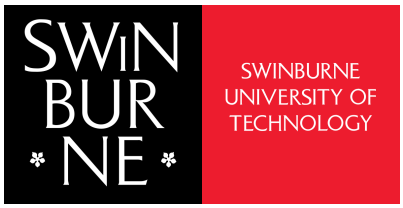

Swinburne University of Technology

Software Development for Mobile Devices

Extension: Performance

Marco Giacoppo (104071453)

Thursday 12:30

2023 Semester 2

Introduction

The performance of mobile applications is a crucial component of providing a seamless user experience in today's digital landscape. We were required to use the Profiler tool to evaluate an Android application's performance as part of our evaluation. This report examines the effects of various factors on the performance of the app while giving an overview of our experiments and findings.

The app is built with a list-based user interface that uses a RecyclerView to show a list of strings along with accompanying icons. Our study aims to comprehend how various factors impact the app's performance, and the findings will be helpful in enhancing its functionality.

Experiment

Scenario 1: A generated icon created on bind.

This scenario exhibits a unique memory behaviour with memory reduction after multiple clicks. This may indicate some memory management or garbage collection optimizations.

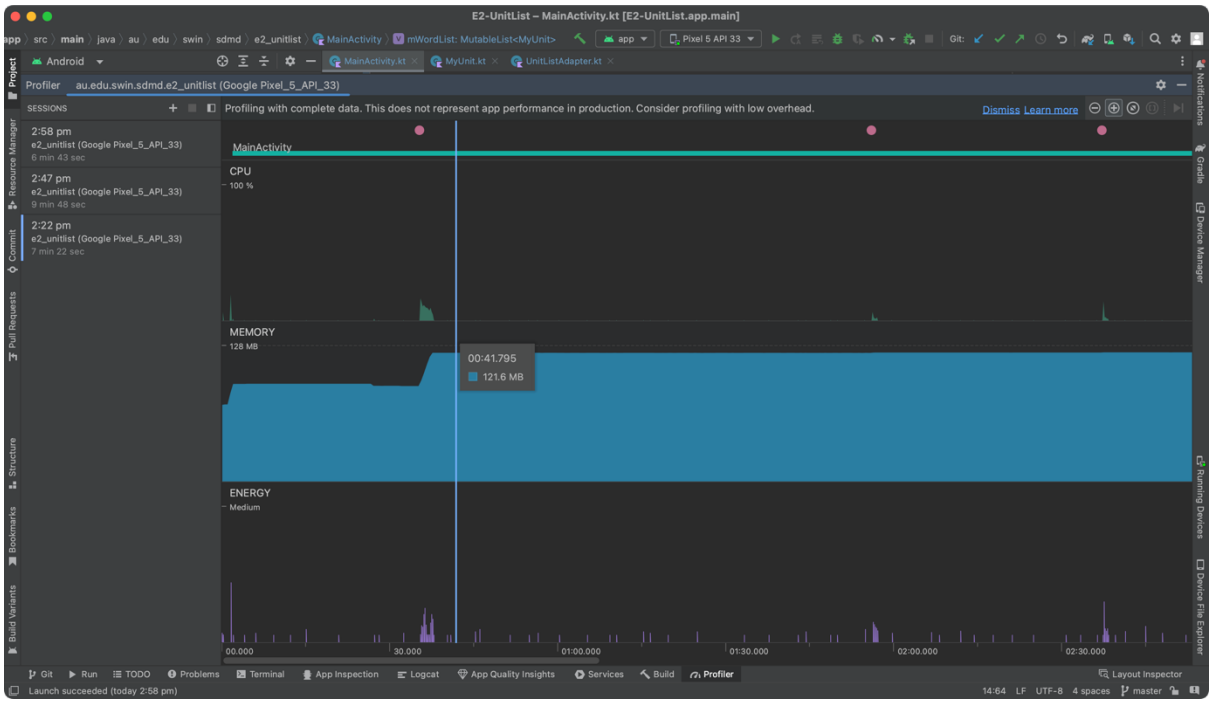

Figure 1: Scenario 1 on start + click on button.

On initialising the app, we can see that the CPU had a spike to 20% and the RAM went from 73.1MB to 92.5MB. When I pressed on the FAB, the CPU had a spike again towards 17% and the memory usage went up significantly to 121.6 and stays there until the next time I pressed the FAB.

