
* E.g.: DAO insert(), RecyclerView binding, LiveData updates

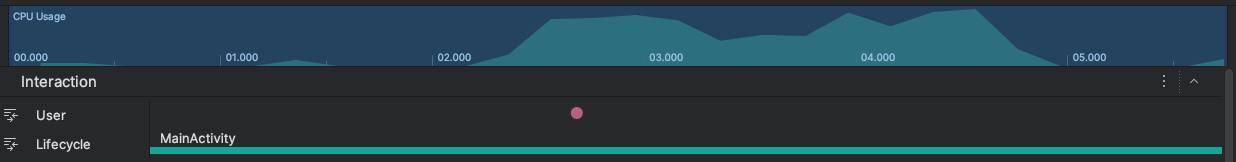
Cpu usage and top methods consuming CPU:  
  
Insert:

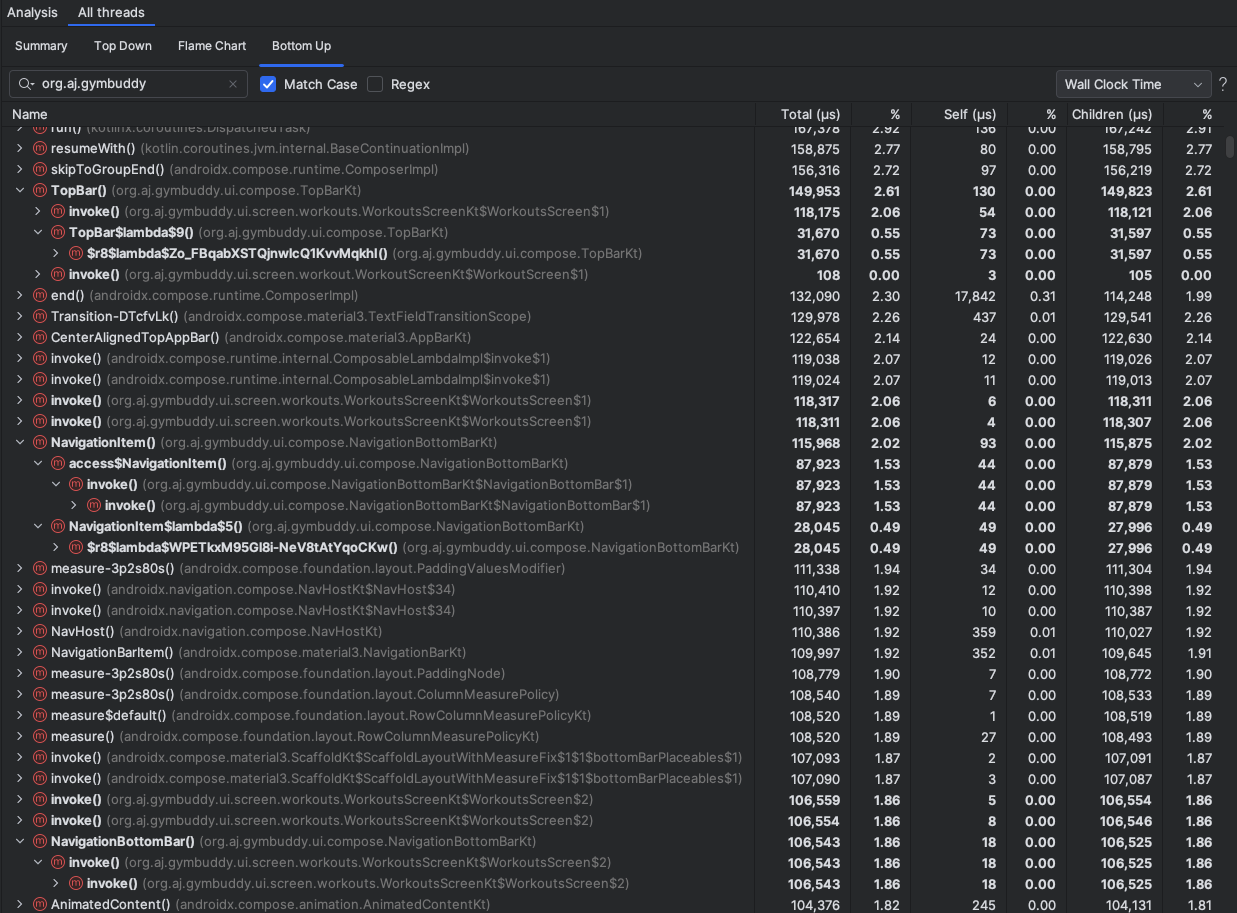


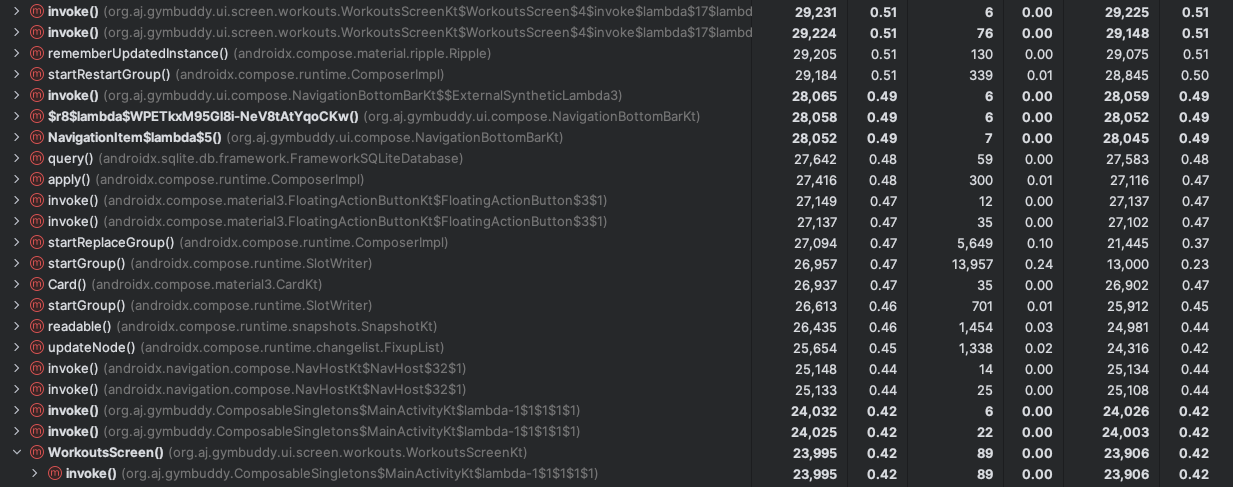






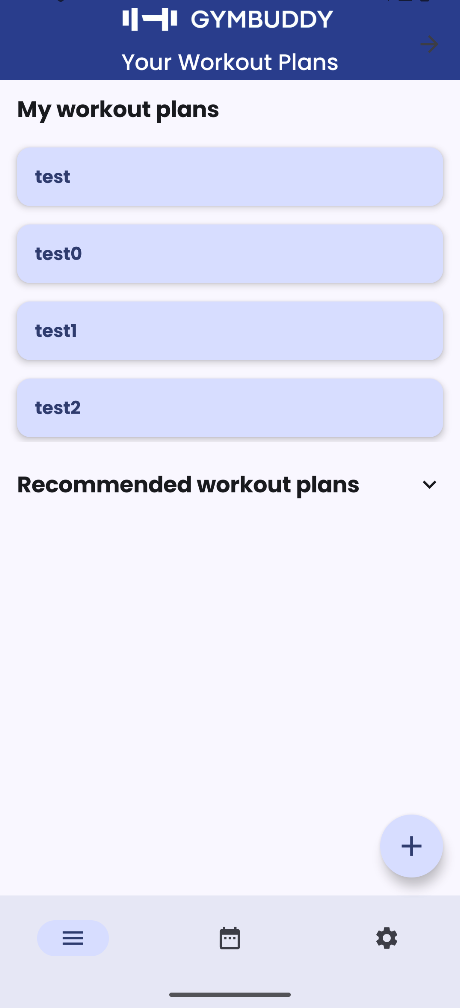