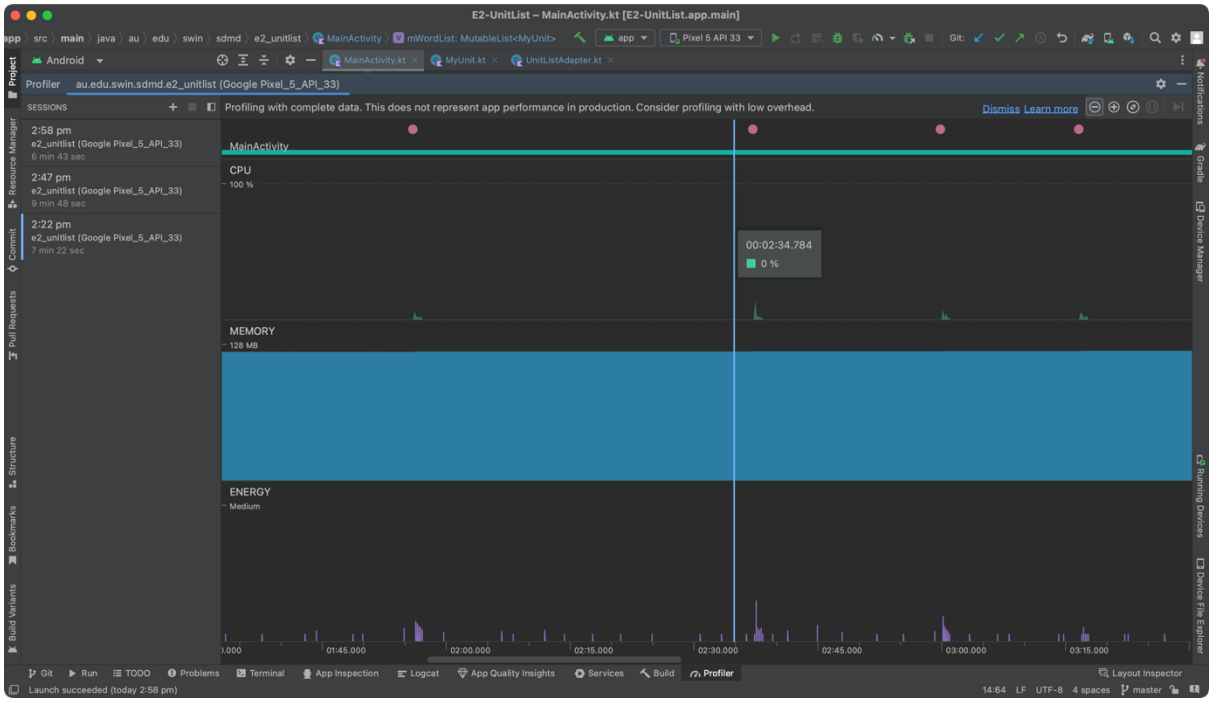

Figure 2: Scenario 1 on the next few clicks.

On the next few clicks, nothing significant happened. The CPU usage varies between 2% until 8%. The memory usage only went up 0.2 or 0.3 MB each click.

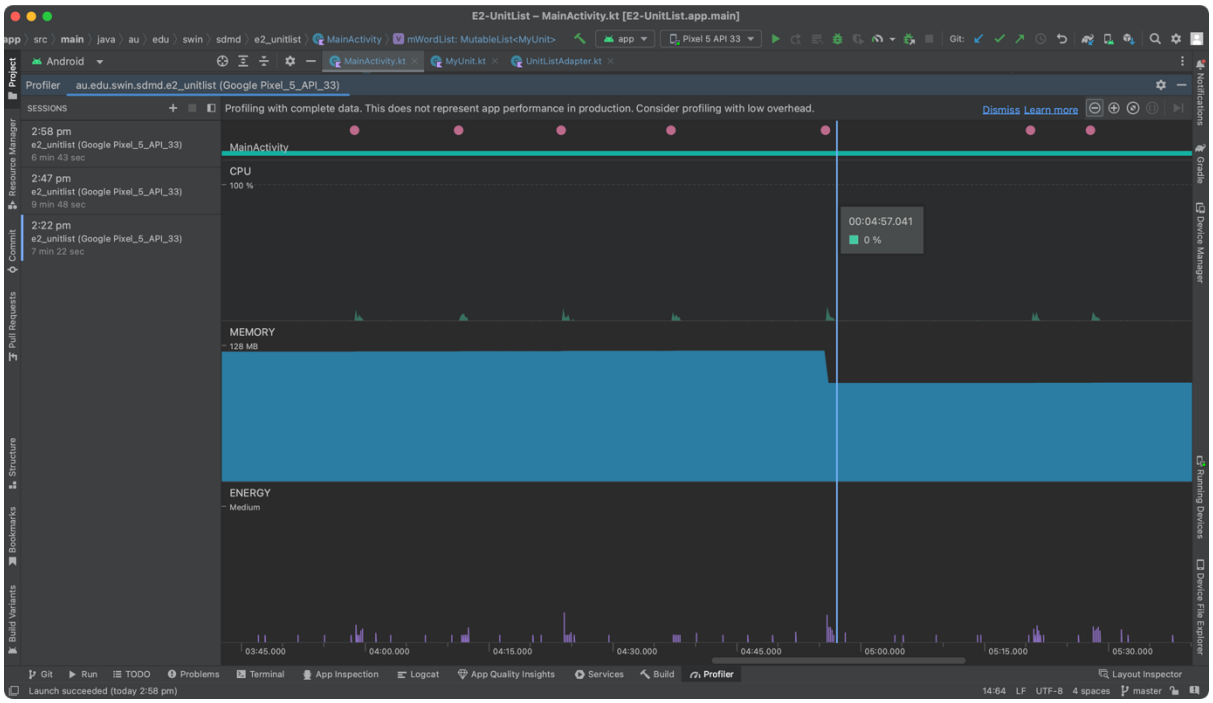

Figure 3: Scenario 1 on the 11th click.

On the 11th click however, the memory usage went from 123.7 MB down to 93.3 MB. This may indicate that some sort of memory optimization of garbage collection may be occurring after a certain number of clicks.

Scenario 1 summary:

- In this scenario, each time 'bind()' is called, 'item.drawIcon()' is used to set the icon. It implies that every time a bind operation is performed, a new icon is generated.
- The initial memory usage may have increased because icons for each item were repeatedly created and loaded during the initial rendering. The subsequent memory decrease raises the possibility of a heap dump and raises questions about how memory is refreshed.

Key Observations:

- A high CPU load at first may affect how quickly a device responds.
- Memory reduction following several clicks suggests potential memory management or garbage collection improvements.
- If dynamic icon generation is required, the scenario might be appropriate, but memory management needs to be carefully taken into account.

Scenario 2: A generated icon but created on initialisation.

This scenario shows consistent memory growth with no memory reduction.