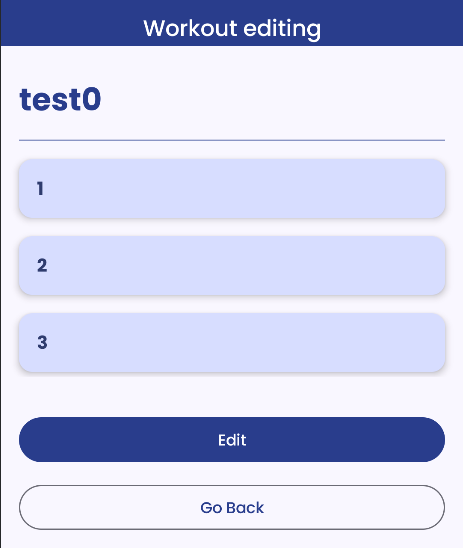


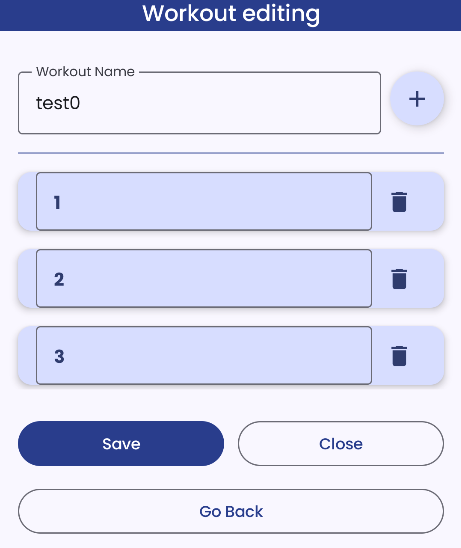


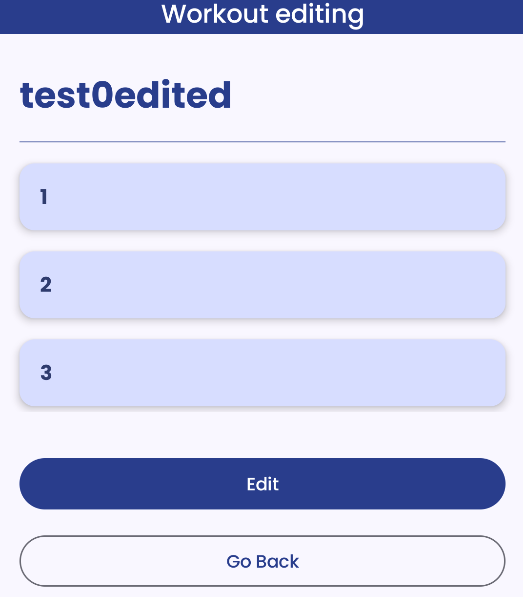
Cpu usage percentage spiked to 80-85%. Top methods: TopBar, invoke, NavigationItem.

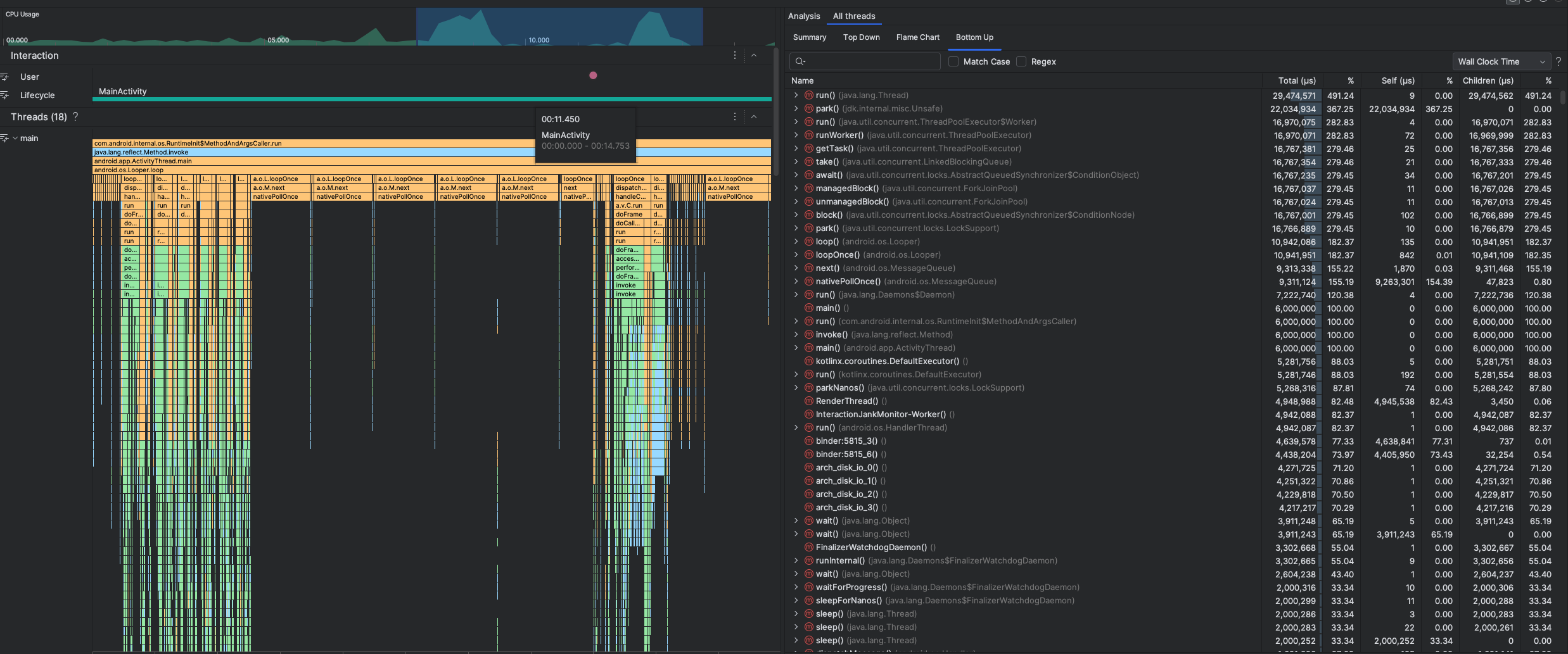
Edit:

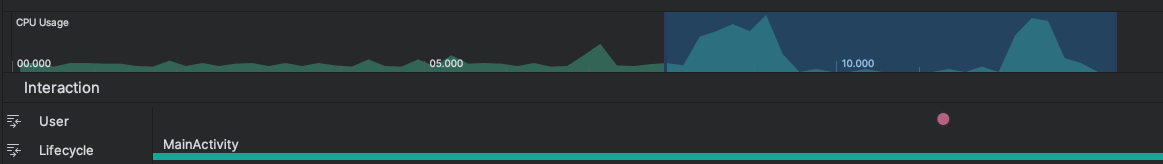


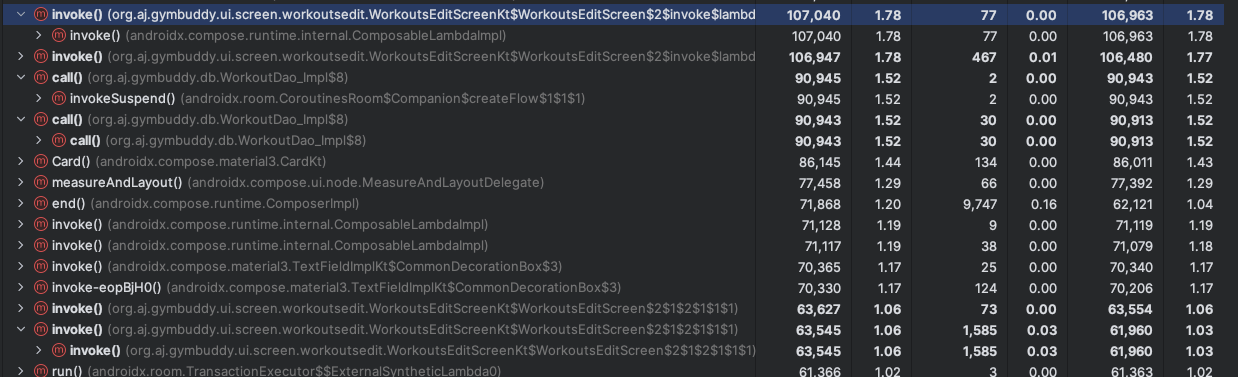


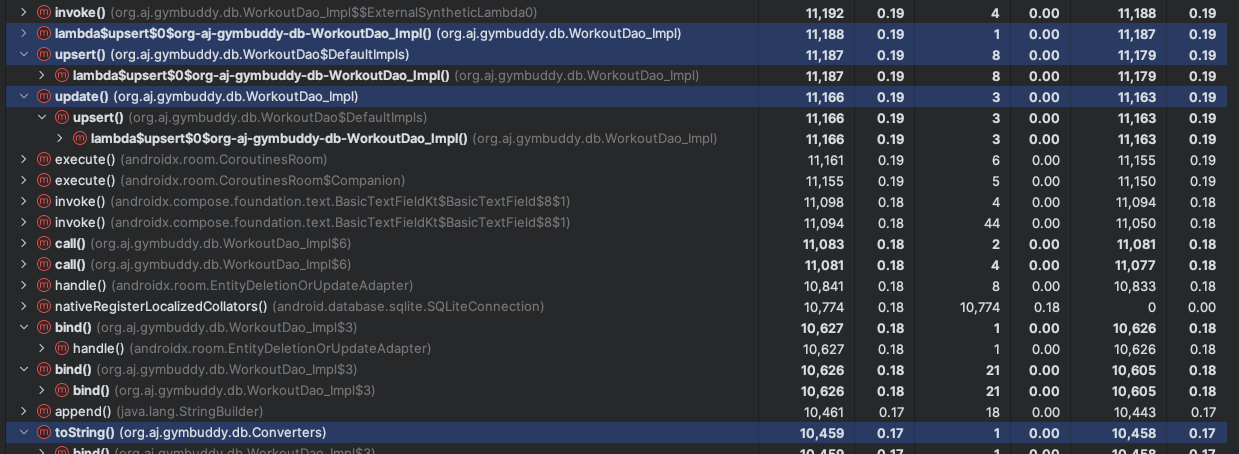






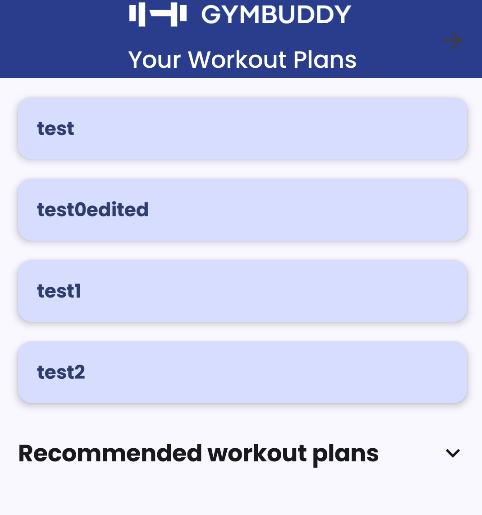


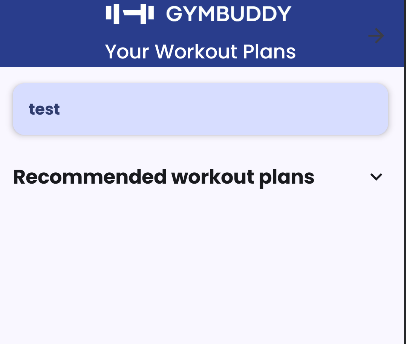


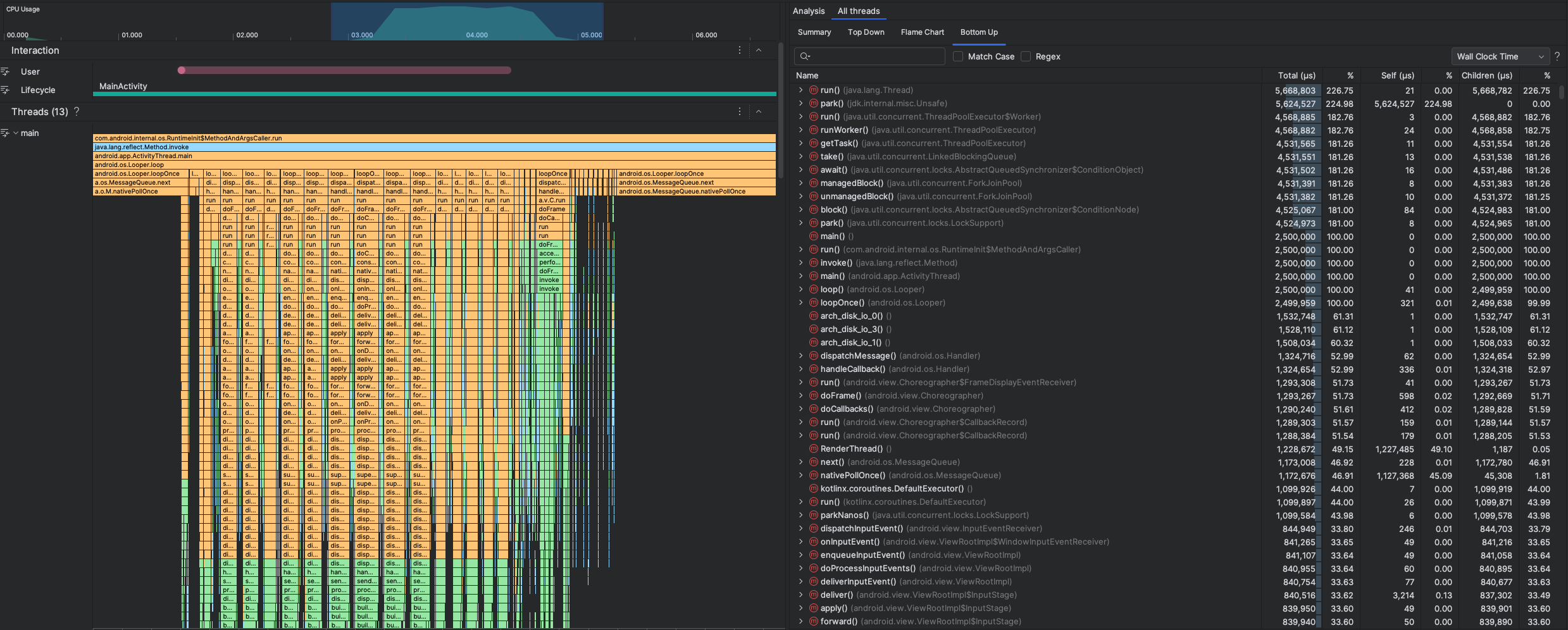


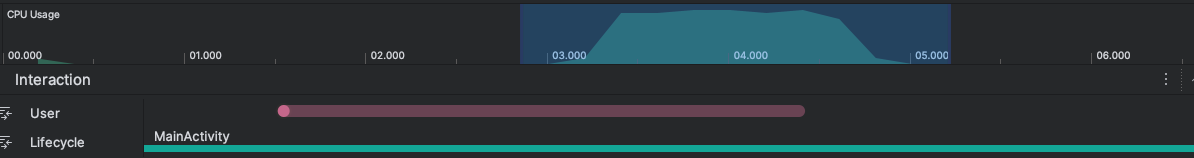
Two spikes because in one the text was being edited and in another the edited text was saved. While editing cpu usage spiked to 59-80% and on save it spiked to 61-90%. Top methods: invoke(), call, upsert, update.

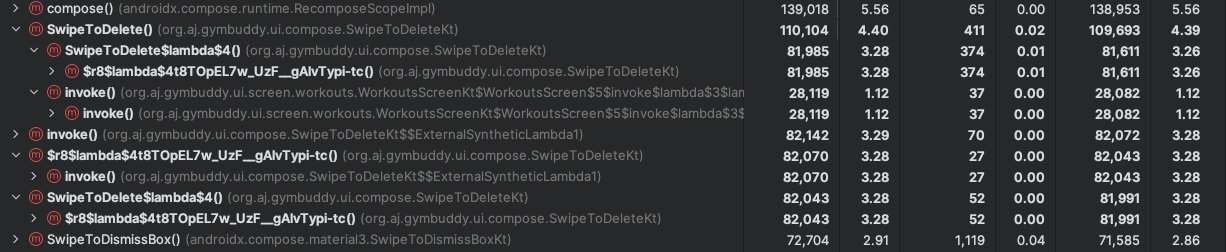
Delete:

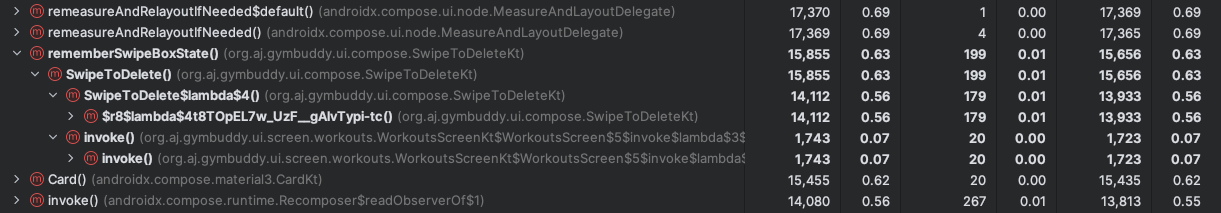






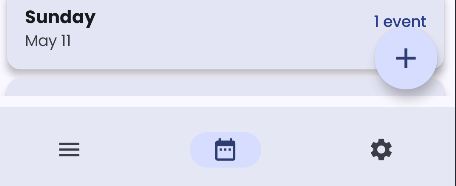


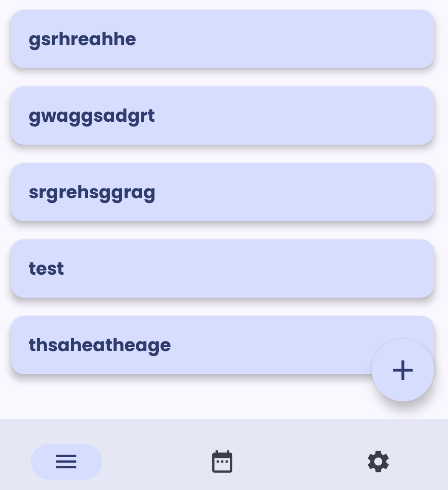


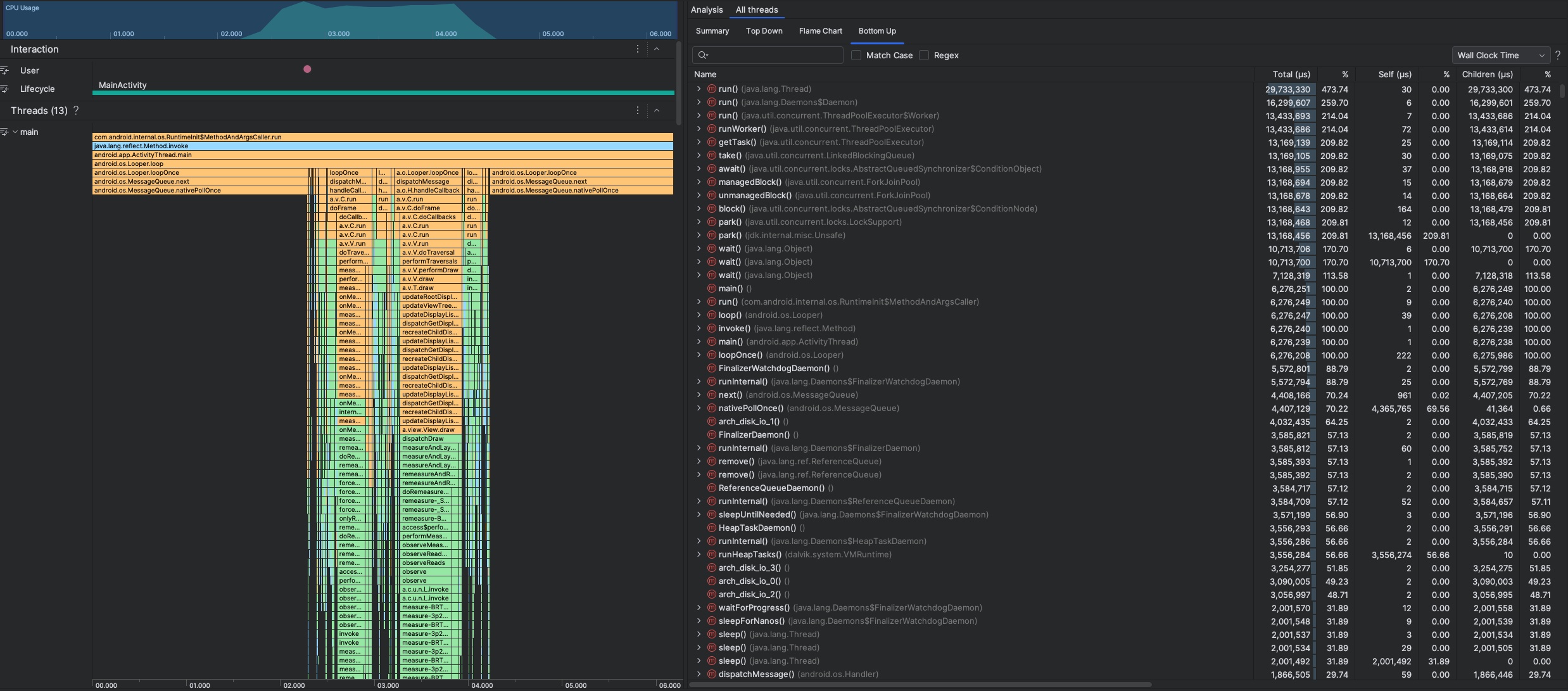


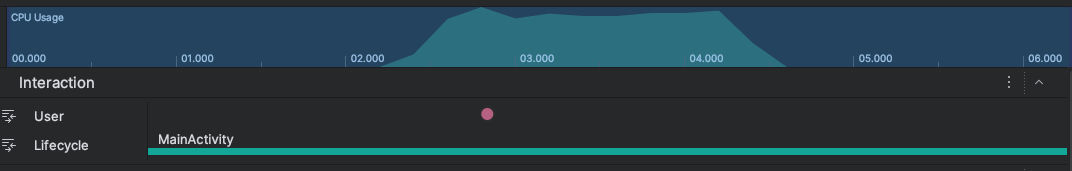
Spiked from 80-90%. Top methods: swipetodelete, invoke, rememberswipeboxstate

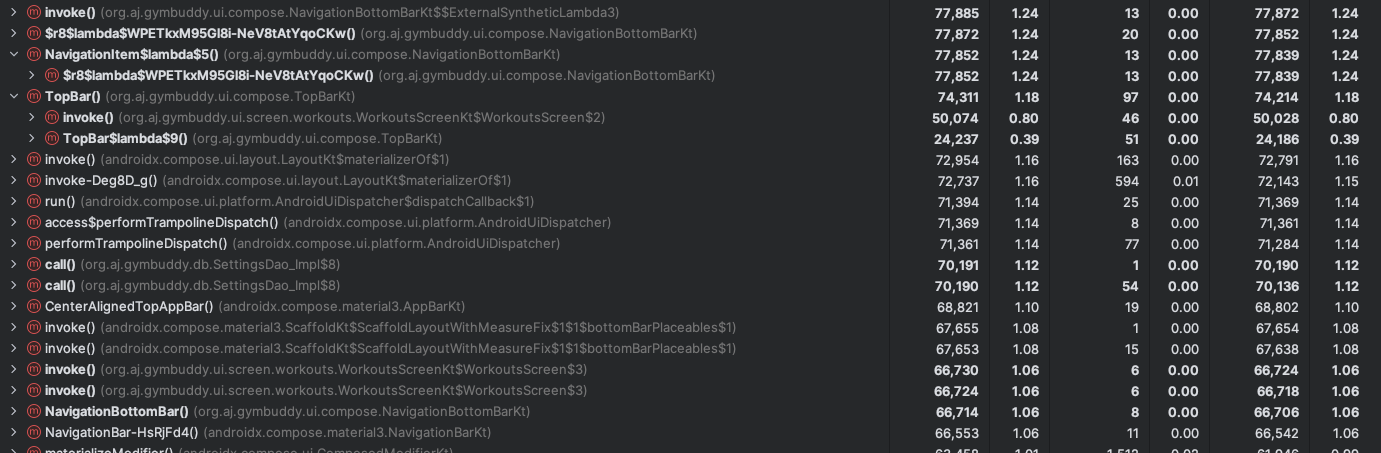
Load a list:





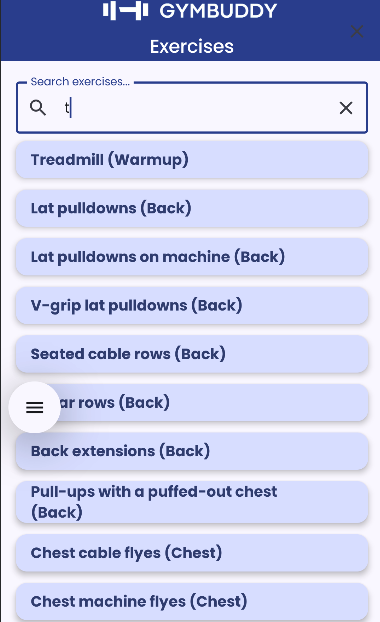


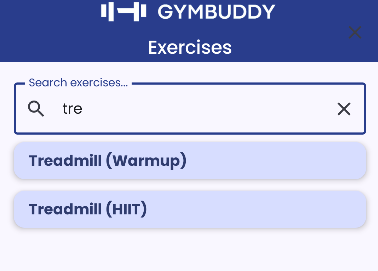


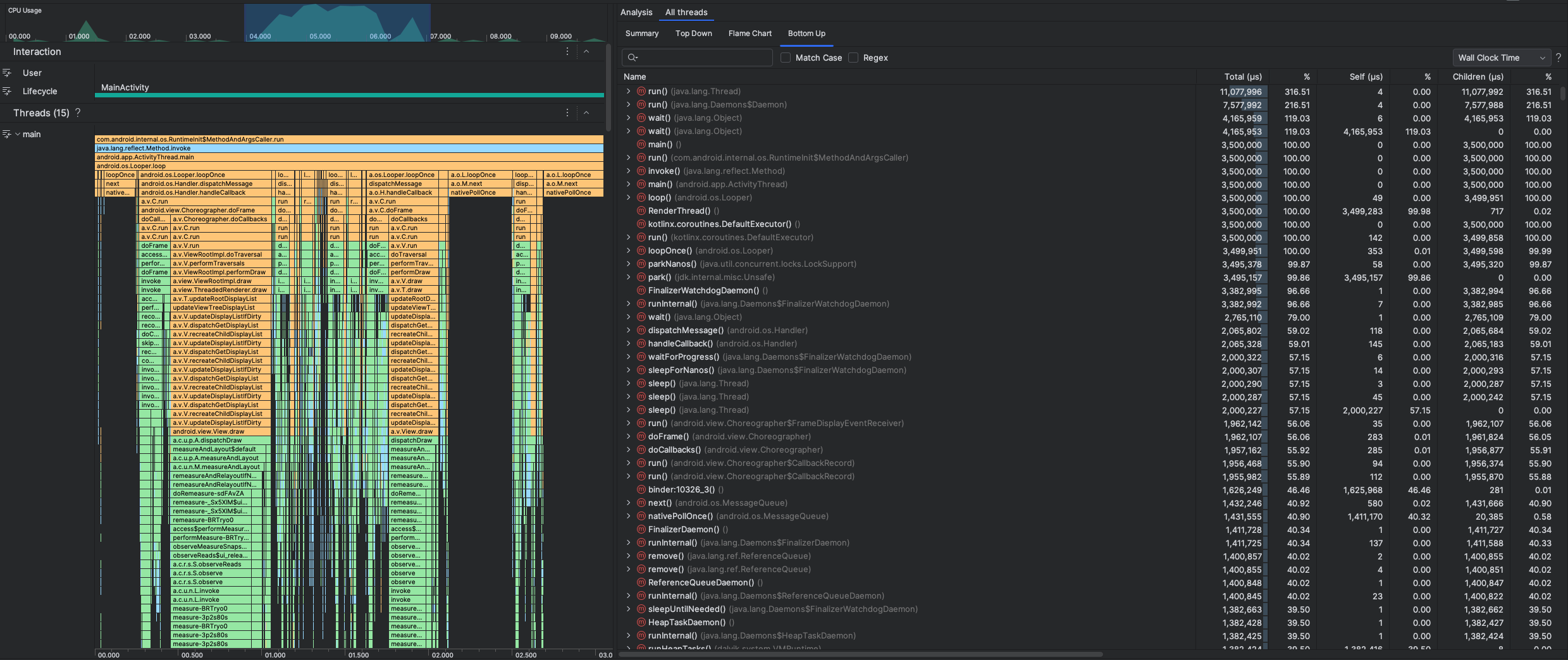


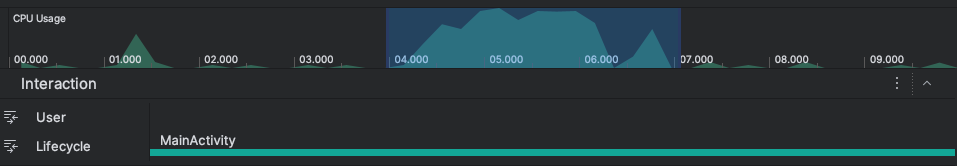
Spiked 80-100% hundred because made a bigger list and emulator takes it longer. Top methods: invoke, navigationitem, topbar, call, navigationbuttonbar.