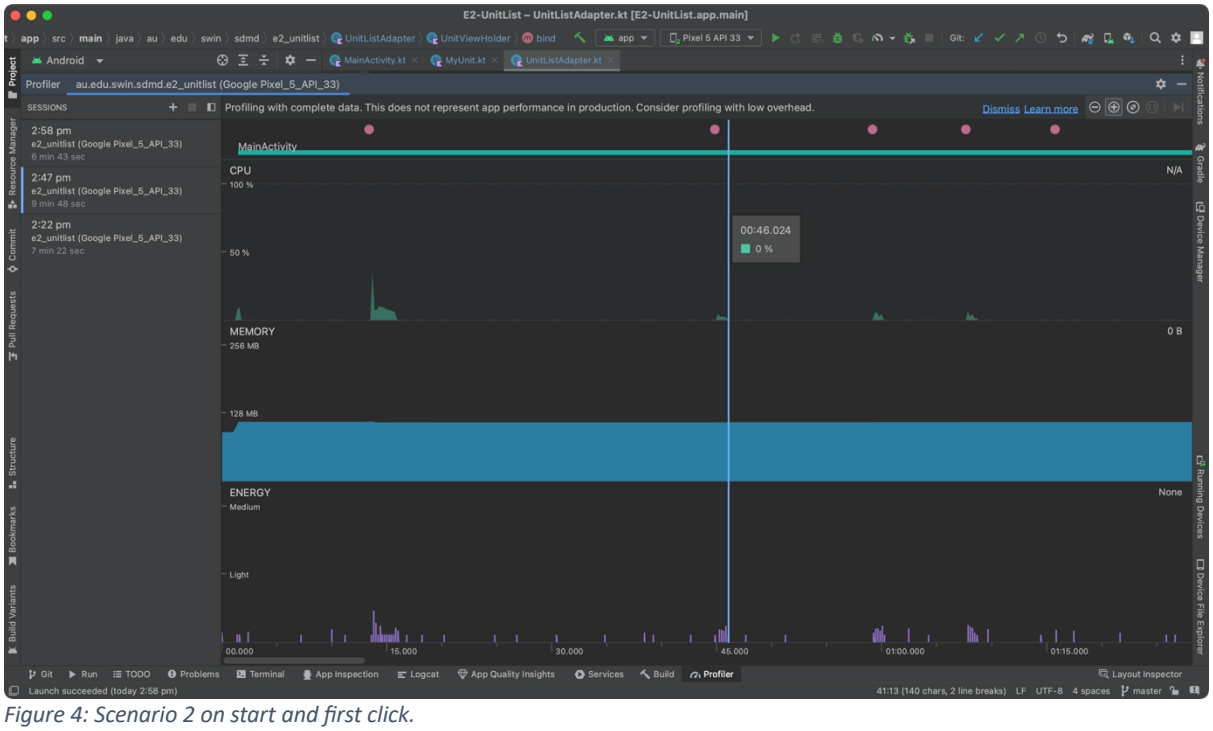

Figure 4: Scenario 2 on start and first click.

On initialising the second scenario, the CPU usage starts at 16% and the memory started at 93.6MB then went up to 112.8MB. I'm pretty sure it went up to 120.4MB on my previous tests. After clicking the FAB for the first time, the CPU usage had a spike to 36% but the allocated RAM somehow went down to 111.7MB – 111.9MB.

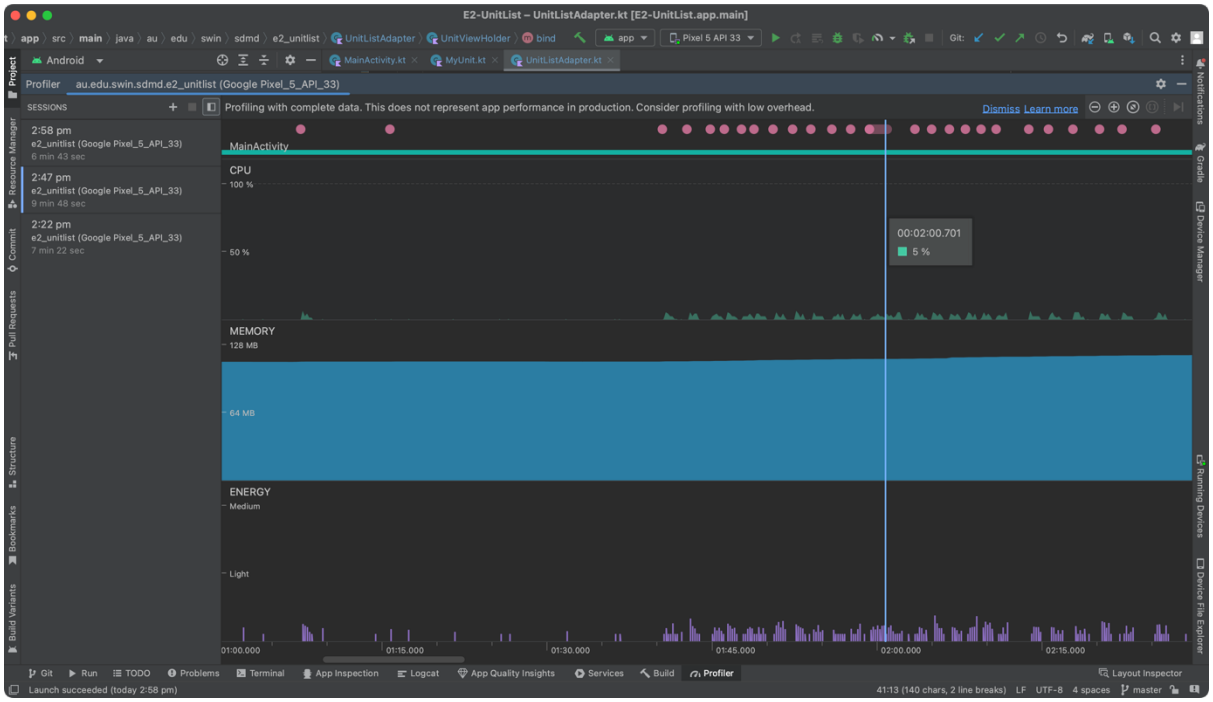

Figure 5: Scenario 2 on the next 20+ clicks.

After trying to click for more than 20 times, and also trying to scroll up and down the list, I can't find anything significant on this scenario. The memory usage increases consistently by 0.2MB per FAB click. There is no significant memory reduction observed.

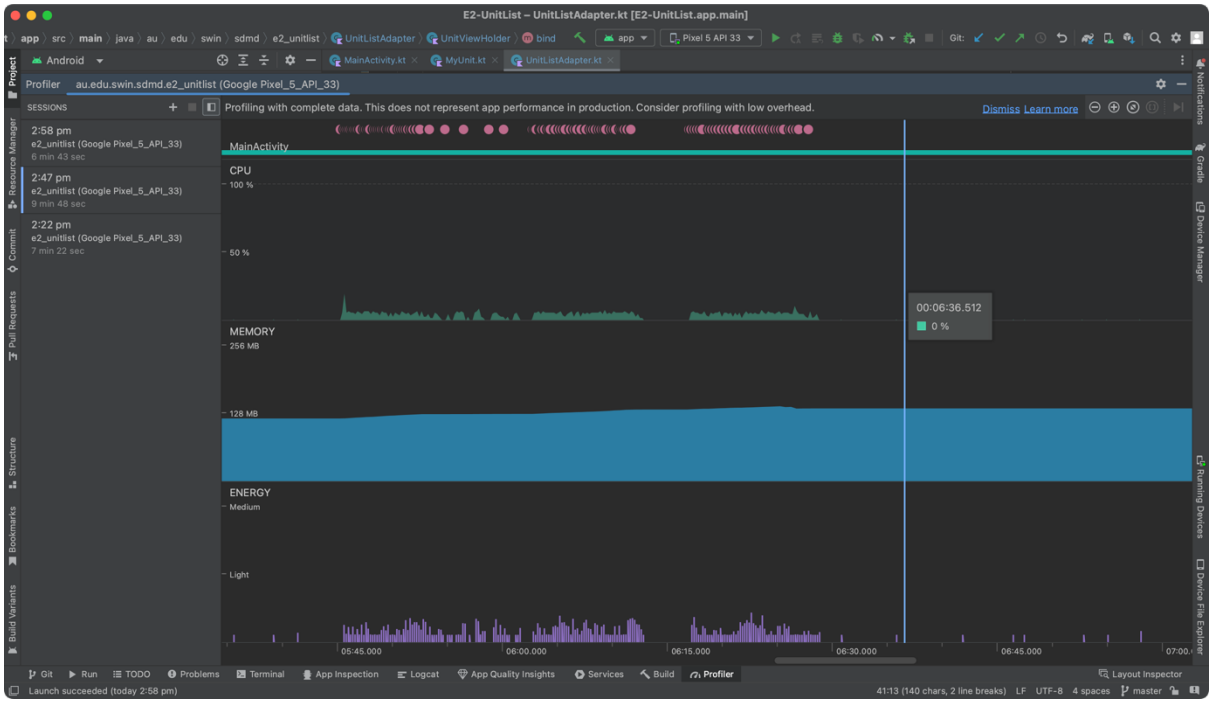

Figure 6: Scenario 2 after more than 50 clicks and scrolling.

As we can see here, after I tried to do more actions to see if there will be any action like Scenario 1, the only thing I can find is that the memory increases gradually until 142MB then went down to 138MB and stays there until I terminated the app.

Scenario 2 summary:

- In this case, it is likely that the list objects' initialisation is when the icons are generated, and they are constant across all items. This makes the memory usage predictable and consistent.
- The extra 0.2MB of memory required for each click is probably the result of resource management overhead or the building up of temporary resources with each click.

Key observations:

- Long-term memory problems may result from constant memory growth without memory reduction.
- Excessive CPU use during initialisation and FAB click could affect how responsive the app is.

Scenario 3: A constant icon.

This scenario has fluctuating CPU usage and memory growth, with occasional memory drops, which might be due to memory management or resource recycling.