Execute search:

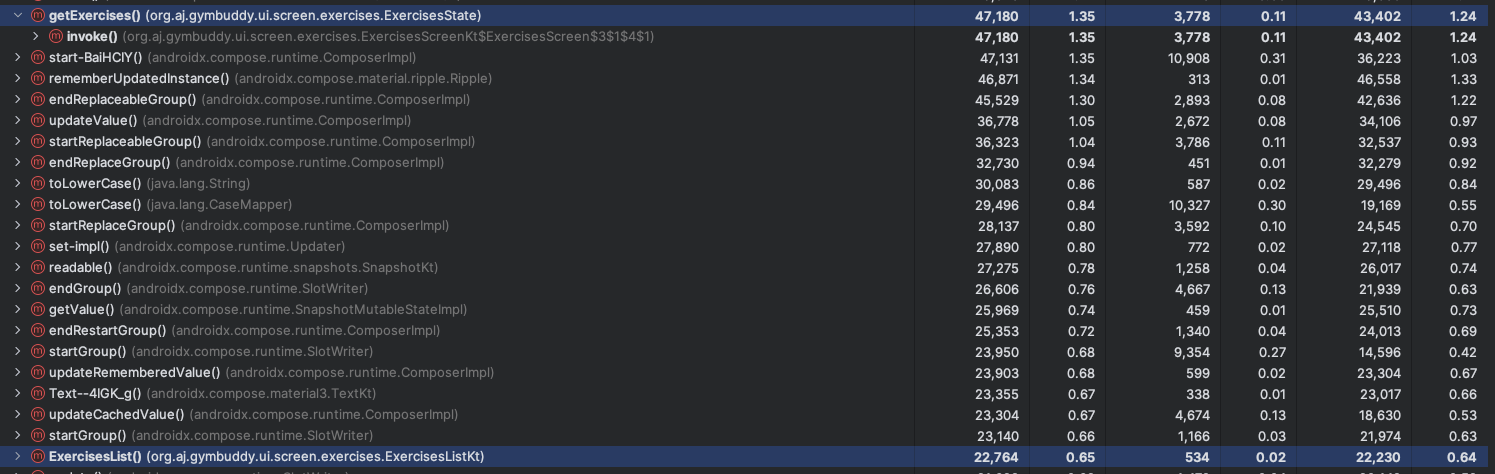












When executing the first search and the bigger list is presented for the results cpu usage spiked 90-100% and when continuing my search 90-95%. Top methods: invoke, exercisesRow, getexercises, exerciseslist.

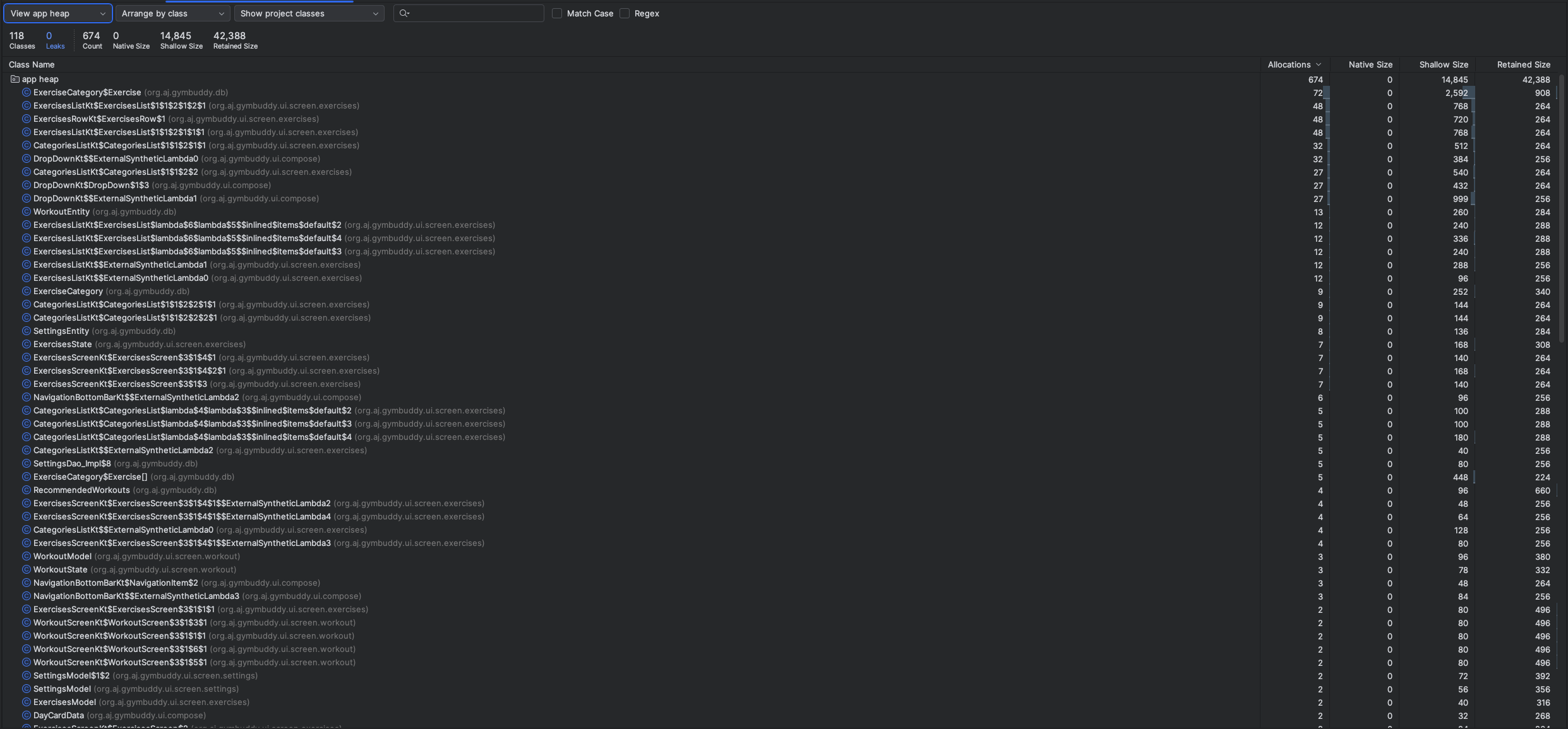
All tests show high cpu usage percentage because for tests we used emulator which tends to show higher cpu usage.