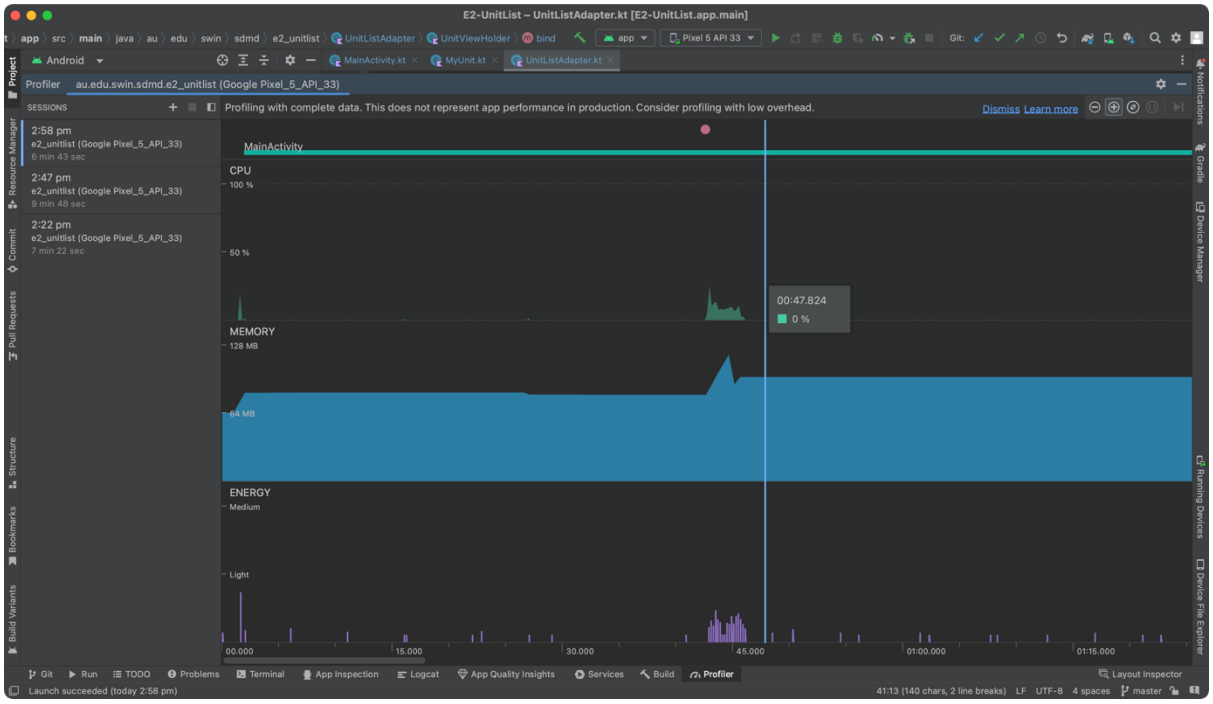

Figure 7: Scenario 3 and first click.

On this scenario, the CPU usage starts at 20%, the memory starts at 65MB then went up towards 83.9MB and suddenly drops to 81.7MB before I clicked the FAB. On first click of the button, the CPU usage spikes up to 26% and there was also a spike on the memory allocated to 119.4MB but went straight down to 98.5MB and stays there until the next time the button is clicked.