## Memmory usage analysis

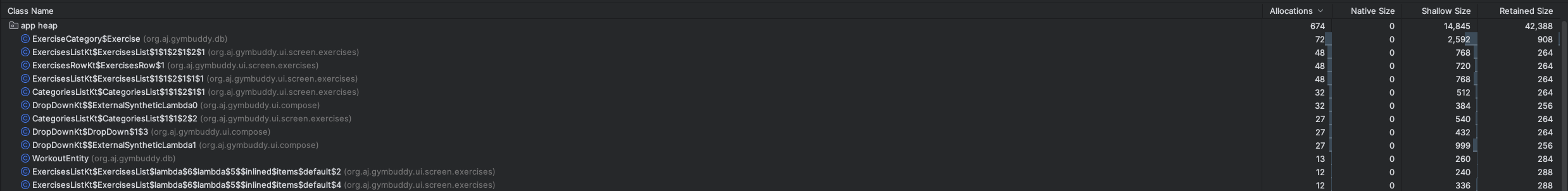
 Method used: Heap dump + analysis

 Findings:

* Most allocated objects (e.g., entity classes, strings)
* Memory leaks (if any)

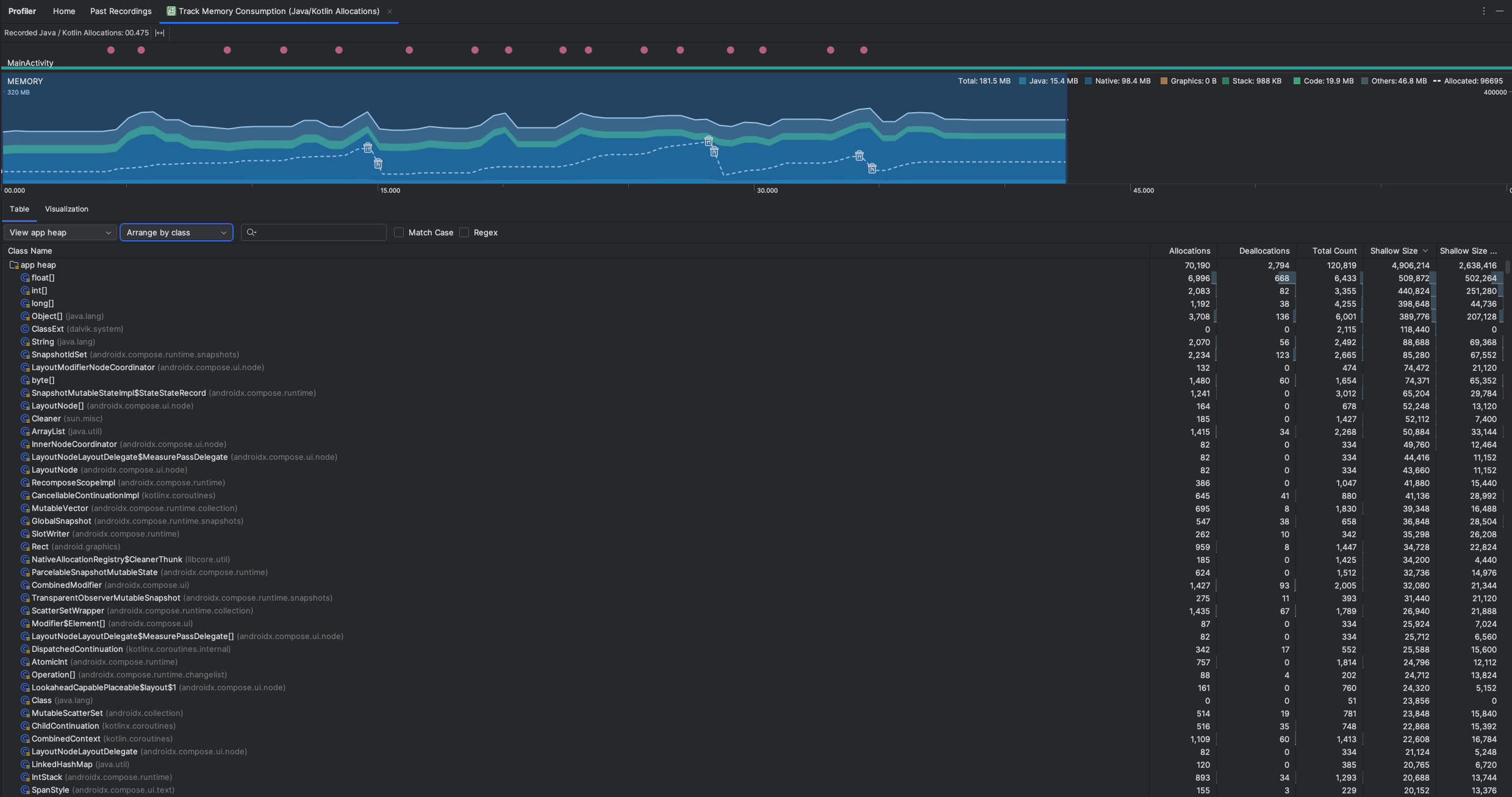


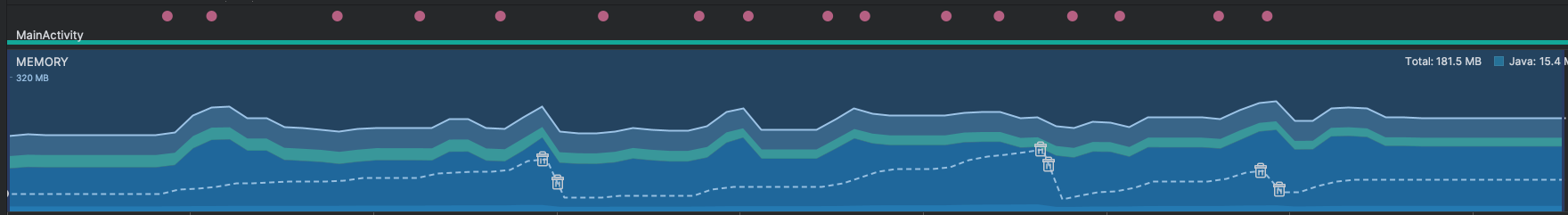




Most alocations^

No memory leaks





Insert^, delete^, edit^, save^, load list^, search^ (on spikes all these)

# Load/Stress testing

## Simulated load stategy

 App doesn't have a REST API; simulated load by:

* Creating 1000 insert/read operations over 5 minutes
* Using Kotlin coroutines or multi-threading

 (Optional) Used local REST mock (Retrofit + MockWebServer) if required for module spec

## Results

 Response time per operation (avg/min/max)

 Throughput (ops/sec)

 Observed behavior under load (UI freezes, crashes, slowdowns)

## Profiler during load

 Profiler attached during load testing

 Tracked:

* CPU spikes
* Memory usage patterns
* Any method bottlenecks

# Findings and insights

 Summary of major findings:

* E.g., Room insert is lightweight, RecyclerView can lag under large datasets
* No major leaks found / minor memory spike during heavy DB operations

 Performance bottlenecks (if any)

 Areas for optimization (e.g., pagination, background threads, caching)

# Conclusion

 Overall app performance assessment

 Whether it meets expected thresholds

 Next steps (e.g., implement pagination, optimize DB calls, UI responsiveness)