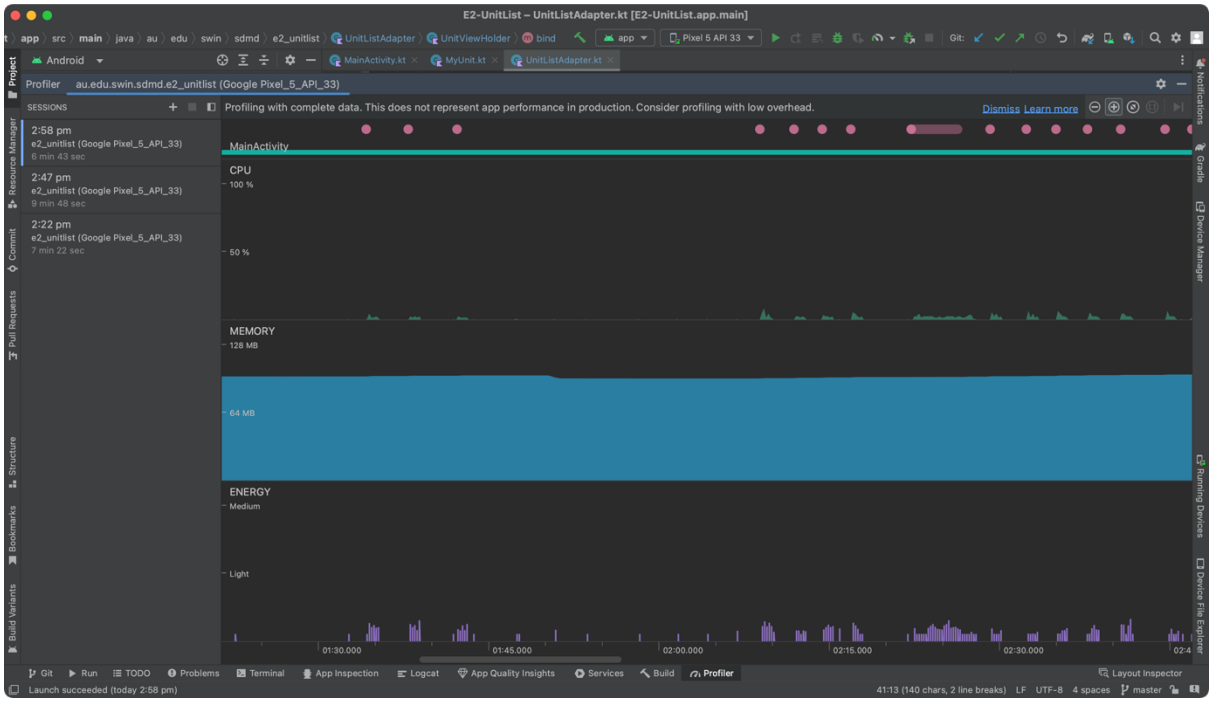

Figure 8: Scenario 3 on the next few clicks.

On the next few clicks however, the CPU usage was relatively low compared to the 1st click. The next few clicks only use 2% until 5% of CPU. The memory growth also increased by 0.3MB to 0.4MB per FAB click.

Figure 9: Scenario 3 after tons of clicks and scrolling action.

After I tried clicking the button multiple times and also tried scrolling, I can see a big difference on the memory usage compared to the last 2 scenarios. The memory increases by 2-3MB per actions which would cause the program to run slower than the other 2 methods. After reaching it's peak at 127.7MB, the memory usage dropped back to

84.5MB. Then when I tried to click on the button again, the CPU usage spiked at 18% and the memory added was 0.3MB per FAB click.

Scenario 3 summary:

- 'BitmapFactory.decodeResource()' is used to load the icon in this case from a fixed resource ('R.drawable.trencher'). This process is fairly straightforward and reliable.
- The fluctuating CPU usage and memory growth, followed by sharp memory drops, point to a potential periodic resource release or optimisation by Android's memory management system that would reduce memory.

Key observations:

- Low CPU usage makes the app more responsive and energy efficient.
- Fluctuating memory usage suggests periodic memory management or resource recycling.
- This scenario demonstrates more stable performance with reasonable memory growth.

Conclusion

Icons are generated, loaded, and managed differently in each scenario, which is why the results are inconsistent. These results are also influenced by Android's resource management and garbage collection mechanisms. The observed behaviours highlight the significance of taking these factors into account when enhancing app performance for a more seamless user experience. They reflect the complexity of resource handling in Android apps.

In my opinion, the effective management of a fixed resource is responsible for Scenario 3's performance. When compared to creating icons dynamically during each 'bind()' operation (as in Scenario 1) or during initialisation (as in Scenario 2), loading an existing icon from resources is typically more memory-efficient. Since this is the best course of action, in my opinion.

The worst scenario, however, is Scenario 2. It appears to have performed less well because it generates icons during initialisation while keeping constant icons for all items. It showed increased CPU activity and steady memory growth without appreciable memory reduction.

Another important point is that progress reporting to the user is crucial when using concurrency. Items that are loaded asynchronously can update the user interface with progress bars, keeping users informed of ongoing background processes. Additionally, concurrency enables multiple tasks to run simultaneously, possibly taking advantage of

multiple CPU cores. You can divide the tasks into parallel threads when loading a large number of items from a file. Faster loading times result from the ability of each thread to handle a portion of the file. This maintains a responsive user interface while also enhancing performance.

All in all, careful consideration of how images are generated and managed, especially in list-based interfaces, is crucial for achieving optimal performance. Balancing memory usage, CPU efficiency, and user experience is a key challenge when working with lists and images in Android apps.